A Consistency of `Bounce`

In this section, we prove the consistency of the `Bounce` algorithm. The proof is based on Papenmeier et al. [48] and Eriksson and Poloczek [20].

Theorem 1 (`Bounce` consistency)**.** *With the following definitions*

Def. 1. $(\boldsymbol{x}_k)_{k=1}^{\infty}$ *is a sequence of points of decreasing function values;*
Def. 2. $\boldsymbol{x}^* \in \arg\min_{\boldsymbol{x}\in\mathcal{X}}$ *is a minimizer of* f *in* $\mathcal{X}$*;*

and under the following assumptions:

Ass. 1. D *is finite;*
Ass. 2. f *is observed without noise;*
Ass. 3. The range of f *is bounded in* $\mathcal{X}$*, i.e.,* $\exists C \in \mathbb{R}_{++}$ *s.t.* $|f(\boldsymbol{x})| < C \ \forall \boldsymbol{x} \in \mathcal{X}$*;*
Ass. 4. For at least one of the minimizers $\boldsymbol{x}_i^*$ *the (partial) assignment corresponding to the continuous variables lies in a (continuous) region with positive measure;*
Ass. 5. One `Bounce` *reached the input dimensionality* D*, the continuous elements of the initial points* $\{\boldsymbol{x}_{\mathit{cont}_i}\}_{n=1}^{n_{\mathit{init}}}$ *after each TR restart are chosen*
  (a) uniformly at random for continuous variables; and
  (b) such that every realization of the combinatorial variables has positive probability;

then the `Bounce` *algorithm finds a global optimum with probability 1, as the number of samples* N *goes to* ∞*.*

Proof. The range of f is bounded per Assumption 3, and `Bounce` only considers a function evaluation a 'success' if the improvement over the current best solution exceeds a certain constant threshold. `Bounce` can only have a finite number of 'successful' evaluations because the range of f is bounded per Assumption 3. For the sake of a contradiction, we suppose that `Bounce` does not obtain an optimal solution as its number of function evaluations $N \to \infty$. Thus, there must be a sequence of failures, such that the TRs in the current target space, i.e., the current subspace, will eventually reach its minimum base length. Recall that in such an event, `Bounce` increases the target dimension by splitting up the 'bins', thus creating a subspace of $(b+1)$-times higher dimensionality. Then `Bounce` creates a new TR that again experiences a sequence of failures that lead to another split, and so on. This series of events repeats until the embedded subspace eventually equals the input space and thus has dimensionality D. See lines $12 - 16$ in Algorithm 1 in Sect. 3.

Still supposing that `Bounce` does not find an optimum in the input space, there must be a sequence of failures such that the side length of the TR again falls below the set minimum base length, now forcing a restart of `Bounce`. Recall that at every restart, `Bounce` samples a fresh set of initial points uniformly at random from the input space; see line 18 in Algorithm 1. Therefore, with probability 1, a random sample will eventually be drawn from any subset $\mathcal{Y} \subseteq \mathcal{X}$ with positive Lebesgue measure ($\nu(\mathcal{Y}) > 0$):

$$1 - \lim_{k\to\infty} \left(1 - \mu(\mathcal{Y})\right)^k = 1, \tag{2}$$

where μ is the uniform probability measure of the sampling distribution that `Bounce` employs for initial data points upon restart [91].

Let

$$\alpha = \inf\left\{t : \nu\left[x \in \mathcal{X} \ \mid \ f(x) < t\right] > 0\right\}$$

denote the essential infimum of f on $\mathcal{X}$ with ν being the Lebesgue measure [91].

Following Solis and Wets [91], we define the optimality region, i.e., the set of points whose function value is larger by at most ε than the essential infimum:

$$R_{\varepsilon,M} = \{x \in \mathcal{X} \mid f(x) < \alpha + \varepsilon\}$$

with $\varepsilon > 0$ and $M < 0$. Because of Ass. 4, at least one optimal point lies in a region of positive measure that is continuous for the continuous variables. Therefore, we have that $\alpha = f(\boldsymbol{x}^*)$. Note that this is also the case if the domain of f only consists of combinatorial variables (Ass. 5). Then, $R_{\varepsilon,M} = \{\boldsymbol{x} \in \mathcal{X} \mid f(\boldsymbol{x}) < f(\boldsymbol{x}^*) + \varepsilon\}$.

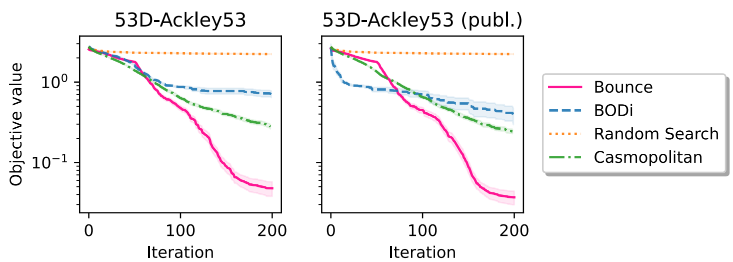

Figure 7: `Bounce` the other algorithms on the synthetic `Ackley53` benchmark function. `Bounce` outperforms all other algorithms and quickly finds excellent solutions. `BODi`'s performance degrades upon randomization.

Let $(\boldsymbol{x}_k^\star)_{k=1}^\infty$ denote the sequence of best points that `Bounce` discovers with $\boldsymbol{x}_k^\star$ being the best point up to iteration k. This sequence satisfies Def. 1 by construction. Note that $\boldsymbol{x}_k^\star \in R_{\varepsilon,M}$ implies that $\boldsymbol{x}_{k'}^\star \in R_{\varepsilon,M}$ for all $k' \geq k+1$ [91] because observations are noise-free. Then,

$$\begin{aligned}\mathbb{P}\left[\boldsymbol{x}_k^\star \in R_{\varepsilon,M}\right] &= 1 - \mathbb{P}\left[\boldsymbol{x}_k^\star \in \mathcal{X} \setminus R_{\varepsilon,M}\right] \\ &\geq 1 - \left(1 - \mu(R_{\varepsilon,M})\right)^k,\end{aligned}$$

and,

$$1 \geq \lim_{k\to\infty} \mathbb{P}\left[\boldsymbol{x}_k^\star \in R_{\varepsilon,M}\right] \geq \underbrace{1 - \lim_{k\to\infty}\left(1 - \mu(R_{\varepsilon,M})\right)^k}_{=1,\text{ Eq. (2)}} = 1,$$

i.e., $\boldsymbol{x}_k^\star$ eventually falls into the optimality region [91]. By letting $\varepsilon \to 0$, $\boldsymbol{x}_k^\star$ converges to the global optimum with probability 1 as $k \to \infty$.

