# Optimization Framework with Minimum Description Length Principle for Probabilistic Programming

Alexey Potapov[1,2,3], Vita Batishcheva[2,3], and Sergey Rodionov[3,4]

[1]ITMO University, St. Petersburg, Russia
[2]St. Petersburg State University, St. Petersburg, Russia
[3]AIDEUS, Russia
[4]Aix Marseille Université, CNRS, LAM (Laboratoire d'Astrophysique de Marseille) UMR 7326, 13388, Marseille, France
{pas.aicv, elokkuu, astroseger}@gmail.com

**Abstract.** Application of the Minimum Description Length principle to optimization queries in probabilistic programming was investigated on the example of the C++ probabilistic programming library under development. It was shown that incorporation of this criterion is essential for optimization queries to behave similarly to more common queries performing sampling in accordance with posterior distributions and automatically implementing the Bayesian Occam's razor. Experimental validation was conducted on the task of blood cell detection on microscopic images. Detection appeared to be possible using genetic programming query, and automatic penalization of candidate solution complexity allowed to choose the number of cells correctly avoiding overfitting.

**Keywords:** probabilistic programming, MDL, image interpretation, AGI

## 1 Introduction

Occam's razor is the crucial component of universal algorithmic intelligence models [1], in which it is formalized in terms of algorithmic information theory. In practice, Occam's razor is most widely used in the form of the Minimum Description/Message Length (MDL/MML) principles [2, 3], which can also be grounded in algorithmic information theory [4], but usually are applied loosely using heuristic coding schemes instead of universal reference machines [5].

Another form of Occam's razor is the Bayesian Occam's razor. In its simplest form, it penalizes complex models assigning them lower prior probabilities. However, these priors can be difficult to define non-arbitrarily. Some additional principles such as the maximum entropy principle were traditionally used to define priors, but algorithmic information theory providing universal priors resolves this difficulty more generally and elegantly [6] absorbing this simple form of the Bayesian Occam's razor. Real alternative to the information-theoretic interpretation of Occam's razor is 'a modern Bayesian approach to priors' [7], in which model complexity is measured by its flexibility (possibility to fit to or generate different data instances) estimated on the second level of inference.

Interestingly, Bayesian Occam's razor arises naturally without special implementation in probabilistic programming languages (PPLs) with posterior probability inference [8]. Programs in PPLs are generative models. They require a programmer to define prior probabilities for some basic random variables, but the total probability distribution is derived from the program. One can easily obtain universal priors by writing a function like (define (gen) (if (flip) '() (cons (if (flip) 0 1) (gen)))), where (flip) equiprobably returns #t or #f, and interpreting generated binary lists as programs for Universal Turing Machine (UTM).

Universal priors appear here from the natural structure of the program, and a concrete form of the selected distributions for the basic random choices only shifts them as the choice of a concrete UTM does. Similar situation appears in the case of models specifying Turing-incomplete spaces – higher-order polynomials with concrete coefficients will naturally have smaller prior probabilities than lower-order polynomials even if the degree of polynomials is uniformly sampled from a certain range.

Inference methods implemented in PPLs are intended for evaluating posterior probabilities incorporating priors defined by a program. Thus, instead of manually applying the MDL principle, one can simply use PPLs, which provide both the overlearning-proof criterion and automatic inference methods.

However, existing PPLs don't solve the problem of efficient inference in a general case, although they provide more efficient inference procedures than blind search. Now, different attempts to improve inference procedures are being made (e.g. [9, 10]). Most of them are done within the full Bayesian framework. The optimization framework, in which only maximum of posterior distribution (or other criterion) is sought, can be much more efficient and is enough in many practical tasks, but is much less studied in probabilistic programming.

Optimization queries require some criterion function to be defined instead of a strict condition. It is usually straightforward to define precision-based criteria. Actually, in some tasks, strict conditions are defined as stochastic equality based on likelihood (otherwise it will be necessary to blindly generate and fit noise), so the latter is more basic. Of course, if there is no appropriate quantitative criterion, the optimization framework is not applicable. However, if one uses stochastic equality, priors will be automatically taken into account by conditional sampling (since samples will be generative in accordance with prior probabilities and then kept proportionally to likelihood), while optimization queries will directly maximize the given criterion and will be prone to overfitting if this criterion is precision-based.

