Notes: The Window Sticker Framework

September 20, 2023

1 The `Stochastic-Benchmark` Implementation

`Stochastic-Benchmark` is an open-source package implementing the methodology in the Window Sticker framework. This open-source package implements a statistical analysis methodology for evaluating and comparing the performance of parameterized stochastic optimization solvers. We present the key features and contributions of the Window Sticker, emphasizing its relevance in addressing the unique characteristics and challenges associated with parameterized stochastic optimization methods. By incorporating data visualization techniques and robust statistical analysis, the `Stochastic-Benchmark` implementation of the Window Sticker provides researchers with a comprehensive framework to assess solver performance and facilitate informed decision-making, especially in quantum-inspired optimization.

The Window Sticker framework is relevant for analyzing quantum-inspired methodologies, which often produce populations of solutions as outputs. Empirical observations reveal that quantum and analog solvers can have a distinct advantage over random search, producing probability distributions of outputs that consistently yield high-quality solutions. However, these solvers might struggle to generate samples of the global optimum and cannot guarantee optimality even when they do. Existing techniques to address this issue, such as error mitigation techniques, primarily focus on enhancing the quality of scalar observables, such as the expectation values of functions, rather than correcting bitstrings themselves.

Addressing these challenges can be done in several ways. Firstly, improving the distribution of solution quality can be achieved through pre-processing techniques and tuning the algorithm's parameters. This is an issue for practical expected performance since good parameter settings might not be generalizable to other problem instances, success metrics, or available resources. Moreover, the parameter-tuning strategy is resource-consuming and usually is not reported when discussing the expected performance of the solvers. Secondly, by designing solvers that leverage the quantum or analog solver's capabilities to enhance the expectation value itself, the weaknesses of these methods can be mitigated by algorithmic approaches. Assessing the performance of such methods becomes challenging as the single solution of specific sub-problems only accounts for a portion of the actual optimization problem solution. The analysis framework here addresses these issues by providing a general performance comparison and parameter-setting strategy evaluation platform.

The framework presented in this study was designed to facilitate the analysis of parameterized stochastic optimization solvers. In this context, we use the term "solver" to refer to any end-to-end combination of hardware (the device) and software running an algorithm and the operations occurring within the device, utilized together to attempt to find solutions to optimization problems. Solvers typically feature multiple parameters that significantly impact their performance, yet the effects of those parameters are often unknown in advance. We can treat the solver to be benchmarked as a sampler of random variables from an unknown distribution for our analysis, sometimes referred to by the classical optimization community as stochastic optimization methods [5]. This setting of stochastic optimization methods is particularly appealing to the case of quantum heuristics and Ising machines, as they can be characterized within this class of optimization methods.

We describe a general abstraction of these solvers in the following pseudo-algorithm. The raw output of such stochastic methods is a string of N bits or binary values $(z_1 \dots z_N)$ obtained by a single measurement at the end of the computation [1]. We associate the latter with a vector variable $\mathbf{z} = (z_1, \dots, z_N)$. Note that

[1] In case of an Ising model framework, it is a spin configuration $\{\sigma_i \mid \sigma_i \in \{-1, 1\}\}$, however, without loss of generality both representations are equivalent up to a linear transformation, e.g., $z_i = (\sigma_i + 1)/2$.

Algorithm 1 Solution of an optimization problem via a Parameterized Stochastic Optimization Solver

Require: Solver s
Require: Parameters θ
Require: Optimization problem I that admits as a solution a bit-string of size N
Require: Resources R available for finding a solution.
Require: Number of samples Z, returned by the solver
1: Initialize distribution $D(s(\theta))$ as a uniform random distribution over bitstrings of size N
2: **for** r in R **do**
3: Update the distribution $D(s(\theta))$
4: **end for**
5: **return** Z samples $\mathbf{z} \in \{0,1\}^N \sim D(s(\theta))$

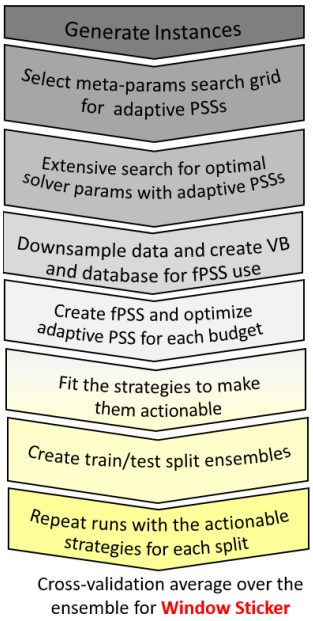

Figure 1: Flow chart showing the main steps followed by the Window Sticker framework.

the algorithm description does not specify how the distribution is updated or how the samples are obtained.