□

B Additional experiments

We compare `Bounce` to the other algorithms on three additional benchmark problems: `Ackley53` and `MaxSAT60` [18]. Moreover, we run two additional studies to further investigate the performance of `Bounce`. First, we run `Bounce` on a set of continuous problems from Papenmeier et al. [48] to showcase the performance and scalability of `Bounce` on purely continuous problems. We then present a "low-sequency" version of `Bounce` to showcase how such a version can outperform its competitors on the original benchmarks by introducing a bias towards low-sequency solutions.

B.1 `Bounce` and other algorithms on additional benchmarks

B.1.1 The synthetic `Ackley53` benchmark function

`Ackley53` is a 53-dimensional function with 50 binary and 3 continuous variables. Wan et al. [61] discretized 50 continuous variables of the orginal Ackley function, requiring these variables to be either zero or one. This benchmark was designed such that the optimal value of 0.0 is at the origin $\boldsymbol{x} = (0, \ldots, 0)$. Here, we perturb the optimal assignment of combinatorial variables by flipping each binary variable with probability $1/2$. Figure 7 summarizes the performances of the algorithms. `Bounce` outperforms all other algorithms and proves to be robust to the location of the optimum point. `Casmopolitan` is a distanced runner-up. `BODi` initially outperforms `Casmopolitan` on the published benchmark version but falls behind later.

B.1.2 Contamination control

The `Contamination` benchmark models a supply chain with 25 stages [32]. At each stage, a binary decision is made whether to quarantine food that has not yet been contaminated. Each such intervention is costly, and the goal is to minimize the number of contaminated products and prevention cost [6, 45]. Figure 8 shows the performances of the algorithms.

`Bounce`, `Casmopolitan`, and `BODi` all produce solutions of comparable objective value. `Bounce` and `Casmopolitan` find better solutions than `BODi` initially, but after about 100 function evaluations, the solutions obtained by the three algorithms are typically on par.

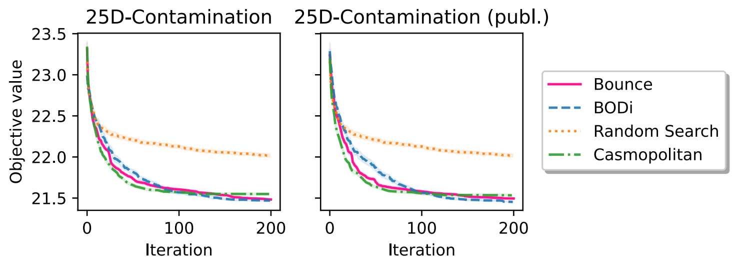

Figure 8: `Bounce` and the other algorithms on the 25-dimensional contamination problem. `Bounce` performs on par with `Casmopolitan` and `BODi` on both versions of the benchmark.

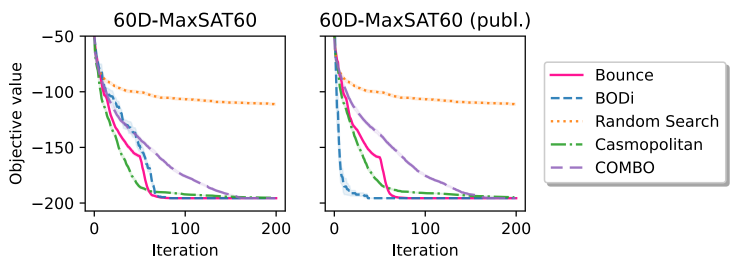

Figure 9: `Bounce` and other algorithms on the 60-dimensional weighted maximum satisfiability problem. `Bounce` is the first to find an optimal solution (left). On the published version (right), `Bounce` comes in second after `BODi`.

B.1.3 The `MaxSAT60` benchmark

`MaxSAT60` is a 60-dimensional, weighted instance of the Maximum Satisfiability (MaxSAT) problem. MaxSAT is a notoriously hard combinatorial problem that cannot be solved in polynomial time (unless $\mathbb{P} = \mathbb{NP}$). The goal is to find a binary assignment to the variables that satisfies clauses of maximum total weight. For every i in $[d]$, this benchmark has one clause of the form x_i with a weight of 1 and 638 clauses of the form $\neg x_i \lor \neg x_j$ with a weight of 61. Following [18, 45, 61], we normalize these weights to have zero mean and unit standard deviation. This normalization causes the one-variable clauses to have a *negative weight*, i.e., the function value improves if such a clause is not satisfied, which is atypical behavior for a MaxSAT problem. Since the clauses with two variables are satisfied for $x_i = x_j = 0$ and the clauses with one variable of negative weights are never satisfied for $x_i = 0$, the normalized benchmark version has a global optimum at $\boldsymbol{x}^* = (0, \ldots, 0)$ by construction. The problem's difficulty is finding an assignment for variables such that *all* two-variable clauses are satisfied and *as many* one-variable clauses as possible is not captured by normalized weights.

Figure 9 summarizes the performances of the algorithms. The general version that attains the global optimum for a randomly selected binary assignment is shown on the left. The special case where the global optimum is set to the all-zero assignment is shown on the right.

We observe that `Bounce` requires the smallest number of samples to find an optimal assignment in general, followed by `BODi` and `Casmopolitan`. Only in the special case where the optimum is the all-zero assignment, `BODi` ranks first, confirming the corresponding result in Deshwal et al. [18].

B.2 Parallel evaluations on continuous problems

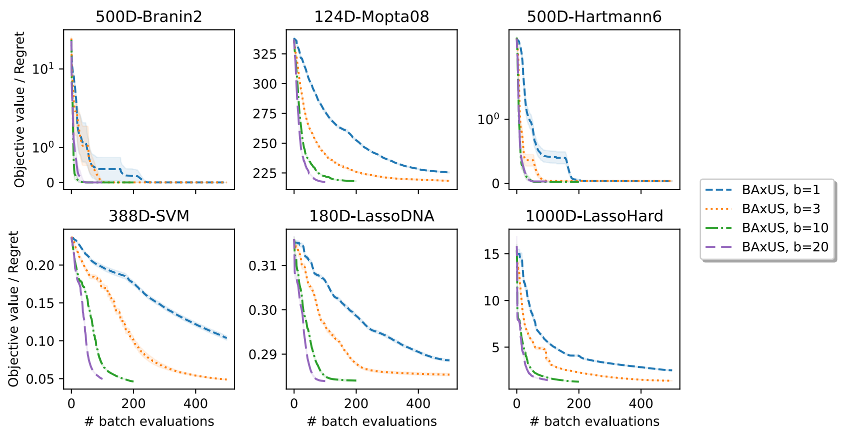

Figure 10: Bounce on continuous problems with different batch sizes.

