




# Evaluating Branching Heuristics in Interval Constraint Propagation for Satisfiability

Calvin Huang<sup>1</sup>, Soonho Kong<sup>2</sup>, Sicun Gao<sup>3</sup>, and Damien Zufferey<sup>4</sup> 

<sup>1</sup> Scale Labs Inc., San Francisco, USA

<sup>2</sup> Toyota Research Institute, Cambridge, USA

<sup>3</sup> University of California, San Diego, USA

<sup>4</sup> Max Planck Institute for Software Systems, Kaiserslautern, Germany  
zufferey@mpi-sws.org

**Abstract.** Interval Constraint Propagation (ICP) is a powerful method for solving general nonlinear constraints over real numbers. ICP uses interval arithmetic to prune the space of potential solutions and, when the constraint propagation fails, divides the space into smaller regions and continues recursively. The original goal is to find paving boxes of all solutions to a problem. Already when the whole domain needs to be considered, branching methods do matter much. However, recent applications of ICP in decision procedures over the reals need only a single solution. Consequently, variable ordering in branching operations becomes even more important.

In this work, we compare three different branching heuristics for ICP. The first method, most commonly used, splits the problem in the dimension with the largest lower and upper bound. The two other types of branching methods try to exploit an integration of analytical/numerical properties of real functions and search-based methods. The second method, called smearing, uses gradient information of constraints to choose variables that have the highest local impact on pruning. The third method, lookahead branching, designs a measure function to compare the effect of all variables on pruning operations in the next several steps.

We evaluate the performance of our methods on over 11,000 benchmarks from various sources. While the different branching methods exhibit significant differences on larger instance, none is consistently better. This shows the need for further research on branching heuristics when ICP is used to find a unique solution rather than all solutions.

## 1 Introduction

Interval Constraint Propagation (ICP) is used to find solutions to logic formulas that contain continuous, typically nonlinear, real functions. It is used in a variety of domains such as design and verification of software controller, optimization of parameters of control laws, system biology, etc.

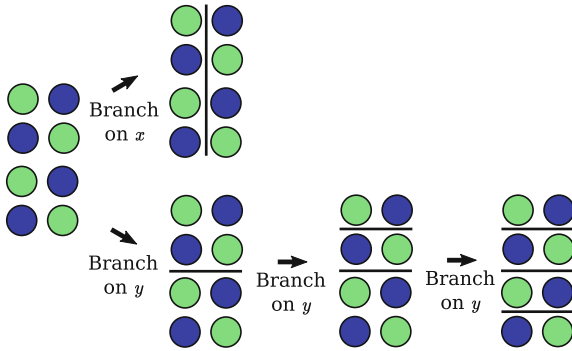
ICP finds solutions to problems of the form:

$$\exists \mathbf{x} \in \mathbb{R}^n. \bigwedge_{i=1}^n l_i \leq x_i \leq u_i \wedge \bigwedge_{i=1}^m c_i(\mathbf{x})$$

where  $c_i$  are arbitrary nonlinear constraints.  $l_i$  and  $u_i$  are lower and upper bounds that limits the search space. When a solution exists, the algorithm returns *satisfiable* and it returns *unsatisfiable* when the constraints are inconsistent. Along with a satisfiable answer, ICP also returns a solution to the problem.

Typical ICP algorithms uses the branch-and-prune paradigm. Interval extensions of functions are used to prune out sets of points that are not in the solution set and branch on intervals when such pruning cannot be done. This process repeats recursively until a small enough box that may contain a solution is found or inconsistency is observed.

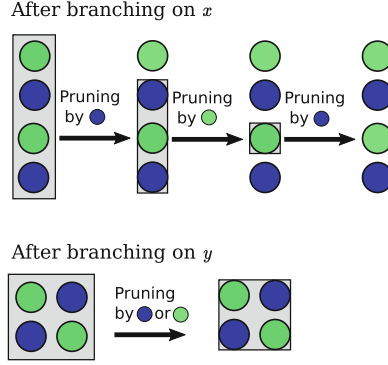
Unfortunately, ICP is exponential in the number of dimension of the solution space and, thus, scales poorly. The original goal of ICP algorithms was to *pave* all solutions and, therefore, covering the entire search space. However, for many applications, finding a single solution is sufficient. In this case, the heuristics that choose the dimension on which to branch are critical for the performance of an ICP solver.



**Fig. 1.** An example with two variables  $x$  and  $y$  where branching on  $x$  leads to termination in one step, whereas branching on  $y$  does need three steps. The example has two constraints represented by (1) the blue circles and (2) the green circles. The pruning steps are shown in Fig. 2. (Color figure online)

*Example 1.* Figure 1 shows the solution set for the two constraints being the union of four circles each. The two sets of circles do not intersect and, therefore, there is no solution. The order on which the branching is performed changes the number of step performed in the ICP algorithm.

