A Motivating Example

Figure 6 shows a graphic of a droplet microfluidic reactor [49] which is our main motivating example.

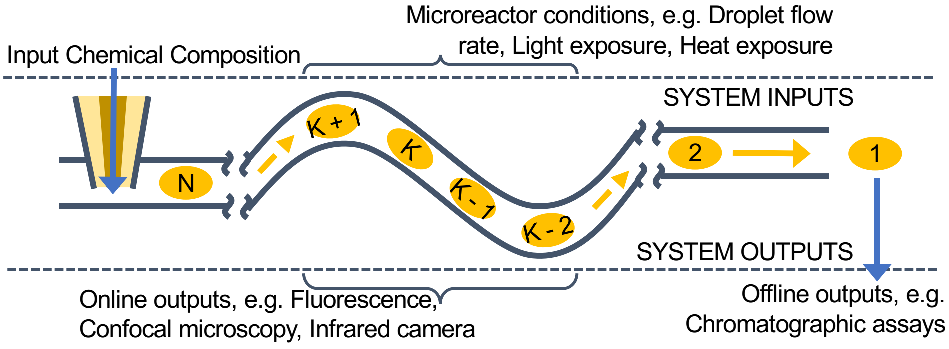

Microreactor conditions, e.g. Droplet flow rate, Light exposure, Heat exposure
Input Chemical Composition
SYSTEM INPUTS
N
K + 1
K
K - 1
K - 2
2
1
SYSTEM OUTPUTS
Online outputs, e.g. Fluorescence, Confocal microscopy, Infrared camera
Offline outputs, e.g. Chromatographic assays

Figure 6: Motivating example. Droplets flow into the micro-reactor where we control conditions such as temperature and flow rate. The cost to change function inputs arises from how adjacent droplets are coupled, e.g. rapidly changing the temperature after droplet K means waiting for system equilibration before taking new measurements. Asynchronicity arises from choosing drops $2, 3, ..., N$ before getting the results of droplet 1.

B General Approach

While we present SnAKe as a detailed algorithm, the ideas behind it are far more general. For example, in Section C we create our batch by taking samples from three different GPs. Algorithm 3 presents a more general algorithm that could be used for any similar task.

Algorithm 3 General Ordering-Based Optimization

```
input: Optimization budget of $T$ samples. Method for creating batch of queries. Method for creating an ordering from a batch of queries. Method for updating paths.
begin: Create initial batch of size $T$ and ordering, $S$
for $t = 1, 2, 3, ..., T$ do
    if there is new information then
        Update surrogate model
        Choose a batch of new points to query
        Create a new path, $\tilde{S}$
        $S \leftarrow \tilde{S}$
    end if
    Choose next query point from ordering: $x_t \leftarrow S_t$
    Evaluate $f(x_t)$
end for
```

C Ypacarai Lake Experiment

A second motivating example for our research comes from optimization requiring spatially continuous exploration. In this section, we examine the problem more closely, and show that SnAKe can easily be extended to optimize multiple black-box functions *simultaneously* in the same search space.

The problem in question is an adaptation of the case study introduced by Samaniego et al. [41]. The application is to use autonomous boats to find the *largest* source of contamination in Lake Ypacarai in Paraguay. Figure 7 shows a visualization of the Lake, and three objectives we want to optimize over. Each objective corresponds to a different measure of water contamination, examples might include pH, turbidity, CO2 levels, and more. In Section 4.3 we explored optimizing over these examples individually.

Table 2: Ypacarai experimental results. We present the average regret from 10 runs ± the standard deviation multiplied by 10^3. MO1, MO2, MO3 represent the different maximums (one for each function). All runs are terminated after the cost exceeds 10 units (approximately 100 km). SnAKe is the only method with good regret in all metrics, as TrEI performs poorly in the first objective, and EIpu performs poorly in the second.

	Regret MO1	Regret MO2	Regret MO3
TrEI	13.636 ± 35.842	0.802 ± 0.828	0.162 ± 0.108
EIpu	0.995 ± 0.661	19.308 ± 55.901	0.212 ± 0.137
SnAKe	0.996 ± 2.743	1.740 ± 4.813	0.064 ± 0.053

In this section, we extend the problem to optimize over all of these *simultaneously*. This is important because it is inefficient to run multiple objective runs when we are exploring the same search space. At every iteration we choose one query x_t, and obtain three outputs $f_1(x_t)$, $f_2(x_t)$, and $f_3(x_t)$ one for each objective.

We model each objective using an independent GP, meaning we can *create Thompson Samples from any objective*. Therefore we extend SnAKe by creating the batch using a mixture of samples from each objective. For the experiment we use the ratio 1:1:1. For TrEI and EIpu, we adapt to the multi-objective case by optimizing the first objective until we have travelled 3.3 units (approximately 33km), after which we change to the subsequent objective. Table 2 shows the results.

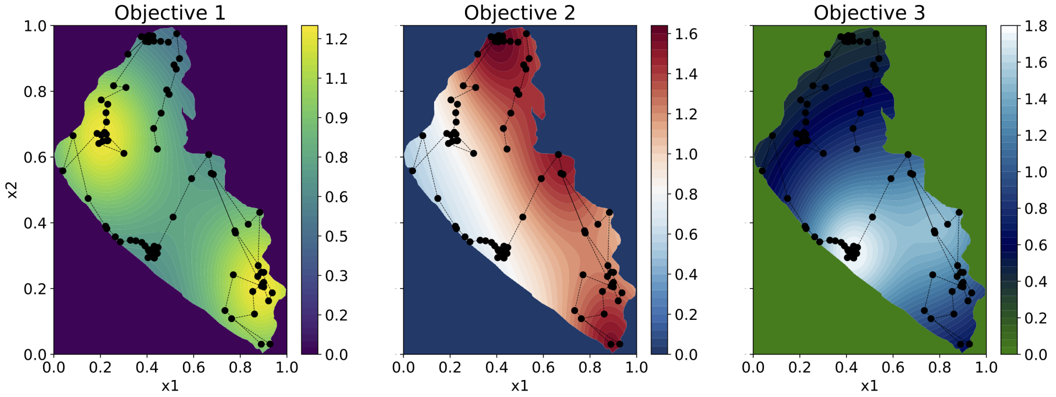


Figure 7: Visualization of Ypacarai Lake, and the corresponding objectives we optimize over. We show an example of an optimization path.

D Empirical Analysis of Escape Probability

D.1 Areas with stationary points

Figure 8 estimates the non-escape probability (see Definition 3.2) from the interval $A = [0.1, 0.2]$. The optimization objective is a bi-modal function. Once we have 15 samples in the interval $[0.1, 0.2]$, we estimate the escape probability to converge to $p \approx 0.74$.

D.2 Areas without stationary points

We now repeat the same experiment as in section D.1, this time we change the interval to $A = [0.0, 0.1]$ which does not contain any stationary points. One can observe a clear difference in the behavior of p_t as we include more information. This time, $p_t \to 0$ very fast.

D.3 Resampling vs Point Deletion

Figure 10 empirically confirms the analysis of Section 3.7 showing the effectiveness of ϵ-Point Deletion, and displaying the effect of increasing the budget from 100 to 250 iterations.

