Applying Machine Learning to Heuristics for Real Polynomial Constraint Solving

Christopher W. Brown(✉) and Glenn Christopher Daves

United States Naval Academy, Annapolis, USA
{wcbrown,m201362}@usna.edu

Abstract. This paper considers the application of machine learning to automatically generating heuristics for real polynomial constraint solvers. We consider a specific choice-point in the algorithm for constructing an open Non-uniform Cylindrical Algebraic Decomposition (NuCAD) for a conjunction of constraints, and we learn a heuristic for making that choice. Experiments demonstrate the effectiveness of the learned heuristic. We hope that the approach we take to learning this heuristic, which is not a natural fit to machine learning, can be applied effectively to other choices in constraint solving algorithms.

Keywords: Non-linear polynomial constraints · Machine learning

1 Introduction

In [2] the first author proposed Non-uniform Cylindrical Algebraic Decomposition (NuCAD) as an alternative to the well-known Cylindrical Algebraic Decomposition (CAD) as a data-structure for representing sets of points in Euclidean space defined by boolean combinations of real polynomial equalities and inequalities. The process of constructing a NuCAD involves many points at which an arbitrary choice needs to be made—a choice that does not affect correctness, but can have a considerable impact on running time, memory usage, and quality of solution. The purpose of this work is to consider one such choice, and attempt to use machine learning to automatically learn a successful heuristic. This paper will introduce the problem, explain why the application of machine learning to the problem is exceptionally challenging, describe the process we developed to handle not just this heuristic-learning problem, but others with similar challenges, and report experimental results.

2 The Problem

A *semi-algebraic set* is a set of points in Euclidean space defined by a boolean combination of polynomial equalities and inequalities (known as a *Tarski formula*). Non-uniform Cylindrical Algebraic Decomposition (NuCAD) is a data structure providing an explicit representation of semi-algebraic sets. From this

This is a U.S. government work and not under copyright protection in the U.S.;
foreign copyright protection may apply 2020
A. M. Bigatti et al. (Eds.): ICMS 2020, LNCS 12097, pp. 292–301, 2020.
https://doi.org/10.1007/978-3-030-52200-1_29

data structure, a variety of questions can be answered. In this article we only address the question of whether the semi-algebraic set is non-empty or, equivalently, whether the associated Tarski formula is satisfiable. For example, to determine the satisfiability of the Tarski formula $[x^2+y^2 < 1 \wedge y > x^2 \wedge 3x > 2y^2+1]$, we would construct a NuCAD data structure representing the decomposition of $\mathbb{R}^2$ depicted in Fig. 1.

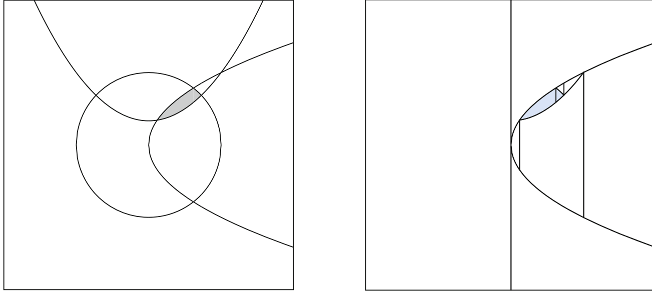

Fig. 1. Depicted on the left is the semi-algebraic set defined by the Tarski formula $[x^2 + y^2 < 1 \wedge y > x^2 \wedge 3x > 2y^2 + 1]$ along with the curves defined by the three polynomials in the formula. On the right is depicted the decomposition described by a NuCAD data structure produced from the formula. This is not unique, as different choices during the construction lead to different NuCADs.

The NuCAD data structure contains a sample point for each cell in the decomposition it represents. In this case, two cells have sample points at which the input formula is satisfied. Thus, not only do we learn that the formula is satisfiable, but we also have witness points to prove it.

NuCADs are constructed by a simple refinement process. At each step a cell is selected, and from amongst the constraints that are violated at its sample point, one is chosen to be the basis for splitting the selected cell into subcells—thus refining the decomposition. The key idea is that the subcell containing the original sample point has the property that the chosen constraint is violated throughout the subcell. To illustrate, suppose that in the previous example our initial sample point was (0, 0). Two constraints (the parabolas) are violated at this point, so we must choose one. It is this choice that we concentrate on in the present paper. We would like to learn a heuristic for it. To be precise, we give the following definition of a *choice function*.

