Search Strategies for Floating Point Constraint Systems

Heytem Zitoun[1(✉)], Claude Michel[1], Michel Rueher[1], and Laurent Michel[2]

[1] CNRS, I3S, Université Côte d'Azur, Nice, France
{heytem.zitoun,claude.michel,michel.rueher}@i3s.unice.fr
[2] University of Connecticut, Storrs, CT 06269-2155, USA
ldm@engr.uconn.edu

Abstract. The ability to verify critical software is a key issue in embedded and cyber physical systems typical of automotive, aeronautics or aerospace industries. Bounded model checking and constraint programming approaches search for counter-examples that exemplify a property violation. The search of such counter-examples is a long, tedious and costly task especially for programs performing floating point computations. Indeed, available search strategies are dedicated to finite domains and, to a lesser extent, to continuous domains. In this paper, we introduce new strategies dedicated to floating point constraints. They take advantage of the properties of floating point domains (e.g., domain density) and of floating point constraints (e.g., floating point arithmetic) to improve the search for floating point constraint problems. First experiments on a set of realistic benchmarks show that such dedicated strategies outperform standard search and splitting strategies.

1 Introduction

A key issue while verifying programs with floating point computations is the search of floating point arithmetic errors that produce results quite different from the expected result over the reals. Consider `foo`, a program doing floating point computations:

```
void foo(){
   float a = 1e8f;
   float b = 1.0f;
   float c = -1e8f;
   float r = a + b + c;
   if(r >= 1.0f)
      doThenPart();
   else doElsePart();
}
```

This work was partially supported by ANR COVERIF (ANR-15-CE25-0002).

J.C. Beck (Ed.): CP 2017, LNCS 10416, pp. 707–722, 2017.
DOI: 10.1007/978-3-319-66158-2_45

Over the reals, r is equal to 1.0 and the `doThenPart` function is called. However, over the floats with a"round to the nearest" rounding mode, an absorption phenomenon occurs: $a + b$ is equal to a and, thus, r is assigned to 0. As a result, the `doThenElse` function is called instead of the `doElsePart` function. This simple example illustrates how the flow of a very simple program over the floats ($\mathbb{F}$) can differs from the expected flow over the reals ($\mathbb{R}$). Such a flow discrepancy might have critical consequences if, for instance, the condition is related to decide whether to brake or not in an ABS system.

Constraint programming has been used to verify such properties [5,16] in a bounded model checking framework [6,7]. However, the search of such counter-examples is a long, tedious and costly task especially for programs performing floating point computations. The use of standard search technique to solve constraints over $\mathbb{F}$ lacks efficiency. Numerous search strategies over finite domains have been proposed [4,8,14,15,17] and, to a lesser extent, over continuous domains [11,12]. But, such strategies do not adapt well to floating point numbers. A subset of integers bounded by two integers is a finite and uniformly distributed set which can be enumerated. A subset of reals bounded by two floating point numbers is an infinite set of reals that cannot be enumerated and thus, search strategies over continuous domains rely on interval arithmetic, bisection and mathematical properties to prove the existence of solutions in some small interval [1]. A contrario, the set of floating point numbers is a finite set with a huge cardinality and a non-uniform distribution (half of the floating point numbers belongs to the interval $[-1, 1]$). The aforementioned technique like enumeration are not well suited to floating point number density and distribution. Though floating point number approximate real numbers, they do not benefit from the same properties such as continuity. It is thus difficult to reuse search strategies designed for the reals with floating point variables.

The purpose of this paper is to introduce new search strategies dedicated to floating point numbers to ease and, perhaps more importantly, speed-up the solving of verification problems. Preliminary experiments performed on a limited but realistic set of benchmarks show that such dedicated strategies outperform standard search and splitting strategies.

2 Notations and Definitions

2.1 Floating Point Numbers

Floating point numbers were introduced to approximate real numbers. The IEEE754-2008 standard for floating point numbers [10] sets floating point formats, as well as, some floating point arithmetic properties. The two most common formats defined in the IEEE754 standard are *simple* and *double* floating point number precision which, respectively, use 32 bits and 64 bits. A floating point number is a triple (s, m, e) where $s \in \{0, 1\}$ represents the sign, the p bits m, the significant or mantissa and, e the exponent [9]. A *normalized* floating point number is defined by:

$$(-1)^s 1.m \times 2^e$$

To allow gradual underflow, IEEE754 introduces de-normalized numbers whose value is given by:

$$(-1)^s 0.m \times 2^0$$

Note that simple precision are represented with 32 bits and a 23 bits mantissa ($p = 23$) while doubles use 64 bits and a 52 bits mantissa ($p = 52$).

2.2 Absorption

Absorption occurs when adding two floating point numbers with different order of magnitude. The result of such an addition is the furthest from zero. For instance, in C, using simple floating point numbers with a rounding mode set to "round to nearest", $10^8 + 1.0$ evaluates to 10^8. Thus, 1.0 is absorbed by 10^8.

2.3 Cancellation

Cancellation occurs when most of the most significant bits are lost. For instance, it appears when subtracting the close results of two operations. Consequences of cancellation increase with the accumulation of rounding errors. Such a phenomenon is highlighted by subtracting two close operands [18].