Thus, the necessity for MDL-like criteria arises in the optimization approach to probabilistic programming. Necessity for manual specification of such criteria, which incorporate not only precision, but also complexity, makes optimization queries much less usable and spoils the very idea of probabilistic programming. Thus, optimization queries should be designed in such a form that user-defined likelihood criteria are modified using automatically estimated priors.

In this work, we re-implement a functional PPL with optimization queries in the form of C++ library, which have been implemented in Scheme and described in the companion paper [11]. We add a wrapper for OpenCV to this library in order to deal with non-toy problems. In these settings, we develop a procedure to calculate prior probabilities of instantiations of generative models in the form of computation traces used in optimization queries, and study its applicability to avoid overlearning.

## 2 Background

**Minimum Description Length Principle**

Universal induction and prediction models are based on algorithmic complexity and probability, which are incomputable and cannot be directly applied in practice. Instead, the Minimum Description (or Message) Length principle (MDL) is usually applied. Initially, these principles were introduced in some specific strict forms [2, 3], but now are utilized in many applied methods (e.g. [5]) in the form of the following loose general definition [4]: the best model of the given data source is the one which minimizes the sum of

- the length, in bits, of the model description;
- the length, in bits, of data encoded with the use of the model.

Its main purpose is to avoid overfitting by penalizing models on the base of their complexity that is calculated within heuristically defined coding schemes. Such "applied MDL principle" is quite useful, but mostly in the context of narrow AI. Bridging the gap between Kolmogorov complexity and applications of the MDL principle can also be a step towards bridging the gap between general and narrow AI.

**Probabilistic Programming**

In traditional semantics, a program with random choices being evaluated many times yields different results. The main idea behind probabilistic programming is to associate the result of program evaluation not with such particular outcomes, but with the distribution of all possible outcomes. Of course, the problem is to represent and compute such distributions for arbitrary programs with random choices. It can be done directly only for some Turing-incomplete languages. In general case, the simplest way to deal with this problem is via sampling, in which a distribution is represented by the samples generated by a program evaluated many time using traditional semantics.

Crucial feature of PPLs is conditioning, which allows a programmer to impose some conditions on (intermediate or final) results of program evaluation. Programs with such conditions are evaluated to conditional (posterior) distributions, which are the core of Bayesian inference. The simplest implementation of conditional inference is rejection sampling, in which outcomes of the program evaluation, which don't meet the given condition, are rejected (not included into the generated set of outcomes representing conditional distribution). Such rejection sampling can be easily added to most existing programming languages as a common procedure, but it is highly inefficient, so it is usable only for very low-dimensional models. Consequently, more advanced inference techniques are being applied. For example, Metropolis-Hastings method is quite popular. In particular, it is used in Church [8], which extends Scheme with such sampling functions as *rejected-query*, *mh-query*, and some others.

PPLs extend traditional programming languages also adding to them some functions to sample from different distributions. In Church, such functions as *flip*, *random-integer*, *gaussian*, *multinomial*, and some others are implemented.

**Bayesian Occam's Razor in Probabilistic Programming**

As was mentioned, such PPLs as Church naturally support the Bayesian Occam's razor [8]. Let us consider the following very simple example.

(mh-query 1000 100

```
(define n (+ (random-integer 10) 1))
(define xs (repeat n (lambda () (random-integer 10))))
n
(= (sum xs) 12))
```

Here, we want a sum of unknown number *n* of random digits *xs* be equal to the given number, 12. Values of *n* belonging to the specified range are equiprobable a priori. However, the derived posterior probabilities are highly non-uniform – $P(n=2|sum=12)\approx0.9$; $P(n=3|sum=12)\approx0.09$; $P(n=4|sum=12)\approx0.009$.

Underlying combinatorics is quite obvious. However, this is exactly the effect of "penalizing complex solutions" that works in less obvious cases [8], e.g. polynomial approximation using polynomials of arbitrary degree, or clustering with unknown number of clusters.