Figure 2 shows how the pruning operations for the different branches. Initially, the pruning cannot do anything as the box containing the blue circles is



**Fig. 2.** The pruning step for the example in Fig. 1. The gray square represent the domain  $D$  explored by the ICP algorithm. (Color figure online)

the same as the box containing the green ones. After branching on  $x$ , the pruning can progressively remove circles from the domain until it is empty. Let us consider the left side of the branch. Pruning with the blue circles removes the top green circle. Pruning with the remaining green circle removes the two blue circles. Finally, pruning again with the blue circles shows a contradiction. The same process repeats on the right side and the problem is shown unsatisfiable.

On the other hand, branching on  $y$  result in a situation similar to the initial state but smaller. The pruning does not make progress and two more splits on  $y$  are needed before the pruning shows the problem to be unsatisfiable.

In this paper, we compare three branching heuristics for ICP: (1) largest first [5], (2) gradient branching, a method inspired by the maximal smear heuristics [13], (3) lookahead branching [15].

Largest first selects the variable with the largest domain. The gradient branching uses the Jacobian of the constraints and branches on the variable maximizing the magnitude of the gradient. The lookahead branching performs exhaustive splitting over each of the variables, and keeps the branch which results in the largest amount of progress during the subsequent pruning. Both methods introduce parameters that can be fine-tuned based on characteristics of different benchmark sets.

More complex variations of the heuristics have been proposed, e.g., doing random lookahead at the beginning, scoring each variables, and combining the score with another heuristic like maximal smear [16]. However, to better isolate the effect of each heuristics we use simple version of each heuristic and do not combine them.

*Contributions.* In this paper, we study branching heuristics for ICP. More specifically:

- We show the importance of branching heuristics for constraints solving over a *continuous* domain for satisfiability. Branching heuristics have been

intensively studied in solving constraints over discrete domains and over continuous domain for paving. Our experimental evaluations show that different heuristics can lead to very different running times on some instances.

- An implementation of our heuristics in the dReal SMT solver and an experimental evaluation on more than 11,000 benchmarks from varied sources. Surprisingly, all the methods have similar overall performances.

The paper is organized as follow. First, we recall the basics of the ICP algorithm. Then we explain the branching heuristics. Finally, we present an experimental evaluation.

## 2 Preliminaries

We consider first-order formulas interpreted over the real numbers. Our special focus is formulas that can contain arbitrary nonlinear functions that are *Type 2 computable* [20]. Intuitively, Type 2 computability corresponds to *numerical computability*.

We look at  $\delta$ -decision procedures [9] that are based on Interval Constraint Propagation (ICP). The framework of  $\delta$ -decision procedures formulates a relaxed notion of logical decisions, by allowing one-sided  $\delta$ -bounded errors [7]. Instead of asking whether a formula has a satisfiable assignment or not, we ask if it is “ $\delta$ -satisfiable” or “unsatisfiable”. Here, a formula is  $\delta$ -satisfiable if it would be satisfiable under some  $\delta$ -perturbation on the original formula. On the other hand, when the algorithm determines that the formula is “unsatisfiable”, it is a definite answer and no numerical error can be involved.

ICP [3] finds solutions to a system of real constraints using the branch-and-prune method, combining interval arithmetic and constraint propagation. The idea is to use interval extensions of functions to prune out sets of points that are not in the solution set, and recursively branch on intervals when such pruning can not be done, until a small enough box that may contain a solution is found or inconsistency is observed. A high-level description of the decision version of ICP is given in Algorithm 1. The boxes, or interval domains, are written as  $\mathbf{D}$  and  $c_i$  denotes the  $i$ th constraint.

The core ICP algorithm uses a branch-and-prune loop that aims to either find a small enough box that witnesses  $\delta$ -satisfiability, or detect that no solution exists. The loop consists of two main steps: pruning and branching.

**Prune( $\mathbf{D}, c_i$ )** (line 4–6) The pruning operator removes from the current box  $\mathbf{D}$  parts of the state space that does not contain solutions to the constrain  $c_i$ . It uses interval arithmetic to maintain an overapproximation of the solution sets.

**Branch( $\mathbf{D}, i$ )** (line 9) When the pruning operation ceases to make progress, one performs a depth-first search by branching on variables and restarts pruning operations on a subset of the domain. Typically, branching selects the variable with the largest domain, e.g.,  $D_i$ , and partition it into two equally large subintervals. More concretely, line 9 selects  $i = \arg \max_{\{i. |D_i| > \epsilon\}} |D_i|$ .