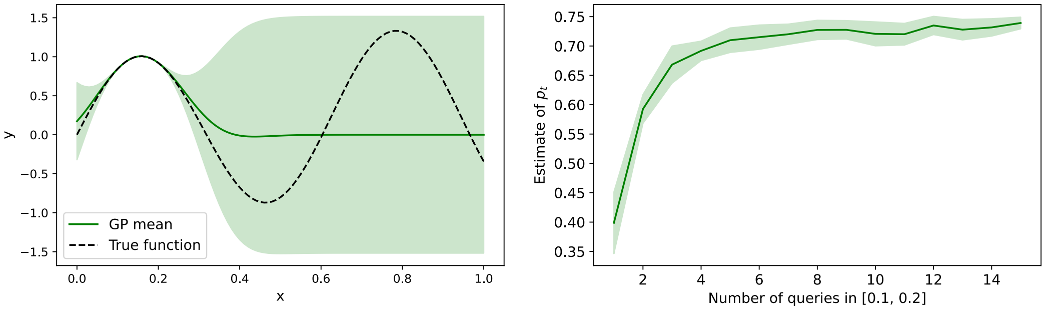


Figure 8: We estimate the probability of non-escape by taking 5000 independent Thompson Samples and counting the number of samples inside A (i.e. the MLE estimator of the Bernoulli distribution). We do this for increasing number of training points in A (which are chosen randomly with a uniform distribution in A). We repeat the experiment 10 times. The left plot shows the underlying function and the Gaussian Process for 15 training points. The right plot shows the evolution of our estimate as we increase training points inside $A = [0.1, 0.2]$ (we plot the mean of each run ± the standard deviation). This example makes it clear that p_t does not converge to zero. Furthermore, it seems to converge to just over 0.7 which a very large probability. This will make fully escaping the local minimum very difficult without Point Deletion.

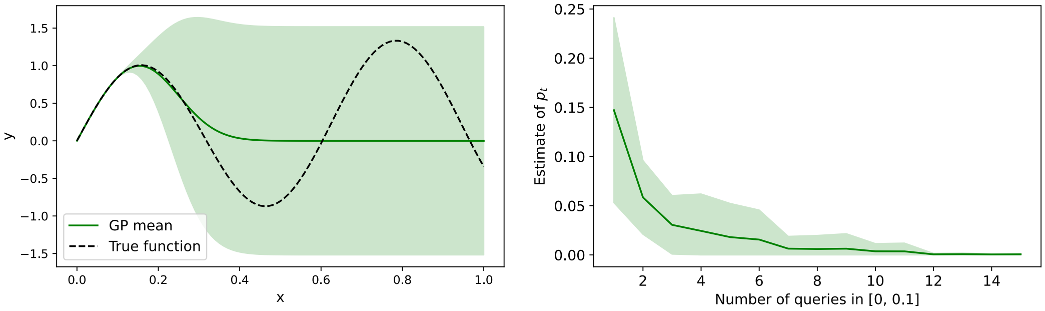


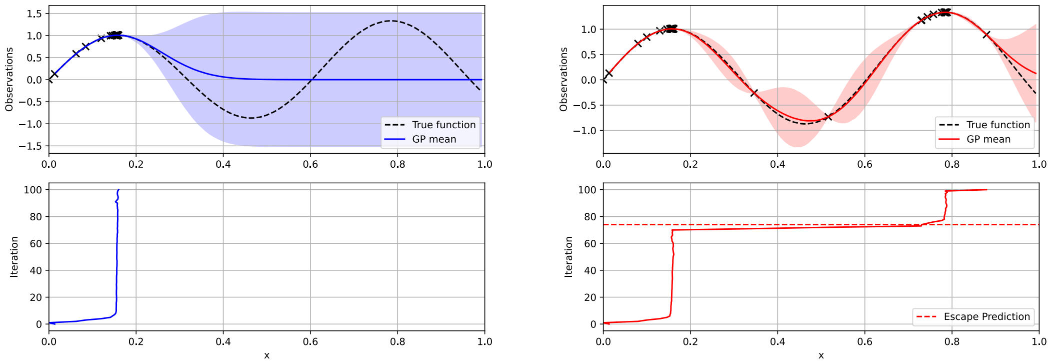
Figure 9: We estimate the probability of non-escape by taking 5000 independent Thompson Samples and counting the number of samples inside A (i.e. the MLE estimator of the Bernoulli distribution). We do this for increasing number of training points in A (which are chosen randomly with a uniform distribution in A). We repeat the experiment 10 times. The left plot shows the underlying function and the Gaussian Process for 15 training points. The right plot shows the evolution of our estimate as we increase training points inside $A = [0, 0.1]$ (we plot the mean of each run ± the standard deviation). We can see that p_t quickly converges to (almost) zero. We almost guaranteed to fully escape the area after 15 time-steps, even for very large budgets.

E Implementation Details

This section outlines the implementation details and hyper-parameter choices for all the methods compared in the paper. The code used in the paper is available at the following link: `https://github.com/cog-imperial/SnAKe`.

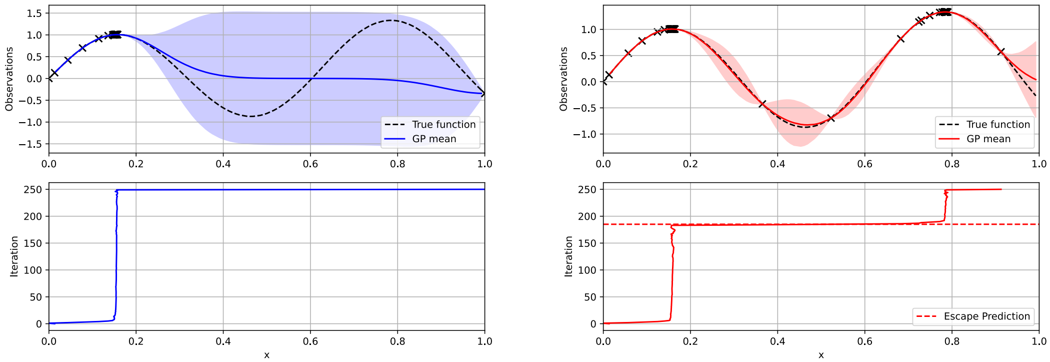
E.1 Computational Considerations (Extended)

Unfortunately, Algorithm 2 (SnAKe) is computationally expensive in two aspects. First, for large budgets, we may struggle to train and sample the GPs. For our experiments, training was not an issue and we were able to use full model GPs. However, we could use Sparse GPs [43] if needed. The Wilson et al. [53] approach allows us to create GP samples efficiently, and possibly optimize them using gradient methods. The sampling can be done in linear time (*after* the GP has been trained). We use the Wilson et al. [53] method to create our samples, and then optimize the samples using Adam [23].



(a) Optimization with Resampling. $T = 100$.

(b) Optimization with 0.1-Point Deletion. $T = 100$.



(c) Optimization with Resampling. $T = 250$.

(d) Optimization with 0.1-Point Deletion. $T = 250$.

Figure 10: We investigate the effect of Point Deletion, and empirically confirm our analysis from Section 3.6. For Point Deletion, we calculate the escape prediction as pT, using $\hat{p} \approx 0.74$, which we estimated in Section D.1. We can see that without Point Deletion, the escape happens until *until the very last* iteration, independently of the budget (see Remark 3.4 for an explanation). With Point Deletion, we can see that our escape predictions are accurate, and the exploration of the actual optimum increases with the budget.

The second bottleneck is solving the TSP, which is NP-hard, and we may need to solve it almost at every iteration (for small values of t_{delay}). There are heuristic solutions that give approximate solutions quickly. We use Simulated Annealing [24] which grows linearly with the budget size, T. For small budgets, simulated annealing should find good solutions, but it could struggle as the budget size increases. To solve this, note that we do not actually require a super-specific solution to the problem: we are only expecting to query the first few points on a path before replacing it by an entirely new path.

We build an adaptive grid (at each iteration) consisting of two separate parts. A very coarse grid, ξ_{global}, covers most of the search space, and a very fine grid, $\xi_{local,t}$, consisting of the N_l samples closest to x_t. This allows us to define the grid $\xi_t = \xi_{global} \cup \xi_{local,t}$. The remaining $T - N_l$ samples will be assigned to the closest point in ξ_t (using the Euclidean distance).