Definition 1. *A* choice function *maps a set of multivariate polynomials and a variable ordering to an element of the input polynomial set. When a cell has been selected to refine, the set of polynomials from constraints that are violated at the selected cell's sample point is given as input to a choice function, and the output of the choice function is the constraint that will be used for the next refining step. Additional input consisting of state information from the selected*

cell and the NuCAD as a whole (e.g. the polynomials defining the selected cell's boundaries) are optionally allowed.

It is important to point out that making good choices vs. bad choices can make orders of magnitude differences in computing time and number of cells in the resulting NuCAD. Of course for very easy problems, the difference is not that great. Likewise, there are combinations of problem and initial point for which, at every sample point, there is only one violated constraint. This means that there never is a real choice, and so the heuristic is not needed. However, as our experiments show, there are also problems for which a good choice heuristic vs. bad choice heuristic means the difference between problems that are solvable within a reasonable amount of time and those that are not.

3 ML and and Why Learning a Choice Function Is Hard

Machine Learning (ML) is a huge field, and one very much in the spotlight at the moment. Moreover, machine learning has been applied to related problems (see for example [5–8]), though in a significantly different ways than what we do in this paper. So we will not try to explain what machine learning is, and trust that any reader who is not sufficiently conversant in the subject will have many options for obtaining the necessary background. Machine learning typically centers around trying to learn a function that maps input feature vectors to output result vectors. The dimensions and component types of these vectors is fixed for a given learning problem. Also pertinent to this discussion is the basic categorizations of learning as "supervised" vs. "unsupervised" and "regressions" vs. "classification".

Unfortunately, learning a choice function does not fit these typical machine learning paradigms well. There is no limit on the size of the input space, since there is no bound on the number of polynomials in the input set, nor is there a bound on the number of variables, degrees, or number of terms in the individual polynomials within the set. So it does not even fit the general framework. It does not fit as a regression problem, because we don't have continuous outputs (the output being one polynomial from the set). It doesn't fit as a classification problem, because each polynomial from the input set constitutes a class label, and there is not a fixed number of polynomials in the set (classification usually assuming a fixed set of labels). It does not fit as a supervised learning problem because we have no ground truth, i.e. we do not know what the "right" decisions

are, so we do not have labeled data[1]. Unsupervised learning, at first, seems like it might be promising. Specifically, we could try to cast learning a choice function as a reinforcement learning problem, in which a decision made during the construction of a NuCAD for an input is rewarded or penalized by the success of the final result, appropriately discounted by its distance from that result. Unfortunately, there are two problems with this approach. First is that we have no basis for determining an award or penalty. If we end up with a decomposition consisting of 500 cells, is that good? Perhaps a different sequence of choices would have yielded a decomposition into 20 cells, or perhaps 500 is in fact the minimum achievable. With no way of knowing, we cannot determine an award or penalty. Second, the choices made during the construction of a NuCAD are not generally linear. The first choice, for example, produces several sibling cells, and the further refinement of each of those sibling cells is independent of the others. It is then unclear how to split up rewards.

Another substantial impediment to applying ML to the problem of producing an effective choice function is the very common problem of finding good data. In the case of polynomial constraints, we have the issue that the problem space is so vast, that it is not feasible to have anything like reasonable coverage of the space. Moreover, it is typically observed that most symbolic algorithms for dealing with polynomial constraints behave very differently on random problems than on "real-world" problems. This means that if we learn from randomly generated inputs we have to be concerned that the result might not transfer well to real-world problems or, indeed, problems that were not generated in a similar way to the training data. While we do have the SMT-LIB [1] QF_NRA library of problems as a valuable resource, we have to be careful learning from it as well. This is specifically true because most problems come from a large group generated by one of a relatively few applications. Typically all the problems within a group are very similar. This means, that if you learn from some of the problems in a given group and you evaluate on others from the same group, the learned function may test well because it has memorized correct actions for that group. This may result in overfitting that would only get exposed if you were to test on problems that were not from any of the groups that were used in