For instance, evaluating[1] `((1.0f - 1.0e-7f) - 1.0f) * 1.0e+7f` in C using simple floating point numbers and a rounding mode sets to "round to nearest" yields 1.1920928955078125 instead of -1.0. Indeed, over $\mathbb{F}$ subtracting 1.0 to the result of $1.0 - 10^{-7}$ leads to loose the most significant bits. The subtraction result is then used in a product that amplifies this loss in the mantissa.

2.4 Notations

In the sequel, x, y and z denote variables and $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{z}$, their respective domains. When required, $x_{\mathbb{F}}$, $y_{\mathbb{F}}$ and $z_{\mathbb{F}}$ denote variables over $\mathbb{F}$ and $\mathbf{x}_{\mathbb{F}}$, $\mathbf{y}_{\mathbb{F}}$ and $\mathbf{z}_{\mathbb{F}}$, their respective domains while $x_{\mathbb{R}}$, $y_{\mathbb{R}}$ and $z_{\mathbb{R}}$ denote variables over $\mathbb{R}$ and $\mathbf{x}_{\mathbb{R}}$, $\mathbf{y}_{\mathbb{R}}$ and $\mathbf{z}_{\mathbb{R}}$, their respective domains. Note that $\mathbf{x}_{\mathbb{F}} = [\underline{x}_{\mathbb{F}}, \overline{x}_{\mathbb{F}}] = \{x_{\mathbb{F}} \in \mathbb{F}, \underline{x}_{\mathbb{F}} \leq x_{\mathbb{F}} \leq \overline{x}_{\mathbb{F}}\}$ with $\underline{x}_{\mathbb{F}} \in \mathbb{F}$ and $\overline{x}_{\mathbb{F}} \in \mathbb{F}$. Likewise, $\mathbf{x}_{\mathbb{R}} = [\underline{x}_{\mathbb{R}}, \overline{x}_{\mathbb{R}}] = \{x_{\mathbb{R}} \in \mathbb{R}, \underline{x}_{\mathbb{R}} \leq x_{\mathbb{R}} \leq \overline{x}_{\mathbb{R}}\}$ with $\underline{x}_{\mathbb{R}} \in \mathbb{F}$ and $\overline{x}_{\mathbb{R}} \in \mathbb{F}$. Let $x_{\mathbb{F}} \in \mathbb{F}$, then $x_{\mathbb{F}}^{+}$ is the smallest floating point number strictly superior to $x_{\mathbb{F}}$ and $x_{\mathbb{F}}^{-}$ is the biggest floating point number strictly inferior to $x_{\mathbb{F}}$. In addition, given a constraint c, $vars(c)$ denotes the set of floating point variables appearing in c.

3 Properties of Floating Point Domains, Variables and Constraints

This section defines properties on floating point domains and constraints that are useful to build dedicated search strategies. Domain properties like cardinality or

[1] One must take care to annotate all literals with 'f' to force floating point constants and to decompose the expression into elementary arithmetic operations to prevent the compiler from evaluating at compile time.

density capture the structure of the domains of the floating point variables. Constraint properties take into account floating point arithmetic properties like absorption or cancellation. They also capture structural properties by, for instance, taking advantage of the derivative.

3.1 Properties of Floating Point Domains and Variables

Definition 1 (Width). *Let* $w(\mathbf{x}_\mathbb{F})$ *the width of domain* $\mathbf{x}_\mathbb{F}$ *be defined as*

$$w(\mathbf{x}_\mathbb{F}) = \overline{x}_\mathbb{F} - \underline{x}_\mathbb{F}$$

The domain width is defined by the distance between its two bounds. It is a rather historical criteria. On finite domains, many strategies rely on this criteria, especially one of the most widespread, namely *minDom* [14]. Selecting variables with the smallest domain aims at focusing on the most constrained variables. However, over the floats, this criteria is questionable because of the non uniformly distributed floating point values. Here, a smaller width does not necessarily mean a smaller number of values.

Example 1 (Width versus size). Let $x_\mathbb{F}$ and $y_\mathbb{F}$ be two simple floating point variables and $\mathbf{x}_\mathbb{F} = [1, 2]$, $\mathbf{y}_\mathbb{F} = [10, 12]$ be their respective domains. While $w(\mathbf{x}_\mathbb{F}) = 1$ and $w(\mathbf{y}_\mathbb{F}) = 2$, $\mathbf{x}_\mathbb{F}$ contains 8388608 values and $\mathbf{y}_\mathbb{F}$ contains 2097152 values. Thus, the most constrained variable is $y_\mathbb{F}$ rather than $x_\mathbb{F}$.