To showcase the performance and scalability of Bounce, we run it on a set of continuous problems from Papenmeier et al. [48]. The 124-dimensional Mopta08 benchmark is a constrained vehicle optimization problem. We adopt the soft-constrained version from Eriksson et al. [21], Eriksson and Jankowiak [74]. The 388-dimensional soft-constrained SVM problem [74] concerns the classification performance with an SVR on the slice localization dataset. The 180-dimensional LassoDNA benchmark [90] is a sparse regression problem on a real-world dataset, the 1000-dimensional LassoHard benchmark optimizes over a synthetic dataset. The 500-dimensional Branin2 and Hartmann6 problems are versions of the 2- and 6-dimensional benchmark problems where additional dimensions with no effect on the function value were added.

We set the number of function evaluations to $\max(2000, 500B)$ for a batch size of B and configure Bounce such that it reaches the input dimensionality after 500 function evaluations. Figure 10 shows the simple regret for the synthetic Branin2 and Hartmann6 problems, and the best function value obtained after a given number of batch evaluations for the remaining problems: Mopta08, SVM, LassoDNA, and LassoHard.

We observe that Bounce always benefits from more parallel function evaluations. The difference between smaller batch sizes such as the difference between $B = 1$ and $B = 3$ or $B = 3$ and $B = 10$ is more remarkable than the difference between larger batch sizes like $B = 10$ and $B = 20$. On SVM and LassoDNA, parallel function evaluations prove especially effective. Here, the optimization performance improves drastically. We conclude that a small number of parallel function evaluations already helps increase the optimization performance considerably.

On the synthetic Branin2 and Hartmann6 problems, Bounce quickly converges to the global optimum. Here, we see that a larger number of parallel function evaluations also helps in converging to a better solution.

B.3 Low-sequency version of Bounce

We show how we can bias Bounce towards low-sequency solutions. We do this by removing the random signs (for binary and continuous variables) and the random offsets (for categorical and ordinal variables) from the Bounce embedding. We conduct this study to show a) that Bounce is able to outperform BODi on the unmodified versions of the benchmark problems if we introduce a similar bias towards low-sequency solutions, and b) that the random signs empirically show to remove biases towards low-sequency solutions. However, we want to emphasize that the results of this section are not representative of the performance of Bounce on arbitrary real-world problems. Nevertheless, if one knows that the problem at hand has a low-sequency structure, then Bounce can be configured to exploit this structure and outperform BODi.

Figure 11 shows the results of the low-sequency version of Bounce on the original benchmarks from Section 4. We observe that Bounce outperforms BODi and the other algorithms on the unmodified

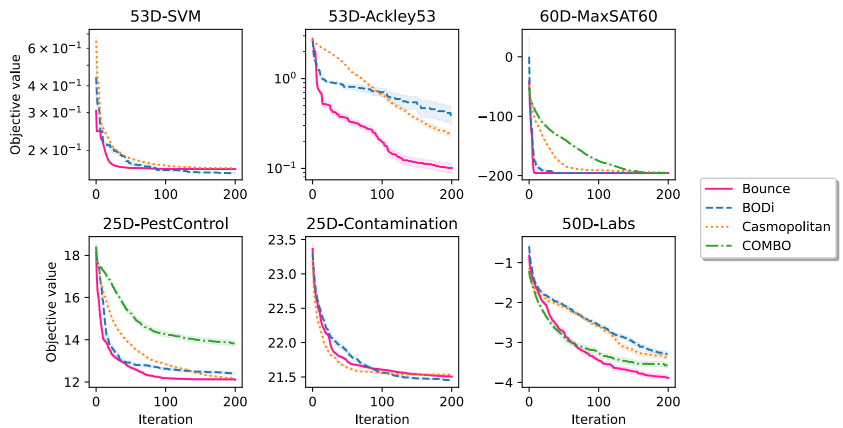

Figure 11: 'Low-sequency' version of `Bounce` on the **original** benchmarks from Section 4: with a bias towards low-sequency solutions, `Bounce` outperforms `BODi` on the original versions of the benchmark problems.

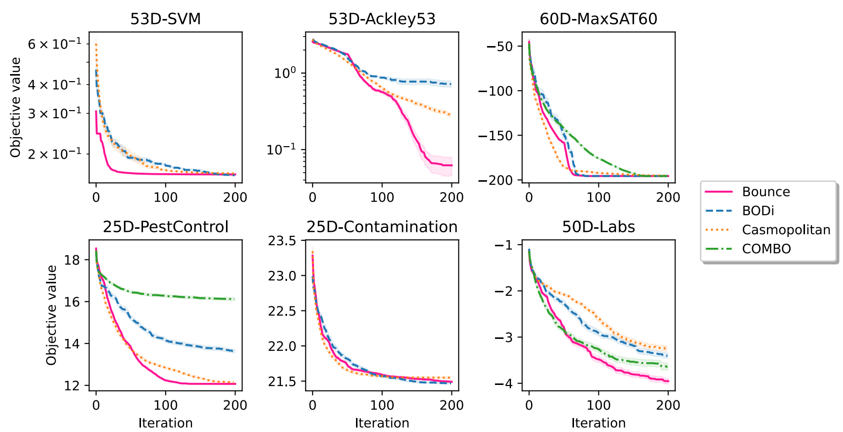

Figure 12: "Low-sequency" version of `Bounce` on the **modified** benchmarks from Section 4.

versions of the benchmark problems. This shows that `Bounce` can outperform `BODi` on the unmodified version of the benchmarks if we introduce a similar bias towards low-sequency solutions.

Figure 12 shows the results of the low-sequency version of `Bounce` on the flipped benchmarks from Section 4. The low-sequency version of `Bounce` is robust towards randomization of the optimal point.

