The present work was submitted to the LuFG Theory of Hybrid Systems

BACHELOR OF SCIENCE THESIS

A HEURISTICAL LOCAL SEARCH APPROACH FOR SOLVING SATISFIABILITY OF POLYNOMIAL FORMULAS

Jan Mattis Lipka

Communicated by
Prof. Dr. Erika Ábrahám

Examiners:
Prof. Dr. Erika Ábrahám
Prof. Dr. Jürgen Giesl

Additional Advisor:
Valentin Promies

Aachen, 08.05.2024

Abstract

Local search (LS) is an incomplete approach to solve satisfiability of formulas by iteratively changing the variable assignments slightly to obtain a solution to a particular problem. In this thesis, we discuss a local search approach in the theory of *Quantifier-Free Non-Linear Real Arithmetic*, i.e. Boolean combinations of multivariate polynomial constraints. The core concept of this algorithm is based on the fact that univariate polynomials only have finitely many roots, thus the domain can be split into finitely many sign-invariant regions for a certain variable. The algorithm then utilizes real root isolation to perform *jumps* between sign-invariant regions for a single polynomial based a heuristic score. Following the ideas introduced by Li et al. in 2023 [LXZ23], we implement a local search procedure and expand it to overcome shortcomings of the initial approach. The implemented approach is then tested on a multitude of problem instances with the focus being set on local search as a stand-alone solver on SMT-LIB and self-generated instances, as well as local search in a DPLL(T) setting and the effects different internal parameters have on the effectiveness of local search. We also combine local search with SMT-RAT's default solving strategy. Benchmarks show that the implemented local search is *not* competitive in instances similar to SMT-LIB as a stand-alone solver, but offers drastic improvements in satisfying high degree polynomial formulas. It is also shown that local search does not thrive in a DPLL(T) setting on SMT-LIB and similar instances and is highly dependent on internal parameters. The combined approach, however, improves the current best solving strategy in SMT-RAT for SMT-LIB and generated instances.

Contents

Chapter 1

Introduction

Solving formulas with multivariate polynomial constraints is one of the more prominent tasks of modern SMT-Solving. In this thesis, we attempt to solve *Satisfiability Modulo the Theory of Quantifier-Free Non-Linear Real Arithmetic*, or QFNRA, via means of a new approach, the so called *Local Search* (LS), as introduced by Li et al. [LXZ23]. QFNRA was shown to be decidable by Tarski in 1951 [Tar51], yet the only practical algorithmic way is Collin's *Cylindrical Algebraic Decomposition* (CAD) [Col74] and its extensions. By definition, it divides the infinite search space into finitely many regions with certain properties. In its original form, the answer can only be determined at the very end after constructing each region and evaluating sample points from the regions. While CAD is a complete solving procedure for QFNRA, its main drawback is its doubly exponential runtime complexity.

Hence, a number of *in*complete procedures have been developed, such as *Subtropical Satisfiability* by Fontaine et al. [FOSV17] or *Interval Constraint Propagation* by Benhamou & Older [BO97]. These procedures cannot show satisfiability or unsatisfiability of *all* possible QFNRA problems, hence the term *incomplete*. The aforementioned local search approach, being the main focus of this thesis, also falls under the category of incomplete solving procedures. To use local search for solving problem instances in the theory of QFNRA, it is necessary to keep an assignment for all variables contained in a given problem instance throughout the searching procedure. This assignment is then iteratively altered throughout the execution of local search. Each iteration tries to find a new assignment similar to the current assignment which is, however, *heuristically closer* to satisfying the given formula. These iterations will continue until a termination condition is met. For this, two core concepts are of importance. Firstly, a set of rules is to be defined which sets guidance as to how the assignment is changed per iteration. These rules yield different classes of *operations*, such that per iteration one operation out of any class is performed. Secondly, a heuristic function is necessary to determine the effectiveness of each possible operation to rate them and select the best operation for the current assignment value according to the heuristic.

This bachelor thesis aims to answer the question whether local search can be viable for solving QFNRA problem instances, while also expanding the SMT-solver SMT-RAT [CKJ+15] currently being developed by the Theory of Hybrid Systems group at RWTH Aachen University, with its own local search procedure. To achieve answering the research question of this thesis, a local search procedure based on Li et al. was implemented as a module in SMT-RAT and adapted to work in the solver's

environment, as well as expanded upon to overcome disadvantages of the initial idea. In addition, the implementation was then benchmarked on a multitude of problem instances for various versions such as a stand-alone solver, in an DPLL(T) strategy, using different internal settings or as a combined solver using SMT-RAT's default strategy. For references, all benchmark sets were also tested on SMT-RAT's default solving strategy. The problem instances for the benchmarks consist of the current set of *satisfiable* QFNRA problem instances from SMT-LIB [BFT16], as well as multiple sets of self-generated instances with different properties.

In the following, the necessary notation for QFNRA and local search will be introduced in Chapter 2. In Chapter 3 the initial algorithm will be presented, as well as the theory behind local search, in particular what the operations are, how and why they work, as well as what the used heuristic is. Following the theory, the initial algorithm will be improved in Chapter 4. Chapter 4 also shows the pseudo-code of the implemented procedure in addition to the various different sets of internal settings. The algorithm is then implemented to be benchmarked and analysed in Chapter 5. Finally, the benchmark results are summarized and discussed in Chapter 6 to form a final conclusion.

Chapter 2

Preliminaries

2.1 Notation

Let a finite variable vector $\bar{x} = \bar{x}_{\mathrm{REAL}} \circ \bar{x}_{\mathrm{BOOL}} = (x_{r,1},...,x_{r,m_{\mathrm{REAL}}}, x_{b,1}...,x_{b,m_{\mathrm{BOOL}}})$ be given as the concatenation of the two vectors $\bar{x}_{\mathrm{REAL}} = (x_{r,1},...,x_{r,m_{\mathrm{REAL}}})$ and $\bar{x}_{\mathrm{BOOL}} = (x_{r,1},...,x_{r,m_{\mathrm{BOOL}}})$ containing m_{REAL} *real*-valued variables and m_{BOOL} *Boolean*-valued variables respectively. In addition, let $n = m_{\mathrm{REAL}} + m_{\mathrm{BOOL}}$ such that $\bar{x}$ can be written as $\bar{x} = (x_1, ..., x_n)$ in use-cases that do not rely on the different types of variables. Furthermore, we allow the notation to use any finite variable vector $\bar{x}$ similarly to a *set* if necessary. In particular, writing $\bar{x}$ in cases that require a set translates to $\{x_i \mid x_i \in \bar{x}\}$ in which $x_i \in \bar{x}$ denotes the element x_i in the vector $\bar{x}$. Furthermore, $\bar{x} \subseteq \bar{x}'$ is defined for a second finite variable vector $\bar{x}'$ such that $\bar{x} \subseteq \bar{x}'$ holds if and only if the vector $\bar{x}'$ contains every variable $x_i \in \bar{x}$, i.e. $\bar{x} \subseteq \bar{x}' \Leftrightarrow \{x_i \mid x_i \in \bar{x}\} \subseteq \{x'_i \mid x'_i \in \bar{x}'\}$. Hence, $\bar{x}'$ might also contain variables *not* in $\bar{x}$. Intuitively, $\bar{x}'$ is a finite extension of $\bar{x}$ with new variables. Lastly, writing a finite variable vector $\bar{x}$ can then be used as a domain in function definitions to map each variable $x_i \in \bar{x}$ to an element in the function's range. For the set of rational numbers $\mathbb{Q}$, let $\mathbb{Q}[\bar{x}_{\mathrm{REAL}}]$ be the polynomial ring consisting of variables $x_{r,i} \in \bar{x}_{\mathrm{REAL}}$ and coefficients $q_j \in \mathbb{Q}$. In addition, let $\mathbb{B}[\bar{x}_{\mathrm{BOOL}}]$ be the set of Boolean variables $x_{b,i} \in \bar{x}_{\mathrm{BOOL}}$ and their respective negations $\neg x_{b,i}$. For a polynomial $p(\bar{x}) \in \mathbb{Q}[\bar{x}_{\mathrm{REAL}}]$, let $VAR(p)$ be the set of contained variables.

Definition 2.1.1 (**Polynomial Formulas**). *A polynomial formula F_{pol} in conjunctive normal form (CNF) has the form*

$$F_{pol} = \bigwedge_{P_i \in \Lambda} \bigvee_{p_{ij} \in P_i} p_{ij} \ \rhd_{ij} \ 0,$$

in which $\Lambda = \{P_1, ..., P_m\}$ denotes a finite set of finite, non-empty subsets $P_i \subset \mathbb{Q}[\bar{x}_{REAL}]$ and $\rhd_{ij} \in \{<, =, >\}$ indicates the relation between the polynomial p_{ij} and 0.

Additionally, $p_{ij} \ \rhd_{ij} \ 0$ will be referred to as an *atomic polynomial formula* and $\bigvee_{p_{ij} \in P_i} p_{ij} \ \rhd_{ij} \ 0$ as a *clause*.

Definition 2.1.2 (**Real Assignment**). *Let F_{pol} be a polynomial formula with real variables $\bar{x}_{REAL}$. Then, the mapping $\alpha : \bar{x}_{REAL} \to \mathbb{R}$ is a real assignment.*

A real assignment α_{REAL} for a polynomial formula F_{pol} assigns each variable $x_{r,i}$ in F_{pol} an element $a_i \in \mathbb{R}$. The assignment α_{REAL} can also be written as an assignment vector $\alpha_{\mathrm{REAL}}(\bar{x}_{\mathrm{REAL}}) = (a_1, ..., a_{m_{\mathrm{REAL}}})$.

In SMT-Solving, not every formula input F is already in CNF. Because the proposed local search procedure should also work on those instances, a conversion to CNF is necessary for the heuristic to work properly. For this, Tseitin's encoding [Tse83] is used, which transforms F into an equi-satisfiable formula F' in CNF. This, however, introduces Boolean variables to the generated formula. Thus, altered definitions for formulas and assignments are necessary and will be used throughout the thesis.

Definition 2.1.3 (**Formula**). *Let $\Lambda = \{L_1, ..., L_m\}$ be a finite set of finite literal sets L_i such that each literal l_{ij} is defined as*

$$l_{ij} ::= p_{ij} \ \rhd_{ij} \ 0 \quad | \quad b_i \tag{2.1}$$

for $p_{ij} \in \mathbb{Q}[\bar{x}_{REAL}]$ a polynomial, $\rhd_{ij} \in \{<, =, >\}$ a relation and $b_i \in \mathbb{B}[\bar{x}_{BOOL}]$ a Boolean variable or its negation. Then, a formula F in CNF for the variable vector $\bar{x}$ has the form

$$F = \bigwedge_{L_i \in \Lambda} \bigvee_{l_{ij} \in L_i} l_{ij}. \tag{2.2}$$

It is easy to see that a formula has the same general structure as a *polynomial* formula, but it also includes Boolean variables as literals. Because of these introduced Boolean variables, the alteration of an assignment must now also include a mapping of Boolean variables. This Boolean assignment is defined similarly to the real assignment.

Definition 2.1.4 (**Boolean Assignment**). *Let $\bar{x}$ be a Boolean variable vector. Then, $\alpha : \bar{x}_{BOOL} \to \{\mathsf{true}, \mathsf{false}\}$ is a Boolean assignment.*

A Boolean assignment α_{BOOL} assigns each variable $x_{b,i}$ in $\bar{x}$ an element $a_i \in \{\mathsf{true}, \mathsf{false}\}$. The assignment α_{BOOL} can also be written as an assignment vector $\alpha_{\mathrm{BOOL}}(\bar{x}_{\mathrm{BOOL}}) = (a_1, ..., a_{m_{\mathrm{BOOL}}})$. Thus, an assignment α for a formula F containing real and Boolean variables adds upon a real assignment α_{REAL} by also providing Boolean values for the Boolean variables that are contained in F using the mapping $\alpha_{\mathrm{BOOL}} : \bar{x}_{\mathrm{BOOL}} \to \{\mathsf{true}, \mathsf{false}\}$.

Definition 2.1.5 (**Assignment**). *Let F be a formula with variable vector $\bar{x} = \bar{x}_{REAL} \circ \bar{x}_{BOOL}$, $\alpha_{REAL}(\bar{x}_{REAL})$ the real assignment and $\alpha_{BOOL}(\bar{x}_{BOOL})$ the Boolean assignment for the variables. Then, the mapping $\alpha : \bar{x} \to \mathbb{R} \cup \{\mathsf{true}, \mathsf{false}\}$ is an assignment using either $\alpha_{REAL}(\bar{x}_{REAL})$ or $\alpha_{BOOL}(\bar{x}_{BOOL})$ for each variable based on their type.*

In addition to Definition 2.1.5, α can also be written as an assignment vector $\alpha(\bar{x}) = (a_1, ..., a_n)$, or even more precisely as $\alpha(\bar{x}) = (a_{r,1}, ..., a_{r,m_{\mathrm{REAL}}}, a_{b,1}, ..., a_{b,m_{\mathrm{BOOL}}})$ denoting the different variable types. If the underlying variable vector $\bar{x}$ is intuitively clear or not changing, the assignment will be referred to solely as α.

2.2 Evaluation

To evaluate a formula F as introduced in Definition 2.1.3 with variable vector $\bar{x}$, an assignment $\alpha(\bar{x}')$ is required for $\bar{x} \subseteq \bar{x}'$. Using $\bar{x}'$, it is ensured that each variable

$x \in \bar{x}$ has a well-defined assignment value a_i. By substituting x_i for a_i in every atomic formula and evaluating them under standard arithmetic, each atomic formula is either true or false. Then, the formula F is easily evaluated using the standard semantics of $\wedge$ and $\vee$ to yield either true or false for the *whole* formula F.

Definition 2.2.1 (**Satisfying/Falsifying Assignment**)**.** *An assignment α is a satisfying assignment for a formula F if and only if F evaluated at α is* true*. It can be written as $\alpha \models F$. If an assignment α for a formula F yields* false*, it is called a falsifying assignment, denoted as $\alpha \not\models F$.*

Definition 2.2.2 (**Satisfiable/Unsatisfiable Formula**)**.** *A formula F is satisfiable if and only if a satisfying assignment α exists for F. If every assignment α mapping exactly the variables $x_i \in \bar{x}$ used in the formula F is a falsifying assignment, then F is unsatisfiable.*

Chapter 3

Local Search

Local search (LS) in the theory of QFNRA is based on the simple idea of iteratively altering an assignment α using operations such that the new assignment α' is near the previous assignment while the input formula evaluated at the new assignment α' is closer to satisfaction according to a heuristic.

In the following chapter, all necessary concepts for LS in QFNRA will be described. This includes the initial local search idea, the search space, the different kinds of operations and how they work, as well as the heuristic. The core concept, operations to change the assignment from Definitions 3.3.1, 3.3.3 and 3.3.4, as well as the initial heuristic are based on the ideas proposed by Li et al. [LXZ23].