Additionally, the stochastic nature of these solvers implies that they generate a distribution of solutions, which necessitates applying post-processing techniques to determine the output required. A comparison between stochastic optimization algorithms and deterministic solution methods, which return the same solution to a problem every time they are executed and might even provide guarantees on the optimality of such a solution, might not be valid in general, given the heuristic nature of sampling associated with the stochastic methods. We can define a transformed real-valued variable $X = fun(\mathbf{z})$, where $fun : \mathbb{B}^N \rightarrow \mathbb{R}$ is known as a pseudo-Boolean function [3]. This real-valued variable, X, can be defined by a scalar function that takes the values of a bitstring and returns a real-valued cost or objective computed by a polynomial function of the bits in $\mathbf{z}$. It is well known that any pseudo-Boolean function can be written uniquely as a multilinear polynomial, i.e., $fun(\mathbf{z}) = c_0 + \sum_{i=1}^{N} c_{1,i} z_i + \sum_{j=i+1}^{N} c_{2,ij} z_i z_j + \sum_{k=j+1}^{N} c_{3,ijk} z_i z_j z_k + \cdots$. This variable X can represent the solver's progress toward solving a single problem. The solver's performance can then be assessed through variable X and used to learn how this solver behaves across different problem instances and compared against other solvers.

Solver performance can vary widely for different problem instances, so to make a more informed benchmarking framework and use the experimental results of using the solver for specific instances to conclude about the solver's usability for a given problem class or family, special care needs to be taken when considering several different problem instances as discussed below. Given our interest in stochastic solvers, we will be

interested in estimating the probability density function of the output variable, X, from the samples acquired during the solution process. We note that the probability density function provides us with an empirical observation of the unknown distribution we are interested in sampling. Improving the output of stochastic solvers results in a change in the solution paradigm compared to deterministic solvers, where the new goal is skewing these distributions towards the desired output and sampling it as efficiently as possible. In this perspective, deterministic solvers search over a Dirac delta distribution centered at the optimal solution, in which case sampling becomes irrelevant, and the deterministic and structured search becomes equivalent to finding such a distribution. Moreover, we will be interested in reporting the solver output for the variable X and the confidence with which a given solution quality can be guaranteed for a new problem instance generated from the instance class of interest.

By augmenting confidence intervals, the Window Sticker provides researchers and practitioners with a valid range of expected performance with unseen instances, providing a robust framework for evaluating solver performance and comparing different algorithms.

In this section, we proceed to explain how the Window Sticker framework operates. We will assume that the following is given:

- Resource available for evaluation $R = [r_0, r_f]$
- Performance metric to be considered P
- Set of instances $I = \{i_1, \ldots, i_{|I|}\}$.
- Set of solvers $S = \{s_1, \ldots, s_{|S|}\}$.
- Set of pre-evaluated parameters for solver s, $\alpha_s = \{\alpha_{s,1}, \ldots, \alpha_{s,|\alpha|}\}$.

As stated in Algorithm 1, the solver returns Z samples, say $\{\mathbf{z}_1, \ldots, \mathbf{z}_Z\} = \{\mathbf{z}_{[\![Z]\!]}\}$, which can be considered to have been sampled from the distribution $D(s(\theta))$. Let $X_i = F(\mathbf{z}_i)$ denote the value of the objective function evaluated at $\mathbf{z}_i$. A performance metric, $perf : \mathcal{D} \to \mathbb{R}$, determines the quality of a distribution, where $\mathcal{D}$ denotes the space of probability distributions over length N bitstrings. Since $D(s(\theta))$ is not known, we only have access to samples $\{\mathbf{z}_{[\![Z]\!]}\}$. In general, we first obtain an approximation $\hat{s}$ for $s(\theta)$ using the samples $\{\mathbf{z}_{[\![Z]\!]}\}$, and then evaluate $\mathcal{D}(\hat{s})$ as an approximation for $\mathcal{D}(s(\theta))$. We can approximate the performance as $Y = \text{perf}(\{\mathbf{z}_{[\![Z]\!]}\})$. Thus, Y is the empirical performance metric and denotes a random variable that serves as a proxy for quantifying the quality of solutions a solver generates. Note that Y is a short-hand for $Y(s, \theta, R, i, Z)$, where the argument s denotes the solver, θ the parameters, R the resources, i the problem instance, and Z the number of samples returned by the solver.