[1] In some other contexts in which machine learning has been applied to learn heuristic choice functions in symbolic computing, for example in work about learning to choose good variable orderings, the approach has been to compute the results of all possible choices, so that one does indeed know ground truth. That is impractical in our context for the following reason: in order to determine the best option for the first choice-point, it is not enough to try each of the options for just that choice; one has to make the optimum decision for each of the follow-on choices until NuCAD construction is complete. This means that one would have to try every option at every choice-point! One can compute that for the three-variable formula $\wedge_{i=1}^{3}(0 < x_i + 1/5 \wedge 0 < x_i + 2/5 \wedge 0 > x_i + 3/5 \wedge 0 > x_i + 4/5)$, which consists solely of linear univariate constraints, there are 5,760 ways that these choices can be made, each resulting in a different NuCAD data structure in the end. By comparison, for three variables there are exactly six variable orders to consider. So this exhaustive approach to finding ground truth to learn from is not feasible for our problem.

training. Finally, the current implementation of NuCAD is for “open” NuCAD only, and only for conjunctions. This means that we are restricted to problems from the SMT-LIB that are conjunctions of strict inequalities.

4 Using ML to Learn Choice Functions

The preceding examination of all the reasons why it might not appear to be promising to apply ML to the problem of learning choice functions is important to understanding why we developed the approach described below.

Reduce to Binary Classification: Trying to learn a choice function directly is problematic in part because the number of classes is not fixed, rather it is determined by the size of the input set of polynomials, which varies from choice-to-choice. In a crucial change of perspective, we redefine our problem to learning a *choice ordering predicate*, i.e. a function that takes two polynomials (and optionally extra information about the underlying cell and NuCAD context), and returns 1 if the first polynomial is a preferable choice to the second, and 0 otherwise. The choice function applied to a set of n polynomials then becomes $n - 1$ applications of the choice ordering predicate, at each step retaining the polynomial preferred by the predicate. If the choice ordering predicate fails to induce a proper total order, the order in which the predicate is applied to the polynomials in the set could affect the result. In the present experiments we accept this fact.

Reduce Inputs to Fixed Size Feature Set: Having restricted the learning problem to learning a choice ordering predicate, we have reduced the dimension of the input vector to the function to be learned, but not yet fixed it to a constant. After all, while there are now always two input polynomials, rather than an arbitrary sized set, those polynomials are still unbounded in the number of variables, degrees, number of terms and coefficient sizes. The solution to this is clear cut from a machine learning perspective: extract from the two polynomials (and optionally from the cell and NuCAD context) a fixed sized set of *features*. Although part of the revolution in deep learning is that the learning algorithm is, in some sense, supposed to do this “feature engineering” for us, in this case our domain forces us to do it. However, because of our restriction to learning choice ordering predicates, there are a number of natural *comparative features*, for example the difference in *level*[2] of the two input polynomials, or the difference in the two polynomials’ total degrees, number of terms, etc.

Reduce to Supervised Learning: The biggest hurdle to applying ML to this problem is that, as described above, it fits neither the supervised nor unsupervised paradigms. To address this, we use the fact that we have total control of the executing algorithm. In particular, for a given input problem, we execute

[2] Prior to constructing a NuCAD, a variable ordering is fixed, and the level of a polynomial is highest index within that ordering of any variable appearing in the polynomial.

NuCAD construction following the "current" choice ordering predicate, but we stop at a random point in the process at which we have more than one constraint violated at a sample point and thus have a choice to make. At this point, we separately try each of the possible choices (i.e. each polynomial), finishing out the entire construction from that point using the "current" choice ordering predicate for all subsequent choices. This gives us the exact number of cells resulting from each polynomial we could have chosen. Thus, for each pairing of polynomials from the set, we construct the associated feature vector and, since we know the correct result for the choice ordering predicate with that pair as input, we have the correct output to go along with it. By collecting these feature vectors and correct output values over many inputs, we produce a data set that can be used for a supervised, binary classification machine learning problem.

Of course, the "correct" results we gathered are only correct for what was then the "current" choice ordering predicate, and by learning we have derived a new choice ordering predicate. This means we should iterate the process until the performance of NuCAD construction using the learned choice ordering predicate converges. It is not clear that this convergence will happen. For example, the initial choice ordering function is chosen at random, and it could easily happen that NuCADs constructed with it are so far off from NuCADs constructed with a good choice ordering predicate, that the data we try to learn from is garbage. However, our experiments indicate that we get good performance in relatively few iterations.

Learn from Randomly Generated Inputs, Evaluate With "Real" Input: Our approach was to learn from randomly generated formulas—formulas in five variables with many constraints, both linear and non-linear, different levels of sparsity, and different numbers of variables appearing. Each input formula was used to analyze only one choice-point. The rationale for this choice is simply that we need a lot of data to learn from. The justification is that our initial experiments show that the learned choice ordering predicate does indeed transfer well to the "real" problems pulled from SMT-LIB, despite the fact that they often have very different shape and that all the problems we tested against had fewer than five variables.