## 3 Optimization Framework for Probabilistic Programming

**Implemented Library**

We aim at the practical, but general implementation of probabilistic programming, so we consider Turing-complete languages and optimization framework. We implemented a subset of Scheme language inside C++ using class constructors instead of function application. For example, such classes as *Define*, *Lambda*, *List*, *Cons*, *Car*, *Cdr*, *Nullp*, *ListRef*, *If*, and others with the corresponding constructors were implemented. All these classes are inherited from the *Expression* class, which has the field *std*::*vector*<*Expression* *> *children*, so expressions can constitute a tree. To create expressions from values, the class *Value* (with the synonym *V*) was added. This class is used for all values dynamically resolving supported types.

Also, such classes as *Add*, *Sub*, *Mult*, *Div*, *Gt*, *Gte*, *Ls*, *Lse*, etc. were added, and such operations as +, –, *, /, >, >=, <, <=, etc. were overloaded to call corresponding constructors. Consequently, one can write something like

*Define*(*f*, *Lambda*(*xs*, *If*(*Nullp*(*xs*), *V*(0), *Car*(*xs*) + *f*(*Cdr*(*xs*)))))

corresponding to

```
(define f (lambda (xs) (+ (if (null? xs) 0 (+ (car xs) (f (cdr xs)))))))
```

To use symbols *f* and *xs*, one needs to declare them as instances of the class *Symbol* (with the synonym *S*) or to write *S*("*xs*") instead of *xs*. Parentheses operator is also overloaded, so one can write *f*(*xs*) instead of *Apply*(*f*, *xs*), where *Apply* is also the child of *Expression*. Similarly, one can write *xs*[*n*] instead of *ListRef*(*xs*, *n*).

Classes corresponding to the basic random distributions were also added including *Flip*, *Gaussian*, *RndInt*, etc.

We also wrapped some OpenCV functions and data structures in our library. Support for *cv*::*Mat* as the basic type was added, so it is possible to write something like *Define*(*S*("image"), *V*(*cv*::*imread*("test.jpg"))). All basic overloaded operations with *cv*::*Mat* are inherited, so values corresponding to *cv*::*Mat* can be summed or multiplied with other values.

To avoid huge program traces while filling image pixels with random values (each such value will become a node in a program trace), we introduced such classes as

*MatGaussian* and *MatRndInt* for generating random matrices as holistic values. These random matrices can be also generated as deviations from given data.

The mentioned constructors of different classes are used simply to create expressions and arrange them into trees. Evaluation of such expressions was also implemented. A given expression tree is expanded into a program trace during evaluation. This program trace is also an expression tree, but with values assigned to its nodes. Evaluation process and program traces implemented in our C++ library are similar to that implemented in Scheme and described in the companion paper [11], so we will not go into detail here. Also, we re-implemented the optimization queries based on simulated annealing and genetic programming over computation traces. For example, one can write the following program with the result of evaluation shown in Fig. 1

```
Symbol imr, imb;
AnnealingQuery(List()
        << Define(imr, MatRndInt(img.rows, img.cols, CV_8UC3, 256, img))
        << Define(imb, GaussianBlur(imr, V(11.), V(3.)))
        << imr
        << (MatDiff2(imb, V(img)) + MatDiff2(imb, imr) * 0.3));
```

Here, *img* is some *cv*::*Mat* loaded beforehand, *List*() << *x* << *y* << *z* … is equivalent to (list x y z …). Operator << can be used to put additional elements to the list on the step of expression tree creation (not evaluation). *imr* is created as the random 3-channel image with *img* as the initial value. *MatDiff2* calculates RMSE per pixel between two matrices. *AnnealingQuery* is the simulated annealing optimization query, which minimizes the value of its last child, and its return value is set to the corresponding value of its last but one child. Here, the second term in the optimization function prevents from too noisy results. Also, *GPQuery* based on genetic programming is implemented.

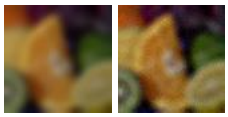

**Fig. 1.** The original blurred image and the result of inference

Simulated annealing is not really suitable to perform search in the space of images, but reasonable result is obtained here in few seconds. It can also be seen that general C++ code can be easily used together with our probabilistic programming library. Of course, this code is executed before or during construction of expression tree or after its evaluation, but not during the process of evaluation. The latter can be done by extending the library with new classes that is relatively simple, but slightly more involved.