Definition 2 (Cardinality). *Let* $|\mathbf{x}_\mathbb{F}|$ *denotes the cardinality of domain* $\mathbf{x}_\mathbb{F}$. *Given* $\mathbf{x}_\mathbb{F} = [\underline{x}_\mathbb{F}, \overline{x}_\mathbb{F}]$ *with* $\underline{x}_\mathbb{F} \geq 0$ *one can define* $|\mathbf{x}_\mathbb{F}|$ *with*

$$|\mathbf{x}_\mathbb{F}| = 2^p * (e_{\overline{x}_\mathbb{F}} - e_{\underline{x}_\mathbb{F}}) + m_{\overline{x}_\mathbb{F}} - m_{\underline{x}_\mathbb{F}} + 1$$

where $e_{\overline{x}_\mathbb{F}}$ *and* $e_{\underline{x}_\mathbb{F}}$ *are the exponents of, respectively,* $\overline{x}_\mathbb{F}$ *and* $\underline{x}_\mathbb{F}$, *and* $m_{\overline{x}_\mathbb{F}}$ *and* $m_{\underline{x}_\mathbb{F}}$ *are the mantissa of, respectively,* $\overline{x}_\mathbb{F}$ *and* $\underline{x}_\mathbb{F}$ *while* p *is the length of the mantissa.*

This formula can be extended to other cases by exploiting symmetries. Figure 1 illustrates how the cardinality of a floating point interval is computed. The bold double ended arrow represents the interval $\mathbf{x}_\mathbb{F}$. The main idea is to compute the number of floating point values contained in the interval $[2^{e_{\underline{x}_\mathbb{F}}}, 2^{e_{\overline{x}_\mathbb{F}}}]$ (computed by $2^p * (e_{\overline{x}_\mathbb{F}} - e_{\underline{x}_\mathbb{F}}) + 1$) represented by the simple double ended arrow. Then, it withdraws the number of floats in $[2^{e_{\underline{x}_\mathbb{F}}}, \underline{x}_\mathbb{F})$ (i.e., $m_{\underline{x}_\mathbb{F}}$ floats) and adds the number of floats in $(2^{e_{\overline{x}_\mathbb{F}}}, \overline{x}_\mathbb{F}]$ (i.e., $m_{\overline{x}_\mathbb{F}}$ floats).

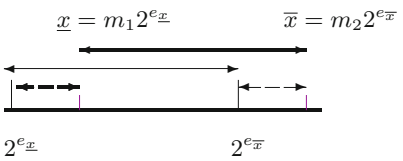

Fig. 1. Computing cardinality of floating point intervals

Notice that, over finite domains, width and cardinality return nearly the same values (especially when there are no 'holes' in the finite domains). However, over the floats, these two properties are not correlated. Width and cardinality play different roles over the floats. Cardinality could be used to identify either the domain with the smallest number of floats, i.e., the variable that constraint the most the problem, or the domain with the biggest number of floats, i.e., the variable with a high potential of solutions.

Definition 3 (Density). *Let* $\rho(\mathbf{x}_\mathbb{F})$ *the density of* $\mathbf{x}_\mathbb{F}$ *be defined as*

$$\rho(\mathbf{x}_\mathbb{F}) = \frac{|\mathbf{x}_\mathbb{F}|}{w(\mathbf{x}_\mathbb{F})}$$

Intuitively, density captures the proximity of floating point values within a given domain. It helps identifying domains that have a small number of values on a big domain or a big number of values on a small domain (with respect to the width). The former allows to reach easily values that should correspond to various behaviors while the latter potentially contains many values corresponding to the same behavior. Remember that, over the floats, density increases near zero.

Definition 4 (Magnitude). *Let* $mag(\mathbf{x}_\mathbb{F})$ *be the magnitude of* $\mathbf{x}_\mathbb{F}$ *and defined as*

$$mag(\mathbf{x}_\mathbb{F}) = \frac{e_{\underline{x}_\mathbb{F}} + e_{\overline{x}_\mathbb{F}}}{2 \cdot e_{max}}$$

where e_{max} *is the biggest exponent in* $\mathbb{F}$*.*

In practice, the magnitude of $[0, 1]$ should be near zero while magnitude of $[10^{36}, 10^{37}]$ should be near 1. In essence, the property helps identifying domains that mainly hold big values or small values. More precisely, magnitude has a dual purpose. First, the property helps selecting variables involved in an absorption, for instance when a big magnitude domain and a small magnitude domain are both involved in an addition. This is easier to implement but less precise than the dedicated property defined in the upcoming Definition 8. Second, this property might help selecting domains with extreme values. Extreme values are those that are often associated to undesirable behaviors.

Definition 5 (Degree). *Let* $degree(x_\mathbb{F})$ *denote the degree of a variable* $x_\mathbb{F}$ *and be defined as the number of constraints in which* $x_\mathbb{F}$ *appears. It is defined as*

$$degree(x_\mathbb{F}) = \sum_{c \in C} (x_\mathbb{F} \in vars(c))$$

where C *is the set of constraints.*

Naturally, the degree definition mirrors its counterpart in finite-domain solvers. It is a static property. The higher the degree of $x_\mathbb{F}$, the more $x_\mathbb{F}$ plays an important role in the solving process. Many strategies over finite domains take advantage of this property like the weighted degree strategy [4].

Definition 6 (Occurrences). *Let* $occur(x_{\mathbb{F}})$ *denote the maximum number of occurrences of* $x_{\mathbb{F}}$ *among all constraints in a set* C *be defined as*

$$occur(x_{\mathbb{F}}) = \max_{c \in C} count(x_{\mathbb{F}}, c)$$

where $count(x_{\mathbb{F}}, c)$ *is the number of* $x_{\mathbb{F}}$ *occurrences in constraint* c.

Multiple occurrences is a recurring problem in handling floating point variables. While solutions have been proposed to handle this problem [2,13], identifying variables with multiple occurrences, might help by, for instance, choosing a more adapted filtering process and fixing these variables as soon as possible.