5 Experiments

The Tarski[3] system [9] was used in our experiments for its implementation of NuCAD. For our machine learning, we used the Keras [4] package.

Random formulas were generated to consist of 13 "$<$" constraints, eight quadratic and five linear, with between two and four terms. Below is an example of one of them.

$$[0 < 18v - 9w \wedge 0 < 9vx + 4vy - 3y + 3 \wedge 0 < 5yz + 8y + 2 \wedge 0 < -4x^2 + 6w - 6y - 7 \wedge 0 < -8wz + 9z^2 + 8w \wedge 0 < -3vz + 6w - 8z - 4 \wedge 0 < v^2 + 22vy + 13y^2 + 4 \wedge 0 < -27w^2 - 72wz - 48z^2 - 20v + 14w - 8z \wedge 0 < -10w - 10y - 5z \wedge 0 < 9v + 8w - 4x \wedge 0 < -8v + 7w + 7x \wedge 0 < -8v - 5w + 1 \wedge 0 < -4w - 8x + 1] \quad (1)$$

[3] Tarski is available from https://www.usna.edu/Users/cs/wcbrown/tarski/.

Given a choice-point in a NuCAD construction and two polynomials, we construct a vector of 22 features capturing individual and comparative properties of the two polynomials and the context of the NuCAD construction. These features are described in the table below. The first six are comparative features based on the two polynomials under consideration (p_1 and p_2). Note that "spsize" is a function that measures the size of the internal representation of a polynomial—it is roughly proportional to the print length, but without any dependency on the length of variable names. Features 6–10 are comparative, but based on "p_1^*" and "p_2^*". Here p_1^* is the same as p_1, but with any term removed for which p_2 has a term with the same power product, and p_2^* is defined analogously. For example, if $p_1 = 2xz^2 - 3zy + x + 1$ and $p_2 = -xz^2 - 2y^2 + y - 5x$ then $p_1^* = -3zy + 1$ and $p_2^* = -2y^2 + y$. Features 11–14 involve a process we call "pseudo-projection", which gives estimates of the size of the projection set arising from choosing a particular polynomial with which to refine the current cell. It gives a very rough estimate, as no resultants or discriminants are computed. Features 15–18 are based on geometric information concerning the roots of $p_1(\alpha_1, \ldots, \alpha_{n-1}, z)$ and $p_2(\alpha_1, \ldots, \alpha_{n-1}, z)$, where α is the cell's sample point. Feature 21 is an arbitrary feature—in the context of learning, it is pure noise.

0	$\texttt{ite}\,\mathrm{tdeg}\,p_1 = 1 \wedge \mathrm{tdeg}\,p_2 > 1, -1, (\texttt{ite}\,\mathrm{tdeg}\,p_1 > 1 \wedge \mathrm{tdeg}\,p_2 = 1, +1, 0)$
1	$(\mathrm{level}\,p_1 - \mathrm{level}\,p_2)/(n-1)$, where $n =$ number of variables
2	$(\mathrm{tdeg}\,p_1 - \mathrm{tdeg}\,p_2)/5$
3	$\mathrm{spsize}\,p_1 - \mathrm{spsize}\,p_2$, where spsize is the internal data structures size
4	degree of p_1 in its main variable
5	degree of p_2 in its main variable
6−10	same as 1–5 except with p_1^* and p_2^*
11−14	pseudo-projection sizes and weighted sizes for p_1 and p_2
15	number of roots of $p_1(\alpha_1, \ldots, \alpha_{n-1}, z)$ inside cell
16	number of roots of $p_2(\alpha_1, \ldots, \alpha_{n-1}, z)$ inside cell
17	± 1 according to which polynomial gives a weaker lower bound over α
18	± 1 according to which polynomial gives a weaker upper bound over α
19	number of constraint polynomials known to be sign-invariant in cell
20	number of constraint polynomials not known to be sign-invariant in cell
21	± 1 based on hashes of p_1 and p_2

There is an existing hand-crafted heuristic used in TARSKI, called "BPC", against which we will compare our learned heuristic. It is expressible in terms of features zero, one and three of the feature vector F for polynomials p_1 and p_2, but we describe it more directly here as follows: if exactly one of p_1 and p_2 has total degree one, choose it; otherwise choose the polynomial with lower level, breaking ties by choosing the polynomial with smaller "spsize". The first feature in our feature vector was included so that other, less capable, learning paradigms (like decision lists) would still be expressive enough to learn BPC. For neural nets we could have replaced features zero and one with $\mathrm{tdeg}\,p_1$ and $\mathrm{tdeg}\,p_2$, which would have been more natural, but these same feature vectors were used for experiments outside of the scope of this paper.