C Implementation details

We implement `Bounce` in Python using the `BoTorch` [5] and `GPyTorch` [77] libraries. For the GP model, we use as similar construction as the CoCaBo kernel [53]. We model the continuous and combinatorial variables with two separate kernels where we use automatic relevance determination (ARD) for the continuous variables and the same lengthscale for the combinatorial variables. We also simply use a Matérn kernel for the combinatorial variables. Following Ru et al. [53], we use a mixture of the sum and the product kernel:

$$k(\boldsymbol{x}, \boldsymbol{x}') = \lambda k_{\text{cmb}}(\boldsymbol{x}_{\text{cmb}}, \boldsymbol{x}'_{\text{cmb}}) k_{\text{cnt}}(\boldsymbol{x}_{\text{cnt}}, \boldsymbol{x}'_{\text{cnt}}) + (1-\lambda)(k_{\text{cmb}}(\boldsymbol{x}_{\text{cmb}}, \boldsymbol{x}'_{\text{cmb}}) + k_{\text{cnt}}(\boldsymbol{x}_{\text{cnt}}, \boldsymbol{x}'_{\text{cnt}})),$$

where $\boldsymbol{x}_{\text{cnt}}$ and $\boldsymbol{x}_{\text{cmb}}$ are the continuous and combinatorial variables in $\boldsymbol{x}$, respectively, and λ is between 0 and 1. The kernel for the continuous variables, k_{cnt} is an ARD-Matérn kernel with $\nu = 2.5$, and the kernel for the combinatorial variables, k_{cmb} is a Matérn kernel without ARD with $\nu = 2.5$. The trade-off parameter λ is learned jointly with the other hyperparameters during the likelihood maximization.

We employ a $\Gamma(1.5, 0.1)$ prior on the lengthscales of both kernels and a $\Gamma(1.5, 0.5)$ prior on the signal variance. We further use a $\Gamma(1.1, 0.1)$ prior on the noise variance.

Motivated by Wan et al. [61] and Eriksson et al. [21], we use an initial trust region baselength of 40 for the combinatorial variables, and 0.8 for the continuous variables. We maintain two separate TR shrinkage and expansion parameters (γ_{cmb} and γ_{cnt}) for the combinatorial and continuous variables, respectively such that each TR base length reaches its respective minimum of 1 and 2^{-7} after a given number of function evaluations. When `Bounce` finds a better or worse solution, we increase or decrease both TR base lengths.

We use the author's implementations for `COMBO`[1], `BODi`[2], and `Casmopolitan`[3]. We use the same settings as the authors for `COMBO` and `BODi`. For `Casmopolitan`, we use the same settings as the authors for benchmarks reported in Wan et al. [61] and set the initial trust region base length to 40 otherwise.

Due to its high-memory footprint, we ran `BODi` on NVidia A100 80GB GPUs for 300 GPU/h. We ran `Bounce` on NVidia A40 GPUs for 2,000 GPU/h. We ran the remaining methods for 20,000 GPU/h on one core of Intel Xeon Gold 6130 CPUs with 60GB of memory.

C.1 Optimization of the acquisition function

We use different strategies to optimize the acquisition function depending on the type of variables present in a problem.

Continuous problems. For purely continuous problems, we follow a similar approach as Eriksson et al. [21]. In particular, we use the lengthscales of the GP posterior to shape the TR. We use gradient descent to optimize the acquisition function within the TR bounds with 10 random restarts and 512 raw samples. For a batch size of 1, we use analytical EI. For larger batch sizes, we use the `BoTorch` implementation of qEI [5, 66, 88].

Binary problems. For all problems with a combinatorial search space, we use local search to optimize the acquisition function. Similar to [61], we use discrete TRs around the current best solution. The TR is defined as the set of all solutions that differ from the current best solution with a certain Hamming distance.

When starting the optimization, we first create a set of $\min(5000, \max(2000, 200 \cdot d_i))$ random solutions. The choice of the number of random solutions is based on Eriksson et al. [21]. For each candidate, we first draw L_i indices uniformly at random from $[d_i]$ without replacement, where L_i is the TR length at the i-th iteration. We then sample d_i values $\in \{0, 1\}$ and set the candidate at the sampled indices to the sampled values. All other values are set to the values of the current best solution. Note that this construction keeps each solution in the TR of the current best solution. We add all neighbors (i.e., points with a Hamming distance of 1) of the current best solution to the set of candidates. This is inspired by [18]. We find the 20 candidates with the highest acquisition function value and use local search to optimize the acquisition function within the TR bounds. At each local search step, we create all neighbors that do not coincide with the current best solution or would violate the TR bounds. We then move the current best solution to the neighbor with the highest acquisition function value. We repeat this process until the acquisition function value does not increase anymore. Finally, we return the best solution found during the local search.

Categorical problems. We use the same approach as for purely binary problems, i.e., we first create a set of random solutions with the same size as for purely binary problems and start the local search on the 20 best initial candidates. We use one-hot encoding for categorical variables.

Suppose the number of categorical variables of the problem is smaller or equal to the current TR length. In that case, we sample, for each candidate and each categorical variable, an index uniformly at random from $[|v_i|]$ where $|v_i|$ is the number of values of the i-th categorical variable. We then set the candidate at the sampled index to 1 and all other values to 0.

If the number of categorical variables of the problem is larger than the current TR length L_i, we first sample L_i categorical variables uniformly at random from $[d_i]$ without replacement. For each initial

[1] `https://github.com/QUVA-Lab/combo`, unspecified license, last access: 2023-05-04

[2] `https://github.com/aryandeshwal/bodi`, no license provided, last access: 2023-05-04

[3] `https://github.com/xingchenwan/casmo`, `MIT` license, last access: 2023-05-04

candidate and each sampled categorical variable, we sample an index uniformly at random, at which we set the categorical variable to 1 and all other values to 0. The values for the variables that were not sampled are set to the values of the current best solution.

As for the binary case, we add all neighbors of the current best solution to the set of candidates and we sample the 20 candidates with the highest acquisition function value.

We then use local search to optimize the acquisition function within the TR bounds while neighbors are created by changing the index of one categorical variable. Again, we repeat until convergence and return the best solution found during the local search.

Ordinal problems. The construction for ordinal problems is similar to the one for categorical problems.

Suppose the number of ordinal variables of the problem is smaller or equal to the current TR length. In that case, we sample an ordinal value uniformly at random to set the ordinal variable for each candidate and each ordinal variable. Otherwise, we choose as many ordinal variables as each candidate's current TR length and sample an ordinal value uniformly at random to set the ordinal variable. We add all neighbors of the current best solution, all solutions where the distance to the current best solution is 1 for one ordinal variable, to the set of candidates. We then sample the 20 candidates with the highest acquisition function value and use local search to optimize the acquisition function within the TR bounds. In the local search, increment or decrement the value of a single ordinal variable.

Mixed problems. Mixed problems are effectively handled by treating every variable type separately. Again, we create a set of initial random solutions where the values for the different variable types are sampled according to the approaches described above. Note that this can lead to solutions lying outside of the TR bounds. We simply remove these solutions and find the 20 best candidates only across the solutions within the TR bounds.

When optimizing the acquisition function, we differentiate between continuous and combinatorial variables. We optimize the continuous variables by gradient descent with the same settings as purely continuous problems. When optimizing, we fix the values for the combinatorial values.