The adaptive grid means we expect to have multiple samples assigned to the same point, specifically in the coarse areas of our grid. But this is not important, because we expect our immediate attention will be in the area around our current input where there should be little to no repetition. This will allow the algorithm to focus on testing solutions which are relevant to our problem.

The adaptive grid introduces two hyper-parameters: the size of the global and local grids, respectively N_g and N_l. Using this method, we run the TSP heuristics on a graph with at most $\min(N_g + N_l, T)$ nodes. For the experiments, we create the global grid using a simple Sobol grid [45].

E.2 Gaussian Processes

For every GP, we use the RBF Kernel with an output-scale, θ_0:

$$k_{RBF}(x_1, x_2) = \theta_0 \exp\left(-\frac{1}{2}(x_1 - x_2)^T \Theta^{-2}(x_1 - x_2)\right)$$

where $\Theta = \text{diag}(\ell_1, ..., \ell_d)$ and ℓ_i denotes the length-scale of the ith variable. For the prior mean, we used a constant function with trainable value, μ_0. We implemented them all using the package GPyTorch [11].

E.3 Training the hyper-parameters of the Gaussian Processes

Our method is well suited for physical systems. Hence, we assume that there is good prior knowledge of the hyper-parameters. In particular, we found it reasonable that each hyper-parameter would be given a lower and upper bound. Normally, we would simply have a large initialization sample. However, we believe this goes against the nature of the problem because we want to explore the space slowly to avoid large input costs. So any type of initialization would be costly.

We simulate this in the following way: we first randomly sample $\max(T/5, 10d)$ points from the d-dimensional input space, and train a GP on these data-points. The hyper-parameters of the GP are optimized by maximizing the marginal log-likelihood [40] over 500 epochs using Adam [23] with a learning rate of 0.01. The resulting hyper-parameters will correspond to the ‘educated guess’. We then set the following bounds:

a) For the length-scale, the lower bound is half the educated guess, and the upper bound double the ‘educated guess’.

b) For the output-scale, θ_0, the lower bound is half the educated guess, and the upper bound double the ‘educated guess’.

c) For the initial mean, μ_0, the lower bound is the educated guess minus a third of the initial variance, the upper bound is the educated guess plus a third of the initial variance.

d) The noise parameter we simply set to be greater than 10^{-5}.

We (partially) enforce the constraints by setting a SmoothedBoxPrior on each parameter, with a variance of 0.001. Finally, under the constraints defined above, we re-estimate the hyper-parameters every time we obtain 25 new observations.

To make sure all models receive fair initializations, we set the same seed for each run and function pair.

E.4 SnAKe

We used Simulated Annealing [24] to solve the Travelling Salesman Problem. We implemented it using the NetworkX package [16]. We initalized the cycle with the ‘greedy’ sub-algorithm and used all default options.

We generated the Thompson Samples using the method introduced in Wilson et al. [53] which we implemented ourselves. To optimize the samples we used Adam [23] and PyTorch [36] over $10d$ epochs, with a learning rate of 0.01. We used $10d$ multi-starts for each sample. To create the samples, we used $\ell = 1024$ Fourier bases.

For ℓ-SnAKe, we define an adaptive deletion constant, $\epsilon_t = \min(\ell_{1,t}, ..., \ell_{d,t})$, where $\ell_{i,t}$ denotes the length scale of the ith variable at time t (recall we are re-training the hyper-parameters every new 25 observations, so the length scales change with time).

For the adaptive grid, we use $N_l = 25$ local samples, and a corse global Sobol grid [45] of $N_g = 100$ points.

E.5 Classical Bayesian Optimization

We used BoTorch [3] to implement all methods in this section. We optimized the acquisition functions across 150 epochs using Adam [23] with a learning rate of 0.0001 using 7500 random multi-starts.

E.5.1 Expected Improvement

Expected Improvement [33] optimizes the acquisition function:

$$EI(x) = \mathbb{E}\left[\max(y - y_{best}, 0)\right], \quad y \sim f(x)$$

where y_{best} is our best observation so far.

Expected Improvement per Unit Cost [44] optimizes the acquisition function:

$$EI(x, x_{t-1}) = \frac{EI(x))}{C(x, x_{t-1}) + \gamma}$$

We set $\gamma = 1$ in all experiments to address the fact that not moving incurs zero cost. Note that as $\gamma \to \infty$, the method will behave closer to normal Expected Improvement, and as $\gamma \to 0$ the method will tend to stay closer and closer to the current input.

Truncated Expected Improvement [41] first maximizes the normal Expected Improvement acquisition function. It then travels a distance of at most ℓ towards the proposed point, where ℓ is the GPs length-scale. We exclude this method from synchronous SnAr, as it is not suited for general cost functions.

E.5.2 Upper Confidence Bound

Upper Confidence Bound [46] optimizes the acquisition function:

$$UCB(x) = \mu_t(x) + \beta_t \sigma_t(x)$$

We set $\beta_t = 0.2d \log(2t)$ following Kandasamy et al. [21].

E.5.3 Probability of Improvement

Probability of Improvement [25] optimizes the acquisition function:

$$PI(x) = \mathbb{P}(y \geq y_{best}), \quad y \sim f(x)$$

E.5.4 Truncated Expected Improvement

Truncated Expected Improvement was developed by [41] trying to solve the task of using an automated boat to monitor the water quality of Ypacari Lake in Paraguay. The method seeks to take distance travelled into account when doing Bayesian Optimization. It does by selecting a point using Expected Improvement, setting a direct path from our current input to the new point, and then *truncating* the path one length-scale away. That is:

$$p_t = \arg\max_{x \in \mathcal{X}} EI(x \mid D_t)$$

$$x_{t+1} = x_t + \frac{p_t - x_t}{||p_t - x_t||} \min(\ell, ||p_t - x_t||)$$

E.6 Asynchronous Bayesian Optimization

E.6.1 Local Penalization

We use the penalization method as is described in González et al. [14]. For the Lipschitz constant, we estimate it by calculating the gradient of μ_t (using auto-differentiation) in a Sobol grid [45] of $50d$ points and selecting L to be the maximum gradient in the grid.

UCB with Local Penalization We set $\beta_t = 0.2d \log(2t)$ and $M = y_{best}$.

EIpu with Local Penalization We set $\gamma = 1$.

E.6.2 Thompson Sampling

We use the sample procedure as in E.4, except we only optimize a single sample at every iteration.

E.7 Description of Benchmark Functions

We chose the benchmark functions to observe the behavior of SnAKe in a variety of scenarios. More details of all the benchmark functions can be found in Surjanovic and Bingham [47].

E.7.1 Branin2D

The two-dimensional Branin function is given by. The function has three global maximums:

$$f(x) = a(x_2 - bx_1^2 + cx_1 - r)^2 + s(1-t)\cos(x_1) + s$$

where we optimize over $\mathcal{X} = [-5, 10] \times [0, 15]$. We set $a = -1$, $= 5.1/(4\pi^2)$, $c = 5/\pi$, $r = 6$, $s = -10$, and $t = 1/(8\pi)$.

E.7.2 Ackley4D

The four-dimensional Ackley function has a lot of local optimums, with the optima in the center of the search space. The function is given by:

$$f(x) = a\exp\left(-b\sqrt{\frac{1}{4}\sum_{i=1}^{4} x_i^2}\right) + \exp\left(\frac{1}{d}\sum_{i=1}^{4}\cos(cx_i)\right) - a - \exp(1)$$

where we slightly shift the search space and optimize over $\mathcal{X} = [-1.8, 2.2]^4$. This is to avoid having the optimum exactly at a point in the Sobol grid and giving SnAKe an unfair advantage. We set $a = 20$, $= 0.2$, and $c = 2\pi$.

E.7.3 Michaelwicz2D

The two-dimensional Michalewicz function is characterized by multiple local maxima and a lot of flat regions. The function is given by:

$$f(x) = \sum_{i=1}^{2} \sin(x_i)\sin^{2m}\left(\frac{ix_i^2}{\pi}\right)$$

where we set $m = 10$ and we optimize on the region $\mathcal{X} = [0, \pi]^2$.