3.1 Initial Local Search Approach

First, we describe the initial outline of local search. It can be broken down into 4 main steps that are executed along the order of appearance or based on guidance presented in the steps. It is noteworthy that, as local search is an incomplete approach, it does need a termination condition such that it does not run indefinitely in cases where it is impossible for local search to find a solution. Hence, local search is a *timed* solver that only will try to solve the problem for a given amount of time. For the understanding of local search, it is necessary to briefly introduce *operations*. An operation changes the current assignment to a different assignment. They are classified into *single* operations changing only the assignment for a single variable, and *multi* operations changing the assignment for multiple variables at once for a given direction vector. These operations will be properly defined in the following Section 3.3.

Definition 3.1.1 (**Local Search Outline** [LXZ23])**.** *Given an input formula F, the initial approach tries to solve F after initializing every real variable with 1 and every Boolean variable with* true *as follows:*

1. *Check termination condition:*
   Formula is satisfied by assignment, return sat
   Allotted time has run out, return unknown

2. *Generate all single cell jump operations and rate them heuristically. If no operation op with $h(op) > 0$ exists, go to* (3)*. Else, update assignment and go to* (1)*.*

3. *Update clause weights using PAWS.*

4. *Generate direction vectors, then generate all multi cell jump operations based on those direction vectors and rate them heuristically. If no operation op with* $h(op) > 0$ *exists, return* unknown. *Else, update assignment and go to* (1).

The approach stays very simple and it is thus easy to see that is attempts solving polynomial formulas by iteratively satisfying false constraints until either the formula is satisfied or the time limit is met. Hence, it always terminates. Its simplicity, however, also causes some drawbacks and leaves space for improvements which will be addressed in Chapter 4.

3.2 Setup

In this section, we will introduce all the necessary concepts and ideas to be able to define the operations in Section 3.3 out of which to build the local search. The *search space* for any as previously defined problem instance F is

$$S = \{(s_{r,1}, ..., s_{r,m_{\text{REAL}}}, s_{b,1}, ..., s_{b,m_{\text{BOOL}}}) \mid s_{r,1},..., s_{r,m_{\text{REAL}}} \in \mathbb{R} \wedge s_{b,1}, ..., s_{b,m_{\text{BOOL}}} \in \{\mathsf{true}, \mathsf{false}\}\}. \tag{3.1}$$

Intuitively, any assignment vector α yields a point $(a_1, ..., a_n) \in S$ for which real variables are mapped to $a_i \in \mathbb{R}$ and Boolean variables are mapped to $a_i \in \{\mathsf{true}, \mathsf{false}\}$. It is now important to find points, i.e. assignment vectors, that have the potential to provide a different truth value for F evaluated at the given point. Because the search space is infinitely large, we want to partition S into regions, i.e. *cells*, with certain properties such that only a *single* point α out of any cell would need to be checked to decide whether *all* or *no* points α' from a given cell are satisfying assignments.

Definition 3.2.1 (Cell). *Let* F *be a formula with variable vector* $\bar{x} = \bar{x}_{REAL} \circ \bar{x}_{BOOL}$. *Then, a cell is a connected region* R *in* S *such that*

$$\begin{aligned} \forall \quad & a = (a_{r,1}, ..., a_{r,m_{REAL}}, a_{b,1}, ..., a_{b,m_{BOOL}}) \in R, \\ & b = (b_{r,1}, ..., b_{r,m_{REAL}}, b_{b,1}, ..., b_{b,m_{BOOL}}) \in R, \\ & p(\bar{x}_{REAL}) \in F : \\ & sgn(p(a_{r,1}, ..., a_{r,m_{REAL}})) = sgn(p(b_{r,1}, ..., b_{r,m_{REAL}})) \wedge \\ & (a_{b,1}, ..., a_{b,m_{BOOL}}) = (b_{b,1}, ..., b_{b,m_{BOOL}}) \end{aligned} \tag{3.2}$$

Following this definition, a cell C for a formula F can also be seen as a region in S such that no literal $l_{ij} \in F$ will change its truth value evaluated at any point in C. Because of this, a formula F only has to be evaluated at a single point from C to cover the whole cell C. It is noteworthy that a cell does not have to be of maximum size, i.e. it does not have to be non-expandable. Furthermore, it is also important to show that the *whole* search space S can be partitioned into finitely many cells. Otherwise, there would be a dense part of S that we could not easily cover with a single sample point, but only infinitely many.

Theorem 3.2.1 (Finite Cell Covering). *Let* F *be a formula. Then, the search space* S *can be partitioned into finitely many cells.*

Proof. By the theory of CAD, $\mathbb{R}^{m_{\text{REAL}}}$ can be divided into finitely many sign-invariant regions $C = \{C_1, ..., C_k\}$. But each point $a = (a_{r,1}, ..., a_{r,m_{\text{REAL}}}, a_{b,1}, ..., a_{b,m_{\text{BOOL}}}) \in S$ also contains m_{BOOL} many Boolean variables. Hence, adding true or false for each Boolean variable yields the cells $C' = C \times \{\mathsf{true}, \mathsf{false}\}^{m_{\text{BOOL}}}$ such that $|C'| = k \cdot 2^{m_{\text{BOOL}}} \in \mathbb{N}$. Thus, S can be partitioned into finitely many cells. □

Theorem 3.2.1 yields the fact that for any satisfiable formula F, there exists a cell C such that any point $r \in C$ is a satisfying assignment for F. A local search approach that switches between cells might switch to a satisfying cell, thus finding a solution. The way of switching cells used in our local search approach is based on satisfying a currently falsified constraint. By doing so, the sign condition in Definition 3.2.1 is violated, thus changing to a new cell. For this simple cell-switching, *sample points* are necessary at which to test a given currently falsified polynomial for satisfaction. We will start by finding sample points for *univariate* polynomials via means of real root isolation.

Definition 3.2.2 (**Root Isolating Intervals** [LXZ23])**.** *Let $p(x) \in \mathbb{Q}[x]$ be a univariate polynomial with s real roots. Then, $I = \{(a_1, b_1), ..., (a_s, b_s)\}$ are the root isolating intervals of p such that*

$$\begin{aligned} &\forall i \in \{1, ..., s\} : \; a_i, b_i \in \mathbb{Q} \;\wedge\; a_i < b_i \;\wedge\; (a_i, b_i) \textit{ contains exactly one real root of } p \\ &\forall i \in \{1, ..., s-1\} : \; b_i < a_{i+1} \end{aligned} \tag{3.3}$$

It is easy to see that every root r_k of a univariate polynomial $p(x) \in \mathbb{Q}[x]$ is contained in exactly one interval $(a_k, b_k) \in I$ from its root isolating intervals I. Because of the open bounds, r_k cannot be located at an exact bound a_k or b_k but must be strictly between the bounds. Figure 3.1 depicts such intervals graphically. Out of these root isolating intervals, we can construct a set of sample points which might satisfy the current falsified constraint.

Definition 3.2.3 (**Sample Points**)**.** *Let $I = (a_1, b_1), ..., (a_s, b_s)$ be the root isolating intervals for a univariate polynomial $p(x) \in \mathbb{Q}[x]$. Then, the set of sample points is defined as*

$$SP = \{a_1, b_s\} \cup \bigcup_{i=1}^{s-1} \left\{ b_i, \frac{b_i + a_{i+1}}{2}, a_{i+1} \right\}. \tag{3.4}$$

Each point $sp_i \in SP$ is a sample point for $p(x)$. If p evaluated at sp_i yields a positive value, sp_i is a *positive* sample point. If p evaluated at sp_i yields a negative value, sp_i is a *negative* sample point. It is noteworthy that the set of sample points is an over-approximation, i.e. it contains more points than necessary to have a single sample point for each sign-invariant region that is not a root. Having more sample points, however, will yield more potential points to satisfy the currently falsified constraint and thus also potential different heuristic scores. Figure 3.2 shows the positive and negative sample points for the example polynomial introduced in Figure 3.1. Note that a univariate polynomial might not have a single root, thus it might not produce any sample point.

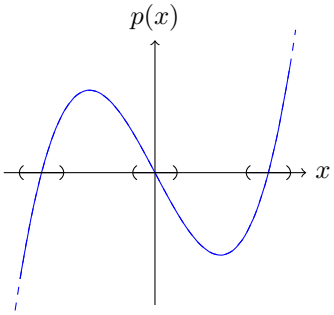

Figure 3.1: Root isolating intervals for a univariate polynomial $p(x)$ containing 3 real roots.

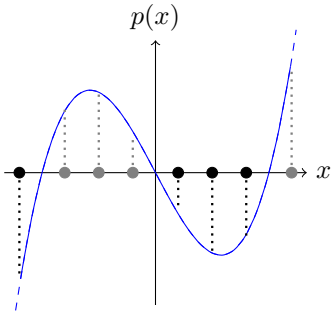

Figure 3.2: Positive (*gray*) and negative (*black*) sample points for a univariate polynomial $p(x)$.

3.3 Operations

The *sample points* introduced in Definition 3.2.3 can now be used to create operations *op* that change the assignment α to a different assignment corresponding to a point in a different cell. This new assignment might have a different truth value for the formula F by changing at least the truth value for one literal l_{ij}. These operations will satisfy a given polynomial constraint falsified under α using the relation $\rhd \in \{<, =, >\}$ or a false Boolean literal by adjusting one or multiple variables accordingly, thus moving the current assignment to a different cell. In the following, two different types, *single-* and *multi- cell-jump* operations, based on the number of variables the operation changes will be introduced.

Definition 3.3.1 (Operation: Coordinate Axis Cell Jump [LXZ23]). *Let F be a formula with variable vector $\bar{x} = \bar{x}_{REAL} \circ \bar{x}_{BOOL}$, current assignment α, $l = (p(\bar{x}_{REAL}) \rhd 0)$ a polynomial constraint in F so that $\alpha \not\models l$ for $\rhd \in \{<, >\}$ and $x_{r,i} \in VAR(p)$. Let $p_i(x_{r,i})$ be the polynomial p after substituting $a_j \in \alpha$ for $x_{r,j}$ for every variable $x_{r,j} \neq x_{r,i}$, $x_{r,j} \in VAR(p)$ and $l' = (p_i(x_{r,i}) \rhd 0)$ be the corresponding univariate constraint. Lastly, let SP_i be the set of sample points for $p_i(x_{r,i})$. Then, $\textsc{cjump}(x_{r,i}, l)$ is the coordinate axis cell jump operation which assigns $x_{r,i}$ the sample point $sp_i \in SP_i$ closest to $a_i \in \alpha$ satisfying l'.*

The operation is constructed for a polynomial constraint currently falsified under an assignment α and a real variable contained in the constraint. It aims to change the variable's assignment to a rational value such that the constraint is satisfied if the variable is assigned its new value and every other variable keeps its assignment value. The operation substitutes every real variable $x_{r,j} \neq x_{r,i}$ for its value a_j in the current assignment in the polynomial constraint, thus making the constraint's polynomial univariate for which the sample point set is then constructed. Then, the closest positive or negative sample point to a_i is taken from the set of sample points depending on the type of the relation to 0. Intuitively, one can think of this operation $\textsc{cjump}(x_{r,i}, l)$ as searching for a point a_i along $x_{r,i}$'s coordinate axis in $\mathbb{R}^{m_{\text{REAL}}}$ that satisfies the constraint posed by the literal l. It is noteworthy that if no sample point exists that satisfies the corresponding univariate constraint l', we say that the operation $\textsc{cjump}(x_{r,i}, l)$ does not exist. Lastly, as only the variable $x_{r,i}$ is being changed in the coordinate axis cell jump operation $\textsc{cjump}(x_{r,i}, l)$, it is classified

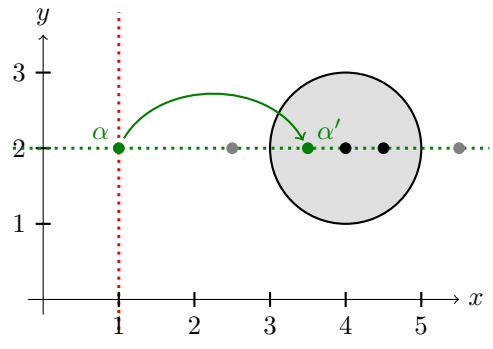

Figure 3.3: Completed CJUMP(x, l) and attempted CJUMP(y, l) for $l = \big((x-4)^2 + (y-2)^2 - 1 < 0\big)$ under assignment $\alpha = (1{,}2)$ using sample points from Figure 3.4.

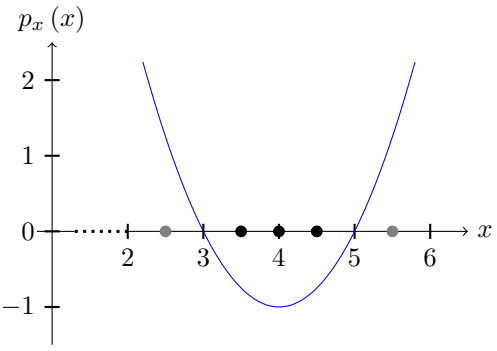

Figure 3.4: Sample points for $l = \big((x-4)^2 + (y-2)^2 - 1 < 0\big)$ under assignment $\alpha = (1{,}2)$ evaluated for x by substituting $y = 2$ into l yielding constraint $x^2 - 8x + 15 < 0$.

as a *single*-CJUMP-operation.

Example 3.3.1 (**Coordinate Axis Cell Jump**). *Consider the two dimensional polynomial constraint* $l = \big((x-4)^2 + (y-2)^2 - 1 < 0\big)$ *currently falsified under assignment* $\alpha = (x \to 1, y \to 2)$. *Figure 3.3 shows the search intuition for* CJUMP(x, l) *(green) and* CJUMP(y, l) *(red) along the coordinate axes for the variables* x *and* y*, as well as the constraint's solution space and all potential new assignments. It also shows the successful jump* CJUMP(x, l) *that creates the new assignment* $\alpha' = \big(x \to \frac{7}{2}, y \to 2\big)$*, while* CJUMP$(y, l)$ *is not successful. Figure 3.4 shows the univariate polynomial* $p_x(x) = x^2 - 8x + 15$ *which was obtained after substituting the assignment value* $y = 2$ *into the original polynomial, as well as the set of sample points* $\{\frac{5}{2}, \frac{7}{2}, 4, \frac{9}{2}, \frac{11}{2}\}$ *for* x *on* $p_x(x)$ *out of which* $\frac{7}{2}$ *is chosen as it is the nearest satisfying sample point to* x*'s previous assignment.*

It is now left to show that the coordinate axis cell jump operation functions as intended, i.e. if and only if a single variable $x_{r,i}$ that is contained in a false polynomial literal l can be adjusted under the current assignment α such that l becomes satisfied, a coordinate axis cell jump operation will be found that produces an assignment which also satisfies l.

Theorem 3.3.1 (**Existence of Coordinate Axis Cell Jump** [LXZ23]). *Given a false polynomial constraint* $l = (p(\bar{x}_{REAL}) \rhd 0)$ *under an assignment* $\alpha = (a_1, ..., a_n)$ *for* $\rhd \in \{<, >\}$ *and a real variable* $x_{r,i} \in \bar{x}_{REAL}$*, there exists a* CJUMP$(x_{r,i}, l)$ *operation if and only if the set*

$$L = \{\alpha' = (a_{r,1}, ..., a_{r,i-1}, \theta, a_{r,i+1}, ..., a_{r,m_{REAL}}, a_{b,1}, ..., a_{b,m_{BOOL}}) \mid \theta \in \mathbb{R} \wedge \alpha' \vDash l\} \tag{3.5}$$

is non-empty.