**Algorithm 1.**  $\text{ICP}(c_1, \dots, c_m, \mathbf{D} = D_1 \times \dots \times D_n, \delta)$ 


---

```

1:  $S.\text{push}(\mathbf{D})$ 
2: while  $S \neq \emptyset$  do
3:    $\mathbf{D} \leftarrow S.\text{pop}()$ 
4:   while  $\exists 1 \leq i \leq m, \mathbf{D} \neq_\delta \text{Prune}(\mathbf{D}, c_i)$  do
5:      $\mathbf{D} \leftarrow \text{Prune}(\mathbf{D}, c_i)$ 
6:   end while
7:   if  $\mathbf{D} \neq \emptyset$  then
8:     if  $\exists 1 \leq i \leq n, |D_i| \geq \varepsilon$  then  $\triangleright \varepsilon$  is some computable factor of  $\delta$ 
9:        $\{\mathbf{D}_1, \mathbf{D}_2\} \leftarrow \text{Branch}(\mathbf{D}, i)$ 
10:       $S.\text{push}(\mathbf{D}_1)$ 
11:       $S.\text{push}(\mathbf{D}_2)$ 
12:    else
13:      return sat
14:    end if
15:  end if
16: end while
17: return unsat

```

---

**Notation 1.** To simplify the presentation, throughout this paper we assume that  $0/0 = 0$ .

**Notation 2.** In the rest of this paper, we denote the pruning fixed-point, i.e., the line 4–6 of Algorithm 1, by  $\text{Prune}(\mathbf{D})$ .

### 3 Gradient Branching

Branching on the size of the interval does not exploit information specific about the constraints in the problems. Instead, we can use a branching heuristic based on the gradient that exploits analytical/numerical properties of the constraints. One method uses gradient information of constraints to choose variables that have the highest local impact on pruning. The method we present is a variation of the maximal smearing method [13]. On top of this method, we add weight to different elements (gradient, size of the intervals). Also, we directly work in the log-space and sum rather than multiply the coefficients.

The algorithm is shown in Algorithm 2. The final score is a combination of two factors: (1) the size of the interval and (2) the gradient. The size of the interval (line 2) is a term similar to the standard heuristic that picks the larger interval for branching. The importance of that factor is controlled by the parameter  $p_3$ , usually a small value. The main factor is the gradient at the center of the current box (line 7). The midpoint method returns the point at the center of the box  $\mathbf{D}$ . The gradient is obtained using the Jacobian of the constraints.  $\text{Jacobian}(c_j)(p)$  computes the Jacobian of  $c_j$  at the point  $p$  and return a vector  $g$  of size  $n$  where  $g_i$  is the derivative value of the  $i$ th variable.

Algorithm 2 integrates in Algorithm 1 at line 4. The algorithm depends on three parameters  $p_1$ ,  $p_2$ , and  $p_3$ . We set the parameters to 1000, 1000, and 0.01

respectively. We found these values empirically by exploring a range of values and keeping the ones performing the best on a small set of examples.

*Remark 1.* Since the ranges of the variables can be vastly different from one another, we do not want to give disproportionate weight to the variables where the range is much larger, so we use the `asinh` function to score the range of a variable. We choose `asinh` rather than `log` because it has better behavior for small values.

---

**Algorithm 2.**  $\text{Gradient}(c_1, \dots, c_m, \mathbf{D} = D_1 \times \dots \times D_n)$

---

```

1:  $p \leftarrow \text{midpoint}(\mathbf{D})$ 
2: for all  $i \in [1; n]$  do  $\text{scores}[i] \leftarrow \text{asinh}(|D_i|p_1)p_3$ 
3: end for
4: for all  $j \in [1; m]$  do
5:    $g \leftarrow \text{Jacobian}(c_j)(p)$ 
6:   for all  $i \in [1; n]$  do
7:      $\text{scores}[i] \leftarrow \text{scores}[i] + \text{asinh}(\text{abs}(g_i)|D_i|p_2)$ 
8:   end for
9: end for
10: return  $\arg \max_i \text{scores}[i]$ 
```

---

## 4 Lookahead Branching

Due to the nonlinearity of the constraints, it is difficult to predict a priori the effect of branching on the subsequent pruning steps. An alternative is to use a posteriori information, i.e. try to branch on multiple variables, prune, and keep the branching which results in the most progress. This is the lookahead strategy [15].

The lookahead strategy makes locally optimal branching choices, but it is computationally expensive as the results of many pruning steps are discarded. However, a *bad* choice in branching, i.e. a branching leading to little or no pruning also has a cost on the total running time.

*Example 2.* Making a bad choice can double the amount of subsequent computations. Let us consider an unsatisfiable set of constraints  $\mathcal{C}(X)$  over the set of variables  $X$ . Furthermore, assume that the ICP algorithm takes  $T$  seconds to return `unsat`.

Now let us add a fresh variable  $y \in [-\delta; \delta]$  and let  $\mathcal{C}'(X \cup \{y\})$  be  $\mathcal{C}(X) \wedge y = y$ . In the best case, the ICP algorithm running on  $\mathcal{C}'$  does not branch on  $y$  and runs on  $\mathcal{C}'$  like it ran on  $\mathcal{C}$ . In the worst case, it first branch on  $y$ . After branching on  $y$ ,  $\mathcal{C}'$  is the same as  $\mathcal{C}$ . However, the ICP needs to show that *both* sides of the branch do not contain any solution. In the first case solving  $\mathcal{C}'$  takes  $T$  seconds and in the second case it takes  $2T$  seconds. The more the ICP branches on  $y$ , the more the running time increases.

We present a modified ICP algorithm which uses lookahead. The main components are (1) a way of measuring the progress made during pruning, (2) a lookahead algorithm which preserves some of the progress made during the pruning, and (3) a modified ICP loop which can mix lookahead and normal branching decisions.