E.7.4 Hartmann

We select this function to see how the algorithms behave in *similar* functions as dimension increases. We do three versions of the Hartmann function, with dimensions $d = 3, 4$, and 6. The equation is given by:

$$f(x) = \sum_{i=1}^{4} \alpha_i \exp\left(-\sum_{j=1}^{d} A_{ij}(x_j - P_{ij})\right)$$

where $\alpha = (1, 1.2, 3, 3.2)^T$. For $d = 3$ we use:

$$A = \begin{pmatrix} 3 & 10 & 30 \\ 0.1 & 10 & 35 \\ 3 & 10 & 30 \\ 0.1 & 10 & 35 \end{pmatrix} \qquad P = 10^{-4}\begin{pmatrix} 3689 & 1170 & 2673 \\ 4699 & 4387 & 7470 \\ 1091 & 8732 & 5547 \\ 381 & 5743 & 8828 \end{pmatrix}$$

for $d = 4$ and $d = 6$ we use:

$$A = \begin{pmatrix} 10 & 3 & 17 & 3.5 & 1.7 & 8 \\ 0.05 & 10 & 17 & 0.1 & 8 & 14 \\ 3 & 3.5 & 1.7 & 10 & 17 & 8 \\ 17 & 8 & 0.05 & 10 & 0.1 & 14 \end{pmatrix} \qquad P = 10^{-4}\begin{pmatrix} 1312 & 1696 & 5569 & 124 & 8283 & 5886 \\ 2329 & 4135 & 8307 & 3736 & 1004 & 9991 \\ 2348 & 1451 & 3522 & 2883 & 3047 & 6650 \\ 4047 & 8828 & 8732 & 5743 & 1091 & 381 \end{pmatrix}$$

They are all evaluated on the unit cube $[0, 1]^d$.

E.7.5 Perm10D

We select the 10-dimensional version of the Perm benchmark to test the capabilities of the algorithms in a very high-dimensional setting (by BO standards). The equation is given by:

$$f(x) = -10^{-21} \sum_{i=1}^{10} \left(\sum_{j=1}^{10} (j^i + \beta) \left(\left(\frac{x_j}{j} \right)^i - 1 \right) \right)^2$$

where we set $\beta = 10$. We evaluate it on $\mathcal{X} = [-10, 10]^d$.

E.7.6 SnAr4D

We implement the simulation using Summit [10]. We control temperature between 40 and 120 degrees, concentration from 0.1 to 0.5 moles per liter, and residence time between 0.5 and 2 minutes. We set the cost parameters for temperature $\alpha_{temp} = 5, \beta_{temp} = 1, \gamma_{temp} = 1$, for concentration $\alpha_{conc} = 2, \beta_{conc} = 0.01, \gamma_{conc} = 1$, and for residence time $\alpha_{residence} = 3$, $\beta_{residence} = 0.05$ and $\gamma_{residence} = 1$. We further optimize over the equivalents of pyrrolidine between 1 and 5 units, but we assume changing it incurs no input costs.

Since the SnAr benchmark is a multi-objective problem, we optimize a weighted sum of the two objectives:

$$\text{SnAr}(x) = \omega_1 \times \text{yield} - \omega_2 \times \text{e-factor}$$

where we set $\omega_1 = 10^{-4}$ and $\omega_2 = 0.1$. We optimize over $\mathcal{X} = [40, 120] \times [0.1, 0.5] \times [0.5, 2] \times [1, 5]$.

E.8 Ypacarai Implementation Details

As per [41] we model the Lake using the Shekel function which is given as:

$$f(x) = \sum_{i=1}^{m} \left(\sum_{j=1}^{2} (10x_j - C_{ji})^2 + \beta_i \right)^{-1}$$

For the objective we use $m = 2$, $m = 3$, and $m = 2$. The other parameters are:

$$C^{(1)} = \begin{pmatrix} 2 & 6.7 \\ 9 & 2 \end{pmatrix} \qquad C^{(2)} = \begin{pmatrix} 7 & 6 \\ 3.8 & 9.9 \\ 9 & 0.1 \end{pmatrix} \qquad C^{(3)} = \begin{pmatrix} 4 & 3 \\ 8.5 & 4 \end{pmatrix}$$

$$\beta^{(1)} = \begin{pmatrix} 9 & 9 \end{pmatrix} \qquad \beta^{(2)} = \begin{pmatrix} 10 & 8 & 8 \end{pmatrix} \qquad \beta^{(3)} = \begin{pmatrix} 7 & 9 \end{pmatrix}$$

Where $C^{(i)}$ and $\beta^{(i)}$ represent the parameters of the ith objective. All objectives are optimized across on a subset of $\mathcal{X} = [0, 1]^2$. The subset is defined by creating grid of points mapping a high-resolution black and white image of Lake Ypacarai onto $\mathcal{X}$. For simplicity, we assume the cost of moving from one point to another is simply the distance between the points, even though in practice we might need to take a longer route to avoid land. For Truncated Expected Improvement, we simply project the truncation into the closest point grid, to avoid sampling points outside of the Lake.

F Full Experiment Results

F.1 Tables of Results

F.1.1 Synchronous Experiments

This section includes the full tables of results of the synthetic synchronous experiments. The results are shown in Table 3 and 4.

F.1.2 Asynchronous Experiments

In this section we include the full table results of all asynchronous experiments. The results are shown in Table 5 and 6.

Table 3: Comparison of 2-norm cost for different BO benchmark functions. The best three performances are shown in bold, and the best one in italics. We can see that SnAKe constantly achieves low cost, especially for larger budgets. The best cost performance is achieved by 0.0-SnAKe, however, we do this at the expense of worse regret. The only function for which SnAKe struggles is the very high dimensional Perm10D.