Proof. $\Rightarrow$ Assume that a CJUMP$(x_{r,i}, l)$ operation exists. Per Definition 3.3, this operation only changes the assignment α for the variable $x_{r,i}$. As the operation *exists* per assumption, the sample point $sp \in \mathbb{Q}$ that $x_{r,i}$ is being mapped to must also exist. Using Definition 3.2.3 for sample points, the correctly chosen

sample point sp will satisfy the polynomial constraint l after every other variable has been substituted for its own assignment value. Hence, the new assignment $\alpha' = (a_{r,1}, ...,a_{r,i-1}, sp, a_{r,i+1}, ..., a_{r,m_{\text{REAL}}}, a_{b,1}, ..., a_{b,m_{\text{BOOL}}})$ with $x_{r,i}$'s mapping to sp will also satisfy l. Because this new assignment exists and satisfies l, it is contained in the set L, thus making L non-empty.
$\Leftarrow$ Assume that L is non-empty. This means that there exists a value $\theta \in \mathbb{R}$ such that the new assignment $(a_{r,1}, ...,a_{r,i-1}, \theta, a_{r,i+1}, ..., a_{r,m_{\text{REAL}}}, a_{b,1}, ..., a_{b,m_{\text{BOOL}}})$ satisfies the constraint l. After substituting every variable's assignment value for $x_j \neq x_{r,i}$, the polynomial p from the constraint l becomes the univariate polynomial $p_i(x_{r,i})$ in the constraint $l' = (p_i(x_{r,i}) \rhd 0)$. The following cases for the number of roots of $p_i(x_{r,i})$ exist.
(1) $p_i(x_{r,i})$ *has no root.* In this case, no value for $x_{r,i}$ will satisfy l, because l is per definition a false polynomial constraint and there exists no value for $x_{r,i}$ such that $p_i(x_{r,i})$ will change its sign. However, this case cannot occur because the lack of existence for such a value for $x_{r,i}$ also yields that L is empty.
(2) $p_i(x_{r,i})$ *has k roots* r_k. This creates the sign-invariant regions $(-\infty, r_1)$, $[r_1, r_1]$, (r_1, r_2), $[r_2, r_2]$, ..., and (r_k, ∞). Because of the constraint's relation of $<$ or $>$, θ cannot be located in any region that is a point interval. θ must be in an open interval region R_j with a root r_j on at least one interval endpoint, i.e. $(-\infty, r_1)$, (r_a, r_{a+1}) or (r_k, ∞). Due to the Definition 3.2.2 of root isolating intervals, there must exist a root isolating interval $I = (i_l, i_r)$ that covers the root r_j. Because the root isolating intervals are not point intervals, there must be an overlap between I and the root's region interval R_j. Because I cannot cover multiple roots, it cannot extend to a region that is adjacent to R_j. Hence, either i_l or i_r is also in R_j and thus either i_l or i_r also satisfies l'. Due to the Definition 3.2.3 of sample points, i_l and i_r are both sample points that will be considered when creating $\text{CJUMP}(x_{r,i}, l)$. Thus, every root yields one sample point that $\text{CJUMP}(x_{r,i}, l)$ could jump to. Assigning $x_{r,i}$ the closest satisfying sample point to $a_{r,i}$ completes the operation. □

Recall that Boolean literals were introduced by converting a polynomial formula F into an equi-satisfiable formula F' in CNF using Tseitin's encoding in Chapter 2. The next operation is an addition to the ideas introduced by Li et al. to also cover Boolean variables in the formulas. It aims to change the Boolean assignment of a Boolean variable to satisfy the given constraint.

Definition 3.3.2 (**Operation: Boolean Cell Jump**). *Let F be a formula, $\alpha = (a_{r,1}, ..., a_{r,m_{REAL}}, a_{b,1}, ..., a_{b,m_{BOOL}})$ an assignment and $l = x_{b,i}$ (or its negation $l = \neg x_{b,i}$) a falsified Boolean literal under α. Then, the Boolean cell jump operation $\text{CJUMP}(x_{b,i}, l)$ yields the new assignment $\alpha' = (a_{r,1}, ..., a_{r,m_{REAL}}, a_{b,1}, ..., a_{b,i-1}, \neg a_{b,i}, a_{b,i+1}, ..., a_{b,m_{BOOL}})$.*

As only the variable $x_{b,i}$ is being changed, the operation $\text{CJUMP}(x_{b,i}, l)$ is also classified as a *single* cell-jump operation. It is trivial to see that this operation always exists and changes the truth value for the literal from false to true.

Following two single cell jump operations, we will now introduce a *multi* cell jump operation. This operation aims to satisfy a given falsified constraint l under α by changing multiple variables along a given direction vector $\bar{d}$ to find a satisfying new assignment α' for l.

Definition 3.3.3 (**Operation: Fixed Line Cell Jump** [LXZ23]). *Let F be a formula with variable vector $\bar{x} = \bar{x}_{REAL} \circ \bar{x}_{BOOL}$, current assignment α, $l = (p(\bar{x}_{REAL}) \rhd 0)$ a*

polynomial constraint in F so that $\alpha \not\models l$ for $\rhd \in \{<, >\}$. Let $\bar{d} = (d_{r,1}, ..., d_{r,m_{REAL}}) \in \mathbb{Q}^{m_{REAL}}$ be a direction vector and $t \notin VAR(p)$ a new real variable. Let $p_t(t)$ be the univariate polynomial obtained by substituting $x_{r,i} = a_{r,i} + d_{r,i} \cdot t \ \forall\, x_{r,i} \in VAR(p)$ in $p(\bar{x}_{REAL})$ and $l' = (p_t(t) \ \rhd 0)$ be the corresponding univariate constraint. Lastly, let SP_t be the set of sample points for $p_t(t)$. Then, $\textsc{cjump}(\bar{d}, l)$ is the fixed line cell jump operation which assigns t the sample point $sp_t \in SP_t$ closest to 0 satisfying l' yielding $\alpha' = (a_{r,1} + d_{r,1} \cdot sp_t, ..., a_{r,m_{REAL}} + d_{r,m_{REAL}} \cdot sp_t, a_{b,1}, ..., a_{b,m_{BOOL}})$.

This operation introduces the auxiliary variable t, then substitutes each real variable $x_{r,i}$ in l with its corresponding line equation starting at α dependent on t, i.e. $x_{r,i} = a_{r,i} + d_{r,i} \cdot t$. The resulting univariate polynomial $p_t(t)$ in the constraint $l' = (p_t(t) \ \rhd \ 0)$ with the same relation as l can now be checked for positive or negative sample points depending on the type of relation in the constraint l. Then, the closest sample point sp_t for t to 0 is chosen that satisfies the constraint l'. Re-substituting the values according to $x'_{r,i} = a_{r,i} + d_{r,i} \cdot sp_t$ yields the new assignment α'. It is noteworthy that such a satisfying sample point sp_t must not always exist, i.e. if the variables cannot be adjusted in $\bar{d}$'s indicated direction to satisfy l'. In this case, we say that $\textsc{cjump}(dir, l)$ does not exist for such a direction. Intuitively, it is checked whether the line intersecting the assignment α with slope $\bar{d}$ hits the solution space for the given falsified literal l. It is also easy to see that the coordinate axis cell jump operation is just a special case of the fixed line cell jump operation in which the direction vector only contains a single non-zero entry thus marking the coordinate axis that is being searched. Due to the direction vector $\bar{d}$, this operation might find satisfying new assignments which were not found using a single cell jump operation. In particular, it might satisfy a falsified constraint l which cannot be satisfied using a single cell jump operation because no coordinate axis intersects l's solution space.

Because the operation $\textsc{cjump}(\bar{d}, l)$ can change more than a single variable dependent on the direction vector $\bar{d}$, it is classified as a *multi* cell-jump operation.

Example 3.3.2 (**Fixed Line Cell Jump**). *Consider the two dimensional polynomial constraint $l = \big((x-4)^2 + (y-2)^2 - 1 < 0\big)$ introduced in Example 3.3.1 and direction vector $\bar{d} = (3, 1)$. l is falsified under the assignment $\alpha = (x \to 1, y \to 1)$. Figure 3.5 shows a successful fixed line cell jump operation $\textsc{cjump}(\bar{d}, l)$ (green) and the search intuition for two failed fixed line cell jump operations (red) along direction vectors that do not intersect l's solution space. Figure 3.5 also shows l's solution space and all potential new assignments along direction $\bar{d}$. Substituting $x = 1 + 3t$ and $y = 1 + t$ into l yields the univariate constraint $l' = \big(10t^2 - 20t + 9 < 0\big)$ with roots $\left\{1 - \frac{\sqrt{10}}{10}, 1 + \frac{\sqrt{10}}{10}\right\}$ and sample points $\left\{\frac{3}{5}, \frac{4}{5}, 1, \frac{6}{5}, \frac{7}{5}\right\}$. The univariate polynomial $p_t(t)$ and sample points are depicted in Figure 3.6. The operation chooses the sample point $\frac{4}{5}$, re-substituting yields $x' = 3 + \frac{2}{5}$ and $y' = 1 + \frac{4}{5}$ and thus the new assignment $\alpha' = (3.4, 1.8)$.*

It is now left to show that the fixed line cell jump operation functions as intended, i.e. if and only if a direction $\bar{d}$ exists such that the assignment α can be altered in this direction for a falsified polynomial literal l under α, then the fixed line cell jump operation $\textsc{cjump}(\bar{d}, l)$ will be found that produces a satisfying assignment α' for l.

Theorem 3.3.2 (**Existence of Fixed Line Cell Jump** [LXZ23]). *Given a false polynomial constraint $l = (p(\bar{x}_{REAL}) \ \rhd \ 0)$ under assignment $\alpha = (a_1, ..., a_n)$ for $\rhd \in \{<, >\}$, a direction vector $\bar{d} \in \mathbb{Q}^{m_{REAL}}$ and a new real variable $t \notin VAR(p)$, there*

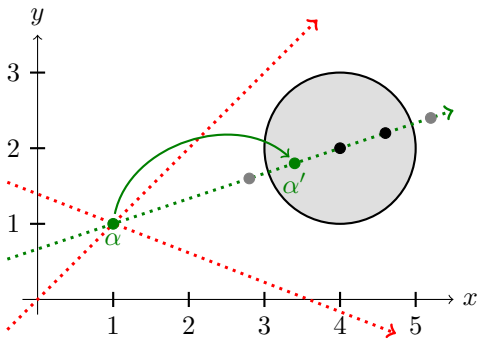

Figure 3.5: Completed CJUMP(dir, l) for $l = \big((x-4)^2 + (y-2)^2 - 1 < 0\big)$ under assignment $\alpha = (1{,}1)$ and $\bar{d} = (3, 1)$ using sample points from Figure 3.6.

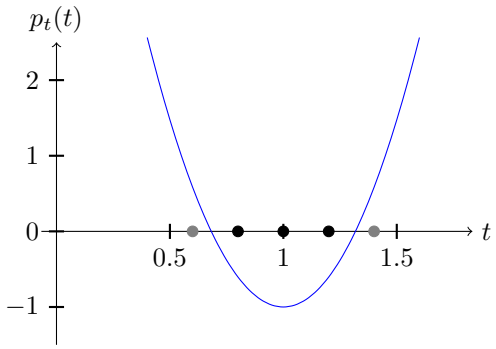

Figure 3.6: Sample points for $l' = \big(10t^2 - 20t + 9 < 0\big)$ under assignment $\alpha = (1{,}1)$ evaluated for t after substituting x and y into l.

exists a CJUMP$\big(\bar{d}, l\big)$ *operation if and only if the set*

$$L = \{\alpha' = (a_{r,1} + d_{r,1} \cdot \theta, ..., a_{r,m_{REAL}} + d_{r,m_{REAL}} \cdot \theta, a_{b,1}, ..., a_{b,m_{BOOL}}) \mid \theta \in \mathbb{R} \wedge \alpha' \vDash l\} \tag{3.6}$$

is non-empty.

Proof. $\Rightarrow$ Assume that CJUMP$\big(\bar{d}, l\big)$ exists. Per Definition 3.3.3, this operation changes the assignment $\alpha = (a_{r,1}, ..., a_{r,m_{\text{REAL}}}, a_{b,1}, ..., a_{b,m_{\text{BOOL}}})$ to $\alpha' = (a_{r,1} + d_{r,1} \cdot sp_t, ..., a_{r,m_{\text{REAL}}} + d_{r,m_{\text{REAL}}} \cdot sp_t, a_{b,1}, ..., a_{b,m_{\text{BOOL}}})$ for an existing sample point $sp_t \in \mathbb{Q}$. Because the operation exists, sp_t must have satisfied the substituted constraint l'. By re-substituting, the new assignment α' now also satisfies l. Hence, the set L is non-empty as it must contain the assignment for $\theta = sp_t$.
$\Leftarrow$ Assume that L is non-empty. This means that there exists a $\theta \in \mathbb{R}$ such that the new assignment $\alpha' = (a_{r,1} + d_{r,1} \cdot \theta, ..., a_{r,m_{\text{REAL}}} + d_{r,m_{\text{REAL}}} \cdot \theta, a_{b,1}, ..., a_{b,m_{\text{BOOL}}})$ satisfies the constraint l. The set L contains all assignments on the line $\alpha + \bar{d} \cdot \theta$ that satisfy l. Hence, every variable $x_{r,i}$ can be *substituted* for $x_{r,i} = \alpha_{r,i} + d_{r,i} \cdot \theta$ creating the univariate polynomial $p_\theta(\theta)$. Next, the existence of satisfying sample points for θ under $p_\theta(\theta)$ is analogous to the corresponding part in Proof 3.3 for coordinate axis cell jumps. Out of the satisfying sample points, the one closest to 0 is chosen as sp_θ. Lastly, re-substituting for the original variables $x_{r,i}$ yields the new assignment α' that must also satisfy l. Hence, $\alpha' = (a_{r,1} + d_{r,1} \cdot sp_\theta, ..., a_{r,m_{\text{REAL}}} + d_{r,m_{\text{REAL}}} \cdot sp_\theta, a_{b,1}, ..., a_{b,m_{\text{BOOL}}})$ is a solution that will be found by CJUMP$\big(\bar{d}, l\big)$.