We use local search to optimize the acquisition function for the combinatorial variables. In this step, we fix the values for the continuous variables and only optimize the combinatorial variables. We create the neighbors by creating neighbors within Hamming distance of 1 for each combinatorial variable type and then combining these neighbors. Again, we repeat the local search until convergence.

We do five interleaved steps, starting with the continuous variables and ending with the combinatorial variables.

D Additional analysis of `BODi` and `COMBO`

D.1 Analysis of `BODi`

Binary problems. `BODi` [18] is based on the idea of using a dictionary of reference points $\boldsymbol{A} = (\boldsymbol{a}_1, \ldots, \boldsymbol{a}_m)$ to encode a candidate point $\boldsymbol{z}$. In particular, the i-th entry of the m-dimensional embedding $\phi_{\boldsymbol{A}}(\boldsymbol{z})$ is obtained by computing the Hamming-distance between $\boldsymbol{z}$ and $\boldsymbol{a}_i$. Notably, the dimensionality of the embedding m is chosen to be 128 in their experiments [18], which is larger than the dimensionality of the benchmark functions themselves.

The dictionary elements $\boldsymbol{a}_i$ are chosen such that they represent a wide range of *sequencies* where the sequency of a binary string is defined as the number of times the string changes from 0 to 1 and vice versa. Deshwal et al. [18] propose two approaches to generate the dictionary elements: (i) by using binary wavelets, and (ii) by first drawing a Bernoulli parameter $\theta_i \sim \mathcal{U}(0, 1)$ for each $i \in [m]$ and then drawing a binary string $\boldsymbol{a}_i$ from the distribution $\mathcal{B}(\theta_i)$. The latter approach is their preferred method.

We will now show that the probability of a point of sequence zero (i.e., $\boldsymbol{a}_i = \mathbf{0}$ or $\boldsymbol{a}_i = \mathbf{1}$) to be sampled is higher than for arbitrary points. We hypothesize that `BODi` benefits from containing such a point with high probability if the optimal point is also of sequency zero (cf. Section 4.6). Since `BODi`'s performance degrades when randomizing the optimal point, we further hypothesize

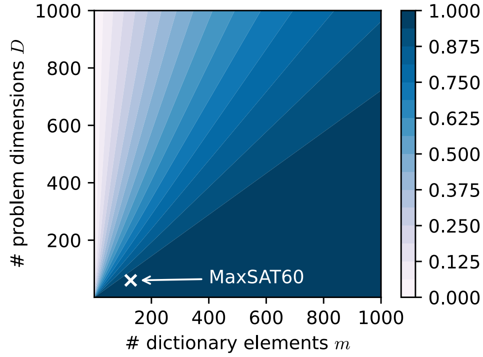

Figure 13: Probabilities of `BODi` to contain a zero-sequency solution for different choices of the dictionary size m and the function dimensionality D.

that `BODi`'s performance on problems with a zero- or low-sequency solution is not representative of problems with an arbitrary solution.

Deshwal et al. [18] choose to have 128 dictionary elements. Given a Bernoulli parameter θ_i, the probability that the i-th dictionary point $\boldsymbol{a}_i$ is a point of sequency zero is given by $\theta_i^D + (1-\theta_i)^D$:

$$\mathbb{P}(\text{“zero sequency”} \mid \theta_i) = \prod_{i=1}^{m} \theta_i^D + (1-\theta_i)^D$$

Then, since θ_i follows a uniform distribution, the overall probability for a point of zero sequency is given by

$$\begin{aligned}\mathbb{P}(\text{“zero sequency”}) &= 1 - \underbrace{\prod_{i=1}^{m} \int_0^1 \left(1 - \theta_i^D - (1-\theta_i)^D\right) \underbrace{p(\theta_i)}_{=1} d\theta_i}_{\text{prob. of } m \text{ times not zero sequency}} \\ &= 1 - \prod_{i=1}^{m} \left(\theta_i - \frac{\theta_i^{D+1}}{D+1} + \frac{(1-\theta_i)^{D+1}}{D+1} \right) \Bigg|_{\theta_i = 1} \\ &= 1 - \left(1 - \frac{2}{D+1}\right)^m,\end{aligned}$$

i.e., the probability of at least one dictionary element being of sequency zero is $1 - \left(1 - \frac{2}{D+1}\right)^m$. The probability of `BODi`'s dictionary to contain a zero-sequency point increases with the number of dictionary elements m and decreases with the function dimensionality D (see Figure 13).

For instance, for the 60-dimensional `MaxSAT60` benchmark, the probability that at least one dictionary element is of sequency zero is $1 - \left(1 - \frac{2}{60+1}\right)^{128} \approx 0.986$ (see Figure 13).

Note that there is at least one point $\boldsymbol{z}^*$ with a probability of $\leq 1/2^d$ to be drawn. The probability of the dictionary containing that $\boldsymbol{z}^*$ is less than or equal $1 - \left(1 - \frac{1}{2^d}\right)^m$ which is already less than 0.01 for $d = 14$ and $m = 128$. In Section 4, we have shown that randomizing the optimal point structure leads to performance degradation for `BODi`. We hypothesize this is due to the reduced probability of the dictionary containing the optimal point after randomization.

Categorical problems. We calculate the probability that `BODi` contains a vector in its dictionary where all elements are the same. For categorical problems, `BODi` first samples a vector $\boldsymbol{\theta}$ from the $\tau_{\max}$-simplex $\Delta^{\tau_{\max}}$ for each vector $\boldsymbol{a}_i$ in the dictionary, with $\tau_{\max}$ being the maximum number of categories across all categorical variables of a problem. We assume that all variables have the same number of categories as is the case for the benchmarks in Deshwal et al. [18]. Let τ be the number of categories of the variables. For each element in $\boldsymbol{a}_i$, `BODi` then draws a value from the categorical

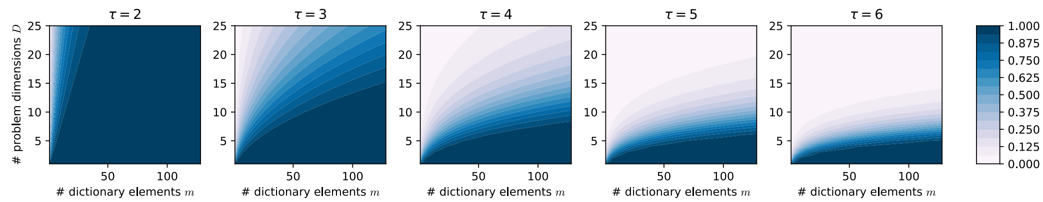

Figure 14: Probabilities of BODi's dictionary to contain at least one categorical point where each category has the same value. The probability increases with the number of dictionary elements m but decreases with the number of categories τ and the number of problem dimensions D.

distribution with probabilities $\boldsymbol{\theta}$. While line 7 in Algorithm 5 in Deshwal et al. [18] might suggest that the elements in $\boldsymbol{\theta}$ are shuffled for every element in $\boldsymbol{a}_i$, we observe that $\boldsymbol{\theta}$ remains fixed based on the implementation provided by the authors[4]. The random resampling of elements from $\boldsymbol{\theta}$ is probably only used for benchmarks where the number of realizations differs between categorical variables.