Using only lookahead is prohibitively expensive on problems with a large number of variables. Therefore, our algorithm has two important features. First, the lookahead procedure preserves some information about the results of the discarded pruning steps. Second, the modified ICP loop can mix lookahead with another cheaper branching policy.

*Measuring Progress.* To evaluate the progress of pruning steps we need a way of quantifying the progress made by pruning steps, i.e., measure the  $\mathbf{D}$ . The most natural measure of a box is its volume. However, the volume is not resistant to degenerate intervals, e.g.,  $[0; 0]$ . The volume of any box containing a degenerate interval is 0. Instead we use the *linear dimension* of the box to measure progress. The linear dimension is defined as the sum of the interval sizes:

$$\text{ld}(\mathbf{D}) = \sum_i |D_i|$$

*Computing the Scores of Variables.* To decide on which variable to branch, we give a score to each variable and pick the variable with the maximal score. A variable's score is computed by branching on that variable, pruning the two resulting boxes, and measuring the progress made during the pruning. Furthermore, the variables scores can be amortized in a decaying sum. Since we do not perform the lookahead at every step, this helps identifying the variables which are globally more important from the local variations. The decaying sum is inspired by the variable state independent decaying sum [14] which is a very successful heuristic for SAT solvers. Algorithm 3 shows the details of the computation.

A crucial optimization of the lookahead procedure is the hull on line 9. Instead of repeatedly branching only on the initial  $\mathbf{D}$ , we take the hull, i.e., smallest box containing  $\mathbf{D}_1$  and  $\mathbf{D}_2$ . This preserves some of the work done during the pruning.

The algorithm depends on the coefficient  $p_1$  that amortises the scoring over the entire run of the ICP algorithm. In our experiment, we set  $p_1$  to 0.5.

*Modified ICP Algorithm.* Algorithm 4 is a modified version of Algorithm 1 to track additional information about the search. The stack  $S$  does not only contains boxes but also keeps track of the depth of the box in the search tree built by the algorithm. Two additional variables  $l$  and  $dl$  keep track of how long ago the last lookahead performed was, and at what depth. This extra information is used to decide when to use lookahead. Between lookahead steps, the same variable is split repeatedly or if its interval is too small, the algorithm can revert to another policy (line 15).

Algorithm 4 depends on three parameters:  $p_1$ ,  $p_2$ , and  $p_3$ . These parameters control how often the lookahead step is performed on line 10 of Algorithm 4.

**Algorithm 3.** Lookahead( $c_1, \dots, c_m, \mathbf{D} = D_1 \times \dots \times D_n$ )

---

```

1:  $scores \leftarrow \mathbf{0}$  ▷ only the first time
2:  $i \leftarrow 1$ 
3: while  $i \leq n \wedge \mathbf{D} \neq \emptyset$  do
4:   if  $|D_i| > \epsilon$  then
5:      $\{D_1, D_2\} \leftarrow \text{Branch}(\mathbf{D}, i)$ 
6:      $D'_1 \leftarrow \text{Prune}(D_1)$ 
7:      $D'_2 \leftarrow \text{Prune}(D_2)$ 
8:      $score[i] \leftarrow score[i] p_1 + (1 - p_1) \left( \frac{\text{ld}(D_1)}{\text{ld}(D'_1)} + \frac{\text{ld}(D_2)}{\text{ld}(D'_2)} \right)$ 
9:      $\mathbf{D} \leftarrow \text{hull}(D'_1, D'_2)$ 
10:   end if
11:    $i \leftarrow i + 1$ 
12: end while
13: return  $(\mathbf{D}, \arg \max_i score[i])$ 

```

---

**Algorithm 4.** ICP2( $c_1, \dots, c_m, \mathbf{D} = D_1 \times \dots \times D_n, \delta$ )

---

```

1:  $l \leftarrow -p_1$  ▷ how long since the last lookahead
2:  $dl \leftarrow 0$  ▷ depth from the last lookahead
3:  $i \leftarrow 0$  ▷ branching index
4:  $S.\text{push}(0, \mathbf{D})$ 
5: while  $S \neq \emptyset$  do
6:    $(d, \mathbf{D}) \leftarrow S.\text{pop}()$ 
7:    $\mathbf{D} \leftarrow \text{Prune}(\mathbf{D})$ 
8:    $l \leftarrow l + 1$ 
9:   if  $\mathbf{D} \neq \emptyset$  then
10:    if  $l < 0 \vee l > p_2 \vee d \leq dl - p_3$  then
11:       $(\mathbf{D}, i) \leftarrow \text{Lookahead}(c_1, \dots, c_m, \mathbf{D})$ 
12:       $l \leftarrow \min(l, 0)$ 
13:       $dl \leftarrow d$ 
14:    end if
15:    if  $|D_i| \geq \epsilon \vee \exists 1 \leq i \leq n, |D_i| \geq \epsilon$  then
16:       $\{D_1, D_2\} \leftarrow \text{Branch}(\mathbf{D}, i)$ 
17:       $S.\text{push}(d + 1, D_1)$ 
18:       $S.\text{push}(d + 1, D_2)$ 
19:    else
20:      return sat
21:    end if
22:  end if
23: end while
24: return unsat

```

---

The lookahead is performed at a fixed frequency, except at the start where it is performed more often, and after backtracking.