□

It is noteworthy that the given operations cannot handle equality constraints. Hence, a *fourth* operation is introduced that handles the cases of equality in the input formula. This operation, however, is not be able to solve equality constraints arbitrarily, but only for linear variables in the constraint. For equality, the operation requires a root to satisfy the constraint, but irrational roots are harder to compute with than rational ones. Hence, we want the assignment α to be rational such that isolating a linear variable will yield a rational root. By keeping α rational, this will speed up the local search by minimizing the computational overhead caused by irrational roots. While an assignment α by definition can include real values for their variables, by

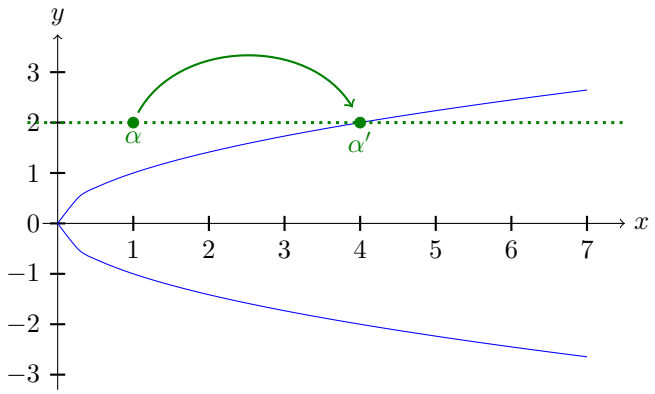

Figure 3.7: Completed CJUMP(x, l) for $l = \left(x - y^2 = 0\right)$ under assignment $\alpha = (1{,}2)$.

starting at a rational assignment and only altering this assignment using the described operations, by Proposition 3.3.3 the assignment will stay rational.

Proposition 3.3.3 (**Rational Assignment**). *Let F be a formula with real variables $\bar{x}_{REAL}$ and $\alpha = (a_{r,1}, ..., a_{r,m_{REAL}}, a_{b,1}, ..., a_{b,m_{BOOL}})$ an assignment such that $\forall x_{r,i} \in \bar{x}_{REAL} : a_{r,i} \in \mathbb{Q}$. Finally, let op be any operation described in Definition 3.3.1, 3.3.2 or 3.3.3. Then, for the new assignment α' the property $\forall x_{r,i} \in \bar{x}_{REAL} : a'_{r,i} \in \mathbb{Q}$ holds.*

Using Proposition 3.3.3, we can now define the equality cell jump operation to solve an equality constraint using a linearly occurring variable.

Definition 3.3.4 (**Operation: Equality Cell Jump** [LXZ23]). *Let F be a formula with real variable vector $\bar{x}_{REAL}$, current assignment α and $l = (p(\bar{x}_{REAL}) = 0)$ a polynomial constraint in F so that $\alpha \not\models l$. Let $x_{r,i} \in VAR(p)$ be an only linearly occurring variable in p such that $x_{r,i} = p_{x_{r,i}}$ is the constraint l solved for $x_{r,i}$. Let q be the value obtained by substituting $a_j \in \alpha$ for $x_{r,j} \; \forall \; x_{r,j} \in VAR(p)$ in $p_{x_{r,i}}$. Then, CJUMP$(x_{r,i}, l)$ is the equality cell jump operation which assigns q to $x_{r,i}$.*

The operation works by first deciding whether the given variable $x_{r,i}$ is linear in the constraint l. If it is, then l will be solved for the variable $x_{r,i}$. By now substituting every other assignment value a_j for $j \neq i$, a rational value q for $x_{r,i}$ is found that provides the new solution for l as $\alpha' = (a_{r,1}, ..., a_{r,i-1}, q, a_{r,i+1}, ..., a_{r,m_{\text{REAL}}}, a_{b,1}, ..., a_{b,m_{\text{BOOL}}})$. Intuitively, the operation CJUMP$(x_{r,i}, l)$ searches along the linear variable $x_{r,i}$'s coordinate axis to fix a value such that the constraint is satisfied. Such a value must always exist. It is very similar to the normal coordinate axis cell jump, the important difference being the linearity of $x_{r,i}$ and the type of constraint being an equality.

As the equality cell jump operation CJUMP$(x_{r,i}, l)$ only changes a single variable, it is also classified as a single cell jump operation.

Example 3.3.3. *Consider the two dimensional polynomial constraint $l = \left(x - y^2 = 0\right)$ falsified under assignment $\alpha = (x \mapsto 1, y \mapsto 2)$ as shown in Figure 3.7. The only linear variable in l is x, thus only CJUMP(x, l) exists. Solving l for x yields $x = y^2$ for which substituting $y = 2$ results in $x = 4$. Hence, the new assignment is $\alpha' = (x \mapsto 4, y \mapsto 2)$.*

It is now left to show that the equality cell jump operation works as intended, i.e. if and only if the assignment value for a linear variable $x_{r,i}$ can be adjusted to satisfy a given constraint l, the equality cell jump operation exists and will be found.

Theorem 3.3.4 (**Existence of Equality Cell Jump**). *Given a false equality constraint* $l = (p(\bar{x}_{REAL}))$ *under assignment* $\alpha = (a_{r,1}, ..., a_{r,m_{REAL}}, a_{b,1}, ..., a_{b,m_{BOOL}})$ *and an only linearly occurring real variables* $x_{r,i} \in VAR(p)$*, there exists a* CJUMP$(x_{r,i}, l)$ *operation if and only if the set*
$S = \{\alpha' = (a_{r,1}, ..., a_{r,i-1}, \theta, a_{r,i+1}, ..., a_{r,m_{REAL}}, a_{b,1}, ..., a_{b,m_{BOOL}}) \mid \theta \in \mathbb{R} \wedge \alpha' \vDash l\}$ *is non-empty.*

Proof. $\Rightarrow$ Assume that CJUMP$(x_{r,i}, l)$ exists. Per definition, this operation only changes the assignment value for $a_{r,i}$. Because the operation exists, there exists a value r_i that $x_{r,i}$ can be changed to such that l is satisfied. Thus, the new assignment $\alpha' = (a_{r,1}, ..., a_{r,i-1}, r_i, a_{r,i+1}, ..., a_{r,m_{\text{REAL}}}, a_{b,1}, ..., a_{b,m_{\text{BOOL}}})$ is contained in S.
$\Leftarrow$ Assume that S is non empty. This means that there exists a value θ such that the new assignment satisfies l. Because $x_{r,i}$ is only contained linearly in l, solving for $x_{r,i}$ can only yield a single solution. Because all the other assignment values stay the same, this value θ is found by substituting the assignment values, except for $x_{r,i}$, thus creating the CJUMP$(x_{r,i}, l)$ operation. □

It is noteworthy that the introduced equality cell jump operation can only solve a certain class of equality constraints, namely constraints in which a variable occurs linearly. But there might exist equality constraints in which no variable only occurs linearly while still having rational equality solutions. Consider the simple literal $l = \left(x^2 - y^2 = 0\right)$. This is trivially true for any x and y such that $|x| = |y|$, however starting from a falsifying assignment, the equality cell jump operation could not find a solution. Hence, this operation is not suitable for general equality constraints, yet it is an easy way to cover a range of equality constraints.

Furthermore, it is important to note that if neither a single nor a multi cell-jump operation is found for a constraint l and a set of direction vectors, it does *not* mean that the constraint cannot be satisfied - only that no single variable can be adjusted and under the considered directions no multiple variables can be changed to satisfy l. Recall that Figure 3.5 shows two multi cell-jumps with direction vectors that do not intersect the solution space. If only the two red direction vectors were considered, then neither a single, nor a multi operation would have found a solution for l.

3.4 Heuristic

Given all types of operations, it is now left to rank the individual existing operations created from the variables and literals from the input formula F, as well as the direction vectors, to choose the *best* operation that brings F the closest to satisfaction. This ranking has to be done heuristically as in the LS setting, there is no way to determine which operation is the definitive best one.

Definition 3.4.1 (**Heuristic Function**). *Let* F *be a formula,* α *an assignment and* OP *the set of all existing operations for* F *under* α*. Then, a heuristic function is a mapping* $h : OP \to \mathbb{Q}$ *such that*

$$\begin{aligned} h(op) > 0 &\Leftrightarrow op \text{ is suitable to be used} \\ h(op) \leq 0 &\Leftrightarrow op \text{ is unsuitable to be used} \end{aligned} \tag{3.7}$$

rates each operation.

In the following, the deployed heuristic will be described. Note that the heuristic itself is easily exchangeable as it must only obey the requirements set by Definition 3.4.1. The heuristic used in local search is based on a literal level that is then being projected upwards through the formula. In the original approach by Li et al., the literals were not weighted. However, we can use the formula itself to extract information about the importance of certain literals. Hence, we use Jeroslow-Wang literal weights to introduce a *static* component to the heuristic that yields an exponentially higher weight to literals in shorter clauses, as well as a liner weight to the number of occurrences of the literals.

Definition 3.4.2 (**Jeroslow-Wang Literal Weights** [JW90] [HV95])**.** *Let F be a formula, l a literal in F and C_l the set of clauses $c \in F$ containing l. Then,*

$$J(l) = \sum_{c \in C_l} 2^{-|c|}$$

is the Jeroslow-Wang literal weight for l.

Using the literal weights from Definition 3.4.2, a heuristic score for each literal l under the assignment α can be defined as the *distance to truth*.

Definition 3.4.3 (**Distance to Truth** [LXZ23])**.** *Let l be a literal with literal weight $J(l)$, α an assignment and $pp \in \mathbb{Q}_{>0}$ and $b_{offset} \in \mathbb{Q}_{>0}$ two positive parameters. Then,*

$$\textit{DTT}(l, \alpha) = J(l) \cdot \begin{cases} 0 & \textit{if } \alpha \textit{ is a solution to } l \\ (|p(a_1, ..., a_n)| + pp) & \textit{else if } l \textit{ is polynomial} \\ (b_{offset} + pp) & \textit{otherwise/Boolean} \end{cases}$$

is the distance to truth $\textit{DDT}(l, \alpha)$*.*

The aforementioned *distance to truth* from Definition 3.4.3 is thus an altered version of the polynomial-only idea introduced in [LXZ23] that also handles the case of Boolean literals and utilizes literal weights. Intuitively, the distance to truth is 0 if the current assignment satisfies the literal, otherwise it is a positive non-zero value that depicts how far away the current assignment is from a satisfying assignment, as well as how important the literal is based on the literal weights. The smaller a value is, the heuristically closer this assignment is to satisfying the literal. The parameter $pp \in \mathbb{Q}_{>0}$ assures that $\textit{DDT}(l, \alpha)$ can only be 0 if and only if $\alpha \vDash l$ while b_{offset} is used as an adjustment for the Boolean case for which an absolute polynomial value cannot exist.

The literal heuristic scores from Definition 3.4.3 can now be lifted to the clause level such that a clause is satisfied under an assignment if and only if the heuristic score for this clause is 0. Again, the smaller a heuristic score for a clause is, the heuristically closer the current assignment is to satisfying the clause.

Definition 3.4.4 (**Distance to Satisfaction** [LXZ23])**.** *Let $c \in F$ be a clause in a formula F and α an assignment. Then,*

$$\textit{DTS}(c, \alpha) = \min_{l \in c} \{\textit{DTT}(l, \alpha)\}$$

is the distance to satisfaction $\textit{DTS}(c, \alpha)$*.*

Similarly to the literals, we want to put emphasis on certain clauses. Recall that local search works in iterations that always change the assignment. Hence, we want to capture which clauses are harder to satisfy, i.e. which clauses stay unsatisfied for multiple iterations. For this, clause weights following the PAWS scheme [TPBFJ04] are used in which clause weights are changed in *update iterations*.

Definition 3.4.5 (**Clause Weights**)**.** *Let c be a clause in formula F. Then, $w(c) \in \mathbb{Q}$ is the clause weight for c.*

The clause weights are initialized with the value 1 for each clause. Then, they can be updated as described in Definition 3.4.6.

Definition 3.4.6 (**Clause Weight Updates PAWS** [TPBFJ04], [CS13], [LXZ23])**.** *Let $sp \in (0,1)$ be a smoothing probability, α an assignment, $c \in F$ a clause and $w_i(c)$ the clause weight in update iteration i. Then,*

$$w_{i+1}(c) = \begin{cases} w_i(c) - 1 & \text{with probability } sp \text{ if } \alpha \vDash c \wedge w_i(c) > 1 \\ w_i(c) + 1 & \text{with probability } 1 - sp \text{ if } \alpha \nvDash c \\ w_i(c) & \text{otherwise} \end{cases} \tag{3.8}$$

is the new clause weight for c.

The clause weights according to the PAWS scheme pose a *dynamic* setting for the decision heuristic used in local search. Due to the probability sp, local search might behave differently for the same formula instance F in multiple runs. Finally, the combination of clause weights and distance to satisfaction yield the final heuristic described in Definition 3.4.7.

Definition 3.4.7 (**Local Search Heuristic** [LXZ23])**.** *Let op be an operation, α the current assignment and α' the assignment after performing op. Then,*

$$h(op) = \sum_{c \in F} (\mathit{DTS}(c, \alpha) - \mathit{DTS}(c, \alpha')) \cdot w(c)$$

is the heuristic score an operation.

It is easy to see that the introduced heuristic in Definition 3.4.7 obeys the requirements set in the Definition 3.4.1 for a heuristic function. In conclusion, the heuristic tries to incorporate solving the formula by using the distances to truth and satisfaction, the importance of selected clauses by using dynamic clause weights and the importance of clause lengths and number of literal occurrences by employing literal weights.

Chapter 4

Algorithm

This chapter covers the algorithm briefly introduced in Section 3.1 in more detail, in particular Chapter 4 shows what improvements were made to the initial idea, which internal parameters exist for local search and which sets of internal settings were used for them. Finally, this chapter also covers how to implement local search in pseudo-code.

4.1 Improvements

Starting from the initial approach as presented in Section 3.1, the following improvements are made to the procedure which is then presented in Algorithm 1. In addition, the improvements can be disabled based on internal settings such that the local search can be used for various applications such as a stand-alone solver, in an incremental DPLL(T) architecture or as a combined solving strategy.

Restarts By the definition of the operations, the assignment α is only changed the minimal amount in order to satisfy a given false literal. Intuitively, the local search stays in fact local. Hence, returning unknown in step (4) from Definition 3.1.1 after neither a single, nor a multi cell jump operation was found, is a drawback, as there can still be solutions as previously stated. Thus, an improvement over the initial design is that, instead of returning unknown, the search will be restarted at a different assignment, where the new assignments follow the pattern: In the first restart, each real variable is assigned the closest integer value to either its upper or lower bound that satisfies the bound, if such a polynomial constraint exists in the input formula, i.e. if a constraint $a \cdot x < b_{\text{upper}}$ or $a \cdot x > b_{\text{lower}}$ $(a \in \mathbb{Q} \setminus \{0\}, b \in \mathbb{Q})$ exists. We do not perform any processing to determine bounds that are not explicitly stated. In the second to sixth restart, each real variable is assigned a random value in $\{-1, 1\}$, and in all subsequent restarts i, each real variable is assigned a random value in $[-50 \cdot (i-6),\ 50 \cdot (i-6)]$. In particular, in the first *subsequent* restart $(i = 7)$, we obtain values in $[-50, 50]$. In the second *subsequent* restart $(i = 8)$, we obtain values in $[-100, 100]$. Intuitively, we widen the range in which assignment values are chosen for each *subsequent* restart. In every restart, all Boolean values are initialized with the default Boolean value which is set to true.