The set of parameter values that maximize the performance of a solver, s, when applied to an instance i, are generally not known *a priori*. Hence, a parameter-tuning strategy, such as Bayesian optimization or gradient descent, may be employed to obtain parameter value θ^* that maximizes $\mathbb{E}[Y]$ after averaging over multiple implementations of multiple runs of the solver.

The objective of the Window Sticker framework is multi-fold. First, for any given solver s, the framework generates a performance profile by running it with the same parameter setting using a different number of samples Z or simulating this behavior. Since running the solver might be expensive, we consider using a re-sampling via bootstrapping.

Following the generation of performance profiles for different parameter settings of a solver, we compute a virtual-best performance and strategy, where for each evaluated number of samples and instance, the best parameter setting is chosen in terms of the performance metric. Its performance is reported as an idealized solver, where the best parameters for each instance at each resource quantity were known.

A single set of parameters for a set of instances is chosen based on the median performance of the instance population. This strategy then results in what the usual recommendation in other pieces of literature report as the recommended parameter setting.

Notice that, to avoid over-fitting, we compute the series of parameter settings that lead to the best performance on a subset of instances, which becomes a training set, followed by evaluating the performance on a set of different instances that we call the testing set.

The previous parameter setting supposes that there is a static parameter setting strategy valid for all instances. This assumption might not always be valid; hence, we propose a sequential exploration and

exploitation strategy. In this case, the framework uses black-box optimization methods; in particular, for this implementation, we integrate the Tree of Parzen Method implemented in Hyperopt [2] with our code to determine, not the solver parameters, but the exploration and exploitation meta-parameters. There, the balance of the fraction of exploration time versus a total running budget and the effort of each exploration step is considered.

Finally, to avoid biases from the testing and training datasets selection, a cross-validation procedure is performed by shuffling these subsets, allowing us to sharpen the confidence intervals around the performance predictions we provide.

We provide more details on each step below.

1.1 Re-sampling via bootstrapping

A stochastic solver's performance may depend on the number of samples generated from it. Thus, one may be interested in the expected performance as a function of the number of samples. Instead of generating new samples to compute the performance metric for different resource values, we reuse data and compute these values given a single set of samples from the solver via bootstrapping.

1.2 Virtual Best Baseline

The virtual best baseline can be viewed as an oracle value representing the solver's best possible performance over all parameter settings. It is computed as the best performance on each problem instance over all parameters evaluated in the data set.

1.3 Fixed Suggested Parameters

This experiment evaluates the effectiveness of a static parameter setting as a function of resource value. The recommended parameter settings are computed by simply recording the best parameter values over the training set for each resource value and evaluating those values on the test set. Suppose the recommended parameters are not found in the collected data. In that case, they are projected to the closest ones included (hence, why it is referred to as the projection experiment).

1.4 Sequential Exploration-Exploitation

In the data collection process, we used the Hyperopt package for Bayesian optimization of the parameter values. The Sequential Exploration-Exploitation strategy with Hyperopt is designed to simulate a process where Hyperopt is used to explore new parameter settings for `ExploreFrac` fraction of the total resources, and the remaining resources are applied towards sampling values using the best parameter settings found during the exploration phase. The two meta-parameters optimized for each resource are the quantity of resources used to evaluate each new parameter setting, `tau`, and the fraction of total resources that should be used for exploring new parameters, `ExploreFrac`.

1.5 Cross-Validation

The conclusions obtained from different test-train splits could vary depending on which instances are in test or training sets. To obtain more reliable results, we first choose multiple test-train splits. We obtain a performance profile, parameter strategy, and hyper-parameter recommendation for each split. For each value of resource, we combine the center static and confidence intervals obtained from the different splits into a combined central static and confidence interval for the parameter recommendation and the performance.

