Bayesian Optimisation of Solver Parameters in CBMC

Chaitanya Mangla, Sean B. Holden, and Lawrence C. Paulson

University of Cambridge
{cm772,sbh11,lp15}@cl.cam.ac.uk

Abstract

Satisfiability solvers can be embedded in applications to perform specific formal reasoning tasks. CBMC, for example, is a bounded model checker for C and C++ that embeds SMT and SAT solvers to check internally generated formulae. Such solvers will be solely used to evaluate the class of formulae generated by the embedding application and therefore may benefit from domain-specific parameter tuning. We propose the use of Bayesian optimisation for this purpose, which offers a principled approach to black-box optimisation within limited resources. We demonstrate its use for optimisation of the solver embedded in CBMC specifically for a collection of test harnesses in active industrial use, for which we have achieved a significant improvement over the default parameters.

1 Introduction

F* is a programming language aimed at formal program verification (Nikhil et al. [17]). Dafny is another such language, with built-in specification constructs and a corresponding static analyser for formal verification (Leino et al. [9]). CBMC is a static analysis tool for the C and C++ programming languages that uses bounded model checking to formally verify programs (Clarke et al. [2]). All three systems share an architectural similarity: formal verification is achieved by transforming the properties in question into logical formulae, which are then solved using external tools. This architecture is advantageous, because it allows the authors of a variety of tools to exploit the advancing capabilities of high performance solvers to perform complex logical reasoning, while allowing them to focus on tasks specific to their applications.

Modern SAT and SMT solvers rely on a variety of parameterized algorithms to achieve good performance, subject to a choice of good parameters for the target application. The space of parameters may be large; the Lingeling solver, for example, has 323 parameters [1]. Also, individual parameters can have significant influence on the performance of the solver. For example, Huang [5] has shown that the restart policy has a significant impact on the performance of a clause learning SAT solver. Finally, the best parameter choice is often sensitive to the class of target problems, as was shown to be the case for restart parameters by Sinz and Iser [15].

Any solver embedded in an application will in effect be used to check the specific class of formulae generated by that application. Could the performance of such a solver be improved by tuning its parameters according to the target application? We show this to be true in the case of a test suite that uses CBMC to formally verify properties of a library written in C. In this work, we use Bayesian optimisation to find an improved parameter configuration for the solver in CBMC, that is tuned especially for that test suite. Bayesian optimisation (Shahriari et al. [14]) provides a black-box parameter optimisation method that is founded on the principles of Bayesian statistics and practical to use within limited resource budgets.

Our work joins a large chorus of prior work on automatic parameter optimisation of Satisfiability solvers. Hutter et al. [6] used *ParamILS* to optimise parameters for their SAT solver *Spear*, which yielded speed-ups of 4.5 times on bounded model checking instances and 500 times on software verification problems, in comparison to manually tuned parameters. ParamILS is an automatic parameter optimisation tool for general algorithms, developed by Hutter et al. [8]. ParamILS begins with a random collection of parameter settings, searches around the neighbourhood of these parameters, and uses a system of randomisation and restarts to avoid getting lost in local minima. In addition to being a black-box method, ParamILS is also model-free, since it does not explicitly attempt to model the algorithm it is optimising. In contrast, Bayesian optimisation is a model-based method, wherein a probabilistic model of the optimisation target is constructed and improved iteratively, and that model is used to guide the parameter search. The Sequential Model-based Algorithm Configuration (SMAC) method, also developed by Hutter et al. [7], is another model-based parameter optimisation method that has been used to optimise SAT solvers. Although published research on the use of Bayesian optimisation for SMT solvers is limited, it has been successfully utilised in varied disciplines and is a topic of active research in the machine learning community. Shahriari et al. [14] have noted some interesting applications, such as A/B testing, policy parametrisation for robots, highway traffic congestion management and combinatorial optimisation. In our previous work with Słowik et al. [18], Bayesian optimisation was used to tune parameters of the SInE heuristic, commonly used for premise selection in first-order logic theorem provers. Recently, Oh et al. [12] have published work related to our work presented here, on the use of Bayesian optimisation for adapting a static analyser to given programs.

In the following sections, we describe the relevant aspects of CBMC and Bayesian optimisation in detail, and our experiments.