Using restarts, this ensures that the whole allotted time is actually being used for searching instead of prematurely returning. It also allows for finding solutions that

would be impossible to find in the current state the solver is in. Intuitively, this means that the current assignment corresponds to a point in the search space that is too far away from a solution space for the formula such that a local operation will never reach the solution space. This improvement has already been made by the initial authors of the procedure [LXZ23].

Cycling Assignments Due to the nature of searching algorithms in combination with changing weights, and thus a changing heuristic, it might happen that the solver enters a loop or a long stretch alternating between cells and different points in their respective cells. In order to prevent this, a *forbidding* strategy is employed. This means that for any variable x_i that has been changed, k following iterations of operations cannot change the same variable x_i in the opposite direction. Using a forbidding map has also been introduced by the authors of the initial approach. In addition to this, however, is the new functionality of *disabling* the forbidding strategy after a set amount of restarts FORBIDDING_K. The reasoning behind this is that after FORBIDDING_K restarts, there have been a number of iterations in which no suitable operation was found that is obeying the forbidding strategy. Restarting FORBIDDING_K times has not yielded any satisfactory results either, so maximizing the amount of available operations has priority as the restarts have shown not to benefit the search. Furthermore, it seems useful to distinguish between *Boolean* and *real* cell jump operations such that they can be forbidden for a different amount of iterations.

Literal Weights The initial approach of the heuristic only contains clause weights. It is, however, also important to factor in clause *length*. Due to the nature of CNF, every clause needs to be satisfied. However, in shorter clauses, fewer literals are available to be satisfied. Hence, instead of relying on the heuristic in general to satisfy short clauses, Jeroslow-Wang literal weights [JW90] are used to give exponentially higher weight to literals in shorter clauses which makes them more desirable to solve using a dedicated operation.

Selection of Literals As an improvement proposed in the article [LXZ23], the process of selecting literals to satisfy is also of importance to a fast algorithm. While generating *all* cell jump operations for every false literal as shown in the initial approach yields the best operation from a given assignment, it might induce a lot of computational overhead. Namely, solving a false literal in an already satisfied clause is a valid operation, but it is very unlikely that the heuristic will score this operation higher than an operation that satisfies a falsified literal in a currently falsified clause, thus satisfying the clause. As an improvement, only falsified literals which are currently contained in falsified clauses are considered when generating the cell jump operations. Only if there exists no operation that the heuristic deems suitable, the falsified literals in currently satisfied clauses are also considered.

Update of Clause Weights Due to the nature of the introduced local search procedure, the clause weights will only be updated if there exists no single cell jump operation. This, however, can lead to example runs in which the clause weights are almost never updated, while certain clauses remain falsified throughout the search as there is no added incentive to solve those certain clauses directly. Hence, as an

addition to the proposed initial approach, the clause weights will be updated every CLAUSE_K $\in \mathbb{N}$ iterations regardless of how many single cell jump operations exist.

Saving Clause Weights With more importance set to clause weights as described in the previous paragraph, after a restart it might seem useful to *not* reset the clause weights as well. This is to keep information about clauses that are hard to satisfy or to keep satisfied such that the search from a new assignment does not need to re-learn the importance of selected clauses.

Rational Roots Based on the introduced operations, if a polynomial constraint is an equality constraint without any linear variable, local search in its pure form would not be able to satisfy the constraint, even if a rational root were to exist for a variable contained in the constraint. Hence, if the tool used for root isolation yields a rational root r for an equality constraint, r is then also used as a normal sample point to satisfy the constraint.

Incrementality To extend local search past a stand-alone solver, it can be updated to be used in a DPLL(T) environment. In short, the original formula is transformed into a Boolean skeleton for which a satisfying Boolean assignment is determined using a SAT solver. Then, this Boolean assignment is lifted to the theory domain to check if the corresponding literals in the original formula are consistent, i.e. if the conjunction of literals expected to be satisfied as denoted by the Boolean assignment can be satisfied in the theory. If it cannot be satisfied, the Boolean assignment is altered slightly to a different satisfying Boolean assignment which then results in a slightly different conjunction of literals to satisfy in the theory. For local search to be used this way, as local search is run with slightly changing conjunctions of literals (i.e. clauses), the assignment, as well as the current clause weights are kept so that they only need to be updated for new clauses, literals and variables such that the search only needs to incrementally satisfy a new constraint starting at the current assignment.

4.2 Settings

With local search and its improvements being dependent on multiple parameters, the used *settings* for local search in general should be discussed. These settings are a result of an initial guess which is then refined through multiple benchmarks as described in Section 5.5. Multiple sets of settings are shown in Figure 4.1 for different use cases. Settings_1 is the initial set of values, Settings_2 are the refined settings for local search as a stand-alone solver. $\text{Settings}_{\text{Incr.}}$ are the incremental settings derived from Settings_2 for local search to be used in an incremental solving strategy. These are also tested in the benchmarking Chapter 5.

In the settings, SP denotes the smoothing probability used for updating clause weights as shown in Definition 3.4.6, CLAUSE_RESET is a Boolean deciding whether clause weights should be reset if a restart occurred, PP and B_OFFSET are integer values used for the heuristic in Definition 3.4.3, DIR_NUM denotes the number of direction vectors generated to check for multi cell jump operations, FORBIDDING_DISABLE is a Boolean deciding if the forbidding strategy should be disabled after FORBIDDING_K restarts, FORBIDDING_REAL and FORBIDDING_BOOL are integer values indicating for how many iterations a real or Boolean valued variable is not to be updated according

Settings Member	Settings_1	Settings_2	$\text{Settings}_{\text{Incr.}}$
SP	0.003	0.05	0.05
CLAUSE_RESET	true	true	false
CLAUSE_UPDATES	true	true	true
CLAUSE_K	20	20	20
FORBIDDING_DISABLE	false	true	true
FORBIDDING_K	10	3	3
FORBIDDING_REAL	10	3	3
FORBIDDING_BOOL	10	3	3
PP	1	5	5
B_OFFSET	3	8	8
DIR_NUM	12	8	8
DURATION	30000	30000	500

Table 4.1: Local search settings used in benchmarking Chapter 5.

to the forbidding strategy, CLAUSE_UPDATES is a Boolean setting to use additional clause updates, CLAUSE_K is the interval in which additional clause updates are done and lastly DURATION is the allotted time for local search in milliseconds.

While not being a drawback nor a setting, it is also important how the direction vectors are generated. The direction vectors are calculated uniformly randomly such that for each entry d_i of a direction vector $\bar{d}$ the property $d_i \in [-100, 100]$ holds. These vectors are calculated once per iteration and are then used for all multi cell jump operations in this single iteration.

4.3 Pseudo-Code Implementation

Finally, utilizing the mentioned improvements and settings yields Algorithm 1. It shows the general pseudo-code of the main solving function, where f is the formula to solve, α is the assignment, cW and lW are the clause weights respectively literal weights used by the heuristic, and forbiddingMap is the map containing values forbidding certain changes based on the forbidding strategy. Due to simplicity and readability, the checking whether an improvement is disabled is portrayed by "*based on settings*". Furthermore, Procedure 1 and Procedure 2 are two auxiliary functions to generate cell jump operations respectively update the necessary variables.

It is easy to see how the procedure in Algorithm 1 follows the guidelines set by the initial approach, yet the only improvements that are easy visible are the *restart* after not finding any operations, as well as the *distinction* between generating cell jump operations using only literals in currently falsified clauses, or literals in only satisfied clauses and the added updates of clause weights. The *forbidding strategy* is only visible in the updates of the forbidding map, as well as an input parameter for the generation of operations, while the *literal weights* solely occur as a parameter.

Algorithm 1 Local Search Procedure in SMT-RAT

```
1: f, α, cW, lW, forbiddingMap, remaining_time ← initResources
2: while remaining_time > 0 do
3:     if α ⊨ f then
4:         return sat, α
5:     end if
6:     if iteration is multiple of clause_k based on settings then
7:         cW ← update clause weights using PAWS scheme
8:     end if
9:     dirs ← ∅
10:    for typeJump ∈ [single, multi] do
11:        for typeClause ∈ [falsified, satisfied] do
12:            if typeJump == multi ∧ typeClause == falsified then
13:                dirs ← generate direction vectors
14:                cW ← update clause weights using PAWS scheme
15:            end if
16:            ops, h_ops ← getOperations (typeJump, typeClause, dirs, cW, lW)
17:            if ops ≠ ∅ then
18:                α, forbiddingMap ← update (ops, h_ops, forbiddingMap)
19:                goto while loop start
20:            end if
21:        end for
22:    end for
23:    α ← restart search assignment
24:    forbiddingMap ← disable based on settings
25:    cW ← reset clause weights based on settings
26: end while
27: return unknown
```

Procedure 1 Construction of Operations

```
1: procedure getOperations(typeJump, typeClause, dirs, cW, lW)
2:     ops ← generate all typeJump cell jump operations
3:             for literals in typeClause clauses using dirs if necessary
4:     h_ops ← heuristically score each operation op ∈ ops using cW and lW
5:     ops ← delete op with h_ops[op] < 0
6:     return ops, h_ops
7: end procedure
```

Procedure 2 Update Assignment and Forbidding Map Based on Given Operations

```
1: procedure update(ops, h_ops, forbiddingMap)
2:     op ← operation op ∈ ops with highest heuristic score h_ops [op]
3:             not blocked by forbiddingMap
4:     α ← update model based on op
5:     forbiddingMap ← update forbiddingMap based on op
6:     return α, forbiddingMap
7: end procedure
```

Chapter 5

Benchmarks

The local search procedure introduced in Chapter 4 is implemented as a *module* in SMT-RAT, the SMT-Solver developed by the Theory of Hybrid Systems group at RWTH Aachen University. Simulations were performed with computing resources granted by RWTH Aachen University under project thes1685. For this, LuFG Theory of Hybrid System's own benchmarking tool BENCHMAX[1] is used.

The benchmark set is divided into two main subsets. The first subset consists of all QFNRA problem instances from SMT-LIB [BFT16] with status SAT, while the second subset only contains self-generated problem instances with unknown status. It is important to recall that local search can only prove satisfiability of formulas, but not unsatisfiability. Hence, only benchmarking the satisfiable instances is important in order not to waste computing time for instances known not to be solvable using local search. To achieve benchmark results, all input instances are subject to be solved with the implemented LS approach, as well as with SMT-RAT's default strategy - a strategy containing preprocessing, subtropical solving and MCSAT introduced by de Moura and Jovanovic [dMJ13] as shown in Figure 5.1, or with an incremental approach containing preprocessing, an incremental SAT solver, an optional local search module and covering and CAD modules as depicted in Figure 5.2. The covering module follows the approach presented by Ábrahám et al. [ADEK21]. Lastly, we will refer to SMT-RAT's default strategy as Default, and to the implemented local search approach as Local Search or Incremental Local Search.

In the following, the two benchmark sets will be analysed independently for Local Search. In addition, Local Search will be used as a solver in an incremental DPLL(T) strategy and as a combined solver with Default. Lastly, Local Search benchmarks with different settings will be shown indicating the effects of different parameter values. All benchmarks were run with a memory limit of 4GB at a cut-off time of 30s using $Settings_2$ for Local Search introduced in Table 4.1 or $Settings_{Incr.}$ for Incremental Local Search. As per settings $Settings_2$, Local Search is run for 30s, and any solving attempt slightly above this threshold is still deemed a valid answer. Only if the execution time of any solving strategy for any instance greatly exceeds 30s, it is seen as a timeout for which 45s are set as the runtime.

[1] `https://ths-rwth.github.io/smtrat/dd/d0f/benchmax.html`

Default-Strategy:

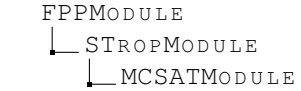

Figure 5.1: Default strategy in SMT-RAT.

Incremental-Strategy:

Figure 5.2: Incremental strategy in SMT-RAT with optional Local Search.

In addition to intuitive plots, *performance profiles* as introduced in the article [DM02] are also used to give a general overview of a solver's performance and their comparison to different solvers as the set of solvers S. In short, the ratio $r_{p,s}$ of a solver's s runtime $t_{p,s}$ to the best runtime for each instance $p \in \mathcal{P}$ for the problem set $\mathcal{P}$ is calculated as

$$r_{p,s} = \frac{t_{p,s}}{\min\{t_{p,s} \mid s \in S\}}.$$

Note that if a solver $s \in S$ cannot solve an instance $p \in \mathcal{P}$, the ratio $r_{p,s}$ is set to a parameter $r_M \in \mathbb{N}$ which is sufficiently large. Then, the performance profile $\rho_s(\tau)$ depicted in Figures 5.5, 5.9, 5.16 and 5.21 for a solver $s \in S$ dependent on $\tau \in \mathbb{N}$ is calculated as

$$\rho_p(\tau) = \frac{1}{|\mathcal{P}|} \cdot |\{p \in \mathcal{P} \mid r_{p,s} \leq \tau\}|,$$

such that $\rho_s(\tau)$ indicates the percentage of instances with a ratio within a factor τ of the best ratio. For example, the value at $\tau = 1$ shows which solver performs the best on all instances as the solver with the highest performance profile $\rho_s(1)$ has the highest percentage of instances solved within a ratio of 1, i.e. has the highest percentage of instances with the best ratio. All other values for τ indicate how close a solver is at solving instances within a given factor. Note that for $\tau \to \infty$, all solvers will reach the same performance profile value, i.e. only a small range for τ is relevant dependent on what is deemed *competitive*.

5.1 SMT-LIB-Benchmarks

The SMT-LIB benchmark set contains a multitude of different problem instances varying in complexity and ease of solvability. In this section, the SMT-LIB instances are solved by the default strategy as shown in Figure 5.1 and Local Search to determine the effectiveness of Local Search. Recall that Local Search can only show satisfiability of polynomial formulas, hence all shown benchmark results only contain the known satisfiable instances from SMT-LIB.

Total = 5248	LS_{30s}	$Default_{30s}$
SAT	1798	**4883**
UNKNOWN	**3403**	0
Timeout	47	**273**
Memout	0	**92**

Table 5.1: SMT-LIB benchmark results. Larger numbers marked.

Average [s] / [MB]	LS_{30s}	$Default_{30s}$
Time: Overall	20.01	**3.30**
-Solved	**0.20**	0.27
-Solved by both	0.17	**0.08**
Memory Overall	**13.9**	94.3
-Solved	**6.0**	15.7
-Solved by both	**6.0**	8.0

Table 5.2: SMT-LIB runtime and memory averages. Smaller numbers marked.

Table 5.1 shows a general overview of the results from Local Search and Default on the SMT-LIB instances. It is easy to see that Local Search performs underwhelmingly with only 1798 solved instances compared to Default's 4883, which corresponds to 36.8%. Noteworthy are also the timeout cases. These happen for instances in which calculating the heuristic is especially slow, thus creating timeouts while testing a potential operation near the time limit. Considering the average runtimes depicted in Table 5.2, Local Search does not offer any improvement over Default. While the average for solved instances is smaller for Local Search, as Local Search solves significantly less instances, this cannot be seen as an improvement. Furthermore, on all instances and instances solved by both strategies, Local Search takes considerably more time.