Method	Budget	0.0-SnAKe	0.1-SnAKe	1.0-SnAKe	ℓ-SnAKe	EI	EIpu	TrEI	UCB	PI	Random
Branin2D	15	4.5 ± 1.8	5.4 ± 1.6	5.8 ± 1.7	5.7 ± 1.5	6.7 ± 1.5	5.3 ± 1.2	5.1 ± 1.0	**4.4 ± 1.6**	***1.0 ± 0.9***	**4.1 ± 0.4**
	50	**5.8 ± 3.0**	9.3 ± 3.1	10 ± 4	9.8 ± 3.2	17 ± 6	**7.3 ± 1.7**	13.4 ± 2.3	15 ± 7	***4.0 ± 3.2***	7.5 ± 0.4
	100	***5.4 ± 2.9***	10 ± 4	10 ± 4	11 ± 4	37 ± 13	**9.1 ± 1.7**	25 ± 4	33 ± 17	12 ± 7	**10.2 ± 0.5**
	250	***7.1 ± 2.3***	15.4 ± 3.4	16 ± 4	**15.3 ± 2.8**	112 ± 32	**12.2 ± 2.0**	59 ± 13	$(9 \pm 5) \times 10^1$	32 ± 22	16.5 ± 0.7
Ackley4D	15	**14.7 ± 1.2**	**14.9 ± 1.3**	16.0 ± 1.0	16.4 ± 0.9	21.1 ± 0.8	15.4 ± 2.6	21.4 ± 0.8	19.5 ± 2.1	18.5 ± 3.4	***8.0 ± 0.4***
	50	**25 ± 5**	**28 ± 4**	31 ± 6	30 ± 5	66 ± 6	60 ± 8	69.4 ± 1.9	60 ± 9	52 ± 9	***19.5 ± 0.6***
	100	***23 ± 5***	27 ± 6	**24 ± 6**	**23 ± 5**	128 ± 9	123 ± 10	135.8 ± 3.2	107 ± 16	93 ± 10	32.6 ± 0.9
	250	***25 ± 8***	**32 ± 9**	**33 ± 8**	33 ± 6	302 ± 28	311 ± 17	330 ± 6	221 ± 30	210 ± 10	59.9 ± 1.1
Michaelwicz2D	15	**1.7 ± 1.1**	**1.9 ± 0.6**	1.9 ± 0.8	2.1 ± 1.1	11.8 ± 2.6	***0.8 ± 0.4***	2.3 ± 0.4	6.7 ± 1.6	4.2 ± 3.3	3.9 ± 0.4
	50	**2.1 ± 0.8**	3.0 ± 0.9	**2.7 ± 0.8**	3.1 ± 1.1	23 ± 4	***1.58 ± 0.33***	7.7 ± 0.9	24 ± 7	8 ± 12	7.5 ± 0.4
	100	**2.4 ± 1.2**	3.9 ± 0.8	**3.6 ± 1.0**	3.7 ± 1.0	30 ± 11	***2.1 ± 0.5***	15.6 ± 2.0	36 ± 4	17 ± 23	10.5 ± 0.4
	250	***2.8 ± 1.4***	5.8 ± 1.6	**4.7 ± 1.6**	5.4 ± 1.0	$(6 \pm 4) \times 10^1$	**3.1 ± 0.4**	40.3 ± 2.6	107 ± 22	$(6 \pm 5) \times 10^1$	16.4 ± 0.6
Hartmann3D	15	***2.5 ± 1.0***	3.4 ± 1.2	4.2 ± 1.3	3.6 ± 1.1	6.7 ± 3.0	**2.6 ± 0.8**	**3.1 ± 0.6**	5.9 ± 2.7	4.4 ± 3.1	6.24 ± 0.32
	50	***4.3 ± 2.1***	**5.7 ± 2.2**	6.9 ± 3.4	7.0 ± 3.3	15.1 ± 3.4	**4.9 ± 2.6**	5.8 ± 1.7	14 ± 4	9 ± 4	13.7 ± 0.5
	100	***4.9 ± 2.6***	9 ± 5	**8 ± 4**	9 ± 5	46 ± 8	**7.0 ± 2.8**	9.4 ± 2.8	26 ± 10	29 ± 12	21.5 ± 0.6
	250	***4.9 ± 2.3***	**10 ± 4**	**8.5 ± 3.5**	9.8 ± 3.4	96 ± 4	13.1 ± 3.5	20 ± 4	53 ± 31	$(9 \pm 4) \times 10^1$	38.5 ± 1.3
Hartmann6D	15	***6 ± 4***	**6 ± 4**	8 ± 4	8 ± 4	18 ± 4	**6.9 ± 3.1**	9 ± 4	17 ± 4	17 ± 5	10.5 ± 0.5
	50	***11 ± 5***	**11 ± 4**	12 ± 6	**12 ± 4**	61 ± 11	39 ± 9	32 ± 14	54 ± 9	49 ± 14	29.6 ± 0.8
	100	***11 ± 5***	**13 ± 6**	15 ± 9	**12 ± 6**	117 ± 21	94 ± 19	65 ± 29	102 ± 11	91 ± 28	51.8 ± 1.0
	250	***13 ± 6***	**15 ± 7**	15 ± 8	**15 ± 9**	$(2.7 \pm 0.6) \times 10^2$	$(2.5 \pm 0.7) \times 10^2$	$(1.6 \pm 0.7) \times 10^2$	224 ± 24	$(2.1 \pm 0.7) \times 10^2$	107.6 ± 1.3
Perm10D	15	22.2 ± 1.0	22.2 ± 1.1	22.3 ± 1.1	22.2 ± 1.2	23.7 ± 2.8	**6.3 ± 0.5**	***2.0 ± 0.6***	21.6 ± 2.8	**14 ± 4**	15.05 ± 0.27
	50	65.7 ± 2.6	65.7 ± 2.3	67.6 ± 2.3	66.0 ± 2.7	85 ± 11	**29.4 ± 3.5**	***6.8 ± 2.0***	76 ± 7	48 ± 14	**45.2 ± 0.7**
	100	118.5 ± 3.3	118.0 ± 3.2	129 ± 9	118 ± 4	173 ± 24	**67 ± 7**	***14 ± 4***	155 ± 12	98 ± 32	**82.5 ± 0.9**
	250	254 ± 12	254 ± 11	282 ± 11	251 ± 9	$(4.3 \pm 0.5) \times 10^2$	**202 ± 24**	***35 ± 10***	400 ± 32	$(2.6 \pm 0.8) \times 10^2$	**183.2 ± 1.2**

Table 4: Comparison of final log(regret) for different BO benchmark functions. The best three performances are shown in bold, and the best one in italics. We can see that SnAKe constantly achieves regret comparable with classical Bayesian Optimization methods. The worst performance happens when $\epsilon = 0$, this could be explained by the method getting stuck in local optimums.