3.2 Properties of Floating Point Constraints

This section introduces properties that take advantage of floating point arithmetic operators used within constraints. The properties will be helpful to define constraint-driven branching strategies.

To appreciate the first property, consider a floating point addition constraint $z_{\mathbb{F}} = x_{\mathbb{F}} \oplus y_{\mathbb{F}}$ in which the rounding more is set to "round to nearest even". If the domain $x_{\mathbb{F}}$ has a significantly larger magnitude than $y_{\mathbb{F}}$, some values in $y_{\mathbb{F}}$ may simply be absorbed when carrying out the addition. Measuring which *fraction* of $y_{\mathbb{F}}$ is obliterated in this way is the purpose of the absorption property.

Definition 7 (Absorption). *Let* $absorb(\mathbf{y}_{\mathbb{F}}, \mathbf{x}_{\mathbb{F}})$ *denote the absorption of* $\mathbf{y}_{\mathbb{F}}$ *by* $\mathbf{x}_{\mathbb{F}}$ *and be defined as:*

$$absorb(\mathbf{y}_{\mathbb{F}}, \mathbf{x}_{\mathbb{F}}) = \frac{|[-2^{e_{max}-p-1}, 2^{e_{max}-p-1}] \cap \mathbf{y}_{\mathbb{F}}|}{|\mathbf{y}_{\mathbb{F}}|}$$

Namely, it is the number of $\mathbf{y}_{\mathbb{F}}$ *values that are absorbed by at least a value of* $\mathbf{x}_{\mathbb{F}}$. *In the above,* e_{max} *is the exponent of* $max\{abs(\underline{x}_{\mathbb{F}}), abs(\overline{x}_{\mathbb{F}})\}$.

Note how $\mathbf{u}_{\mathbb{F}} = [-2^{e_{max}-p-1}, 2^{e_{max}-p-1}] \cap \mathbf{y}_{\mathbb{F}}$ captures the part of $\mathbf{y}_{\mathbb{F}}$ that is absorbed by the biggest value in magnitude in $\mathbf{x}_{\mathbb{F}}$. Thus, if none of the values of $\mathbf{y}_{\mathbb{F}}$ are absorbed by $\mathbf{x}_{\mathbb{F}}$, $\mathbf{u}_{\mathbb{F}}$ will be empty and $absorb(\mathbf{y}_{\mathbb{F}}, \mathbf{x}_{\mathbb{F}})$ will be equal to 0. On the contrary, when all values of $\mathbf{y}_{\mathbb{F}}$ are absorbed by $\mathbf{x}_{\mathbb{F}}$, $\mathbf{u}_{\mathbb{F}}$ will be equal to $\mathbf{y}$ and $absorb(\mathbf{y}_{\mathbb{F}}, \mathbf{x}_{\mathbb{F}})$ will be equal to 1. Selecting variables that are involved in an absorption could help improving the quality of the software and providing counter-examples that instantiate an absorption (see Fig. 2).

The next property only applies to subtraction constraints.

Definition 8 (Cancellation). *Given a floating point subtraction constraint* $z_{\mathbb{F}} = x_{\mathbb{F}} \ominus y_{\mathbb{F}}$ *where* $\ominus$ *is the floating point subtraction with the rounding more set to "round to nearest even", let cancellation denote the number of bits canceled by the subtraction and be defined as*

$$cancellation = max\{e_{\underline{x}_{\mathbb{F}}}, e_{\overline{x}_{\mathbb{F}}}, e_{\underline{y}_{\mathbb{F}}}, e_{\overline{y}_{\mathbb{F}}}\} - min\{e_{z_{\mathbb{F}}}, z_{\mathbb{F}} \in \mathbf{z}_{\mathbb{F}}\}$$

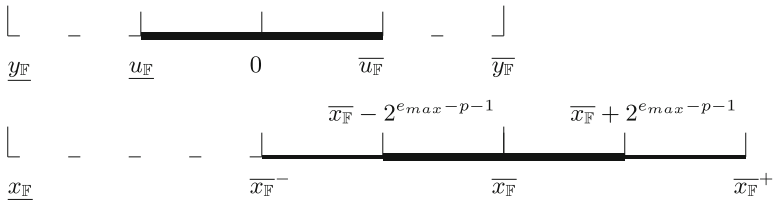

Fig. 2. Illustration of absorption phenomena

The *cancellation* definition was extracted from [3]. It increases with the number of canceled bits and whenever it becomes strictly positive, some bits are potentially lost.

Definition 9 (Derivative). *Given a constraint* $c : e_1 \lozenge e_2$ *in which* $\lozenge \in \{=, \leq, \geq, <, >\}$, c *can be rewritten as* $f : e_1 - e_2 \lozenge 0$. *If* f *is a monovariate function, its derivate can be evaluated using interval arithmetic and gives rise to the definition of* c*'s derivative as*

$$derive(c) = \mathbf{f}'(\mathbf{x}) \approx \frac{\mathbf{f}(\mathbf{x}+h) - \mathbf{f}(\mathbf{x})}{h}$$

The approach generalizes to the case where f is a multivariate function. Its jacobian J gives the variation of each of the variables of f according to other variables of f. Using this matrix, either component-wise or by computing an aggregation of the variation of each variable according to the others, the involvement of a variable of f in the variation of f can be estimated.