Figure 5.3 depicts all runtimes of Local Search and Default graphically showing the runtime for each instance by both solvers. It is easy to see that the majority of problem instances are solved within a very short amount of time by Local Search and Default respectively, yet a trend in favour of the default strategy is also shown indicating that Default performs much better on multiple instances. On the right, there exists a large column showing the instances Default can solve, but not Local Search. Furthermore, the column also shows that for these instances, Default has various different solving times. Overall, the general performance measured on runtime is shown in the performance profile depicted in Figure 5.5 with Local Search barely reaching the 30% mark with more than a factor of 10 within the best solver's times. It also shows that Local Search is the *better* solving strategy in only approximately 15% of instances.

Only in the peak memory comparison shown in Figure 5.4 and 5.2 Local Search performs consistently better than the default strategy. Local Search has a smaller peak memory usage in 99.9% of instances and utilizes on average only 14.7% of Default's average. In fact, for some instances, the default strategy uses more than two orders of magnitude more memory. It is important to note, however, that for instances that are solved by both solving strategies, the difference between peak memory consumption is a lot closer. In contrast to Default using a lot more memory than Local Search, there are also a few instances in which Local Search and Default have approximately the same peak memory usage, yet Default can solve them while Local Search cannot.

Lastly, not comparing Local Search to Default but for the interest of the reader, Figure 5.6 shows the relation of used single cell jumps and multi cell jumps in the Local Search for solved and unsolved instances. It is easy to see that the Local Search prefers single cell jumps in almost all instances and thus, the focus for further improvements should be set to single cell jumps. It is also shown that Local Search can only solve problems if it uses relatively few cell jump operations. This indicates that the selection heuristic might select cell jumps that are not too beneficial for the whole search or

that a satisfying assignment needs a larger number of changes made to the initial assignment.

Despite Local Search's lack of solved instances, there are 17 instances that Local Search can solve, but not Default. These instances are particularly interesting as they offer an actual improvement over Default. For these, Local Search has an average runtime of 3.4s and utilizes 6.2MB of memory. A closer look at those instances yields that they are mostly *meti-tarski atan* and *sin* problems for which distinguishing characteristics could not be found. However, the other instances have a relatively high average variable degree. Logically, Local Search only has to isolate roots of polynomials, but not perform extensive work with them. Thus, in the following Section 5.2, instances with high polynomial degree will be analysed. Still, Local Search does perform very poorly on SMT-LIB instances and is not a viable stand-alone solver for instances similar to those covered by SMT-LIB.

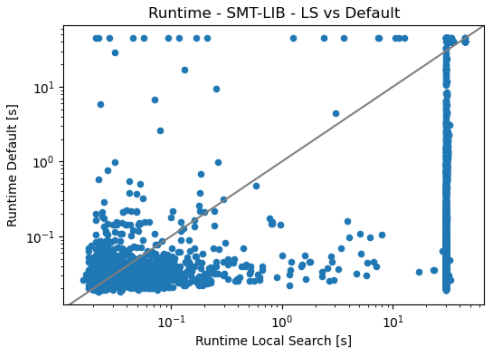

Figure 5.3: Scatter plot comparing run times between Local Search and Default for SMT-LIB instances.

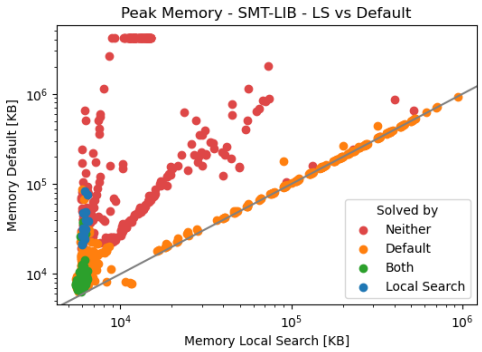

Figure 5.4: Scatter plot comparing peak memory consumption between Local Search and Default for SMT-LIB instances distinguished by the solvers' answers.

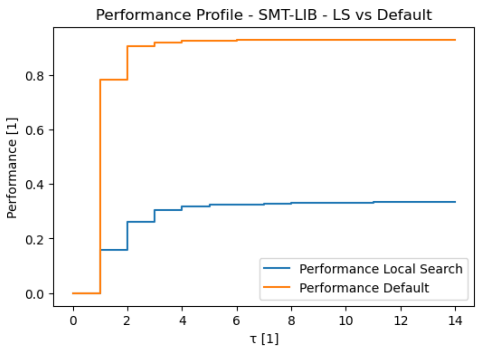

Figure 5.5: Performance profile for SMT-LIB instances regarding runtime comparing Local Search and Default.

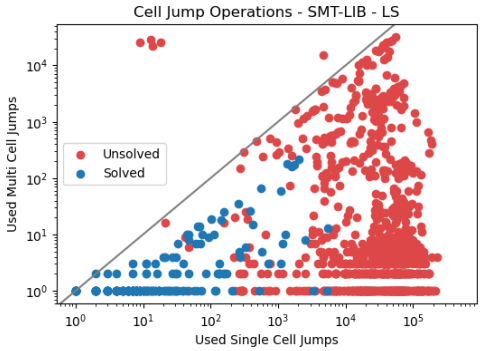

Figure 5.6: Scatter plot showing the number of cell jumps used by Local Search for solved (blue) and unsolved (red) instances.

5.2 Generated Benchmarks

The generation of problem instances with high polynomial degree follows the generation described in the article [LXZ23]. In short, a random instance is generated obeying the following rules in which $\text{RAND}(a, b)$ denotes a uniformly distributed random integer in $[a, b]$. Firstly, base conditions for the construction of a single instance are defined.

1. Generate $v_n = \text{RAND}(30, 40)$ variables
2. Define number of polynomials $pol_{num} = \text{RAND}(60, 80)$
3. Define number of clauses $cl_{num} = \text{RAND}(40, 60)$

Then, the instance is constructed as follows.

1. Generate pol_{num} polynomials such that each polynomial p_i ($i \in \{1, ..., pol_{num}\}$) is independently constructed as follows
   (a) Define number of variables in polynomial $n_i = \text{RAND}(10, 20)$ to be taken randomly from v_n
   (b) Define degree of the polynomial $d_i = \text{RAND}(20, 30)$
   (c) Define number of monomials in the polynomial $m_i = \text{RAND}(20, 30)$
   (d) Generate m_i monomials M_j such that the first monomial M_1 has degree d_i and all other monomials have a degree less or equal to d_i.
   (e) Construct polynomial p_i as $p_i = \sum_{j=1}^{m_i} c_j \cdot M_j + c_0$ for coefficients $c_j = \text{RAND}(-1000, 1000)\ \forall\ j \in \{1,...,, m_i\}$
2. Generate cl_{num} clauses independently as follows:
   (a) Define number of literals in clause $lit_{num} = \text{RAND}(3, 5)$
   (b) Take lit_{num} polynomials p_i with a randomly added relation $<$, $>$ or $=$ each. If equality is chosen, then check if at least one variable is contained solely linearly in p_i. If this is not the case, substitute the relation with $<$ or $>$.

This procedure generates instances with high polynomial degree, yet also an easy Boolean abstraction. Intuitively, the hard part about solving those instances is *finding* points that make selected literals satisfied, not the *selection* of literals to satisfy. It is important to note that the degrees of the variables in the polynomials are relatively evenly distributed and are not biased towards certain variables, i.e. no variable in any monomial should have a significantly higher degree than the other variables. Furthermore, these instances are not checked to be satisfiable beforehand, hence there is no selection process of satisfiable instances in contrast to the SMT-LIB benchmark Section 5.1. This is because other solvers struggle with these instances, but using Local Search to pre-select satisfiable instances would skew the benchmark results in favour of Local Search.

A general overview of the benchmark results can be found in Figure 5.3 showing the behaviour of Local Search in 30 seconds, as well as Default in 120 seconds for generated instances. Figure 5.7 shows the runtime of Local Search for the generated instances. It is easy to see that the Local Search outperforms Default massively. In fact, Default cannot solve a single instance with four times the allotted time because it exceeds the memory limit of 4GB for every instance. Even without a memory limit,

Total = 500	LS_{30s}	$Default_{120s}$
SAT	**393**	0
UNKNOWN	**107**	0
Timeout	0	0
Memout	0	**500**

Table 5.3: Generated instances benchmark results. Larger numbers marked.

Average [s] / [MB]	LS_{30s}	$Default_{120s}$
Time: Overall	**16.50**	120.00
-Solved	**12.90**	-
-Solved by both	-	-
Memory Overall	**7.6**	4183.8
-Solved	**7.6**	-
-Solved by both	-	-

Table 5.4: Runtime and memory averages for generated instances. Smaller numbers marked.

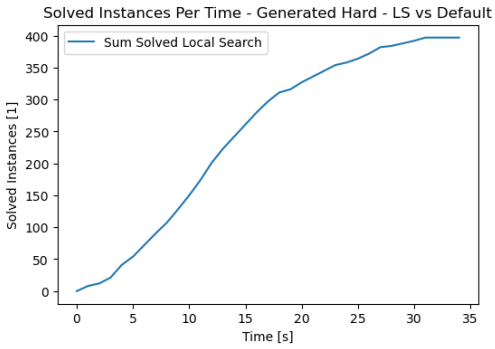

Figure 5.7: Cumulative plot for generated instances depicting the number of instances solved after a certain time.

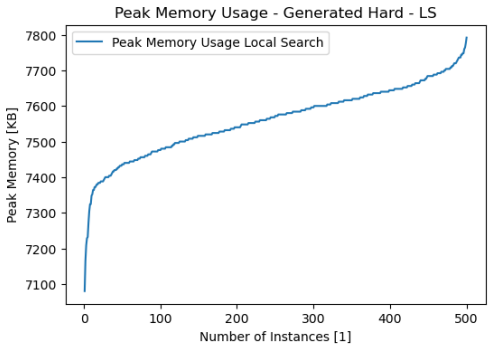

Figure 5.8: Local search peak memory usage for generated instances sorted by memory consumption.

every solving attempt by Default times out. Hence, Default is in no way competitive to Local Search in terms of these generated instances and is easily beaten by the Local Search. The runtimes depicted in Figure 5.7 show that even Local Search can run into timing problems when evaluating those generated instances. Even while Local Search was able to solve a great amount of instances, these instances are still hard to solve for it and can require almost all of the allotted time or even more resulting in a timeout. Local Search is, however, not limited by its memory consumption in contrast to Default, as shown in Figure 5.8.

Mainly because of the results shown in Figure 5.3, a fine comparison between the Local Search and the currently employed default strategy is not possible. Hence, a new set of generated instances is set up such that the instances are *easier* to solve. The changes that were made include limiting the number of clauses from $[40, 60]$ to $[20, 40]$, decreasing the polynomial degree from $[20, 30]$ to $[2, 6]$ and reducing the number of monomials from $[20, 30]$ to $[3, 5]$. Because these instances are easier to solve, the benchmarks were now run with a time limit of 30s for both Local Search, as well as Default.

A general overview of the results for the easier generated instances can be found in Table 5.5. It is easily noticeable that Default now solves a lot more instances, namely 290, and does not reach its memory limit. Local Search, however, also extends its effectiveness to solve these instances to the full benchmark set. Figure 5.10 depicts

the absolute runtime comparison between Local Search and Default for all instances, Figure 5.6 shows average runtimes and peak memory usage for Local Search and Default. Excluding timeouts for Default, most of the runtime values are *slightly* skewed in favour of Default. In fact, Default was faster in about 71% of instances solved by both solvers. However, due to large runtime differences for same instances in favour of Local Search, for instances that Local Search and Default could both solve, the average time is smaller for Local Search. Still, the majority of solved instances are solved within 5s by both Local Search and Default.

Total = 500	LS_{30s}	$Default_{30s}$
SAT	**500**	290
UNKNOWN	0	0
Timeout	0	**210**
Memout	0	0

Table 5.5: Benchmark results for generated easier instances. Larger numbers marked.

Average [s] / [MB]	LS_{30s}	$Default_{30s}$
Time: Overall	**1.42**	20.04
-Solved	**1.42**	1.96
-Solved by both	**1.54**	1.96
Memory Overall	**6.4**	18.6
-Solved	**6.4**	12.6
-Solved by both	**6.4**	12.6

Table 5.6: Runtime and memory averages for generated easier instances. Smaller numbers marked.

The mentioned Figures 5.10 and 5.12 also show the dependency on the sum of polynomial degrees. It is easy to see that this factor is important to both Default *and* Local Search. The higher degree an instance contains, the larger the peak memory consumption. The same trend is also shown for the run times. The higher the sum of polynomial degrees, the longer the runtimes.

The performance profile depicted in Figure 5.9 in regards to runtime show that Local Search quickly dominates Default and that around 75% of instances can be solved by Local Search in at most twice the time of the best time for each instance. It also shows that Default is not able to get close to the Local Search as only approximately 50% of instances are solved within 14 times the best solver's time and that Local Search is the better solver in approximately 60% of instances.

Furthermore, Local Search uses considerably less memory than Default as shown in Table 5.6, Figure 5.11 and Figure 5.12. On average, Local Search uses approximately 52.6% of Default's memory for the same instance. However, the smallest memory usage percentage is 22.9% while the highest is 71.0%. It is also noteworthy that the Local Search scales easily with harder generated instances. Recall that Figure 5.8 shows the memory usage for the harder generated instances which averages to approximately 7.6 MB per instance. The easier generated instances average to 6.4 MB for Local Search. This is only an increase of nearly 18%, while Default spikes from an average of 18.6 MB to more than 4 GB resulting in a memory abort.
It is also noteworthy that in no instance, Local Search had a higher peak memory usage than Default. Furthermore, the memory consumption of Local Search stays relatively consistent, while the memory consumption of Default spikes more than an order of magnitude making it less predictable. In addition to this, as shown in Figure 5.11, there exists a split between instances only Local Search can solve and instances both solvers are able to solve. The instances only Local Search can solve yield a higher peak memory usage for Default in comparison to instances both solver can solve.
In the end, it is concluded that Local Search still outperforms Default on easier

generated instances, however not as drastically as for the harder generated instances. It is easy to see that a main advantage of Local Search is the handling of high degree polynomials with a great amount of monomials where the Boolean abstraction of formulas is not that complex. Furthermore, the scalability of Local Search is shown indicating that it might also solve problem instances with even higher degree than the introduced hard instances.

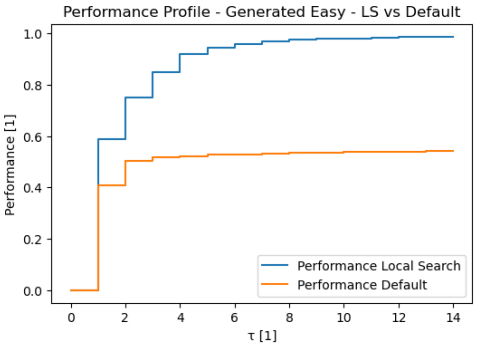

Figure 5.9: Performance profile for generated easier instances regarding runtime comparing Local Search and Default.