Method	Budget	0.0-SnAKe	0.1-SnAKe	1.0-SnAKe	ℓ-SnAKe	EI	EIpu	TrEI	UCB	PI	Random
Branin2D	15	−3.9 ± 1.7	−3.5 ± 0.7	−3.7 ± 1.6	−3.4 ± 0.9	**−4.7 ± 1.5**	**−4.5 ± 1.4**	−4.4 ± 1.3	***−5.0 ± 1.5***	−3.2 ± 1.5	−3.3 ± 0.8
	50	−6.2 ± 2.5	−8.1 ± 2.6	−7.9 ± 2.4	**−8.3 ± 2.3**	***−8.7 ± 1.7***	−7.0 ± 1.9	−6.1 ± 1.6	**−8.5 ± 2.6**	−6.2 ± 2.5	−4.4 ± 1.2
	100	−9.1 ± 3.2	**−11.2 ± 2.2**	**−11.4 ± 2.7**	−10.7 ± 2.2	***−13 ± 5***	−7.4 ± 2.0	−7.0 ± 2.0	−10.7 ± 2.5	−10.8 ± 1.7	−5.1 ± 1.1
	250	−12.3 ± 1.6	−13.2 ± 1.3	**−13.6 ± 1.4**	**−13.5 ± 1.4**	***−15 ± 7***	−7.9 ± 1.7	−8.1 ± 1.5	−11.9 ± 2.5	−12 ± 5	−6.3 ± 1.2
Ackley4D	15	1.62 ± 0.25	1.62 ± 0.25	**1.62 ± 0.25**	1.67 ± 0.18	1.71 ± 0.15	1.69 ± 0.20	1.73 ± 0.19	1.71 ± 0.15	**1.59 ± 0.22**	***1.40 ± 0.11***
	50	1.64 ± 0.19	1.64 ± 0.19	1.64 ± 0.19	1.65 ± 0.19	**1.59 ± 0.22**	1.69 ± 0.20	1.73 ± 0.19	1.70 ± 0.14	**1.34 ± 0.21**	***1.21 ± 0.14***
	100	1.70 ± 0.15	1.69 ± 0.15	1.69 ± 0.15	1.72 ± 0.11	**1.54 ± 0.27**	1.69 ± 0.20	1.73 ± 0.19	1.70 ± 0.14	**1.21 ± 0.11**	***1.06 ± 0.17***
	250	1.52 ± 0.22	**0.9 ± 0.8**	1.2 ± 0.5	1.0 ± 0.5	1.4 ± 0.5	1.69 ± 0.20	1.73 ± 0.19	1.5 ± 0.4	***0.85 ± 0.32***	**0.89 ± 0.20**
Michaelwicz2D	15	−4.2 ± 1.7	−4.8 ± 1.6	−5.1 ± 1.8	−4.5 ± 1.9	**−5.4 ± 1.1**	**−5.8 ± 1.1**	−5.4 ± 1.0	***−5.9 ± 1.3***	−4.3 ± 1.6	−4.5 ± 1.0
	50	−6.4 ± 1.5	**−7.3 ± 1.4**	−6.8 ± 1.3	−6.3 ± 1.4	−6.0 ± 1.3	−6.4 ± 1.1	**−7.0 ± 1.1**	***−7.8 ± 1.9***	−6.2 ± 1.7	−5.4 ± 0.4
	100	−7.0 ± 2.2	−7.2 ± 1.7	**−7.8 ± 1.8**	**−7.6 ± 1.9**	−6.2 ± 0.8	−6.6 ± 1.2	−7.5 ± 1.1	***−8.2 ± 2.0***	−6.6 ± 1.9	−6.0 ± 0.4
	250	−6.6 ± 1.6	**−8.1 ± 2.2**	**−8.5 ± 2.2**	−8.0 ± 2.2	−6.5 ± 0.7	−6.7 ± 1.3	−7.7 ± 1.0	***−8.7 ± 2.5***	−7.0 ± 1.4	−6.5 ± 0.7
Hartmann3D	15	−1.2 ± 1.5	−2.0 ± 1.9	−1.9 ± 1.7	−1.8 ± 1.6	**−2.2 ± 1.6**	**−2.7 ± 1.5**	***−3.2 ± 1.5***	−2.0 ± 1.7	−0.6 ± 1.1	−0.3 ± 0.6
	50	−2.9 ± 2.6	−4.8 ± 3.0	−4.8 ± 2.4	−4.8 ± 2.3	***−7.4 ± 1.6***	−5.0 ± 1.5	−5.1 ± 1.1	**−6.1 ± 2.5**	**−5.2 ± 2.0**	−1.4 ± 0.8
	100	−5.4 ± 2.9	**−8.3 ± 1.5**	−7.9 ± 2.4	−8.2 ± 2.1	***−10.9 ± 1.3***	−5.8 ± 1.5	−5.6 ± 1.0	−8.1 ± 3.2	**−9.9 ± 2.6**	−1.5 ± 0.5
	250	−6 ± 4	−9.8 ± 2.7	−9.2 ± 2.5	−9.4 ± 2.0	***−12.4 ± 2.0***	−6.8 ± 1.8	−6.7 ± 1.2	**−10 ± 4**	**−12.0 ± 2.1**	−2.4 ± 0.7
Hartmann6D	15	0.6 ± 0.5	**0.6 ± 0.5**	0.6 ± 0.5	***0.5 ± 0.6***	0.88 ± 0.33	0.8 ± 0.5	**0.5 ± 0.5**	0.77 ± 0.32	0.8 ± 0.4	0.7 ± 0.4
	50	0.3 ± 0.6	**0.1 ± 0.7**	***−0.0 ± 0.8***	**0.0 ± 0.7**	0.6 ± 0.4	0.3 ± 0.6	0.1 ± 0.6	0.4 ± 0.5	0.5 ± 0.6	0.49 ± 0.26
	100	**−0.2 ± 0.8**	**−0.3 ± 1.0**	−0.1 ± 0.9	***−0.6 ± 0.8***	0.2 ± 0.8	−0.1 ± 0.9	−0.1 ± 0.5	0.0 ± 0.6	0.0 ± 0.7	0.1 ± 0.5
	250	−0.6 ± 0.8	−0.7 ± 1.5	−0.7 ± 1.0	**−0.9 ± 1.0**	−0.7 ± 1.0	−0.5 ± 1.0	−0.4 ± 0.6	***−0.9 ± 0.8***	**−0.8 ± 0.9**	−0.04 ± 0.34
Perm10D	15	−2.0 ± 1.7	−2.0 ± 1.7	−2.1 ± 1.7	−2.3 ± 1.7	−5.4 ± 1.8	−4.9 ± 1.0	−4.3 ± 2.6	***−6.6 ± 1.6***	**−6.1 ± 2.0**	**−5.7 ± 1.1**
	50	−3.2 ± 1.9	−3.2 ± 1.9	−3.6 ± 2.0	−3.3 ± 1.8	−6.0 ± 2.2	**−7.5 ± 1.9**	−5.1 ± 2.5	**−7.5 ± 1.7**	−7.1 ± 1.9	***−7.5 ± 1.7***
	100	−3.4 ± 1.4	−3.4 ± 1.4	−4.7 ± 1.7	−3.7 ± 1.3	−6.4 ± 2.2	**−8.4 ± 1.7**	−6.4 ± 2.0	−8.0 ± 1.7	***−8.5 ± 1.8***	**−8.2 ± 1.2**
	250	−5.0 ± 2.3	−5.0 ± 2.3	−6.4 ± 2.3	−4.1 ± 2.1	−6.8 ± 2.4	**−9.3 ± 1.2**	−8.1 ± 1.0	−8.5 ± 1.9	***−9.6 ± 1.9***	**−9.5 ± 1.6**

F.1.3 SnAr Benchmark

We ran additional experiments on the SnAr benchmark. For the first one (which includes the example looked at in the main paper) we tested on a budget of $T = 100$ iterations for different values of t_{delay}. The results are included in Tables 7 and 8.

We also carried out synchronous results on the benchmark, for different budgets. The results are included in Tables 9 and 10.

F.2 Graphs for results of Section 4.1

We include the full graphs of the sequential Bayesian Optimization experiments. Each row represents a different budget. The left column shows the evolution of regret against the cost used, the middle column shows the evolution of regret with iterations, and the right column shows the evolution of the 2-norm cost. The results encompass Figures 11 to 16. The caption in each figure tells us the benchmark function being evaluated. Each experiment is the mean ± half the standard deviation of 25 different runs.

Table 5: Comparison of 2-norm cost for different benchmark functions in the *asynchronous setting*. The best three performances are shown in bold, and the best one in italics. SnAKe achieves considerable lower cost with respect to other methods, achieving the top 3 lowest costs all but one time.