2 CBMC

CBMC is a bounded model checker for C and C++. CBMC performs code analysis using *symbolic simulation*, that is, it compiles code into logical formulae which are then sent to a solver for verification. With this method, assertions in the code can be verified statically. CBMC can also automatically generate assertions to check for common errors, such as buffer overflows, unsafe pointer access, memory leaks and arithmetic faults.

Test harness. Any project utilising CBMC will typically define a suite of *test harnesses*: carefully written C functions that act as entry-points for CBMC, along with specifications for loop unrolling, function pointer destinations, variables that should be treated as non-deterministic and other optimisations. Each test harness should aim to formally verify some aspect of the code. Test harnesses may be viewed as static analysis unit tests.

aws-c-common. *aws-c-common* is an open source library of commonly used datastructures and algorithms written in the C programming language, developed for use in the Amazon Web Services Software Development Kit and utilised by multiple projects. Given the central role of this library, the developers have committed considerable resources towards its formal verification using CBMC. At the time of writing, the library contains 166 CBMC test harnesses.

An example test harness from aws-c-common is shown in listing 1. This harness was designed to verify certain properties of `aws_string_eq_ignore_case()`: a function exported by the library. Of note, is the use of `nondet_bool()` to specify a non-deterministic value to CBMC. Non-deterministic values are also available for other datatypes, such as `nondet_ushortint()` for unsigned short integers. Such values represent input read from the environment and thus CBMC analyses the test harness for any possible choice of those values.

```
void aws_string_eq_ignore_case_harness() {
    struct aws_string *str_a = nondet_bool() ?
    ↪  ensure_string_is_allocated_bounded_length(MAX_STRING_LEN) : NULL;
    struct aws_string *str_b =
        nondet_bool() ? (nondet_bool() ? str_a : NULL) :
        ↪  ensure_string_is_allocated_bounded_length(MAX_STRING_LEN);
    if (aws_string_eq_ignore_case(str_a, str_b) && str_a && str_b) {
        assert(aws_string_is_valid(str_a));
        assert(aws_string_is_valid(str_b));
        assert(str_a->len == str_b->len);
    }
}
```

Listing 1: An example CBMC test harness from the aws-c-common library.

```
aws_string_eq_ignore_case_harness
    struct aws_string *str_a;
    _Bool return_value_nondet_bool;
    return_value_nondet_bool = NONDET(__CPROVER_bool);
    struct aws_string *tmp_if_expr;
    IF !(return_value_nondet_bool != FALSE) THEN GOTO 1
    struct aws_string *return_value_ensure_string_is_allocated_bounded_length;
    ensure_string_is_allocated_bounded_length((size_t)16);
    return_value_ensure_string_is_allocated_bounded_length =
    ↪  ensure_string_is_allocated_bounded_length#return_value;
    dead ensure_string_is_allocated_bounded_length#return_value;
    tmp_if_expr = return_value_ensure_string_is_allocated_bounded_length;
    GOTO 2
1:  tmp_if_expr = (struct aws_string *)(void *)0;
2:  str_a = tmp_if_expr;
    ...
```

Listing 2: A snippet from the goto file generated for the test harness shown in listing 1 from the aws-c-common library.

Goto files. CBMC uses a modified C compiler to compile source code down to a simplified format that is more amenable to further conversion into logical formulae. One such simplification, for example, is the unrolling of every loop to a specified number of iterations. The output files are called *goto* files in CBMC parlance. Formal verification of each test harness can be viewed as a two-stage process. First the test harness is compiled down to a goto file, and in the next stage the goto file is formally verified by conversion into logical formulae and checked by a solver. This project focuses on optimisation of the solver, and therefore we only need to generate the goto files for every test harness once. Only the second stage needs to be re-executed on any changes to the solver.

The goto files are binaries but a human readable text representation can be generated using CBMC. The text version of the goto file generated from the test harness shown in listing 1 spans 623[1] lines. The large size is in part due to inclusion of other code from the library relevant to the harness in the goto file. A snippet of that text is shown in listing 2, which shows the first few lines of the code generated for the test harness function.

[1]337 lines if the generated comments are excluded

aws-batch-cbmc. Formal verification using CBMC is a computationally intensive task, but each test harness can be verified independently and in parallel. The developers of aws-c-common run their suite of CBMC tests for their continuous integration process and therefore require a fast turnaround from them. Their solution is to run each test harness as an independent batch process on a public cloud service, using the system from another open source project called *aws-batch-cbmc*.