Expression trees can be used not as fixed programs written by a programmer, but as dynamic data structures built automatically. So, such a library can easily be made a part of a larger system (e.g. a cognitive architecture).

Our library is under development and is used in this paper as the research tool, so we will not go into more detail. Nevertheless, the current version can be downloaded from https://github.com/aideus/prodeus

**Undesirable Behavior**

Optimization framework is suitable for many tasks, and optimization queries even without complexity penalty can be applied in probabilistic programming (see some examples in our companion paper [11]). However, even very simple generative models can be inappropriate in this framework. Consider the following program

```
Symbol xobs, centers, sigmas, n, xgen;
AnnealingQuery(List()
        << Define(xobs, V(4.))
        << Define(centers, List(3, -7., 2., 10.))
        << Define(sigmas, List(3, 1., 1., 1.))
        << Define(n, RndInt(Length(centers)))
        << Define(xgen, Gaussian(ListRef(centers, n), ListRef(sigmas, n)))
        << n
        << (xobs – xgen) * (xobs – xgen));
```

Intuitively, this program should simply return the number of the center closest to *xobs* since *AnnealingQuery* will minimize the distance from the generated value to the class center. However, evaluation of this program yields almost random indices of centers. The same model works fine in Church. The following query will return the distribution with $p(n=1)\approx 1$; and in the case of (define centers '(-7., -2., 10.)) it will return $p(n=1)\approx p(n=2)\approx 0.5$.

```
(define (noisy-equal? x y)
    (flip (exp (* -1 (– x y) (– x y)))))
(mh-query 100 100
    (define xobs 4)
    (define centers '(-7., 2., 10.))
    (define sigmas '(1., 1., 1.))
    (define n (random-integer (length centers)))
    (define xgen (gaussian (list-ref centers n) (list-ref sigmas n)))
    n
    (noisy-equal? xobs xgen))
```

It should be noted that noisy-equal? should apply *flip* to the correctly estimated likelihood, if one wants e.g. to get correct posterior probabilities for *xgen*. In particular, it should include such parameter as dispersion or precision. That is, these programs in C++ and Church really include the same information.

Inappropriate result of *AnnealingQuery* originates from its possibility to reduce the given criterion adjusting values of all random variables including both *n* and *xgen* in this model. It is much easier to adjust *xgen* directly since its probability is not taken into account in the criterion. This problem can be easily fixed here, if we will tell *AnnealingQuery* to minimize the distance from the *n*-th center to *xobs*. The program will be simpler, and its result will be correct. However, the general problem will remain. It will reveal itself in the form of overfitting, impossibility to select an appropriate number of cluster or segments in the tasks of clustering and segmentation, necessity to manually define *ad hoc* criteria, and so on. These are exactly the problems, which are solved with the use of the MDL principle.

**Complexity Estimation**

Apparently, if we want optimization queries to work similarly to sampling queries, we need to account for probabilities, with which candidate solutions are generated. Here, we assume that the criterion fed to optimization queries can be treated as the negative log-likelihood. Then, it will be enough to automatically calculate and add minus logarithm of prior probability of a candidate solution to achieve the desirable behavior.

We calculate these prior probabilities by multiplying probabilities in those nodes of the program trace subtree starting from *AnnealingQuery* or *GPQuery*, in which basic random choices are made. Here, we assume that the list of expressions fed to queries is relevant. As the result, each such choice is taken into account only once, even if a variable referring to this choice is used many times.

*AnnealingQuery* and *GPQuery* were modified and tested on the program presented above, and they returned $n$=1 in all cases, so they behave desirably. Of course, optimization queries give less information than sampling queries. For example, in the case of centers '(-7., -2., 10.) the former will return $n$=1 or $n$=2 randomly, while the latter will return their probabilities. However, optimization queries can be much more efficient, and can be used to find the first point, from which methods like *mh-query* can start.