Then, for a fixed $\boldsymbol{\theta}$, the probability that all D elements in $\boldsymbol{a}_i$ for any i are equal to some fixed value $t \in \{1, \ldots, \tau\}$ is given by θ_t^d. The probability that, for any of the m dictionary elements, all D elements in $\boldsymbol{a}_i$ are equal to some fixed value $t \in \{1, \ldots, \tau\}$ is given by

$$\mathbb{P}(\text{"all one specific category"}) = 1 - \prod_{i=1}^{m} \int (1 - \theta_t^D) p(\theta_t) d\theta_t. \tag{3}$$

We note that $\boldsymbol{\theta}$ follows a Dirichlet distribution with $\boldsymbol{\alpha} = \mathbf{1}$ [85]. Then, θ_t is marginally $\text{Beta}(1, \tau - 1)$-distributed [85]. With that, Eq. (3) becomes

$$\begin{aligned}\mathbb{P}(\text{"all one specific category"}) &= 1 - \prod_{i=1}^{m} \mathbb{E}_{\theta_t \sim \text{Beta}(1,\tau-1)} \left[1 - \theta_t^D\right] \\ &= 1 - \prod_{i=1}^{m} 1 - \mathbb{E}_{\theta_t \sim \text{Beta}(1,\tau-1)} \left[\theta_t^D\right]\end{aligned}$$

now, by using the formula $\mathbb{E}[x^D] = \prod_{r=0}^{D-1} \frac{\alpha+r}{\alpha+\beta+r}$ for the D-th raw moment of a $\text{Beta}(\alpha, \beta)$ distribution [85]

$$\begin{aligned}&= 1 - \left(1 - \prod_{r=0}^{D-1} \frac{1+r}{\tau+r}\right)^m \\ &= 1 - \left(1 - \frac{1}{\tau} \cdot \frac{2}{\tau+1} \cdot \ldots \cdot \frac{D}{\tau+D-1}\right)^m \\ &= 1 - \left(1 - \frac{D!\,\tau!}{(\tau+D-1)!}\right)^m\end{aligned}$$

We discussed in Section 4.3 that the PestControl benchmark obtains a good solution at $\boldsymbol{x} = \mathbf{5}$. One could assume that BODi performs well on this benchmark because its dictionary has a high probability to contain this point. However, we observe that the probability is effectively zero for $\tau = 5$, $m = 128$, and $D = 25$ (see Figure 14), which are the choices for the PestControl benchmark in Deshwal et al. [18]. This raises the question of (i) whether our hypothesis is wrong, and (ii) what the reason for BODi's performance degradation on the PestControl benchmark is.

We show that BODi's reference implementation differs from the algorithmic description in an important detail, causing BODi to be considerably more likely to sample category 5 on PestControl (or the "last" category for arbitrary benchmarks) than any other category.

[4] See https://github.com/aryandeshwal/bodi/blob/aa507d34a96407b647bf808375b5e162ddf10664/bodi/categorical_dictionary_kernel.py#L18

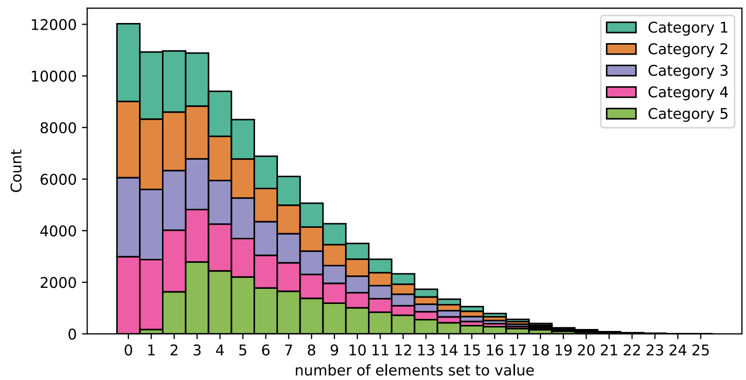

Figure 15: Histograms over the number of dictionary element entries set to each category for 20,000 repetitions of the sampling of dictionary elements for the `PestControl` benchmark. For each of the five categories and each value on the x-axis, the figure shows how often the number of entries in a dictionary element equals the value on the x-axis for the given category. For example, the count for $x = 0$ and category 5 is zero, indicating that all of the 20,000 dictionary points had at least one entry '5'. There is a considerably higher chance for a dictionary element entry to be set to category 5 than to any of the other categories.

In Figure 15, we show five histograms over the number of dictionary elements set to each category. The values on the x-axis give the number of elements in a 25-dimensional categorical vector being set to a specific category. One would expect that the histograms have a similar shape regardless of the category. However, for category 5, we see that more elements are set to this category than for the other categories: The probability of k elements being set to category 5 is almost twice as high as the probability of being set to another category for $k \geq 3$. In contrast, the probability that no element in the vector belongs to category 5 is virtually zero. This behavior is beneficial for the `PestControl` benchmark, which obtained the best value found during our experiments for $\boldsymbol{x}^* = (5, 5, \dots, 5, 1)$ (see Section 4). While we see that the probability of each dictionary entry being set to category 5 is very low, we assume that we sample sufficiently many dictionary elements within a small Hamming distance to the optimizer such that `BODi`'s GP is able to use this information to find the optimizer.

The reason for the oversampling of the last category lies in a rounding issue in the sampling of dictionary elements. In particular, for a given dictionary element $\boldsymbol{a}_i$ and a corresponding vector $\boldsymbol{\theta}$ with $|\boldsymbol{\theta}| = \tau$, for each $i \in \{1, \dots, \tau - 1\}$, Deshwal et al. [18] set $\lfloor D\boldsymbol{\theta}_i \rfloor$ elements to category i. The remaining $D - \sum_{i=1}^{\tau-1} \lfloor D\boldsymbol{\theta}_i \rfloor$ elements are then set to category τ. This causes the last category to be overrepresented in the dictionary elements. For the choices of the `PestControl` benchmark, $D = 25$ and $\tau = 5$, the first four categories had a probability of ≈ 0.1805 while the last one had a probability of ≈ 0.278 for 10^8 simulations[5]. We assume that the higher probability of the last category is the reason for the performance difference between the modified and the unmodified version of the `PestControl` benchmark.