MiniSat. CBMC can use multiple external SMT and SAT solvers to check internally generated formulae, but its default solver is MiniSat — an open source SAT solver by Een et al. [3]. When using a SAT solver, CBMC performs bit-vector flattening of its formulae before verification by the solver. CBMC is compiled with MiniSat using its default parameters. For the purposes of this project, we have modified CBMC to accept solver parameters as runtime options. Although the experiments presented here are for a SAT solver, our methods are similarly applicable for parameter optimisation of SMT solvers.

3 Bayesian Optimisation

Bayesian Optimisation (BO) is a model-based black-box optimisation method, suitable for optimising targets that are expensive to evaluate and lack an analytical form, as is the case with typical SMT solvers. The optimisation target is modelled as an *objective function*, which maps from the parameters of the target system to a performance metric. In this work, we will often refer to the target system being optimised as the objective, with the target system being MiniSat by default. Since the objective is a black-box, the shape of the objective function is modelled using *Gaussian Process* (GP) regression [13], a machine learning method that produces a non-linear regression model using Bayesian statistics. Instead of modelling the objective as a single function, the GP models it as a probability distribution over functions. In effect, every point in the parameter space is mapped by the GP to a probability distribution over the value of the objective function.

Since Bayesian optimisation uses a machine learning model of the objective function, training data must be produced by testing the objective, initially at random. Subsequently, the optimisation process cycles between prediction, testing and training. Using the predictions from the latest model, a point of interest in the parameter space is chosen for testing, based on a criterion explained later in this section. The point is then evaluated on the objective. The result is a new item of data for the GP model to learn from, refining its estimate of the objective. This process is repeated until a pre-defined resource budget is consumed. The resource budget may be, for example, a fixed number of iterations or wall-clock time. The persistent goal throughout the process is to intelligently search for better performing parameters.

Suppose the noisy black-box objective we wish to optimise using Bayesian optimisation is f, such that

$$y = f(x) + \epsilon.$$

Our goal, therefore, is to find the point x that minimises $f(x)$, where x is vector valued. In our setting, x is a vector of MiniSat parameter values, f measures the time spent in MiniSat, and the noise ϵ is due to the inherent non-determinism in the underlying system which influences that measure. After some initial evaluations of f, Bayesian optimisation progresses in the following steps.

1. We treat all previous evaluations of f as training data and use a machine learning algorithm to build a predictive distribution of the values of y for any given x: $p(y|x)$. With the use

of a GP model, this distribution is Gaussian:

$$p(y|x) = \mathcal{N}(y|\mu(x), \sigma^2(x)).$$

We build this distribution instead of a point estimate because many different estimates of f will be able to interpolate the points we have evaluated thus far. Two additional observations reinforce the need for this design: our measurements of the objective may be noisy; and the objective may be expensive to evaluate. When the objective is expensive to evaluate, as is the case with the work presented here, models of the objective will be derived from only a limited number of previous evaluations, and so we must account for the wider distribution of functions that could interpolate them. This is achieved by estimating the variance in addition to the mean.

2. We then use this distribution to build an *acquisition function* α:

$$\alpha(x) = \mathbb{E}_{p(y|x)}[U(y|x)]$$

where U is the *utility function*, discussed in further detail below. The acquisition function is an inexpensive function that measures the expected utility of collecting data about any point x. The point x' that maximises $\alpha(x)$ is therefore the point to evaluate next.

3. We evaluate the objective at point x', which adds to our training dataset, and we return to step 1. However, if the pre-allocated resources for the optimisation process are exhausted, we terminate the optimisation process. The minimum is extracted from the collection of points evaluated so far.