2 Illustrative Example

This section describes results obtained by applying the Window Sticker framework on an illustrative example. First, we briefly describe the problem to be solved and the problem instances we consider. Next, we describe

the two solvers we compare: parallel tempering and coherent Ising machine simulator. Finally, we present the benchmarking results.

2.1 Choice of Problems for Benchmarking: Wishart Instances

We solve a class of zero-field Ising models, i.e., Hamiltonians of the type

$$H = \sum_{i,j=1}^{N} J_{ij} s_i s_j,$$

for $s_i = \pm 1$, with all-to-all connectivity. The ground state energy and its corresponding solution are desired. The values of J_{ij} are picked from the Wishart ensemble. In particular, we use the Wishart planted ensemble [6] to generate problem instances with a priori solutions. The difficulty of the problems generated is controlled by a parameter α, with a non-monotonic easy→hard→easy profile as α is varied. For $0 < \alpha \leq 1$, the problem difficulty increases as α is reduced. We choose $\alpha = 0.5$ for illustrative purposes in the following unless stated otherwise.

We use a Python library called Chook (see Refs. [12, 13] for the GitHub repository and the accompanying manuscript) to generate the problem instances.

2.2 Solver 1: Parallel Tempering

Replica exchange MCMC sampling [7], which is also known as parallel tempering, is a state-of-the-art heuristic for solving Ising-like optimization problems. Parallel tempering aims to overcome the issues faced by simulated annealing [8] by initializing multiple 'replicas' at different temperatures. The replicas undergo some Metropolis-Hastings updates, followed by a temperature swap between two replicas. Here, we briefly describe the solver and the parameter that determines the solver performance and refer the reader to Refs. [15, 10] for more details.

In parallel tempering, a number of replicas n_R are initiated at temperatures ranging between $T_{\min}$ and $T_{\max}$ set by the user. It is usually more convenient to encode these temperatures instead in terms of two probabilities p_{cold} and p_{hot}, that control how likely a spin flip occurs in a Metropolis update at the $T_{\min}$ and $T_{\max}$ respectively. In particular, p_{cold} quantifies the probability of the least likely spin flip at the lower temperature, and p_{hot} denotes the probability of the most likely spin flip at the higher temperature. While both these probabilities depend on the exact value of the J_{ij} matrix, an issue which itself becomes a combinatorial optimization problem, they can be approximated as follows:

$$p_{\text{cold}} = N_{\text{min-gap}} \exp\left(-\frac{-\Delta E_{\text{cold}}}{T_{\min}}\right)$$
$$\text{and } p_{\text{hot}} = \exp\left(-\frac{\Delta E_{\text{hot}}}{T_{\max}}\right).$$

with

$$\Delta E_i^{\text{cold}} = 2 \min_{j|J_{ij}\neq 0} |J_{ij}|$$
$$\Delta E_{\text{cold}} = \min_i \Delta E_i^{\text{cold}}$$
$$N_{\text{min-gap}} = \sum_{i|\Delta E_i^{\text{cold}} = \Delta E_{\text{cold}}} 1$$
$$\Delta E_{\text{hot}} = 2 \max_i \sum_j |J_{ij}|.$$

Thus, given a specific problem instance (which is specified by a J_{ij} matrix) and a choice of the probabilities p_{hot} and p_{cold}, the temperatures $T_{\min}$ and $T_{\max}$ can be inferred from the above formulas.

In addition to n_R, the time of execution is impacted by another parameter, the number of sweeps s, which denotes the number of Metropolis updates to be implemented in the algorithm. Thus, the solver takes four parameters, n_R, s, p_{cold} and p_{hot}.

We use an efficient Python-based implementation of parallel tempering called PySA [1] to generate the data for the benchmarking.

2.3 Coherent Ising Machine

Ising machines are a class of solvers based on the dynamics of physical hardware that aims to solve for the minimum energy solution of the Ising model [11]. Coherent Ising Machines (CIMs) is an example of Ising machines that exploit mixed-state density operators in a quantum oscillator network [14]. A specific type of CIM called the chaotic amplitude control (CAC), or CIM-CAC in short, seems to provide some advantages over other types of CIM [9]. We use a Python-based simulation library [4] to simulate CIM-CAC.