For our learning model, we use a feed forward neural network that takes a dense network with 22 inputs and dimension $22 \times 22 \times 5 \times 5 \times 1$ with ReLU activation functions for internal nodes and sigmoid activation function for the output node. This network, seeded with random weights initially, represents the "current" choice ordering predicate.

Learning proceeded in rounds. In each round the neural net is used as the choice ordering predicate for producing data, as described in the previous section. From this training data, a new set of network weights was learned (starting from a different set of random weights), and this newly learned function served as the "current" choice ordering predicate for the subsequent round. This process ran for 10 rounds, producing the initial (randomly weighted) choice ordering predicate, and 10 learned predicates. Figure 2 shows the performance of the learned heuristics. Also shown is performance of "BPC" the hand-crafted heuristic that is used by default in our implementation of NuCAD. The performance does not smoothly increase from one iteration to the next, nor show the classic "U" shape often seen in machine learning, which warrants further investigation. The neural net resulting from the 6th iteration, "NN06", is the best performing network on these test problems.

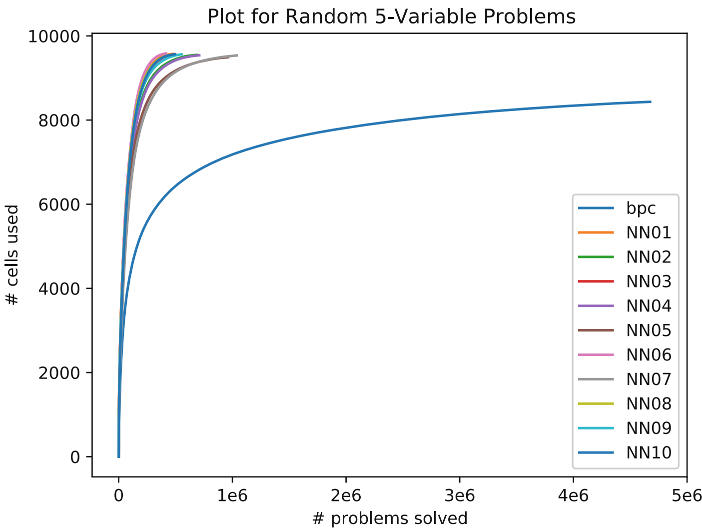

Fig. 2. This plot shows the performance of the trained neural networks for each round on the test data, which consists of problems generated randomly in the same way as the training data. Though difficult to see, "NN06", the network generated after the 6th round performs the best. Also shown is the performance of "BPC" the hand-crafted heuristic that is used by default in our implementation of NuCAD.

The crucial question is whether our learned choice ordering predicate transfers to problems that aren't of the same "shape" as the problems on which we trained. To evaluate this, we considered two data sets. The first is a set of 4,235 problems from [3]. These problems are conjunctions of strict inequalities derived from a subset of problems in the SMT-LIB QF_NRA collection by simplifications and case splittings. Essentially, in these problems "simplification" alone was insufficient, so we need to turn to a solver, like NuCAD. They are a natural choice for us to test with, as they meet the requirements of the current NuCAD implementation, namely that they are conjunctions of strict inequalities. None of these problems are terribly difficult—almost all have three or fewer variables—so we also produced a more challenging set consisting of 1,000 randomly generated four variable problems, generated in a different way than our test problems: each formula consisted of $x > 0 \wedge y > 0$ and nine conditions generated in Maple as `randpoly([x,y,z,w],degree=2,terms=4,coeffs=rand(-9..9))>0`. These examples differ from the training data in a number of ways. They have four variables instead of five. The only linear constraints are $x > 0$ and $y > 0$, whereas the training examples had five linear constraints, each of which was a multi-term constraint, like $-8v+7w+7x > 0$, that ties variables together. Finally the eight non-linear constraints in the training data were generated differently. Figure 3 shows the performance of BPC, NN06, and NN00, a randomly weighted neural net, which serves to show the performance of a random choice function. These experiments indicate that what gets learned from the training set does transfer over to other, unrelated input formulas. Obviously more testing on a wider range of inputs is required to be very confident of this, but these results are promising.