Over the floats, a big variation of f might introduce some holes in the representation of the function while a small variation is often represented by the same floating point value. It thus provides useful information to drive the search.

4 Search Strategies for the Floats

As usual, search strategies over floats are based on a combination of variable selection heuristics and splitting techniques. The next subsection introduces different variable selection heuristics based on the above-mentioned properties. The subsection wraps up with four splitting techniques used in the experiments.

4.1 The Choice of a Variable

Single Property Strategies
Single property based strategies select the variable that either maximizes or minimizes the chosen properties. For instance, one can choose the variable that maximizes the domain density or the one that minimizes this density. That's to say, $maxDens = \max_{x_{\mathbb{F}} \in X} \rho(x_{\mathbb{F}})$ (ditto for $minDens$).

Other constraint properties deserve a more specialized approach. For instance, *absorb* or *cancellation* can be maximized or minimized while the minimization or maximization of *derive* should be done according to its absolute value. The *absorb* property is based on constraints of the form z = x + y. So, to implement this property, we pick up the subset of constraints from C that are additions (form z = x + y) and for which $absorb(y, x) > 0$.

Finally, *degree* and *occur* are static properties whose value stays the same along the search tree.

Multi Property Strategies

In the following, we define two strategies that are based on two properties: `absWDens` and `densWAbs`.

`absWDens` selects the variable that maximizes density from a subset of variables that are involved in an absorption ($absorb > 0$).

`densWAbs` selects the variable that maximizes absorption among the subset of variables that satisfies $density \geq \frac{maxDens+minDens}{2}$.

4.2 Domain Splitting Strategies

Problems over the floating point numbers are characterized by huge domains and non uniformly distributed values. As a result, an enumeration strategy like the one often used in finite domains would fail to quickly find a solution, spending most of the time to exhaustively enumerate all possible combinations of values. It would also fail by missing the opportunity to reduce the size of the domains which is offered when a classical domain splitting strategy like a simple bisection is followed by a filtering process. However, in the presence of a lot of solutions, a simple bisection (Fig. 3a) quickly reaches its limits, the filtering applied after each bisection being unable to reduce domain sizes. On the other hand, problems with no solution should benefit from a simple bisection.

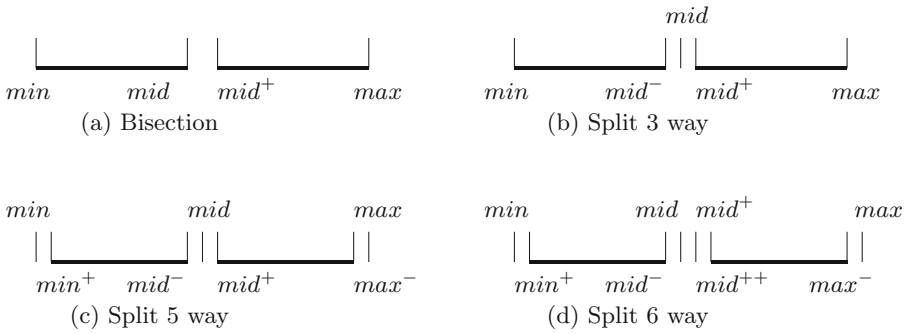

Fig. 3. Different splitting strategies

To overcome these difficulties, we use 3 splitting strategies that mix bisection and enumeration ad that are derived from the strategies introduced in [5]. Instead

of just splitting the domain in two parts, some of the floating point values at the boundaries of the split are isolated and used as enumerated values. Figures 3b–d illustrate the new splitting strategies that combine bisection and enumeration. These combinations begin always with the enumeration of the selected values before handling the two remaining sub-domains. These splitting strategies are called *partial enumeration* splittings.

4.3 Semi-dynamic and Dynamic Strategies

Two alternatives are possible when it comes to composing variable selection and domain splitting. The *semi-dynamic* strategy can first choose a variable and then recursively split that variable until it becomes bound. This approach does not reconsider other variables until the chosen one is grounded. Note that it is possible to leverage any splitting strategy, including the partial enumeration. The *dynamic* strategy adopts a more permissive view. At each node of the search tree, it selects a variable, splits its domain according to some strategy and moves on to possibly select a different variable at the next node. It does not insist on fully instantiating the chosen variable.

5 Experiments

We combined the different variable selection heuristics and splitting techniques on a set of 8 realistic benchmarks. A standard strategy based on a lexicographic order variable selection and *dynamic 2 way* split (i.e., a classical bisection) serves as reference value.

All the experiments were carried out on a MacBook Pro i7 2.3 GHz with 8 GB of memory. All strategies have been implemented in the Objective-CP solver enhanced with floating point constraints. All floating point computations are done with simple precision floats and a rounding mode set to "nearest even".

5.1 Benchmarks

The benchmarks used in these experiments come from test and verification of floating point software.