Utility function. Given a model of the objective function, consider the *utility* of evaluating points in the parameter space. The expectation as well as the uncertainty of the performance of any point in the parameter space as predicted by the model will vary. Some regions of the parameter space may be predicted with high expectation and low variance, wherein it may be useful to *exploit* the knowledge in the model in search of an optimal point. Alternatively, it may be beneficial to *explore* the parameter space in regions of high uncertainty in the model, to reduce the uncertainty in the model and to reveal potentially hidden optimal points. Exploration and exploitation are trade-offs within the pre-determined resource budget allocated for the optimisation process and must be carefully balanced. The choice of the point of interest during Bayesian optimisation at any stage is the point that maximises the acquisition function, which in turn uses the utility function. This acquisition function maps the model of the objective, which is a distribution over functions, to a single function which measures the expected utility of any point in the parameter space. The utility function is user-specified and can be used to control the balance between exploration and exploitation. A commonly used utility function is *expected improvement.* As explained before, we model our objective using GP regression and thereby obtain a distribution over functions f:

$$p(f) = \mathcal{GP}(f; \mu, K)$$

where $\mu(x)$ is the mean function and $K(x, x')$ is the kernel covariance function. Suppose the minimum value of the objective that we have observed so far is v. The expected improvement utility function is then defined as:

$$U(x) = \max(0, v - f(x)).$$

In effect, this function rewards improvements proportionally, but does not reward degradations.

Discrete parameters. The Bayesian optimisation framework presented thus far is defined for continuous arguments to f. SMT solvers however often have discrete parameters. One solution is to map the discrete parameters to integers, and yet model them in the GP regression as any other real-valued parameter. When the objective needs to be evaluated, values supplied for the discrete parameters can be rounded to integers. Rounding is a common workaround for integer parameters in BO, but it can lead to problems, since the points that are actually evaluated are different from the points that are modelled in the GP, as shown in work by Garrido-Merchán et al. [4]. Their suggested solution is to modify the kernel covariance function $K(.,.)$ in the GP to

$$K'(x_i, x_j) = K(T(x_i), T(x_j)),$$

where $T(x)$ rounds all integer-valued variables to the closest integer. Kernel covariance functions are discussed further in appendix A. In this work, we have used a commercially available BO framework described below, which provides built-in facilities for integer parameters.

4 Methodology

In this work, we derive a parameter configuration for MiniSat that is optimised for a subset of the test harnesses in aws-c-common.

Goto files. Using aws-batch-cbmc, we completed one batch of the CBMC tests for aws-c-common. This generated goto files for each of the test harnesses, which we used subsequently for all our tests. The system also reported time spent by the tests in the solver. The 166 tests spend a total of 4252 seconds in the solver, with a median of 1.2 seconds; of these, 159 tests spend less than 100 seconds, and another 4 spend less than 200 seconds. The remaining three tests are notably slower, taking up 488, 687 and 1005 seconds respectively in the solver.

Tests and parameters. Given the resources available to us, we limited our sample of test harnesses to the 125 tests that spend the least amount of time in MiniSat. They represent 254 seconds of total time in the solver before optimisation. CBMC uses the SimpSolver in MiniSat, the parameters of which are shown in Table 1. We decided to limit our optimisation task to the SimpSolver specific parameters, leaving the remaining MiniSat parameters at their defaults. Early experiments revealed that any search for more optimal parameters would maintain the three Boolean parameters `asymm`, `rcheck` and `elim` at their default settings of `false`, `false` and `true` respectively. Therefore, we decided to fix these Boolean parameters to their default values and attempted to optimise only the remaining four parameters.

Objective function. Our goal was to find a single parameter configuration for the whole set of test harnesses that could reduce the total time spent in evaluating the entire set. In more detail, suppose there are N test harnesses, and functions t_i map from the parameters to the evaluation time of the test harness indexed by i, such that

$$t_i : (g, c, s, r) \mapsto \tau$$

where g, c, s and r are the MiniSat parameters `grow`, `cl-lim`, `sub-lim` and `simp-gc-frac` respectively, and τ is the time spent in MiniSat when that test harness is evaluated. For any fixed parameter configuration tuple (g', c', s', r'), the objective function can simply be $\sum t_i(g', c', s', r')$. However, it is necessary to adjust the objective function with a timeout, since some parameter

Table 1: MiniSat SimpSolver parameters

Parameter	Type	Default	Description
`asymm`	Bool	False	Shrink clauses by asymmetric branching
`rcheck`	Bool	False	Check if a clause is already implied
`elim`	Bool	True	Perform variable elimination
`grow`	Int $(-\infty, \infty)$	0	Allow a variable elimination step to grow by a number of clauses
`cl-lim`	Int $[0, \infty)$	20	Variables are not eliminated if it produces a resolvent with a length above this limit
`sub-lim`	Int $[0, \infty)$	1000	Do not check if subsumption against a clause is larger than this
`simp-gc-frac`	Double $(0, \infty)$	0.5	The fraction of wasted memory allowed before a garbage collection is triggered during simplification

configurations will lead to a prohibitively long evaluation time. Therefore, when the objective begins to exceed the pre-defined timeout, we take note and terminate it.