D.2 `COMBO` on categorical problems

On the categorical `PestControl` benchmark, we could observe a similar behavior for `COMBO` [45] as for `BODi`.

[5]The 95% confidence intervals for categories 1–5 are (0.1799, 0.1807), (0.1802, 0.1810), (0.1803, 0.1811), (0.1801, 0.1809), (0.2775, 0.2783). Pairwise Wilcoxon signed-rank tests between categories 1–4 and category 5 gives p values of 0 ($W \approx 4.7 \cdot 10^{10}$ each).

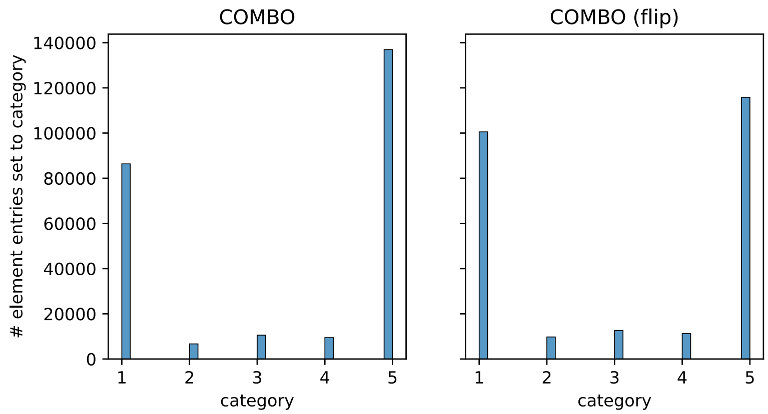

Figure 16: Histograms over the number of dictionary element entries set to each category for 20,000 repetitions of the sampling of dictionary elements for the `PestControl` benchmark. The histogram shows how many element entries of a 25-dimensional dictionary element are set to each of the five categories. There is a considerably higher chance for a dictionary element entry to be set to category 1 or 5 than to one of the other categories.

In Figure 16, we show the histograms over the number of dictionary elements set to each category for both the modified and the unmodified version of the `PestControl` benchmark. We see that the first and the last categories on both versions of the benchmark are overrepresented. As discussed in Section 4, this benchmark attains its best value for $\boldsymbol{x}^* = (5, 5, \dots, 5, 1)$. Therefore, it seems unexpected that `COMBO` sets so many entries to category 1 for the unmodified benchmark version. For the modified benchmark version, this is entirely unexpected as the optimizer has a random structure. Here, one would expect the histogram to be uniform.

We argue that this behavior is at least partially caused by implementation error in the construction of the adjacency matrix and the Laplacian for categorical problems[6]. This error causes categorical variables to be modeled like ordinal variables. According to Oh et al. [45], categorical variables are modeled as a complete graph (see Figure 17).

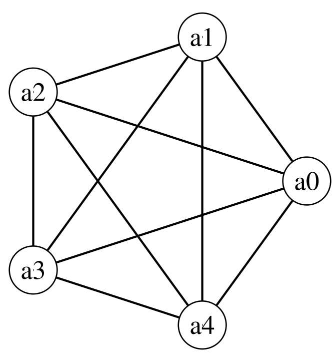

Figure 17: A categorical variable with five categories is modeled is modeled as a complete graph.

However, we find the adjacency matrix for the first category of a categorical variable with five categories is constructed as

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix},$$

[6] `https://github.com/QUVA-Lab/COMBO/blob/9529eabb86365ce3a2ca44fff08291a09a853ca2/COMBO/experiments/test_functions/multiple_categorical.py#L137`, last access: 2023-04-26

which is the adjacency matrix for a path graph with five vertices. We assume that the search space has boundaries due to treating categorical variables as ordinal variables. Due to the high dimensionality of the search space, `COMBO` visits the boundaries of the search space more often than the interior.

E Extended related work

Kim et al. [80] use a random projection matrix to optimize combinatorial problems in a continuous embedded subspace. When evaluating a point, their approach first projects the continuous candidate point to the high-dimensional search space and then rounds to the next feasible combinatorial solution. Deshwal et al. [71] build up on `COMBO` [45] by establishing a closed-form expression for the diffusion kernel and proposing kernels tailored for mixed spaces.

Monte-Carlo Tree Search. Recent approaches employed Monte-Carlo Tree Search (MCTS) to reduce the complexity of the problem. Wang et al. [95] use MCTS to learn a partitioning of the continuous search space to focus the search on promising regions in the search space. Song et al. [92] use a similar approach but instead of learning promising regions in the search space, they assume an axis-aligned active subspace and use MCTS to select important variables.

Non-linear embeddings. Linear embeddings and random linear embeddings [37, 42, 48, 64, 70] only require little or no training data to construct the embedding but assume a linear subspace. Non-linear embeddings allow to learn more complex embeddings but often require more training data. Lu et al. [82] and Maus et al. [83] use variational autoencoders (VAEs) to learn a non-linear embedding of highly-structured input spaces. Tripp et al. [93] also use a VAE to learn a non-linear subspace of a combinatorial search space. By using a re-weighting scheme that puts more emphasis on promising points in the search space, they tailor the embedding toward optimization problems.

Other combinatorial problems. Deshwal et al. [72] propose two algorithms for *permutation spaces* which occur in problems such as compiler optimization [29] and pose a special challenge due to the superexponential explosion of solutions.

References

[5] Maximilian Balandat, Brian Karrer, Daniel R. Jiang, Samuel Daulton, Benjamin Letham, Andrew Gordon Wilson, and Eytan Bakshy. BoTorch: A Framework for Efficient Monte-Carlo Bayesian Optimization. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 33, 2020.

[6] Ricardo Baptista and Matthias Poloczek. Bayesian optimization of combinatorial structures. In *International Conference on Machine Learning*, pages 462–471. PMLR, 2018.

[70] Mohamed Amine Bouhlel, Nathalie Bartoli, Rommel G. Regis, Abdelkader Otsmane, and Joseph Morlier. Efficient global optimization for high-dimensional constrained problems by using the Kriging models combined with the partial least squares method. *Engineering Optimization*, 50(12):2038–2053, 2018.

[71] Aryan Deshwal, Syrine Belakaria, and Janardhan Rao Doppa. Bayesian optimization over hybrid spaces. In *International Conference on Machine Learning*, pages 2632–2643. PMLR, 2021.