Heron. The heron function compute the area of a triangle from the lengths of its sides a, b, and c with Heron's formula: $\sqrt{s*(s-a)*(s-b)*(s-c)}$ where $s = (a+b+c)/2$. The next C program implements this formula, where a is the longest side of the triangle.

```
// Precondition: a > 0 and b > 0 and c > 0 and  a > b and b > c
float heron(float a, float b, float c) {
  float s, squared_area;

  squared_area = 0.0f;
  if ((a + b >= c) && (b + c >= a) && (a + c >= b)) {
```

```
    s = (a + b + c) / 2.0f;
    squared_area = s*(s-a)*(s-b)*(s-c);
  }
  return sqrt(squared_area);
}
```

The first benchmark verifies that if $a \in (5.0, 10.0]$, $b \in (0.0, 5.0]$ and $c \in (0.0, 5.0]$, then $squared\_area < 10^5$. The second verifies that with the same input domains, $squared\_area > 156.25 + 10^{-5}$) [5].

Optimized Heron. Optimized_heron is a variation of heron which uses a more reliable floating point expression to compute squared_area.

```
// Precondition: a > 0 and b > 0 and c > 0 and a > b and b > c
float optimized_heron(float a, float b, float c) {
  float s, squared_area;

  squared_area = 0.0f;

  if ((a + b >= c) && (b + c >= a) && (a + c >= b)) {
    squared_area = (((a+(b+c))*(c-(a-b))*
                    (c+(a-b))*(a+(b-c)))/16.0f);
  }

  return sqrt(squared_area);
}
```

Here, one test verifies that if $a \in (5.0, 10.0]$, $b \in (0.0, 5.0]$ and $c \in (0.0, 5.0]$, then $squared\_area < 10^5$ while the second verifies that with the same input domains, $squared\_area > 156.25 + 10^{-5}$). Note that the latter benchmark has no solution.

Cubic. The solve_cubic benchmark was extracted from the Gnu Scientific Library. It seeks a set of input values that reach the first condition of the program.

```
int solve_cubic (double a, double b, double c,
                 double *x0, double *x1, double *x2) {
  double q = (a * a - 3 * b);
  double r = (2 * a * a * a - 9 * a * b + 27 * c);
  double Q = q / 9;
  double R = r / 54;
  double Q3 = Q * Q * Q;
  double R2 = R * R;
  double CR2 = 729 * r * r;
  double CQ3 = 2916 * q * q * q;
  if (R == 0 && Q == 0) {
    ...
```

Square 2. The next benchmark checks that the square product of a float cannot be equal to 2.

```
// inv_square_int_true-unreach-call.c
int f(int x) {
  float y, z;
  // assume(x >= -10 && x <= 10);
  y = x*x - 2.f;
  // assert(y != 0.f);
  z = 1.f / y;
  return 0;
}
```

As a matter of fact, there is no simple floating point value whose square equal to 2 with the standard rounding mode. Thus, this bench has no solution.

Square 4. A variation checks that the square of a float cannot be equal to 4.

```
// float_int_inv_square_false-unreach-call.c
int g(int x) {
  float y, z;
  // assume(x >= -10 && x <= 10);
  y = x*x - 4.f;
  // assert(y != 0.f);
  z = 1.f / y;
  return 0;
}
```

A solution for this problem is well known and this benchmark has solutions.

Slope. The slope function computes an approximation of the derivative of the square function.

```
  float slope(float x0, float h) {
    float x1 = x0 + h;  float x2 = x0 - h;
    float fx1 = x1*x1;  float fx2 = x2*x2;
    float res = (fx1 - fx2) / (2.0*h);
    return res;
  }
```

The benchmark checks that for `x0` $= 13$, the result is always inferior or equal to 25 for all value of $h \in [10^{-9}, 10^{-6}]$.

5.2 Results

In the tables, the variable choice column contains two columns, the strategy column (short name "strat.") and the dynamic column (short name "dyn."). The strategy column specifies the kind of strategy used to choose the variable whose domain will be split. It is a minimization or a maximization of the defined properties (noted "min" or "max" followed by the first letters of the property

name) or one of the combinations of density and absorption that we have defined. Note that we have not implemented the derivate property yet. The dynamic column takes the value "full" when a different variable is chosen at each node of the search tree or "semi" when the variable choice is postponed until the curent variable is fully instantiated.

Column "split" gives the number of generated values and subdomains. Thus, 2 stands for Fig. 3a, that is to say, a classical bisection, 3 stands for Fig. 3b, 5 for Fig. 3c and 6 for Fig. 3d. Column $\sum t$ gives the total amount of milliseconds required to solve all the benchmarks, or all the benchmarks with or without solutions, according to the selected strategies. When available, the "#OUT" column gives the number of timeout and memory out. Note that the timeout is 180 s and that each memory out is accounted as a time out.

Table 1 gathers three subtables that give the total amount of time required to solve all the benchmarks (Table 1a), all the benchmarks with solutions (Table 1b) and all the benchmarks without solution (Table 1c) according to a given combination of variable choice and splitting strategy. Note that these tables reports only the ten best cases and the ten worst cases among the 144 combinations of variable choice and splitting strategies tested, as well as the time required to solve the related set of benchmarks using the reference strategy.