Implementation. Our implementation of the Bayesian optimisation process for this task is in Matlab [10], a commercially available numerical computing platform and development environment. Matlab 2019 includes a comprehensive Bayesian optimisation framework that was suitable for this task. Since our objective is a measure of elapsed real time on a local machine, small variations can be expected in the objective for any fixed set of parameters. The framework in Matlab can be set to accept such a non-deterministic objective. The framework also accepts integer parameters, which was another key requirement of our project, since all parameters except for the garbage collection factor are discrete.

In the Matlab implementation of Bayesian optimisation, the objective function may return with `NaN` (not a number) to signify an error. Matlab builds an error model to accommodate errors in the objective, and this model can accommodate non-deterministic errors. We chose to model timeouts in our objective function as errors. Modelling timeouts as errors allows the Bayesian optimisation framework to avoid parameter spaces that frequently cause timeouts. In this scenario, the choice of the timeout value has trade-offs that must be considered. A short timeout will be resource efficient, reducing the time spent in parameter spaces that perform worse than the original ones. However, poor performing parameters are also informative of the shape of the objective function, but when they are modelled as errors due to the timeout, the GP model has less data to learn from. Based on the practical resources available to us, we specified a large timeout of 1200 seconds, more than four times the time taken by MiniSat in the original configuration.

Acquisition function & GP kernel. We used the *expected improvement plus* acquisition function from the framework in Matlab. This is similar to the expected improvement function described above, with a modification to avoid excessively exploiting an area. We used the default exploration ratio of 0.5. To obtain the maximum control over the Bayesian optimisation process in Matlab, we used the `bayesopt` function. However, the kernel covariance function used by this is fixed to the Matérn 5/2 kernel, which is briefly discussed in appendix A. The kernel hyperparameters are optimised internally by Matlab, and options to influence them are unavailable.

5 Results

Table 2: Parameter bounds for optimisation.

Parameter	Bounds
`grow`	$[-10000, 10000]$
`cl-lim`	$[0, 1000] \cup \{\infty\}$
`sub-lim`	$[0, 100000] \cup \{\infty\}$
`simp-gc-frac`	$(0, 50.0]$

It is necessary to specify bounds on the parameter space for the optimisation process. Our specified bounds are shown in Table 2. These bounds were chosen to be sufficiently large, again based on preliminary experiments. In the case of `cl-lim` and `sub-lim`, it is possible to set both parameters to -1 in MiniSat, which then removes the clause limit and subsumption limit entirely, and we included that option in our parameter search space.

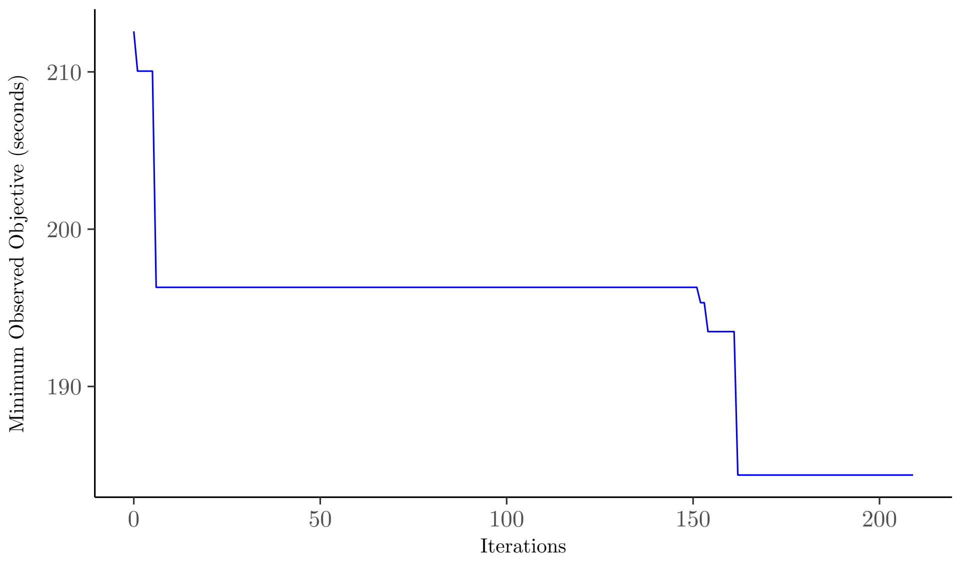

Figure 1: The minimum observed objective as Bayesian optimisation progresses through each iteration.