- $l < 0$  and  $l \leftarrow -p_1$  (line 1) forces the lookahead for the first  $p_1$  iteration of the algorithm. Due to the DFS nature of the ICP algorithm, making good choices



at the early stage of the search is very important. Therefore, we perform more lookahead steps at the beginning.

- $l > p_2$  controls the maximal interval between two lookahead steps.
- $d \leq dl - p_3$  forces lookahead if the depth  $d$  is  $p_3$  less than the depth  $dl$  at which that the lookahead was last computed, i.e. if the search backtracks more than  $p_3$  steps in the search tree. After backtracking, the search may continue in a very different region of the solution space. Therefore, we need to update the score of the variables to make sure it is still relevant to the region currently being explored.

In our experiments we set  $p_1$ ,  $p_2$ , and  $p_3$  to 10.

## 5 Evaluation

We implemented the heuristics presented above in the dReal SMT solver [8] and evaluated their efficiency. dReal already implements the baseline heuristic to bisect the largest dimension by default. Our benchmarks come from:

- the dReal test suite;
- information theory: search for probabilistic encoding minimizing power consumption [19];
- control theory: certifying Lyapunov functions;
- automated theorem proving: proof obligations extracted from the Flyspeck project [9, 11];
- Geometric problems: intersection of objects in high-dimensional spaces, and computing the inverse kinematics for robotic planning;
- The QF\_NRA examples from SMT-LIB [1].

The benchmarks are available at <https://github.com/dreal/benchmarks>.

In this work we focus on branching and our examples do not contain any ordinary differential equations. Examples with ODEs are less interesting as solving the differential part dominates the running time.

We run two experiments. In the first experiment, we run dReal on the whole set of 11789 benchmarks with a 300s timeout. For the second experiment, we keep only the benchmarks with more than 7 variables to filter out the smaller, easier examples. Furthermore, we exclude the `hycomp` benchmarks from the SMT-LIB set because, even though they have many variables, they are solved using pruning only. The second set has 896 benchmarks and we run dReal with a 1800s timeout. We run our experiments in parallel on a server with AMD Opteron 6174 CPUs at 2.2 GHz and 500 GB of RAM. dReal is single threaded and memory consumption is not an issue. Additionally, we run dReal with `--polytope` and `--stat` to use the polytope contractor and to collect statistics of the runs.

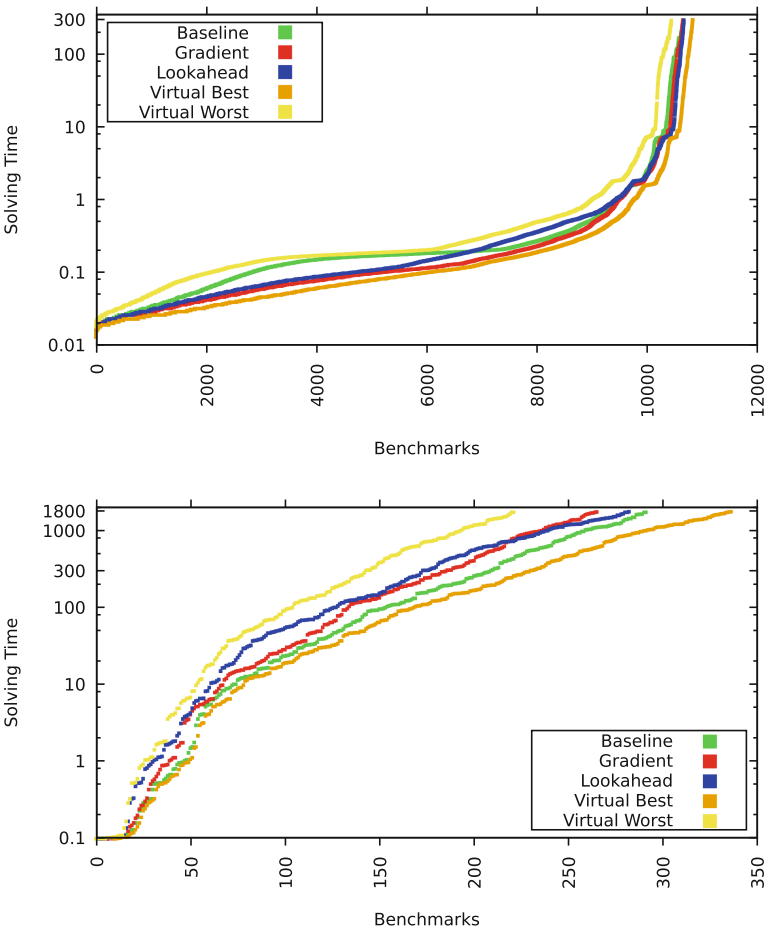
**Table 1.** Summary of the experiments. We show the number of instances solved by each method. The virtual best/worst solver represent the best/worst results of any heuristic on each benchmarks. We also show how many benchmarks could only be solved using a single method.