Method	**Budget**	**Delay**	**0.0-SnAKe**	**0.1-SnAKe**	**1.0-SnAKe**	**ℓ-SnAKe**	**Random**	**TS**	**UCBwLP**	**EIpuLP**
Branin2D	100	10	***7.0 ± 1.9***	**10.0 ± 2.2**	10.6 ± 3.2	**9.8 ± 2.6**	10.4 ± 0.4	49 ± 5	27.5 ± 2.5	22 ± 5
	100	25	***7.8 ± 1.6***	11.2 ± 2.7	**9.6 ± 2.6**	10.6 ± 2.4	**10.2 ± 0.5**	52 ± 6	51 ± 4	25 ± 7
	250	10	***7.6 ± 1.7***	**14.5 ± 1.9**	15.9 ± 3.3	**14.9 ± 2.9**	16.6 ± 0.8	120 ± 12	37 ± 5	37 ± 14
	250	25	***8.8 ± 2.1***	**13.4 ± 1.5**	14.7 ± 3.3	**13.1 ± 2.7**	16.6 ± 0.6	122 ± 10	56 ± 4	48 ± 14
Ackley4D	100	10	***22.2 ± 2.3***	23.7 ± 1.9	**22.5 ± 2.0**	**22.4 ± 2.2**	32.5 ± 0.8	110 ± 6	96 ± 8	100 ± 26
	100	25	***19.6 ± 2.4***	24.0 ± 2.5	**22.9 ± 3.2**	**23.3 ± 3.5**	32.6 ± 0.9	101 ± 10	100 ± 4	59 ± 4
	250	10	**27 ± 4**	30.1 ± 2.9	**26.3 ± 2.6**	***26.2 ± 1.5***	59.6 ± 1.1	219 ± 23	243 ± 25	$(2.5 \pm 0.7) \times 10^2$
	250	25	**28.5 ± 3.3**	32 ± 4	***25.4 ± 2.3***	**25.6 ± 2.3**	59.5 ± 0.9	$(2.2 \pm 0.5) \times 10^2$	240 ± 20	175 ± 5
Michaelwicz2D	100	10	***3.4 ± 1.4***	5.0 ± 1.5	**4.5 ± 1.0**	**4.7 ± 1.0**	10.5 ± 0.4	17.8 ± 2.4	23.0 ± 2.6	54 ± 7
	100	25	***4.2 ± 1.1***	6.0 ± 1.2	**5.7 ± 1.3**	**5.9 ± 1.2**	10.5 ± 0.5	25.8 ± 2.6	33.5 ± 2.9	47.4 ± 3.3
	250	10	***3.5 ± 1.7***	**5.9 ± 0.8**	**5.7 ± 1.3**	6.4 ± 1.6	16.3 ± 0.9	27.2 ± 3.4	37 ± 7	67 ± 9
	250	25	***3.7 ± 1.1***	**6.6 ± 1.1**	6.7 ± 1.2	**6.6 ± 1.0**	16.4 ± 0.7	35.3 ± 2.6	36.0 ± 2.8	96 ± 9
Hartmann3D	100	10	***7.7 ± 3.4***	11 ± 4	**9.7 ± 3.2**	**10 ± 4**	21.7 ± 0.6	20.6 ± 3.3	34 ± 6	28 ± 4
	100	25	***9.6 ± 2.9***	**13 ± 4**	**12.8 ± 3.2**	14 ± 5	21.3 ± 0.5	32 ± 4	55 ± 5	49 ± 5
	250	10	***5.8 ± 2.4***	**11 ± 4**	**9.8 ± 3.1**	11 ± 4	38.6 ± 0.7	27 ± 4	42 ± 5	36 ± 5
	250	25	***7.6 ± 2.3***	**12 ± 4**	**11.2 ± 3.1**	12 ± 4	38.7 ± 0.9	38 ± 4	64 ± 9	57 ± 5
Hartmann4D	100	10	***12 ± 5***	**18 ± 6**	**18 ± 5**	21 ± 8	32.6 ± 0.7	39 ± 13	56 ± 9	47 ± 13
	100	25	***15 ± 4***	**20 ± 4**	**21 ± 5**	23 ± 5	32.5 ± 0.8	42 ± 8	71 ± 9	58 ± 7
	250	10	***13 ± 7***	**22 ± 12**	**20 ± 10**	22 ± 13	59.3 ± 1.0	$(8 \pm 5) \times 10^1$	103 ± 28	$(8 \pm 4) \times 10^1$
	250	25	***17 ± 11***	**28 ± 14**	**24 ± 11**	31 ± 15	59.4 ± 1.1	$(9 \pm 4) \times 10^1$	113 ± 20	96 ± 34
Hartmann6D	100	10	***14.9 ± 3.5***	**17 ± 5**	17.9 ± 3.5	**18 ± 4**	52.1 ± 1.2	39 ± 10	120 ± 19	124 ± 11
	100	25	***20.9 ± 2.7***	**22.4 ± 2.7**	23.9 ± 3.2	**24 ± 4**	51.7 ± 1.0	56 ± 10	109 ± 11	114 ± 9
	250	10	***18 ± 5***	**18 ± 5**	**18 ± 4**	20 ± 7	107.1 ± 1.6	53 ± 19	$(2.6 \pm 0.6) \times 10^2$	$(3.1 \pm 0.5) \times 10^2$
	250	25	***21 ± 5***	**23 ± 6**	24 ± 6	**23 ± 5**	107.0 ± 1.8	74 ± 18	$(2.7 \pm 0.5) \times 10^2$	306 ± 30

Table 6: Comparison of $\log(\text{regret})$ for different benchmark functions in the *asynchronous setting*. The best three performances are shown in bold, and the best one in italics. SnAKe achieves regret comparable with other Bayesian Optimization methods.

Method	**Budget**	**Delay**	**0.0-SnAKe**	**0.1-SnAKe**	**1.0-SnAKe**	**ℓ-SnAKe**	**Random**	**TS**	**UCBwLP**	**EIpuLP**
Branin2D	100	10	−9.6 ± 2.4	−9.7 ± 2.2	**−10.3 ± 2.4**	−10.1 ± 2.4	−5.7 ± 1.8	**−11.7 ± 1.1**	***−12.1 ± 1.5***	−7.3 ± 2.5
	100	25	−7.1 ± 2.9	**−8.3 ± 3.1**	−7.5 ± 2.6	−7.1 ± 2.2	−5.4 ± 1.0	***−11.7 ± 1.6***	**−8.2 ± 2.1**	−5.4 ± 1.7
	250	10	−11.9 ± 1.9	−12.9 ± 0.8	−12.8 ± 1.1	**−13.3 ± 1.2**	−5.9 ± 1.1	**−13.8 ± 1.0**	***−14.2 ± 1.6***	−9.3 ± 2.9
	250	25	−11.6 ± 2.0	−11.6 ± 0.8	**−12.1 ± 0.8**	−12.1 ± 1.1	−6.2 ± 1.1	***−15 ± 6***	**−14.8 ± 1.3**	−8.9 ± 2.7
Ackley4D	100	10	1.0 ± 0.5	**0.9 ± 0.7**	**0.9 ± 0.7**	***0.8 ± 0.7***	1.05 ± 0.18	1.46 ± 0.15	1.08 ± 0.21	1.4 ± 0.4
	100	25	1.21 ± 0.19	**1.1 ± 0.4**	1.15 ± 0.33	1.20 ± 0.23	**1.03 ± 0.17**	1.32 ± 0.16	***1.02 ± 0.23***	1.16 ± 0.17
	250	10	***−0.4 ± 0.8***	**−0.4 ± 1.0**	0.0 ± 0.8	**−0.2 ± 0.7**	0.8 ± 0.4	0.7 ± 0.8	0.1 ± 0.8	1.3 ± 0.4
	250	25	−0.3 ± 0.5	***−0.9 ± 0.5***	**−0.6 ± 0.5**	**−0.6 ± 0.5**	0.95 ± 0.13	0.4 ± 0.9	0.74 ± 0.29	1.02 ± 0.18
Michaelwicz2D	100	10	−7.1 ± 1.8	−7.1 ± 1.4	−6.8 ± 1.7	−7.4 ± 1.7	−6.2 ± 0.7	**−7.9 ± 2.5**	***−9.0 ± 1.1***	**−7.9 ± 1.5**
	100	25	−7.1 ± 2.1	**−7.4 ± 2.0**	−7.0 ± 1.4	−6.8 ± 1.1	−5.86 ± 0.21	−7.4 ± 1.6	***−11.2 ± 1.5***	**−9.2 ± 0.9**
	250	10	−6.8 ± 1.7	−8.3 ± 2.1	−8.5 ± 2.2	−8.4 ± 2.4	−6.5 ± 0.8	**−8.5 ± 3.2**	**−9.0 ± 1.1**	***−9.9 ± 1.7***
	250	25	−6.8 ± 1.3	−8.1 ± 2.0	**−8.6 ± 2.5**	−8.4 ± 2.3	−6.6 ± 0.9	−7.9 ± 2.4	***−11.2 ± 1.5***	**−10.3 ± 1.4**
Hartmann3D	100	10	−5.3 ± 2.8	**−7.8 ± 3.0**	−6.8 ± 3.4	−7.4 ± 2.7	−1.9 ± 0.9	**−9.5 ± 1.1**	***−10.0 ± 1.2***	−5.5 ± 1.3
	100	25	−4.7 ± 2.4	−6.0 ± 2.1	−6.1 ± 1.8	**−6.4 ± 1.7**	−1.6 ± 0.8	***−8.6 ± 1.3***	**−6.3 ± 1.5**	−4.3 ± 1.4
	250	10	−5 ± 4	−7.9 ± 3.5	−8.0 ± 3.5	**−8.6 ± 3.0**	−2.6 ± 0.8	**−10.7 ± 1.1**	***−12.8 ± 1.2***	−6.3 ± 1.3
	250	25	−6 ± 4	−8.4 ± 3.0	−8.8 ± 3.1	**−8.9 ± 2.6**	−2.4 ± 0.5	**−10.5 ± 0.7**	***−12.4 ± 0.9***	−6.2 ± 1.1
Hartmann4D	100	10	−3.1 ± 3.0	−2.9 ± 2.6	**−3.5 ± 2.4**	−2.8 ± 2.1	−1.1 ± 0.6	***−6.5 ± 2.8***	**−4.5 ± 1.9**	−2.6 ± 1.3
	100	25	**−2.1 ± 1.8**	−2.0 ± 1.7	−2.0 ± 1.9	−1.9 ± 1.3	−1.0 ± 0.5	***−4.2 ± 1.4***	−2.1 ± 1.0	**−2.2 ± 1.0**
	250	10	−6 ± 4	**−6 ± 4**	−6 ± 4	−4.2 ± 3.3	−1.2 ± 0.4	***−8.7 ± 2.1***	**−7.2 ± 1.2**	−4.6 ± 2.0
	250	25	−6 ± 4	−6 ± 4	**−6 ± 4**	−5.5 ± 3.2	−1.4 ± 0.5	***−8.7 ± 1.4***	**−6.5 ± 1.6**	−4.2 ± 1.7
Hartmann6D	100	10	**−0.2 ± 0.6**	***−0.3 ± 0.9***	−0.2 ± 0.7	**−0.3 ± 1.0**	0.1 ± 0.4	−0.2 ± 0.9	−0.0 ± 0.7	0.3 ± 0.5
	100	25	**−0.3 ± 0.7**	**−0.2 ± 0.5**	−0.2 ± 0.6	−0.2 ± 0.6	0.07 ± 0.29	***−0.4 ± 0.6***	0.0 ± 0.4	0.0 ± 0.5
	250	10	−0.5 ± 1.1	−0.7 ± 1.0	**−0.8 ± 0.7**	**−0.9 ± 1.6**	−0.2 ± 0.5	−0.5 ± 1.0	***−1.1 ± 0.7***	−0.2 ± 0.8
	250	25	−0.5 ± 0.8	−0.8 ± 0.9	−0.7 ± 1.1	***−1.1 ± 1.5***	−0.05 ± 0.30	**−0.9 ± 0.9**	**−1.0 ± 0.6**	−0.2 ± 0.5