Coupled differential equations determine the time evolution of CIMs. In the case of CIM-CAC, the spin variables x_i and auxiliary variables e_i satisfy

$$\begin{aligned}
\frac{dx_i}{dt} &= (p-1)x_i - \mu x_i^3 + \beta e_i \sum_{j=1}^{N} J_{ij} x_j \\
\frac{de_i}{dt} &= -\xi \left(x_i^2 - a\right) e_i \\
a(t) &= \alpha + \rho \tanh(\delta \Delta H(t)) \\
\xi &= \Gamma(t - t_c)
\end{aligned}$$

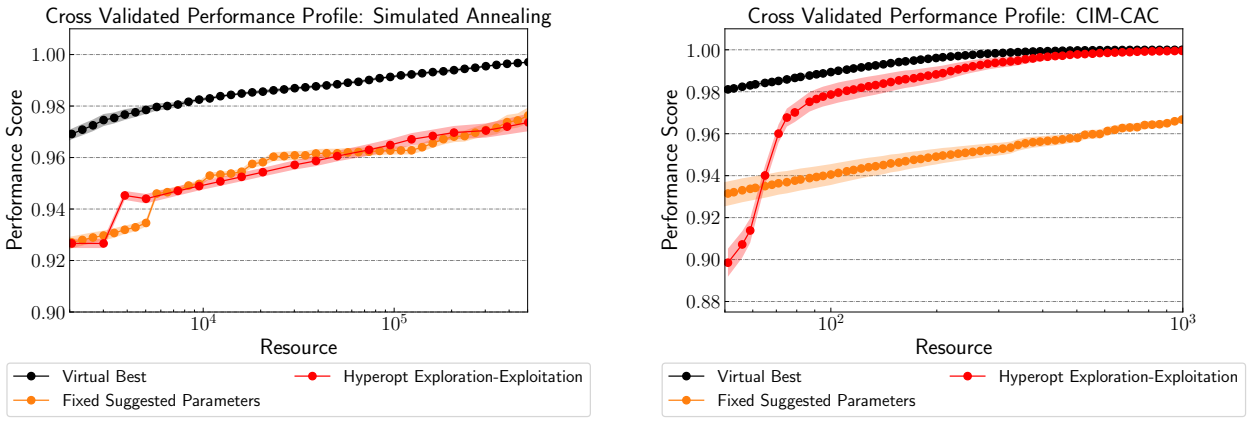

Figure 2: The performance profile for the Parallel Tempering implementation of PySA (left) and the Chaotic Amplitude Control simulation of the Coherent Ising Machine (right). The profile was inferred for each case from 10 test-train splits of 50 Wishart instances with $N = 50$ and $\alpha = 0.5$. The profiles of the virtual best baseline are shown, a `Hyperopt`-driven exploration-exploitation strategy, and the fixed best parameters suggested from the experiments.

2.4 Results

This section presents the results of applying the Window Sticker framework to a class of binary optimization problem instances, using Parallel Tempering and a Chaotic Amplitude Control (CAC) simulation of a Coherent Ising Machine (CIM). We specifically choose problems from the Wishart planted ensemble.

For each technology, we analyze the performance of the solver on 50 randomly chosen instances from the Wishart planted ensemble at various values of the problem size N (i.e., the number of binary variables) and of the parameter α, which defines the ensemble and controls the difficulty of solving a typical problem instance from the ensemble. In the following plots, we show the results obtained from projection experiments and hyperopt exploration-exploitation strategy, in addition to offering the best achievable average performance at a given resource, which will call the virtual best baseline.