Tables 2, 3 and 4 give the total amount of time to solve all benchmarks, all benchmarks with solutions, and all benchmarks without solution according to one of the criteria introduced in our search strategies, i.e., respectively, the variable choice strategy, the nature of the variable choice (semi- or fully-dynamic)

Table 1. Total time to solve benchmarks according to variable choice and splitting

variable choice		split.	$\sum$ t
strat.	dyn.		(ms)
maxAbs	semi	6	4883
maxAbs	full	6	4930
maxDens	semi	6	5059
densWAbs	full	6	7517
maxCard	semi	6	180191
densWAbs	semi	6	180194
maxDegree	full	6	180307
maxDegree	semi	6	180310
maxAbs	full	5	184613
maxDens	semi	5	184796
...			
ref			550988
...			
minDegree	semi	3	906285
minOcc	semi	3	906285
maxWidth	semi	3	906607
minCard	semi	3	911526
maxMagn	semi	3	1077852
absWDens	full	3	1080002
maxWidth	semi	2	1080004
minDens	semi	3	1080005
absWDens	full	5	1080147
absWDens	full	5	1440000

(a) all

variable choice		split.	$\sum$ t
strat.	dyn.		(ms)
maxAbs	semi	6	187
maxCard	semi	6	189
densWAbs	semi	6	191
densWAbs	full	6	196
maxAbs	full	6	202
maxDens	semi	6	217
maxDegree	full	6	305
maxDegree	semi	6	307
maxWidth	full	6	31244
minDens	full	6	38332
...			
ref			540011
...			
minDens	semi	3	720005
minDegree	semi	2	720005
minDegree	full	2	720005
minAbs	full	3	720005
maxDens	full	3	720006
minOcc	semi	2	720006
minAbs	full	2	720006
absWDens	full	5	720147
maxWidth	semi	2	900002
absWDens	full	5	1080000

(b) with solutions

variable choice		split.	$\sum$ t
strat.	dyn.		(ms)
maxAbs	semi	2	2376
maxAbs	full	2	2379
maxAbs	full	3	2410
maxDens	semi	2	2439
maxCard	full	3	4405
maxAbs	semi	5	4451
maxAbs	full	5	4467
maxCard	full	2	4594
maxDens	semi	5	4626
maxAbs	semi	6	4696
...			
ref			10977
...			
maxMagn	semi	3	360000
minMagn	semi	3	360000
maxDegree	semi	3	360000
minDegree	semi	3	360000
minOcc	semi	3	360000
absWDens	semi	2	360000
absWDens	semi	3	360000
absWDens	semi	5	360000
absWDens	semi	6	360000
densWAbs	semi	3	360000

(c) without solution

Table 2. Total time to solve benchmarks according to variable choice strategy

Variable choice	All		With solution		Without solution	
Strat.	$\sum$ t (ms)	#OUT	$\sum$ t (ms)	#OUT	$\sum$ t (ms)	#OUT
maxWidth	4330019	21	2962680	14	1367339	7
minWidth	3762938	19	3470297	18	292641	1
maxCard	3231581	16	1962573	9	1269008	7
minCard	4315427	25	4023103	24	292324	1
maxDens	2573614	13	2323093	12	250521	1
minDens	4936905	27	3316894	18	1620011	9
maxMagn	4881081	24	3261049	15	1620011	9
minMagn	3722681	19	2761916	14	960765	5
maxDegree	3413676	17	1793656	8	1620020	9
minDegree	5259904	27	3639886	18	1620018	9
maxOcc	3360986	17	3071415	16	289571	1
minOcc	5259433	27	3639415	18	1620018	9
maxAbs	2728996	15	2521099	14	207897	1
minAbs	3784212	20	3492984	19	291228	1
maxCan	3360698	17	3071986	16	288712	1
minCan	3356934	17	3068356	16	288578	1
absWDens	5065493	27	3391300	18	1674193	9
densWAbs	2344948	11	1418791	6	926157	5

Table 3. Total time to solve benchmarks according to semi or full dynamic search

Variable choice	All		With solution		Without solution	
Dyn.	$\sum$ t (ms)	#OUT	$\sum$ t (ms)	#OUT	$\sum$ t (ms)	#OUT
Semi	37278081	192	27144829	138	10133252	54
Full	33851636	173	27485876	141	6365760	32

Table 4. Total time to solve benchmarks according to splitting strategy

Split.	All		With solution		Without solution	
	$\sum$ t(ms)	#OUT	$\sum$ t(ms)	#OUT	$\sum$ t (ms)	#OUT
2	23444527	124	20325639	108	3118888	16
3	23539280	123	17177874	89	6361406	34
5	13654772	72	10155196	54	3499576	18
6	10491138	46	6971996	28	3519142	18

and the number of fragments created by splits. Thus, each line of Table 2 sum 64 cases, each line of Table 3, 576 cases and each line of Table 4 288 cases.

5.3 Analysis

As shown in Table 1a, the best strategy outperforms the standard strategy by a factor of more than 110. These performances are even better for problems with solution (see Table 1b) where the gain factor is of more than 2800. On the other hand, the improvement for benchmarks without solution is only 4 times. Thus, the best tested strategies can significantly improve the search of a first solution whenever such a solution exist.

Combining wisely two properties can also be helpful to select useful solutions: the `densWabs` combination improves the *density* property while selecting solution that provide an absorption phenomena.

Thanks to Table 2, we can compare the different variable choice strategies. Here, the tested combination of strategies have the overall best behavior, especially, on benchmarks with solutions.