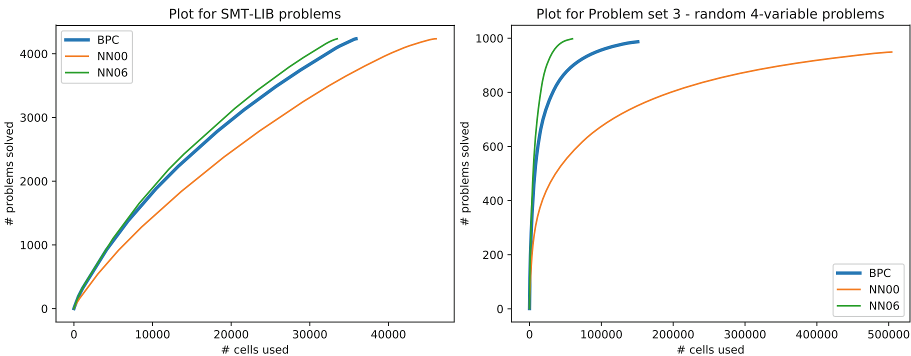

Fig. 3. Plots showing the performance of the hand-crafted heuristic BPC, the randomly weighted network NN00, and NN06, the trained network that performed best on the original set of training data. The left plot shows performance for the problem set stemming from the SMT-LIB, and the right plot shows performance on the four-variable randomly generated problem set. Note that NN00 failed to solve 51 problems within the 60 s timeout, BPC failed 13 problems within the 60 s timeout, and NN06 failed to solve two problems within the 60 s timeout.

6 Future Work and Acknowledgments

There are several avenues for future work that should be considered. The first is to improve upon what we've already done by training on a broader range of input formula "shapes"—different numbers of variables, different distributions for the constraints within formula, etc.—and by evaluating on a wider range of data sets, especially more "real" rather than randomly generated problems. The second avenue is to apply the basic approach outlined here to different problems, for example to the problem of selecting a variable ordering in Cylindrical Algebraic Decomposition, considered in [7], or choosing pivots in parametric Gaussian elimination.

Acknowledgements. This research was supported in part by the DOD High Performance Computing Modernization Program. We recognize and appreciate its support in enhancing our undergraduate education and research. Parts of this work were supported by National Science Foundation Grant 1525896.

References

1. Barrett, C., Fontaine, P., Tinelli, C.: The satisfiability modulo theories library (SMT-LIB) (2016). www.SMT-LIB.org
2. Brown, C.W.: Open non-uniform cylindrical algebraic decompositions. In: Proceedings of the 2015 ACM on International Symposium on Symbolic and Algebraic Computation, ISSAC 2015, pp. 85–92. ACM, New York (2015)
3. Brown, C.W., Vale-Enriquez, F.: From simplification to a partial theory solver for non-linear real polynomial constraints. J. Symb. Comput. **100**, 72–101 (2020). Symbolic Computation and Satisfiability Checking
4. Chollet, F., et al.: Keras (2015). https://keras.io
5. Florescu, D., England, M.: Algorithmically generating new algebraic features of polynomial systems for machine learning (2019)
6. Huang, Z., England, M., Davenport, J.H., Paulson, L.C.: Using machine learning to decide when to precondition cylindrical algebraic decomposition with Groebner bases. In: 18th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2016), pp. 45–52, September 2016
7. Huang, Z., England, M., Wilson, D., Davenport, J.H., Paulson, L.C., Bridge, J.: Applying machine learning to the problem of choosing a heuristic to select the variable ordering for cylindrical algebraic decomposition. In: Watt, S.M., Davenport, J.H., Sexton, A.P., Sojka, P., Urban, J. (eds.) CICM 2014. LNCS (LNAI), vol. 8543, pp. 92–107. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08434-3_8
8. Kobayashi, M., Iwane, H., Matsuzaki, T., Anai, H.: Efficient subformula orders for real quantifier elimination of non-prenex formulas. In: Kotsireas, I.S., Rump, S.M., Yap, C.K. (eds.) MACIS 2015. LNCS, vol. 9582, pp. 236–251. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-32859-1_21
9. Vale-Enriquez, F., Brown, C.W.: Polynomial constraints and unsat cores in Tarski. In: Davenport, J.H., Kauers, M., Labahn, G., Urban, J. (eds.) ICMS 2018. LNCS, vol. 10931, pp. 466–474. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96418-8_55