Instances	Small ( $\Delta$ )	Large ( $\Delta$ )
#Benchmarks	11789	896
Solved Baseline	10654	<b>292</b>
Solved Gradient	10654 (+0)	266 (−26)
Solved Lookahead	<b>10667</b> (+13)	283 (−9)
Virtual Best	10827 (+173)	337 (+45)
Virtual Worst	10439 (−206)	222 (−70)
Unique Baseline	34	19
Unique Gradient	<b>65</b>	5
Unique Lookahead	19	<b>31</b>

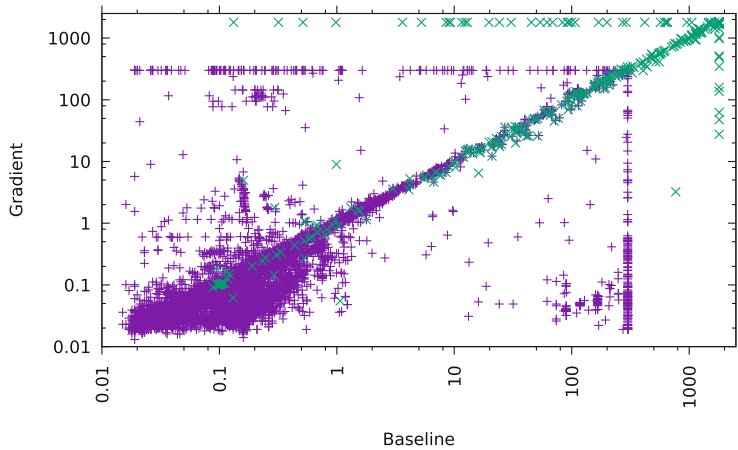
The results are shown in Table 1 and Figs. 3, 4, 5, 6 and 7. The table includes the number of instances solved by each method. We also include the virtual best and virtual worst solver for the three methods. In the total number of instances solved, the three branching heuristics give roughly similar results. The lookahead is better overall and, surprisingly, the baseline performs better on the large instances. However, there is a large variation from benchmark to benchmark. This can be seen with the virtual best/worst solver, the unique instances, i.e., the instances solved by only one method, and this is also visible on the Figures.

Figures 4 and 5 compare the gradient and lookahead against the baseline heuristic. There is a large cluster of easy instances in the bottom left corner. However, the most interesting feature of these graphs are the number of points which are on the timeout lines. Also, we can observe that the gradient is faster than the baseline on the easy instances and the lookahead is slower on average but times out less often.

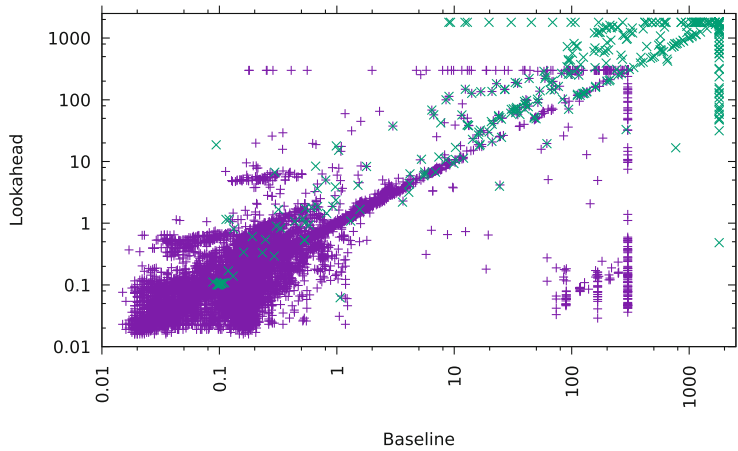
Figures 6 and 7 show more information about the internals of the search. Each solved problem is shown by a  $(x, y)$  dot where  $y$  is the number of dimensions in the problem and  $x$  is the number of pruning or branching steps performed by the solver. The gradient and baseline are roughly comparable with an average of around 9,000 branches and 1,400,000 pruning steps. However, the lookahead search is quite different with around 5,300 branches and 3,072,000 pruning steps, trading off branching for pruning. It is easy to see that the lookahead search performs much more pruning. On the other hand, it branches less often. The branching difference is less visible, but the cost of additional branches is more important, see Example 2.