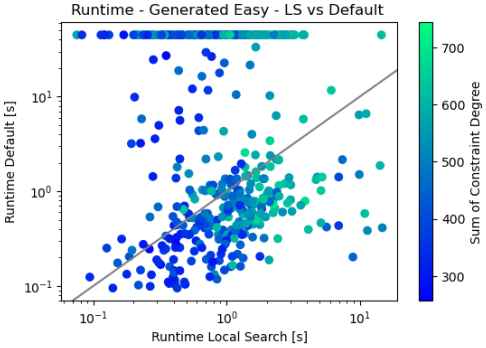

Figure 5.10: Scatter plot runtime comparison for generated easier instances between Local Search and Default for all instances dependent on sum of constraint degree.

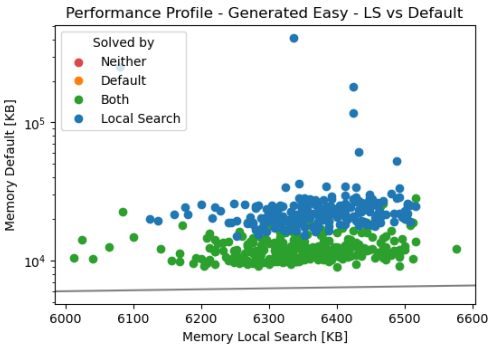

Figure 5.11: Scatter plot peak memory usage comparison for generated easier instances between Local Search and Default for all instances.

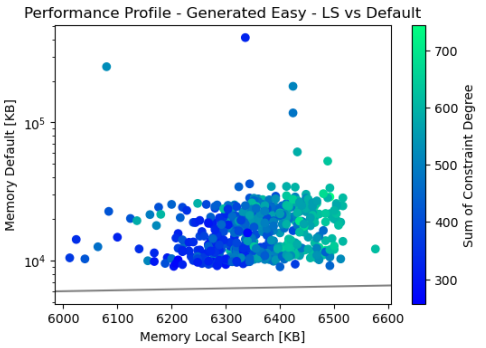

Figure 5.12: Scatter plot peak memory usage comparison for generated easier instances between Local Search and Default for all instances dependent on sum of constraint degree.

5.3 Incremental Local Search

A theoretical benefit of the local search approach is that it can find solutions to hard problems relatively fast and easily *by chance* instead of requiring extensive work before useful information is found. Furthermore, Local Search is easily adaptable to be used incrementally as shown in Section 4.1. Hence, Local Search's effectiveness used in an incremental approach is to be tested on the SMT-LIB instances. For this, we compare Local Search in a DPLL(T) approach with a pure DPLL(T) approach not containing Local Search. The used strategies for this are shown in Figure 5.2 and will be referred to as Incremental Local Search (Incr. LS) and Incremental (Incr.) respectively.

Table 5.7 shows general information about Local Search used in a DPLL(T) strategy for SMT-LIB instances. Note that Incremental Local Search performs slightly worse with 110 solved instances less and more timeouts. In addition to this, Incremental Local Search also takes more time to solve an instance in general. As shown in Table 5.8, the average time to solve an instance for Incremental Local Search is more than three times greater than the average time to solve an instance for Incremental. Considering instances solved by both strategies even yields an average more than four times greater.

Figure 5.13 depicts this graphically showing the run times for Incremental Local Search and Incremental in relation to each other. Noteworthy is that the approach including Local Search was able to solve *easy* instances consistently in less time than Incremental. This is shown in the left of Figure 5.13. However, once the Local Search part of Incremental Local Search is not able to find a solution easily, Incremental is advantageous as it does not have the added drawback of running Local Search without it finding a solution. This behaviour can be seen in the middle part of Figure 5.13 in which *columns* with a spacing of the Local Search's maximum run time are depicted indicating that the Local Search part was not able to find a solution, but Incremental could. Lastly, to the far right is a column which depicts that there are multiple instances the Incremental could solve within various time frames, but not Incremental Local Search. Still, there are 9 instances that were only solved by the Local Search part in Incremental Local Search, but not by Incremental.

Figure 5.16 shows the overall performance profile based on the run times of Incremental and Incremental Local Search. Most importantly, it is shown that the Incremental Local Search is the better solving strategy in only around 22% of instances. In contrast to the previously shown performance profiles, the worse solver performs better within the same range of factors τ, i.e. it shows that the difference of solved instances is a lot smaller and that also the runtime differences for these instances is not too drastic. This is also shown in Figure 5.15 showing the number of solved instances per time.

In contrast to the results from Sections 5.1 and 5.2, Incremental Local Search does not have a large advantage in regards to memory consumption as shown in Figure 5.14, which plots the peak memory consumption for both solving strategies per instance. Both solving strategies utilize around the same peak memory. It is noteworthy, however, that there are multiple outliers for both Incremental Local Search and Incremental which skew the average in favour of Incremental Local Search. In fact, Incremental Local Search used less memory in only 36.0% of all instances.

In the end, using Local Search in an incremental strategy has not shown to be beneficial over a pure incremental strategy, although the result differences between the used strategies is not as drastic as in the previous sections.

Total = 5248	Incr. LS_{30s}	$Incr._{30s}$
SAT	4678	**4788**
UNKNOWN	5	7
Timeout	**422**	313
Memout	**143**	140

Table 5.7: Benchmark results for SMT-LIB instances using a DPLL(T) approach. Larger numbers marked.

Average [s] / [MB]	Incr. LS_{30s}	$Incr._{30s}$
Time: Overall	5.35	**3.51**
-Solved	0.72	**0.20**
-Solved by both	0.70	**0.16**
Memory Overall	**15.9**	16.1
-Solved	**8.3**	13.2
-Solved by both	**8.2**	8.3

Table 5.8: Runtime and memory averages for SMT-LIB instances using a DPLL(T) approach. Smaller numbers marked.

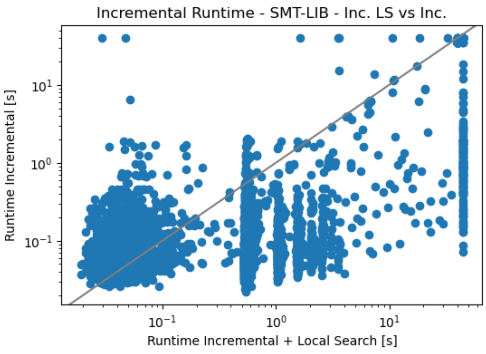

Figure 5.13: Scatter runtime comparison for SMT-LIB instances on a DPLL(T) approach.

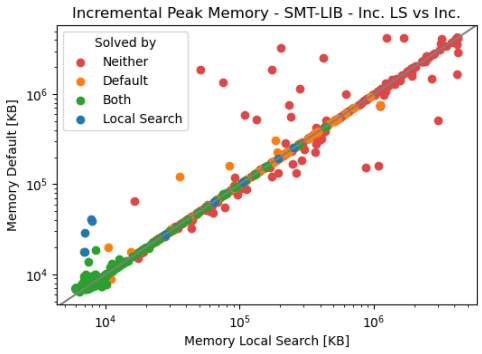

Figure 5.14: Scatter memory comparison for SMT-LIB instances on a DPLL(T) approach.

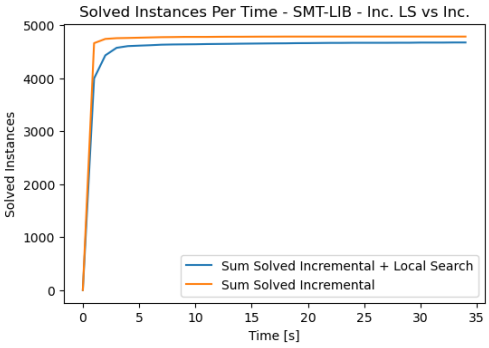

Figure 5.15: Cumulative plot for SMT-LIB instances depicting the number of instances solved after a certain time for a DPLL(T) approach.

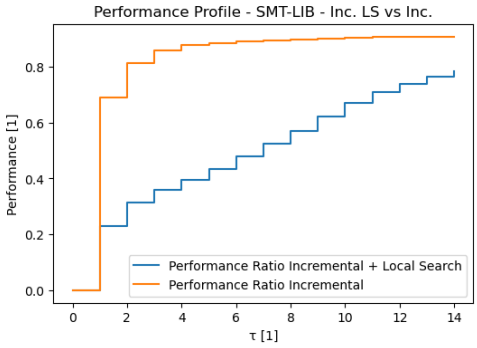

Figure 5.16: Performance profile for a DPLL(T) approach regarding runtimes for SMT-LIB instances.

5.4 Combined Solver

As shown in Section 5.1 and Section 5.3, there are a few instances from SMT-LIB that Local Search can solve, but neither Default, nor Incremental can. Section 5.1 also shows that Local Search is not a viable solver for instances similar to those covered by SMT-LIB. However, combining Local Search *and* Default to a combined strategy, referred to as Combined, might yield results beating each individual solving strategy. We create different instances of this combined strategy in which we use Local Search for a different time $t \in [2\text{s}, 5\text{s}, 10\text{s}, 15\text{s}]$ and Default for the remaining $(30-t)$s, referred to as Combined$_t$ or Comb$_t$. It is noteworthy that the following results are constructed out of Local Search's and Default's individual 30s runs on the SMT-LIB and easier generated benchmark sets.

Total = 5248	Default$_{30s}$	Comb$_{2s}$	Comb$_{5s}$	Comb$_{10s}$	Comb$_{15s}$
SAT	4883	4888	**4889**	4887	4881
–Default	**4883**	3122	3108	3096	3087
–LS	0	1766	1781	1791	**1794**
UNKNOWN	0	0	0	0	0
Timeout	273	268	267	269	**275**
Memout	92	92	92	92	92

Table 5.9: Benchmark results for SMT-LIB instances using Default and Combined with various times for Local Search. Larger numbers marked.

Total = 500	Default$_{30s}$	Comb$_{2s}$	Comb$_{5s}$	Comb$_{10s}$	Comb$_{15s}$
SAT	290	473	499	499	**500**
–Default	**290**	51	16	5	0
–LS	0	422	483	494	**500**
UNKNOWN	0	0	0	0	0
Timeout	**210**	27	1	1	0
Memout	0	0	0	0	0

Table 5.10: Benchmark results for easier generated instances using Default and Combined with various times for Local Search. Larger numbers marked.

Table 5.9 shows a general overview over the results from the SMT-LIB benchmark set. It is noteworthy that all combined solving strategies except Comb$_{15s}$ solve more instances than Default. For SMT-LIB, Comb$_{5s}$ solves the most instances with a total improvement of 6 instances. Table 5.10 shows the corresponding results for the easier generated instances. It is easy to see that for those instances, every combined solving strategy outperforms Default. In fact, Comb$_{15s}$ solves all instances and is thus on the same performance level as Local Search. However, Combined using 5s and 10s respectively only solve a single instance less. Based on total numbers, Comb$_{5s}$ is the strongest Combined approach beating Default on both SMT-LIB and easier generated instances.

Average [s] / [MB]	Def.$_{30s}$	Comb$_{5s}$
Time: Overall	**3.30**	6.26
-Solved	**0.27**	3.41
-Solved by both	**0.23**	3.42
Memory Overall	94.3	**93.5**
-Solved	15.6	**14.9**
-Solved by both	15.6	**15.0**

Table 5.11: Runtime and memory averages for SMT-LIB instances using Default and Combined with 5s for Local Search. Smaller numbers marked.

Average [s] / [MB]	Def.$_{30s}$	Comb$_{5s}$
Time: Overall	20.04	**1.43**
-Solved	1.96	**1.35**
-Solved by both	1.96	**1.47**
Memory Overall	18.6	**6.6**
-Solved	12.6	**6.6**
-Solved by both	12.6	**6.8**

Table 5.12: Runtime and memory averages for easier generated instances using Default and Combined with 5s for Local Search. Smaller numbers marked.

Table 5.11 shows the average runtimes and peak memory consumption for Default and Comb$_{5s}$ on SMT-LIB instances. Default consistently has a smaller average runtime, even averaging approximately half of Comb$_{5s}$'s average's runtime. In particular, on instances solved by both solving strategies, Default's average time is only 6.7% of Comb$_{5s}$'s corresponding average runtime. Only in peak memory consumption is Comb$_{5s}$ slightly advantageous to Default in regards to averages for SMT-LIB instances. Considering the runtime and memory averages from Table 5.12 for the easier generated instances yields that Comb$_{5s}$ beats Default in every category.

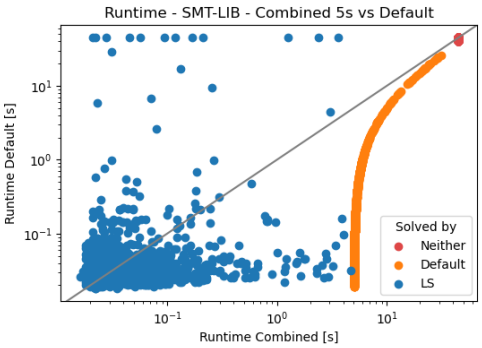

Figure 5.17: Scatter runtime comparison for SMT-LIB instances using Default and Combined with 5s for Local Search.

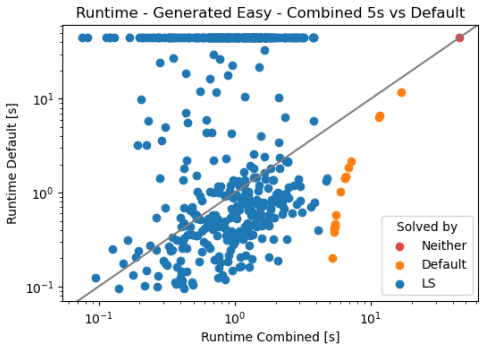

Figure 5.18: Scatter runtime comparison for easier generated instances instances using Default and Combined with 5s for Local Search.

Lastly, Figure 5.17 and Figure 5.18 depict the runtime comparison on an instance basis for Comb$_{5s}$ and Default. They also show whether the Local Search part, the Default part, or neither part solved an instance. In SMT-LIB instances, the runtimes in the Local Search part are skewed in favour of Default while remaining small. Furthermore, the curved line on the right depicts the instances solved by the Default part in Combined for which many were solved in a short amount of time once the Default part was consulted. Lastly, the instance that Combined could solve, but not Default itself, are depicted at the very top.

It is easy to see that Comb$_{5s}$ solves a lot more instances in the easier generated benchmark set as shown in Figure 5.18. These instances are again shown at the very

top. It is also noteworthy that a few instances are solved by the Default part depicted on the right.

Overall, the results show that Comb_{5s} offers an improvement over Default in both SMT-LIB and generated instances by slightly increasing the amount of solved instances for SMT-LIB and greatly increasing the number of solved generated instances. Only the worse runtimes for Comb_{5s} are disadvantageous.

5.5 Settings

With Local Search being dependent on a multitude of parameters, it is also important to see how different settings affect Local Search's ability to solve formulas. Recall that Table 4.1 shows different sets of settings for the parameters. Settings_1 is the initial set of parameter values based on the values presented in the article [LXZ23]. In this section, Local Search with its initial settings (LS_1) will be compared to Local Search with improved settings (LS_2), depicted in Figure 4.1 as Settings_2, to show *why* the improvements were made and *how* they improve Local Search's performance.