## 4 Evaluation

Since we aim at practical probabilistic programming for Turing-complete languages, we consider image analysis tasks which are computationally quite heavy. To the best of our knowledge, the only example of such application is the work [12] (and unfortunately it lacks information about computation time). Thus, possibility to solve image analysis tasks in a reasonable time can be used as a sufficient demonstration of efficiency of the optimization framework. This is also our goal in addition to verification of the automatic MDL criterion calculation procedure.

Consider the task of detection of erythrocytes (our system wasn't aimed to solve this specific task, and it is taken simply as an example; other tasks could be picked). The typical image is shown in Fig. 2. The task is to detect and count cells. This task is usually solved by detecting edge pixels and applying Hough transform, or by tracking contours and fitting circles. Direct application of existing implementations of image processing methods is not enough, and application of non-trivial combinations of different processing functions or even ad hoc implementation of these functions is needed (e.g. [13]).

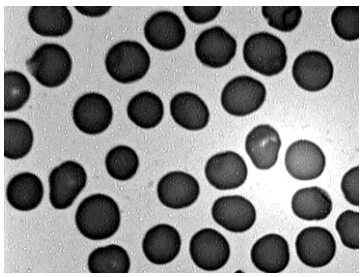

**Fig. 2.** The original image with red blood cells

However, an acceptable solution can be obtained using the following very small generative model:

```
Define(n, RndInt(20) + 10)
Define(circs, Repeat(n, Lambda0(List(RndInt(img.cols),
                                     RndInt(img.rows),
                                     RndInt(12)+6))))
Define(gen, Foldr(Lambda(circ, im,
                         DrawCircle(im, circ[0], circ[1], circ[2], V(168), V(-1))),
                  circs, V(cv::Mat::zeros(img.rows, img.cols, cv::CV_8UC1))))
circs
Log(MatDiff2(gen, V(img))) * V(img.cols * img.rows)
```

Here, *n* is the number of circles to draw, *circs* is the list of random circle centers and radii (*img* is the inverted image to be analyzed), *gen* is the generated image. It is generated starting from an empty image and consequently drawing circles from *circs*. It should be noted that since our library implements a functional quasi-language, such functions as *DrawCircle* don't modify the given image, but return a new one. The last two expressions in the model contain the resulting value and estimation of minus log-likelihood. To increase performance, we also implemented *Drawer* class. During evaluation *Drawer* processes a list of shapes and draws them using one resulting image. The program with *Drawer* instead of *Foldr* and *DrawCircle* was tested.

*AnnealingQuery* failed on the image with many objects, since each step of simulated annealing consists in an attempt to modify coordinates and sizes of all circles simultaneously, and successful modification becomes very unlikely for large number of variables. *GPQuery* showed acceptable results (see Fig. 3), but with some adjustment of the crossover operator.

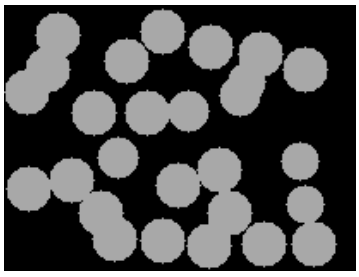

**Fig. 3.** The result yielded by *GPQuery* (population size = 300, number of generations = 100, mutation rate = 0.005)

*GPQuery* yields better results here, since it automatically performs "soft decomposition" of the given problem. However, its results are not optimal, and the search time is not too small (5–30 seconds on i5 2.6 GHz depending on GP parameters). Nevertheless, it is already usable for rapid prototyping.

The search problem is one of the most important problems here, and it is far from being fully solved. However, we are interested in testing the developed method for incorporating the MDL criterion into the optimization queries. Let us consider the calculated value of this criterion on different small images (Fig. 4) for different num-

ber of circles in order to ensure that the found solution is nearly optimal. Table 1 summarizes the obtained results.