After running the optimisation for a total of 210 iterations, which completed in thirty-four hours, the best discovered parameters are shown in Table 3. In comparison to the original parameters, these parameters reduce the time spent in the solver from 254 seconds to 184 seconds. Figure 1 plots the minimum objective that is observed until each iteration during Bayesian optimisation. The times presented here are a measure of the times spent in MiniSat. Each invocation of CBMC to evaluate a goto file will require relatively expensive additional steps such as the preparation of formulae, which are unaffected by changes in MiniSat parameters.

We have optimised a collection of test harnesses all together to find a single parameter configuration that, applied to all the tests, reduces the total time spent in the solver. An

Table 3: Improved parameters discovered using Bayesian optimisation

Parameter	Value
`grow`	1615
`cl-lim`	2
`sub-lim`	61603
`simp-gc-frac`	28.32

alternative is to optimise each test harness individually, and save the discovered parameters along with the test harness. When the suite of test harnesses is evaluated, each test harness can then be evaluated with individualised solver parameters. The trade-off between this approach and our presented approach depends on the resources available and the nature of the test harnesses. For a basic comparison, we optimised two randomly selected tests harnesses individually for the same number of iterations. The results, shown in Table 4, show that the optimised parameters discovered for the two are very different, and further improvement is achieved.

Table 4: Parameters optimised for two test harnesses individually, with durations (original, optimised) in the solver.

					Solver time		
Test harness	`grow`	`cl-lim`	`sub-lim`	`simp-gc-frac`	Orig.	Opt.	Gain
`aws_byte_buf_cat`	0	-1	1000	0.5	56s	51s	9%
`aws_hash_table_put`	-8208	230	36	29.68	78s	59s	24%

6 Future Work

We have focussed on optimisation of the SimpSolver specific parameters in this work. However, another eleven parameters are available in MiniSat, and tuning all parameters together may yield improved results.

7 Conclusions

We have shown that the use of Bayesian optimisation to tune parameters of the default solver in CBMC is effective, and leads to a significant improvement in our experiments. Bayesian optimisation has shown to be successful in a variety of optimisation tasks, and our results show it can be effectively used to tune parameters of satisfiability solvers.

8 Acknowledgements

This work was developed during an internship at the Automated Reasoning Group of Amazon Web Services, supervised by John Harrison. Subsequent research was supported by the Engineering and Physical Sciences Research Council Doctoral Training studentship [reference number 1788755].

References

[1] Armin Biere. Yet another local search solver and Lingeling and friends entering the SAT Competition 2014. *Sat competition*, 2014(2):65.

[2] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176. Springer, 2004.

[3] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, pages 502–518, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[4] Eduardo C. Garrido-Merchán and Daniel Hernández-Lobato. Dealing with categorical and integer-valued variables in Bayesian optimization with Gaussian processes. *Neurocomputing*, 380:20–35, 2020.

[5] Jinbo Huang. The effect of restarts on the efficiency of clause learning. In *Proceedings of the 20th International Joint Conference on Artifical Intelligence*, IJCAI'07, page 2318–2323, San Francisco, CA, USA, 2007. Morgan Kaufmann Publishers Inc.

[6] Frank Hutter, Domagoj Babic, Holger H Hoos, and Alan J Hu. Boosting verification by automatic tuning of decision procedures. In *Formal Methods in Computer Aided Design*, FMCAD'07, pages 27–34. IEEE, 2007.

[7] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In Carlos A. Coello Coello, editor, *Learning and Intelligent Optimization*, pages 507–523, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[8] Frank Hutter, Holger H. Hoos, and Thomas Stützle. Automatic algorithm configuration based on local search. In *Proceedings of the 22nd National Conference on Artificial Intelligence - Volume 2*, AAAI'07, page 1152–1157. AAAI Press, 2007.