[72] Aryan Deshwal, Syrine Belakaria, Janardhan Rao Doppa, and Dae Hyun Kim. Bayesian optimization over permutation spaces. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 6515–6523, 2022.

[18] Aryan Deshwal, Sebastian Ament, Maximilian Balandat, Eytan Bakshy, Janardhan Rao Doppa, and David Eriksson. Bayesian Optimization over High-Dimensional Combinatorial Spaces via Dictionary-based Embeddings. In *International Conference on Artificial Intelligence and Statistics*, pages 7021–7039. PMLR, 2023.

[74] David Eriksson and Martin Jankowiak. High-dimensional Bayesian optimization with sparse axis-aligned subspaces. In Cassio de Campos and Marloes H. Maathuis, editors, *Proceedings of the Thirty-Seventh Conference on Uncertainty in Artificial Intelligence*, volume 161 of *Proceedings of Machine Learning Research*, pages 493–503. PMLR, 27–30 Jul 2021.

[20] David Eriksson and Matthias Poloczek. Scalable Constrained Bayesian Optimization. In *Proceedings of The 24th International Conference on Artificial Intelligence and Statistics*, volume 130 of *Proceedings of Machine Learning Research*, pages 730–738. PMLR, 13–15 Apr 2021.

[21] David Eriksson, Michael Pearce, Jacob Gardner, Ryan D Turner, and Matthias Poloczek. Scalable Global Optimization via Local Bayesian Optimization. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 32, pages 5496–5507, 2019.

[77] Jacob R Gardner, Geoff Pleiss, David Bindel, Kilian Q Weinberger, and Andrew Gordon Wilson. GPyTorch: Blackbox Matrix-Matrix Gaussian Process Inference with GPU Acceleration. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 31, 2018.

[29] Erik Hellsten, Artur Souza, Johannes Lenfers, Rubens Lacouture, Olivia Hsu, Adel Ejjeh, Fredrik Kjolstad, Michel Steuwer, Kunle Olukotun, and Luigi Nardi. BaCO: A Fast and Portable Bayesian Compiler Optimization Framework. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2023.

[32] Yingjie Hu, JianQiang Hu, Yifan Xu, Fengchun Wang, and Rong Zeng Cao. Contamination control in food supply chain. In *Proceedings of the 2010 Winter Simulation Conference*, pages 2678–2681. IEEE, 2010.

[80] Jungtaek Kim, Seungjin Choi, and Minsu Cho. Combinatorial bayesian optimization with random mapping functions to convex polytopes. In *Uncertainty in Artificial Intelligence*, pages 1001–1011. PMLR, 2022.

[37] Ben Letham, Roberto Calandra, Akshara Rai, and Eytan Bakshy. Re-Examining Linear Embeddings for High-Dimensional Bayesian Optimization. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 33, pages 1546–1558, 2020.

[82] Xiaoyu Lu, Javier Gonzalez, Zhenwen Dai, and Neil D Lawrence. Structured variationally auto-encoded optimization. In *International conference on machine learning*, pages 3267–3275. PMLR, 2018.

[83] Natalie Maus, Haydn Thomas Jones, Juston Moore, Matt Kusner, John Bradshaw, and Jacob R. Gardner. Local latent space bayesian optimization over structured inputs. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 35, 2022.

[42] Amin Nayebi, Alexander Munteanu, and Matthias Poloczek. A framework for Bayesian Optimization in Embedded Subspaces. In *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research (PMLR)*, pages 4752–4761, 09–15 Jun 2019.

[85] Kai Wang Ng, Guo-Liang Tian, and Man-Lai Tang. Dirichlet and related distributions: Theory, methods and applications. 2011.

[45] Changyong Oh, Jakub Tomczak, Efstratios Gavves, and Max Welling. Combinatorial Bayesian Optimization using the Graph Cartesian Product. *Advances in Neural Information Processing Systems (NeurIPS)*, 32, 2019.

[48] Leonard Papenmeier, Luigi Nardi, and Matthias Poloczek. Increasing the Scope as You Learn: Adaptive Bayesian Optimization in Nested Subspaces. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 35, 2022.

[88] Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra. Stochastic backpropagation and approximate inference in deep generative models. In *International conference on machine learning*, pages 1278–1286. PMLR, 2014.

[53] Binxin Ru, Ahsan Alvi, Vu Nguyen, Michael A Osborne, and Stephen Roberts. Bayesian optimisation over multiple continuous and categorical inputs. In *International Conference on Machine Learning*, pages 8276–8285. PMLR, 2020.

[90] Kenan Šehić, Alexandre Gramfort, Joseph Salmon, and Luigi Nardi. LassoBench: A High-Dimensional Hyperparameter Optimization Benchmark Suite for Lasso. In *First Conference on Automated Machine Learning (Main Track)*, 2022.

[91] Francisco J. Solis and Roger J-B. Wets. Minimization by Random Search Techniques. *Mathematics of Operations Research*, 6(1):19–30, 1981.

[92] Lei Song, Ke Xue, Xiaobin Huang, and Chao Qian. Monte Carlo Tree Search based Variable Selection for High Dimensional Bayesian Optimization. *Advances in Neural Information Processing Systems (NeurIPS)*, 35, 2022.

[93] Austin Tripp, Erik Daxberger, and José Miguel Hernández-Lobato. Sample-efficient optimization in the latent space of deep generative models via weighted retraining. *Advances in Neural Information Processing Systems (NeurIPS)*, 33:11259–11272, 2020.

[61] Xingchen Wan, Vu Nguyen, Huong Ha, Binxin Ru, Cong Lu, and Michael A. Osborne. Think Global and Act Local: Bayesian Optimisation over High-Dimensional Categorical and Mixed Search Spaces. In *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 10663–10674. PMLR, 18–24 Jul 2021.

[95] Linnan Wang, Rodrigo Fonseca, and Yuandong Tian. Learning Search Space Partition for Black-box Optimization using Monte Carlo Tree Search. *Advances in Neural Information Processing Systems (NeurIPS)*, 33:19511–19522, 2020.

[64] Ziyu Wang, Frank Hutter, Masrour Zoghi, David Matheson, and Nando de Feitas. Bayesian Optimization in a Billion Dimensions via Random Embeddings. *Journal of Artificial Intelligence Research (JAIR)*, 55:361–387, 2016.

[66] James T Wilson, Riccardo Moriconi, Frank Hutter, and Marc Peter Deisenroth. The reparameterization trick for acquisition functions. *NeurIPS Workshop on Bayesian Optimization*, 2017.