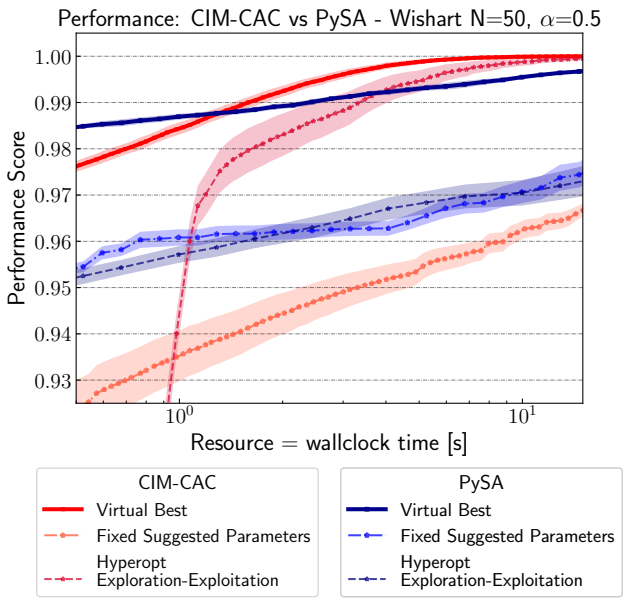

Figure 3: Performance Comparison: The performance profiles of CIM-CAC and PySA overlaid on the same plot, with the resource chosen to be the wall-clock time.

To obtain the performance profile (the Window Sticker), we analyze the performance profiles for 10 test-train splits, with 80% of the instances chosen as training instances and the rest as testing instances. We combine the confidence intervals and the average value (mean or median) for the performance. We quantify the performance using a *normalized performance score* defined as follows:

$$\text{Score} = \frac{\text{(best found solution - random solution)}}{\text{(optimal solution - random solution)}}.$$

Thus, the score ranges from 0, when the solver performs no better than random sampling, to 1, when the solver obtains the optimal solution.

For CIM, we choose the resource to be quantified by the number of shots, and for PySA, we set Resource = Replicas $\times$ sweeps $\times$ shots. The cross-validated performance profiles for both the technologies, obtained from 10 test-train splits of 50 instances chosen from the Wishart planted ensemble corresponding to $N = 50$ and $\alpha = 0.50$ are shown in Fig. 2

To compare the performance profiles of the two technologies, we plot them both, setting the resource as the wall clock time. We find a crossover value of the resource, after which the exploration-exploitation strategy applied to CIM-CAC outperforms the same for PySA (see Fig. 3).

The framework also returns the best average values of parameters and meta-parameters for each technique and have been plotted in Figs. 4 and 5. These values are intended as suggestions generated by the framework for obtaining the best performance on unseen problem instances. To generate these recommendations, instead of splitting the available problem instances into test and train sets, all the problem instances are treated as train sets.

References

[1] PySA: Fast Simulated Annealing in Native Python. NASA, July 2023.

[2] J. Bergstra, D. Yamins, and D. Cox. Hyperopt: Distributed asynchronous hyper-parameter optimization. *Astrophysics Source Code Library*, pages ascl–2205, 2022.

[3] E. Boros and P. L. Hammer. Pseudo-boolean optimization. *Discrete applied mathematics*, 123(1-3):155–225, 2002.