[9] K. Rustan M. Leino and Michał Moskal. Co-induction simply. In Cliff Jones, Pekka Pihlajasaari, and Jun Sun, editors, *FM 2014: Formal Methods*, pages 382–398, Cham, 2014. Springer International Publishing.

[10] The Mathworks, Inc., Natick, Massachusetts. *MATLAB version 9.7.0.1296695 (R2019b) Update 4*, 2019.

[11] Ha Quang Minh, Partha Niyogi, and Yuan Yao. Mercer's theorem, feature maps, and smoothing. In *International Conference on Computational Learning Theory*, pages 154–168. Springer, 2006.

[12] Hakjoo Oh, Hongseok Yang, and Kwangkeun Yi. Learning a strategy for adapting a program analysis via Bayesian optimisation. *ACM SIGPLAN Notices*, 50(10):572–588, 2015.

[13] Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian processes for machine learning.* MIT Press, Cambridge, MA, 2008.

[14] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando De Freitas. Taking the human out of the loop: A review of Bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2015.

[15] Carsten Sinz and Markus Iser. Problem-sensitive restart heuristics for the DPLL procedure. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009*, pages 356–362, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[16] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical Bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

[17] Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. Verifying higher-order programs with the Dijkstra monad. *ACM SIGPLAN Notices*, 48(6):387–398, 2013.

[18] Agnieszka Słowik, Chaitanya Mangla, Mateja Jamnik, Sean Holden, and Lawrence Paulson. Bayesian optimisation for heuristic configuration in automated theorem proving. In Laura Kovacs and Andrei Voronkov, editors, *Vampire 2018 and Vampire 2019. The 5th and 6th Vampire Workshops*, volume 71 of *EPiC Series in Computing*, pages 45–51. EasyChair, 2020.

A Kernel covariance functions in GP regression

A *kernel function* is a real-valued function of two vector-valued arguments,

$$k(x, x') \in \mathbb{R},$$

for $x, x' \in \chi$ where χ is some abstract space. Kernels may be viewed as a measure of similarity between the two arguments when the function is symmetric and non-negative. For example, the following kernel measures cosine similarity:

$$k(x, x') = \frac{x^T x'}{||x||_2 ||x'||_2}.$$

Kernel functions used as covariance functions in Gaussian processes must be positive definite. Such a kernel is called a *Mercer kernel*. Mercer's theorem (Minh et al. [11]) shows that for such kernels there exists a function

$$\phi : \chi \mapsto \mathbb{R}^D$$

such that

$$k(x, x') = \phi(x)^T \phi(x').$$

This kernel is effectively computing an inner product of the two arguments in a D dimensional space, where D is typically larger than the dimensionality of χ and potentially infinite. For example, consider the following kernel when $x, x' \in \mathbb{R}^2$:

$$\begin{aligned} k(x, x') &= (1 + x^T x')^2 \\ &= (1 + x_1 x'_1 + x_2 x'_2)^2 \\ &= 1 + 2x_1 x'_1 + 2x_2 x'_2 + (x_1 x'_1)^2 + (x_2 x'_2)^2 + 2x_1 x'_1 x_2 x'_2 \\ &= \phi(x)^T \phi(x') \end{aligned}$$

where

$$\phi(x) = \left[1, \sqrt{2}x_1, \sqrt{2}x_2, x_1^2, x_2^2, \sqrt{2}x_1 x_2\right]^T.$$

Therefore, this kernel represents a measure of similarity in a six-dimensional space for two-dimensional vectors. The cosine similarity kernel shown above is also a Mercer kernel. As Snoek et al. [16] explain, the squared exponential kernel is often a default choice for the covariance function in Gaussian processes regression:

$$k_{SE}(x, x') = \theta_0 e^{\{-\frac{1}{2} r^2(x, x')\}}, \quad r^2(x, x') = \sum_{d=1}^{D} \frac{(x_d - x'_d)^2}{\theta_d^2}.$$

However, this may lead to sample functions in the GP that are unrealistically smooth for practical optimisation problems. Instead, Snoek et al. recommend the use of the Matérn 5/2 kernel:

$$k_{M52}(x, x') = \theta_0 \left(1 + \sqrt{5r^2(x, x')} + \frac{5}{3} r^2(x, x')\right) e^{\{-\sqrt{5r^2(x, x')}\}}.$$