Table 3 shows that the fully dynamic strategy brings the best results on average, though the semi dynamic strategy is slightly better on benchmarks with solutions. However, these results are somewhat unbalanced by two of the variable strategies, namely the *occurence* and *degree* strategies. Such properties are static properties whose values stay the same along the search tree. As a consequence, once a variable is chosen according to this property, it will be chosen in the next node of the search tree until it cannot be chosen anymore, i.e., when fully instantiated. Thus, these two properties, whether maximized or minimized, behave alike the semi-dynamic strategy and penalize the fully dynamic results.

Table 4 confirms that the 2 splits or bisection is a better choice for problem without solution while the 6 splits have better performances on problems with solutions.

On the whole the most successful strategies are based on the absorption property, a purely floating point property. The goal when maximizing the absorption is to generate floating point errors, that's to say values for which the control flow over the floats differs from the expected flow over the reals. This is precisely the case of the benchmarks derived from Heron's formula.

6 Conclusion

This paper introduced a set of properties to choose a variable in a search for solving constraints over the floating point numbers. These maximized or minimized properties have been used to choose a variable during the search and combined with a semi dynamic and fully dynamic choice of variable, as well as, 4 splitting strategies. Preliminary experiments have shown that some of these combinations outperforms the standard strategy by two order of magnitude for

all kind of benchmarks and three order of magnitude for benchmarks with solutions. Further works include experimenting on a broader set of benchmarks, exploring other properties and evaluating which combination of properties could benefit to the search.

References

1. Alefeld, G.E., Potra, F.A., Shen, Z.: On the existence theorems of Kantorovich, Moore and Miranda. In: Alefeld, G., Chen, X. (eds.) Topics in Numerical Analysis: With Special Emphasis on Nonlinear Problems, vol. 15, pp. 21–28. Springer, Vienna (2001). doi:10.1007/978-3-7091-6217-0_3
2. Belaid, M.S., Michel, C., Rueher, M.: Boosting local consistency algorithms over floating-point numbers. In: Milano, M. (ed.) CP 2012. LNCS, pp. 127–140. Springer, Heidelberg (2012). doi:10.1007/978-3-642-33558-7_12
3. Benz, F., Hildebrandt, A., Hack, S.: A dynamic program analysis to find floating-point accuracy problems. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2012, Beijing, China, 11–16 June 2012, pp. 453–462 (2012)
4. Boussemart, F., Hemery, F., Lecoutre, C., Sais, L.: Boosting systematic search by weighting constraints. In: ECAI 2004, pp. 146–150 (2004)
5. Collavizza, H., Michel, C., Rueher, M.: Searching critical values for floating-point programs. In: Wotawa, F., Nica, M., Kushik, N. (eds.) ICTSS 2016. LNCS, vol. 9976, pp. 209–217. Springer, Cham (2016). doi:10.1007/978-3-319-47443-4_13
6. Collavizza, H., Rueher, M., Van Hentenryck, P.: CPBPV: A constraint-programming framework for bounded program verification. Constraints **15**(2), 238–264 (2010)
7. Collavizza, H., Le Vinh, N., Rueher, M., Devulder, S., Gueguen, T.: A dynamic constraint-based BMC strategy for generating counterexamples. In: 26th ACM Symposium On Applied Computing (2011)
8. Gay, S., Hartert, R., Lecoutre, C., Schaus, P.: Conflict ordering search for scheduling problems. In: Pesant, G. (ed.) CP 2015. LNCS, vol. 9255, pp. 140–148. Springer, Cham (2015). doi:10.1007/978-3-319-23219-5_10
9. Goldberg, D.: What every computer scientist should know about floating-point arithmetic. ACM Comput. Surv. **23**(1), 5–48 (1991)
10. IEEE: IEEE standard for binary floating-point arithmetic. ANSI/IEEE Standard, 754 (2008)
11. Jussien, N., Lhomme, O.: Dynamic domain splitting for numeric CSPs. In: ECAI, pp. 224–228 (1998)
12. Kearfott, R.B.: Some tests of generalized bisection. ACM Trans. Math. Softw. **13**(3), 197–220 (1987)
13. Lhomme, O.: Consistency techniques for numeric CSPs. In: Proceedings of 13th International Joint Conference on Artifical Intelligence, IJCAI 1993, vol. 1, pp. 232–238. Morgan Kaufmann Publishers Inc., San Francisco (1993)
14. Linderoth, J.T., Savelsbergh, M.W.P.: A computational study of search strategies for mixed integer programming. INFORMS J. Comput. **11**(2), 173–187 (1999)
15. Michel, L., Van Hentenryck, P.: Activity-based search for black-box constraint programming solvers. In: Beldiceanu, N., Jussien, N., Pinson, É. (eds.) CPAIOR 2012. LNCS, vol. 7298, pp. 228–243. Springer, Heidelberg (2012). doi:10.1007/978-3-642-29828-8_15

16. Ponsini, O., Michel, C., Rueher, M.: Verifying floating-point programs with constraint programming and abstract interpretation techniques. Autom. Softw. Eng. **23**(2), 191–217 (2016)
17. Refalo, P.: Impact-based search strategies for constraint programming. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 557–571. Springer, Heidelberg (2004). doi:10.1007/978-3-540-30201-8_41
18. Sterbenz, P.H.: Floating-Point Computation. Prentice-Hall Series in Automatic Computation. Prentice-Hall, Upper Saddle River (1973)

Author Index