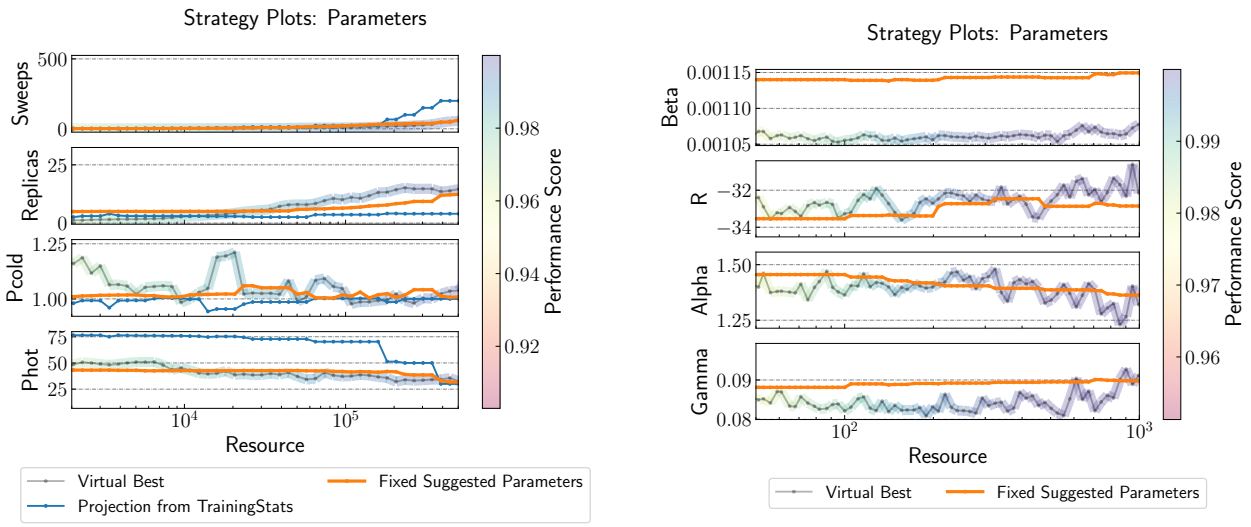

Figure 4: Strategy plots summarizing the findings of the Window Sticker framework, applied to the Parallel Tempering implementation of PySA (left) and the Chaotic Amplitude Control simulation of the Coherent Ising Machine (right). For each case, the profile was inferred from 50 Wishart instances corresponding to $N = 50$ and $\alpha = 0.5$. Shown are the parameters corresponding to the virtual best baseline and parameter recommendations obtained from a `Hyperopt`-driven exploration-exploitation strategy and the fixed best parameters suggested from the experiments.

[4] F. Chen, B. Isakov, T. King, T. Leleu, P. McMahon, and T. Onodera. Cim-optimizer: A simulator of the Coherent Ising Machine, Oct. 2022.

[5] D. Fouskakis and D. Draper. Stochastic optimization: a review. *International Statistical Review*, 70(3):315–349, 2002.

[6] F. Hamze, J. Raymond, C. A. Pattison, K. Biswas, and H. G. Katzgraber. Wishart planted ensemble: A tunably rugged pairwise Ising model with a first-order phase transition. *Physical Review E*, 101(5):052102, May 2020.

[7] K. Hukushima and K. Nemoto. Exchange monte carlo method and application to spin glass simulations. 65(6):1604–1608. Publisher: The Physical Society of Japan.

[8] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, May 1983.

[9] T. Leleu, F. Khoyratee, T. Levi, R. Hamerly, T. Kohno, and K. Aihara. Scaling advantage of chaotic amplitude control for high-performance combinatorial optimization. *Communications Physics*, 4(1):1–10, Dec. 2021. https://www.nature.com/articles/s42005-021-00768-0.

[10] S. Mandrà and H. G. Katzgraber. A deceptive step towards quantum speedup detection. *Quantum Science and Technology*, 3(4):04LT01, July 2018.

[11] N. Mohseni, P. L. McMahon, and T. Byrnes. Ising machines as hardware solvers of combinatorial optimization problems. *Nature Reviews Physics*, 4(6):363–379, June 2022.

[12] D. Perera. Chook, May 2023.

[13] D. Perera, I. Akpabio, F. Hamze, S. Mandra, N. Rose, M. Aramon, and H. G. Katzgraber. Chook – A comprehensive suite for generating binary optimization problems with planted solutions, Mar. 2021.

[14] Z. Wang, A. Marandi, K. Wen, R. L. Byer, and Y. Yamamoto. Coherent Ising machine based on degenerate optical parametric oscillators. *Physical Review A*, 88(6):063853, Dec. 2013. https://link.aps.org/doi/10.1103/PhysRevA.88.063853.