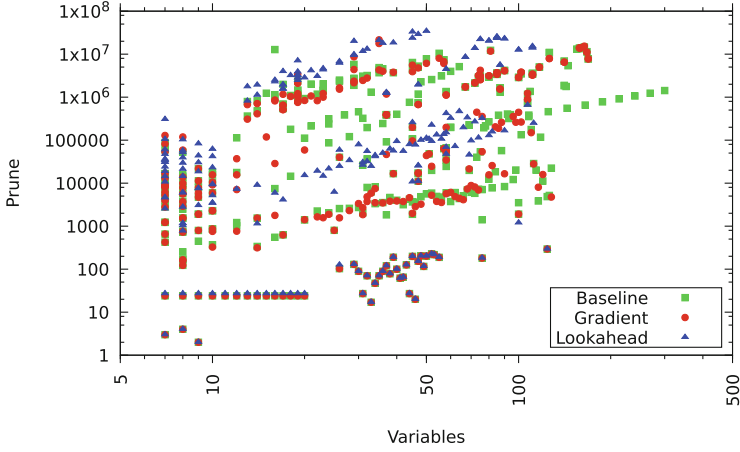
**Fig. 3.** Cactus plot of the running times of the baseline, gradient, and lookahead heuristics. A  $(x, y)$  point means  $x$  instances are solved with a  $y$ -second timeout. The higher graph shows all the benchmarks with a 300 s. timeout. The lower graph shows the large instances with a 1800 s. timeout.



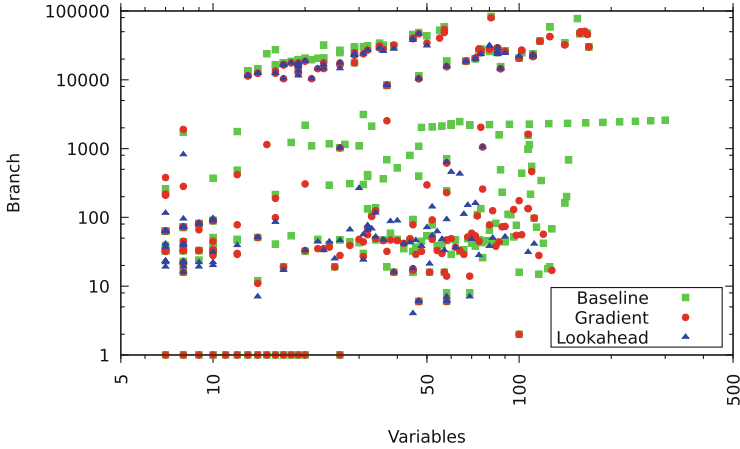
**Fig. 4.** Scatter plot of the running times of the gradient heuristic against the baseline. The + are the tests with a 300 s. timeout and the  $\times$  are the tests with a 1800 s. timeout. The gradient is faster for smaller instances. Notice the large number of instance that can be solved easily by one method but not the other.



**Fig. 5.** Scatter plot of the running times of the lookahead heuristic against the baseline. The + are the tests with a 300 s. timeout and the  $\times$  are the tests with a 1800 s. timeout. The lookahead solves more instances, but is slower overall.



**Fig. 6.** The number of pruning steps against the number of variables. The lookahead performs more pruning and the gradient and baseline are comparable.



**Fig. 7.** The number of branching steps against the number of variables. The lookahead branches less often than the other two heuristics.

## 6 Related Work and Discussion

Branching is extensively studied in the discrete CSP community [17] and many techniques developed in this setting are applicable to an ICP algorithm for continuous domain. For instance, iSAT [6] implements an integrated ICP+SAT algorithm and uses the SAT solver heuristics for branching. In this work, we try to exploit some information specific to the real numbers, such as the gradient, to improve the variable selection for branching. In the future, we plan to evaluate

more complex scoring mechanism for the variables as has been done in the SAT community [4].

Gradient descent methods are common for convex optimisation [18]. Our problems are nonlinear and, therefore, nonconvex. However, we plan to further investigate how technique related to convex relaxation can be integrated in ICP algorithms.

The difference between the different heuristics makes this a prime candidate for combination of heuristics. We could try combination methods using weights [16], alternating between different selection heuristics [10], or portfolio approach [12]. This would allow us to match the results of the virtual best solver. A portfolio approach has another advantage: exploiting parallelism. This also ask questions about identifying when/how to exchange information between solvers. The difficulty stems from the fact that while the solver works with convex objects, i.e. boxes. The part of the search space removed by pruning operations is not convex.

A new direction, we have not yet optimized which branch of the search is explored first after a bisection. Paving always need to explore both sides. However, a satisfiability check may stop after finding a solution on the first side explored.

Another avenue which could be combined to improve the branching is to ask the pruning step to return more information which can be used for decide at which point within an interval the splitting must occur [2].

## 7 Conclusion

In this paper, we show the importance of branching in ICP algorithms for satisfiability of non-linear constraints over the reals. We compare three branching heuristics. One method picks the variable with the largest range and does not use information about the constraints. One using gradient information of constraints and the other method use lookahead to estimate the effect of variables on pruning operations in the next several steps. We have implemented these heuristics in the dReal SMT solver and present an evaluation with over 11,000 problem instances. The different branching methods exhibit significant differences on larger instance. However, no heuristics is consistently better. This shows the need for further research on branching heuristics when using ICP for satisfiability.

**Acknowledgements.** We thank the reviewers for their feedback and comments which helped us improve the paper. Calvin Huang was studying at MIT when that work was done. Damien Zufferey was funded in part by the DFG under Grant Agreement 389792660-TRR 248 and by the ERC under the Grant Agreement 610150.