Total: 5248	$\text{LS}_{1,30s}$	$\text{LS}_{2,30s}$
SAT	1699	**1798**
UNKNOWN	**3499**	3403
Timeout	**50**	47
Memout	0	0

Table 5.13: Benchmark results for SMT-LIB instances using using Local Search with Settings_1 and Settings_2. Larger numbers marked.

Average [s] / [MB]	$\text{LS}_{1,30s}$	$\text{LS}_{2,30s}$
Time: Overall	20.58	**20.01**
-Solved	0.20	0.20
-Solved by both	0.13	**0.10**
Memory Overall	13.88	**13.87**
-Solved	5.98	5.98
-Solved by both	5.98	**5.97**

Table 5.14: Runtime and memory averages for SMT-LIB instances using Local Search with Settings_1 and Settings_2. Smaller numbers marked.

Figure 5.13 shows a general overview of the benchmark results. The improved settings can solve 99 more instances and thus offer an overall improvement without creating too much additional memory overhead. It is noteworthy that these are just the absolute values. LS_1 can solve 31 instances that LS_2 cannot, while LS_2 can solve 130 instances LS_1 is not able to solve. In addition, LS_2 needs less time to solve an instance on average for instances solved by both sets of settings as shown in Figure 5.14. But for instances that are solved by Local Search with either settings, there is no distinguishable time difference. It is noteworthy, however, that the run times for both sets of settings vary strongly on an instance basis, without skewing in favour of one set of settings not considering unsolved instances. This behaviour is shown in Figure 5.19. Most of the instances are still solved within a second. In terms of memory, no set of settings shows a distinct advantage over the other. For most instances, both solvers tend to use the same amount of memory, there are only a few instances in which set of settings outperforms the other. These instances also use very little memory, as shown in Figure 5.20. Hence, the new improved set of settings can solve more instances in the same amount of time without requiring more memory making Local Search with Settings_2 the better overall solver as shown in the performance plot depicted in Figure 5.21. It also shows that the improved set of settings is better in approximately 25% of all instances, while the initial settings were advantageous in only 11%.

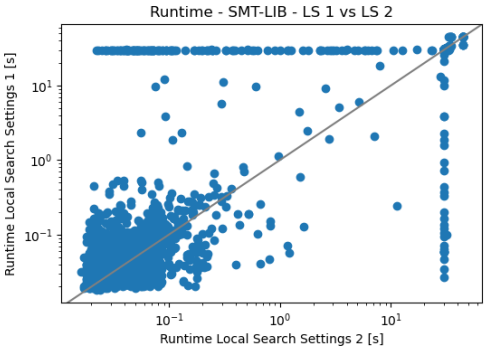

Figure 5.19: Runtime scatter plot for Local Search on SMT-LIB instances using Local Search with Settings_1 and Settings_2.

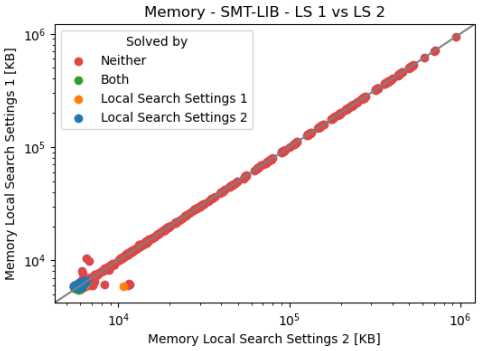

Figure 5.20: Memory scatter plot for Local Search on SMT-LIB instances using Local Search with Settings_1 and Settings_2.

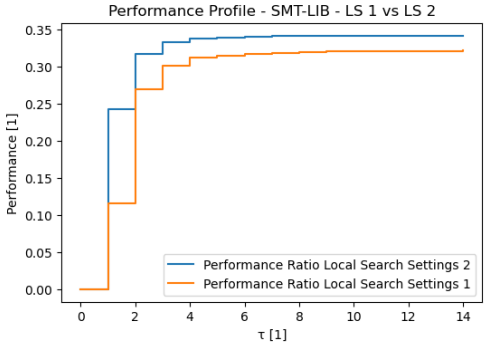

Figure 5.21: Performance profile for Local Search with Settings_1 and Settings_2 regarding runtimes for SMT-LIB instances.

After considering the results from Table 5.15 for the initial set of settings LS_1, it is clear that a lot of operations were forbidden by the forbidding strategy thus creating a lot of forced restarts. Because of this, the parameters for the forbidding strategy were changed in order to forbid fewer operations. The forbidding strategy is to be disabled after a few iterations while it forbids operations for fewer iterations still. In addition to this, through changes made to SP, PP and B_OFFSET, a finer distinction of suitable operations was to be expected. Together, this led to a drastic decrease in forbidden operations, but also to a decrease in forced restarts for both general solved instances and instances solved by both sets of settings. However, the needed iterations and single operations actually increased slightly in instances that were generally solved. This is acceptable though because a lot of instances were solved in addition to the initial set of settings. Compared to instances that were solved by both sets of settings, however, the needed iterations and single operations were decreased. Furthermore, more multi operations were found to solve instances that could not be solved previously, but they do not benefit Local Search in instances previously solved already. In addition to this, Boolean operations have not shown to be beneficial or influential.

These improved settings were then altered to be used in a incremental setting by

adjusting the time based on the mean solving time for solvable instances and disabling clause weights resets such that Local Search can keep information learnt from previous solving attempts to be used in the next incremental call for a similar formula.

Average [1]	$LS_{1,30s}$	$LS_{2,30s}$
Iterations	**23145.2**	42161.4
-Solved	**31.5**	35.8
-Solved by both	21.7	**12.8**
Forbidden Operations	72434.8	**40.2**
-Solved	65.8	**3.2**
-Solved by both	42.8	**2.3**
Forced Restarts	12040.3	**11304.8**
-Solved	10.9	**9.1**
-Solved by both	7.4	**2.8**
Single Operations	**11081.6**	30705.0
-Solved	**19.3**	24.9
-Solved by both	13.1	**8.7**
Multi Operations	**12.8**	151.1
-Solved	**0.3**	0.8
-Solved by both	0.3	0.3
Boolean Operations	**3.6**	6.7
-Solved	0.4	0.4
-Solved by both	**0.2**	0.3

Table 5.15: Averages for SMT-LIB instances using Local Search with $Settings_1$ and $Settings_2$ for Local Search specific statistics. Smaller numbers marked.

Chapter 6

Conclusion

Local search following the ideas proposed in the article [LXZ23] is a heuristical approach for solving quantifier-free non-linear real arithmetic problems. While the search space for these kind of problems is infinite, through the means of Tarski's decision procedure [Tar51] or *Cylindrical Algebraic Decomposition* [Col74], it is possible to decide each problem using only finitely many sample points from the infinite domain. However, *constructing* these points is computationally expensive. Hence, the local search approach uses a heuristic to jump from one sample point to another such that the new point satisfies at least one previous falsified literal without the need to do lots of hard computations. For this, *cell jump operations* are introduced to alter the assignment to either change a single variable, or multiple variables at once. Local Search was then tested on various instances from SMT-LIB, as well as self-generated instances with different properties, to determine its effectiveness.

6.1 Benchmarks

Following the results from Sections 5.1, 5.2, 5.4 and 5.3, it is easy to see that Local Search does not offer any benefit in SMT-LIB instances as a stand-alone solver. Local Search only solves a fraction of all tested instances and just over a third of instances solved by Default. Furthermore, it does not offer any runtime improvements for the solved instances over Default. Only in peak memory usage Local Search outperforms the default strategy, yet *solving* instances is much more important than using less memory in most applications.

Using Local Search in an incremental strategy for SMT-LIB instances has also shown to be not beneficial, yet the differences to the pure incremental strategy are a lot smaller. Local Search is favoured for small and easy instances in the incremental approach, yet once it cannot find a solution, it adds a lot of computational overhead increasing the average run times. In addition, Local Search does not have an advantage in memory consumption either.

Only in the generated instances Local Search performs very well as a stand-alone solver in contrast to Default. For the harder instances, Local Search is able to solve 78.6% of all instances within 30s while Default is not able to solve a single instance. This is because it exceeds the memory limit of 4GB. Even without a memory limit, the default strategy is still not able to solve any instance within 120s. After reducing the polynomial complexity of these instances, Default is able to solve 57.6% of all instances

while Local Search solves every instance. Furthermore, Local Search has a smaller average runtime and uses less memory. Hence, Local Search outperforms Default by a large amount making Default not viable for any of the generated instances.

In addition to this, testing Local Search on a different set of settings has shown to produce different results for SMT-LIB instances. While not changing the memory usage of Local Search, the absolute amount of solved instances could be increased. Local Search with *improved* settings is able to solve 4.3% more instances posing a great improvement without altering the core idea of the local search approach.

Lastly, a combined approach of Local Search and Default has yielded a strategy that outperforms Default on both SMT-LIB and easier generated instances. Combined_{5s} solves 6 more instances in SMT-LIB and 209 more instances for the easier generated benchmark set. Only the runtimes for the combined approach favour Default.

6.2 Discussion

Local Search's simplicity is also its drawback. It only needs to isolate roots of polynomials, but it does not need to perform extensive work with the roots. It does not calculate resultants as an example, nor does it have the ability to calculate infeasible subsets. It does not generate lemmas or propagates implicit information. Local Search is a lightweight, incomplete procedure which can be run for a short amount of time to potentially find a solution to a hard problem.

Because Local Search does not need to store much additional information besides the current assignment, its peak memory usage stays considerably smaller than Default's and does not change over time based on information learnt. Hence, it also easily scales with larger instances. In addition to this, because Local Search does not perform extensive work as previously mentioned, it does not run into problems with high degree polynomials as soon as the default strategy. Combined with an easy Boolean abstraction as in the generated instances, Local Search reaches its peak performance as a stand-alone solver. Compared to SMT-LIB instances which are relatively small in polynomial degree, but have a hard Boolean abstraction, Local Search cannot use its advantage to do easier work because of the smaller polynomial degree, but will also have to restart multiple times because it might run into assignments from which it cannot find a solution using the defined operations and heuristic because of the hard Boolean abstraction. Furthermore, Local Search cannot learn from these restarts or assignments like Default can. Not being able to use previous learnt information about infeasible subsets, and in addition not learning from restarts, is a huge factor for Local Search in a incremental strategy. It is shown that Incremental Local Search does not perform as bad a Local Search in SMT-LIB, but it most importantly it does not offer any improvement.

Lastly, most SMT-LIB and easier generated instances are solved within a short amount of time for Local Search. As shown, it even solves instances unsolvable by Default. Combining Local Search and Default yields promising results as Local Search is only used for a short amount of time to cover many of its solvable instances, while the remaining instances, which are mostly unsolvable for Local Search, are handled by Default. Hence, this combined solver performs better than Default.

6.3 Summary

Stand-alone Local Search is a solving strategy that has only been shown to be very effective for high degree polynomial formulas with easy Boolean abstraction. While Local Search is able to solve a third of SMT-LIB instances, it is not deemed useful for solving those as there are better and faster solvers and strategies for it. Local Search, however, clearly outperforms every other used solving strategy on the generated instances. Thus, for higher degree formulas, Local Search is *the* strategy to use, as long as the Boolean abstraction stays relatively simple.

It is easy to see that Local Search as a stand-alone solver is not a *do-it-all* solver, but highly specific for the use case of high degree polynomials. Hence, a possible user should first determine if Local Search is a solver that might perform well on the respective problem instance, i.e. if the problem instance has characteristically many high degree polynomials such that Local Search can use its advantage over other solvers. Only then Local Search is to be used to its fullest extend.

However, Local Search can be successfully used in a combined strategy with Default to outperform the standard default strategy. It improves the amount of solved instances within 30s for both SMT-LIB and easier generated instances. Hence, for any problem instance the combined approach offers a greater chance of solving the instance.

Further work on Local Search could consist of finding a better heuristic to ensure that the best operation is taken such that fewer restarts happen, as well as adapting Local Search to be able to use equalities more efficiently especially in polynomials that do not have a single linear variable. In addition to this, it might seem seem useful to enforce bounds or single literal clauses pro-actively instead of just using them once in a restart or in general incorporate information learnt along the execution.

Bibliography

[ADEK21] Erika Ábrahám, James H. Davenport, Matthew England, and Gereon Kremer. Deciding the consistency of non-linear real arithmetic constraints with a conflict driven search using cylindrical algebraic coverings. *Journal of Logical and Algebraic Methods in Programming*, 119:100633, February 2021.

[BFT16] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[BO97] Frédéric Benhamou and William J. Older. Applying interval arithmetic to real, integer, and boolean constraints. *The Journal of Logic Programming*, 32(1):1–24, 1997.

[CKJ+15] Florian Corzilius, Gereon Kremer, Sebastian Junges, Stefan Schupp, and Erika Ábrahám. Smt-rat: An open source c++ toolbox for strategic and parallel smt solving. In Marijn Heule and Sean Weaver, editors, *Theory and Applications of Satisfiability Testing – SAT 2015*, pages 360–368, Cham, 2015. Springer International Publishing.

[Col74] George E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition–preliminary report. *SIGSAM Bull.*, 8(3):8090, aug 1974.

[CS13] Shaowei Cai and Kaile Su. Local search for boolean satisfiability with configuration checking and subscore. *Artificial Intelligence*, 204:75–98, 2013.

[DM02] Elizabeth D. Dolan and Jorge J. Moré. Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91(2):201–213, Jan 2002.

[dMJ13] Leonardo de Moura and Dejan Jovanović. A model-constructing satisfiability calculus. In Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 1–12, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[FOSV17] Pascal Fontaine, Mizuhito Ogawa, Thomas Sturm, and Xuan Tung Vu. Subtropical satisfiability. In Clare Dixon and Marcelo Finger, editors, *Frontiers of Combining Systems*, pages 189–206, Cham, 2017. Springer International Publishing.

[HV95] J. N. Hooker and V. Vinay. Branching rules for satisfiability. *Journal of Automated Reasoning*, 15(3):359–383, Oct 1995.

[JW90] Robert G. Jeroslow and Jinchang Wang. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, 1(1):167–187, Sep 1990.

[LXZ23] Haokun Li, Bican Xia, and Tianqi Zhao. Local search for solving satisfiability of polynomial formulas. In Constantin Enea and Akash Lal, editors, *Computer Aided Verification*, pages 87–109, Cham, 2023. Springer Nature Switzerland.

[Tar51] A. Tarski. *A Decision Method for Elementary Algebra and Geometry.* University of California Press, 1951.

[TPBFJ04] John Thornton, Duc Nghia Pham, Stuart Bain, and Valnir Ferreira Jr. Additive versus multiplicative clause weighting for sat. In *AAAI*, volume 4, pages 191–196, 2004.

[Tse83] Grigori S Tseitin. On the complexity of derivation in propositional calculus. *Automation of reasoning: 2: Classical papers on computational logic 1967–1970*, pages 466–483, 1983.