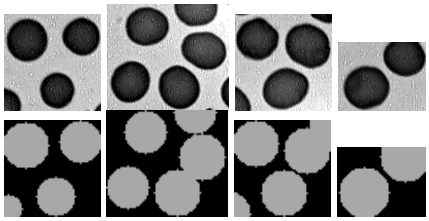

**Fig. 4.** Image fragments and best results for them

Table 1. Total description lengths, bits

| Image # | *n* | | | | | |
|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 |
| 1 | 14650.4 | 14038.0 | 13131.2 | 12687.3 | 12689.3 | 12690.0 |
| 2 | 20201.3 | 19612.1 | 18888.2 | 17955.2 | 17104.2 | 17115.2 |
| 3 | 14680.3 | 13995.2 | 12808.1 | 12391.7 | 12316.6 | 12321.0 |
| 4 | 9270.7 | 8155.1 | 8160.6 | 8162.6 | 8163.2 | 8168.5 |

It can be seen that the total description length starts to slowly increase from some number of circles for each image. Each circle adds around 10 bits of complexity. So, negative log-likelihood slightly decreases, but slower than increase of complexity. Actually, since blood cells are not perfectly circular, additional circles fitted to uncovered parts of cells can increase model complexity lesser than decrease of negative log-likelihood in some cases. However, in these cases, queries calculating posterior probability will also give a strong peak at the same number of circles. In other words, the origin of this result is not in query procedures or criteria, but in the model. In general, the found minima of the description length criteria correspond to the real number of blood cells, and partially presented cells are reliably detected.

## Conclusion

The developed method for automatic usage of the Minimum Description Length principle in probabilistic programming both reduces the gap between the loosely applied MDL principle and the theoretically grounded, but impractical Kolmogorov complexity, and helps to avoid overfitting in optimization queries making them an efficient alternative to more traditional queries estimating conditional probabilities. Experiments conducted on the example of an image analysis task confirmed availability of this approach.

However, even optimization queries being not specialized cannot efficiently solve arbitrary induction tasks especially connected to AGI. Actually, the task of such efficient inference can itself be considered as the “AI-complete” problem. Thus, deeper connections between AGI and probabilistic programming fields are to be established.

## Acknowledgements

This work was supported by Ministry of Education and Science of the Russian Federation, and by Government of Russian Federation, Grant 074-U01.

## References

1. Hutter, M.: Universal Artificial Intelligence: Sequential Decisions Based on Algorithmic Probability. Springer (2005)
2. Wallace, C.S., Boulton, D.M.: An Information Measure for Classification. Computer Journal 11, pp. 185–195 (1968)
3. Rissanen, J.J.: Modeling by the Shortest Data Description. Automatica-J.IFAC 14, pp. 465–471 (1978)
4. Vitanyi, P.M.B., Li, M.: Minimum Description Length Induction, Bayesianism, and Kolmogorov complexity. IEEE Trans. on Information Theory 46 (2), pp. 446–464 (2000)
5. Potapov, A.S.: Principle of Representational Minimum Description Length in Image Analysis and Pattern Recognition. Pattern Recognition and Image Analysis 22 (1), pp. 82–91 (2012)
6. Solomonoff, R.: Does Algorithmic Probability Solve the Problem of Induction? Oxbridge Research, P.O.B. 391887, Cambridge, Mass. 02139 (1997)
7. MacKay. D.J.C.: Bayesian Methods for Adaptive Models. PhD thesis, California Institute of Technology (1991)
8. Goodman, N.D., Tenenbaum, J.B.: Probabilistic Models of Cognition. https://probmods.org/
9. Stuhlmüller, A., Goodman, N. D.: A dynamic programming algorithm for inference in recursive probabilistic programs. In: Second Statistical Relational AI workshop at UAI 2012 (StaRAI-12), arXiv:1206.3555 [cs.AI] (2012)
10. Chaganty, A., Nori A.V., Rajamani, S.K.: Efficiently sampling probabilistic programs via program analysis. Proc. Artificial Intelligence and Statistics, pp. 153–160 (2013)
11. Potapov, A., Batishcheva, V.: Genetic Programming on Program Traces as an Inference Engine for Probabilistic Languages. In: LNAI (2015)
12. Mansinghka, V., Kulkarni, T., Perov, Y., Tenenbaum, J.: Approximate Bayesian Image Interpretation using Generative Probabilistic Graphics Programs. Advances in Neural Information Processing Systems, arXiv:1307.0060 [cs.AI] (2013)
13. Zhdanov, I.N., Potapov, A.S., Shcherbakov, O.V.: Erythrometry method based on a modified Hough transform. Journal of Optical Technology, vol. 80, no. 3, pp. 201–203 (2013)