## References

1. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB) (2019). [www.smtlib.org/](http://www.smtlib.org/)
2. Batnini, H., Michel, C., Rueher, M.: Mind the gaps: a new splitting strategy for consistency techniques. In: van Beek, P. (ed.) CP 2005. LNCS, vol. 3709, pp. 77–91. Springer, Heidelberg (2005). [https://doi.org/10.1007/11564751\\_9](https://doi.org/10.1007/11564751_9)
3. Benhamou, F., Granvilliers, L.: Continuous and interval constraints. In: Rossi, F., van Beek, P., Walsh, T. (eds.) Handbook of Constraint Programming, Chap. 16. Elsevier (2006)
4. Biere, A., Fröhlich, A.: Evaluating CDCL variable scoring schemes. In: Heule, M., Weaver, S. (eds.) SAT 2015. LNCS, vol. 9340, pp. 405–422. Springer, Cham (2015). [https://doi.org/10.1007/978-3-319-24318-4\\_29](https://doi.org/10.1007/978-3-319-24318-4_29)
5. Csendes, T., Ratz, D.: Subdivision direction selection in interval methods for global optimization. SIAM J. Numer. Anal. **34**(3), 922–938 (1997). <https://doi.org/10.1137/S0036142995281528>
6. Fränzle, M., Herde, C., Teige, T., Ratschan, S., Schubert, T.: Efficient solving of large non-linear arithmetic constraint systems with complex Boolean structure. JSAT **1**(3–4), 209–236 (2007)
7. Gao, S., Avigad, J., Clarke, E.M.:  $\delta$ -complete decision procedures for satisfiability over the reals. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS (LNAI), vol. 7364, pp. 286–300. Springer, Heidelberg (2012). [https://doi.org/10.1007/978-3-642-31365-3\\_23](https://doi.org/10.1007/978-3-642-31365-3_23)
8. Gao, S., Kong, S., Clarke, E.M.: dReal: an SMT solver for nonlinear theories over the reals. In: Bonacina, M.P. (ed.) CADE 2013. LNCS (LNAI), vol. 7898, pp. 208–214. Springer, Heidelberg (2013). [https://doi.org/10.1007/978-3-642-38574-2\\_14](https://doi.org/10.1007/978-3-642-38574-2_14)
9. Gao, S., Kong, S., Clarke, E.M.: Proof generation from delta-decisions. In: Winkler, F., et al. (eds.) 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2014, Timisoara, Romania, 22–25 September 2014, pp. 156–163. IEEE Computer Society (2014). <https://doi.org/10.1109/SYNASC.2014.29>
10. Granvilliers, L.: Adaptive bisection of numerical CSPs. In: Milano, M. (ed.) CP 2012. LNCS, pp. 290–298. Springer, Heidelberg (2012). [https://doi.org/10.1007/978-3-642-33558-7\\_23](https://doi.org/10.1007/978-3-642-33558-7_23)
11. Hales, T.C., et al.: A formal proof of the Kepler conjecture. Forum Math. Pi **5**, e2 (2017). <https://doi.org/10.1017/fmp.2017.1>
12. Hamadi, Y., Ringwelski, G.: Boosting distributed constraint satisfaction. J. Heuristics **17**(3), 251–279 (2011). <https://doi.org/10.1007/s10732-010-9134-2>
13. Kearfott, R.B., Novoa, M.: Algorithm 681: INTBIS, a portable interval Newton/bisection package. ACM Trans. Math. Softw. **16**(2), 152–157 (1990). <https://doi.org/10.1145/78928.78931>
14. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: Design Automation Conference, pp. 530–535 (2001). <https://doi.org/10.1145/378239.379017>
15. Purdom, P.W., Brown, C.A., Robertson, E.L.: Backtracking with multi-level dynamic search rearrangement. Acta Inf. **15**(2), 99–113 (1981). <https://doi.org/10.1007/BF00288958>
16. Reyes, V., Araya, I.: Probing-based variable selection heuristics for NCSPs. In: 2014 IEEE 26th International Conference on Tools with Artificial Intelligence, pp. 16–23, November 2014. <https://doi.org/10.1109/ICTAI.2014.14>

17. Rossi, F., van Beek, P., Walsh, T. (eds.): Handbook of Constraint Programming. Elsevier, New York (2006)
18. Snyman, J.: Practical Mathematical Optimization: An Introduction to Basic Optimization Theory and Classical and New Gradient-Based Algorithms. Springer, New York (2005). <https://doi.org/10.1007/b105200>
19. Stanley-Marbell, P., Francese, P.A., Rinard, M.: Encoder logic for reducing serial I/O power in sensors and sensor hubs. In: 2016 IEEE Hot Chips 28 Symposium (HCS), Cupertino, CA, USA, 21–23 August 2016, pp. 1–2. IEEE (2016). <https://doi.org/10.1109/HOTCHIPS.2016.7936231>
20. Weihrauch, K.: Computable Analysis: An Introduction. Texts in Theoretical Computer Science. Springer, Heidelberg (2000). <https://doi.org/10.1007/978-3-642-56999-9>