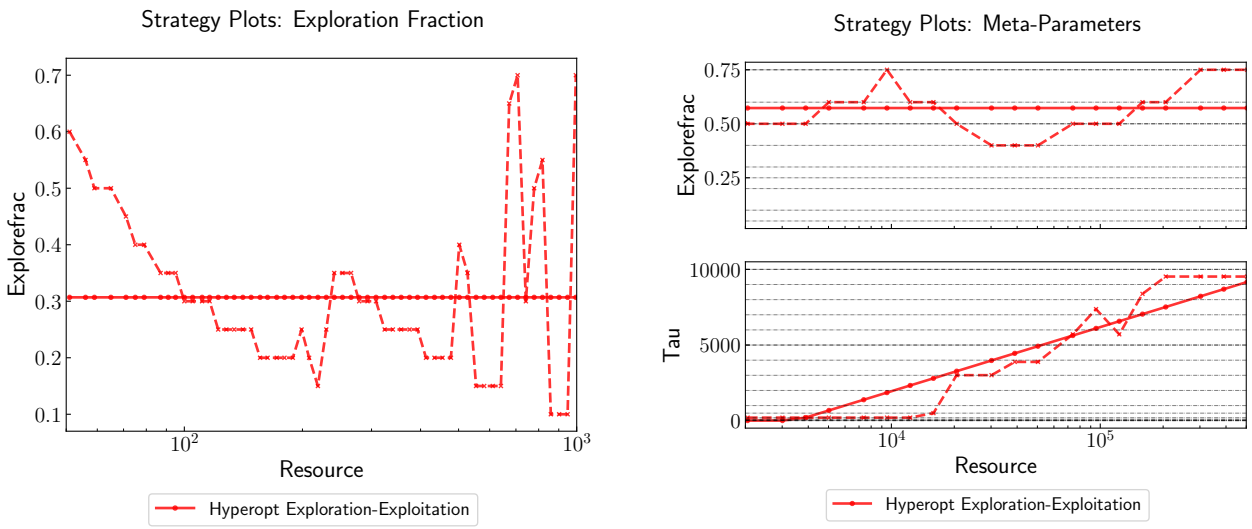

Figure 5: (a) The recommended values of the meta-parameter ExploreFrac obtained generated by the framework, from performance data for 50 instances from the $N = 50$, $\alpha = 0.50$ planted Wishart ensemble. The optimal value of the meta-parameter tau equals 1 throughout the range of resources probed. (b) PySA: Wishart $N = 50$, $\alpha = 0.5$. The horizontal grid lines show the values of meta-parameters that

[15] Z. Zhu, A. J. Ochoa, and H. G. Katzgraber. Efficient Cluster Algorithm for Spin Glasses in Any Space Dimension. *Physical Review Letters*, 115(7):077201, Aug. 2015.

A Parameters used

- Sweeps: qLogUniform$(10^0, 10^4)$
- Replicas:qUniform$(1, 128)$
- pCold: max(logNormal$(10^0, 10^1), 0.01)$
- pHot: max(Normal$(50, 10), 0.1)$
- $\tau = [10, 20, 50, 100, 200, 500, 1000, 2000, 5000, 10000]$
- frac $= [0.05, 0.1, 0.2, 0.3, 0.5, 0.6, 0.75]$

```
budgets = list(range(50, 1000, 5))
taus = list(range(11, 501, 5))
exploration_fracs= list(np.arange(0.05, 1.0, 0.05))

'beta' : hp.uniform('beta', min(beta*0.5, beta*1.5), max(beta*0.5, beta*1.5)),
'r' : hp.uniform('r', min(r*0.1, r*10.0), max(r*0.1, r*10.0)),
'gamma' : hp.uniform('gamma', min(gamma*0, gamma*2.0), max(gamma*0, gamma*2.0)),
'alpha' : hp.uniform('alpha', min(alpha*0.1, alpha*10.0), max(alpha*0.1, alpha*10.0))

With nominal values:
time_step=0.00625,
r=-10.0,
alpha=0.25,
beta=0.0020,
gamma=0.08,
```

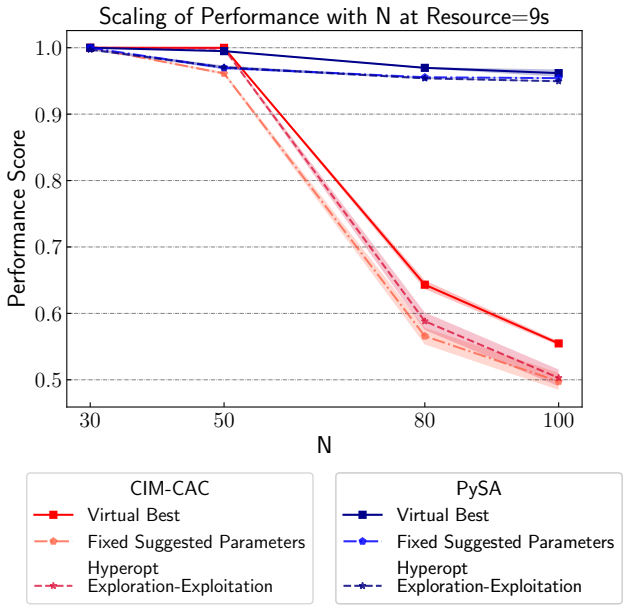

Figure 6: Wishart $N = 50$, $\alpha = 0.5$. Scaling of performance for both technologies.

```
delta=10,
mu=0.5,
rho=5,
tau=2000,
noise=0.5,
T = 5000
```