Table 7: Comparison of cost on SnAr benchmark (asynchronous) for $T = 100$ and different values of t_{delay}. The best three performances are shown in bold, and the best one in italics. SnAKe consistently achieves lower cost than BO methods. EIpuLP achieves lower cost for small delays, suggesting over-exploration in the early stages, and under-exploration once observations arrive.

delay	**0.0-SnAKe**	**0.1-SnAKe**	**1.0-SnAKe**	**ℓ-SnAKe**	**Random**	**TS**	**UCBwLP**	**EIpuLP**
5	$\mathbf{(5.0 \pm 1.1) \times 10^2}$	$(5.7 \pm 0.9) \times 10^2$	$\mathbf{(5.5 \pm 0.8) \times 10^2}$	$(6.2 \pm 0.6) \times 10^2$	596 ± 26	$(1.12 \pm 0.05) \times 10^3$	$(9.2 \pm 0.6) \times 10^2$	$\boldsymbol{(1.6 \pm 0.5) \times 10^2}$
10	$\mathbf{(4.5 \pm 0.8) \times 10^2}$	$\mathbf{(5.3 \pm 0.7) \times 10^2}$	$(5.3 \pm 0.7) \times 10^2$	$(5.8 \pm 0.8) \times 10^2$	600 ± 20	$(1.11 \pm 0.06) \times 10^3$	$(8.9 \pm 0.8) \times 10^2$	$\boldsymbol{(1.9 \pm 0.4) \times 10^2}$
25	$\mathbf{(4.0 \pm 0.7) \times 10^2}$	$(4.8 \pm 0.6) \times 10^2$	$\mathbf{(4.5 \pm 0.8) \times 10^2}$	$(5.1 \pm 0.6) \times 10^2$	603 ± 25	$(1.09 \pm 0.06) \times 10^3$	$(9.3 \pm 0.9) \times 10^2$	***307 ± 31***
50	$\boldsymbol{(4.0 \pm 0.5) \times 10^2}$	$\mathbf{(4.5 \pm 0.4) \times 10^2}$	$\mathbf{(4.2 \pm 0.6) \times 10^2}$	$(4.6 \pm 0.5) \times 10^2$	606 ± 24	$(1.12 \pm 0.05) \times 10^3$	$(1.07 \pm 0.06) \times 10^3$	568 ± 27

Table 8: Comparison of regret on SnAr benchmark (asynchronous) for $T = 100$ and different values of t_{delay}. The best three performances are shown in bold, and the best one in italics. SnAKe consistently achieves regret comparable with BO methods. EIpuLP performs well for small delays, but poorly as the delay is increased.

delay	**0.0-SnAKe**	**0.1-SnAKe**	**1.0-SnAKe**	**ℓ-SnAKe**	**Random**	**TS**	**UCBwLP**	**EIpuLP**
5	$-\mathbf{4.5} \pm \mathbf{1.1}$	-4.5 ± 1.2	-4.3 ± 1.5	-3.5 ± 1.1	-0.93 ± 0.25	$-\mathbf{4.6} \pm \mathbf{1.4}$	$-\boldsymbol{4.9} \pm \boldsymbol{1.4}$	-4.1 ± 0.8
10	-3.5 ± 1.2	-3.9 ± 1.5	-3.7 ± 1.1	-4.0 ± 1.4	-1.1 ± 0.4	$-\boldsymbol{4.3} \pm \boldsymbol{1.3}$	$-\mathbf{4.0} \pm \mathbf{1.2}$	$-\mathbf{4.2} \pm \mathbf{0.8}$
25	$-\mathbf{3.8} \pm \mathbf{1.0}$	$-\mathbf{3.7} \pm \mathbf{1.4}$	-3.3 ± 0.9	-3.6 ± 1.3	-0.9 ± 0.4	$-\boldsymbol{4.2} \pm \boldsymbol{1.3}$	-2.9 ± 0.6	-3.1 ± 1.1
50	-3.2 ± 1.2	$-\mathbf{3.7} \pm \mathbf{1.3}$	-3.1 ± 0.8	$-\mathbf{3.5} \pm \mathbf{0.9}$	-1.1 ± 0.5	$-\boldsymbol{4.3} \pm \boldsymbol{1.2}$	-3.2 ± 0.7	-2.3 ± 1.0

Table 9: Comparison of cost for SnAr benchmark (synchronous) for different budgets. The best three performances are shown in bold, and the best one in italics. SnAKe's performance is poor for small budgets, but improves considerably for the later ones. EIpu achieves the lowest cost in every instance, however Table 10 shows this is due to under-exploration.

Budget	**0.0-SnAKe**	**0.1-SnAKe**	**1.0-SnAKe**	**ℓ-SnAKe**	**EI**	**EIpu**	**UCB**	**PI**	**Random**
10	98 ± 13	101 ± 16	104 ± 12	103 ± 11	$\mathbf{89} \pm \mathbf{11}$	$\boldsymbol{15} \pm \boldsymbol{8}$	103 ± 10	$\mathbf{38} \pm \mathbf{12}$	91 ± 7
25	219 ± 35	230 ± 25	247 ± 21	250 ± 21	230 ± 23	$\boldsymbol{46} \pm \boldsymbol{24}$	289 ± 19	$\mathbf{108} \pm \mathbf{24}$	$\mathbf{202} \pm \mathbf{10}$
50	$(\mathbf{3.7} \pm \mathbf{0.7}) \times \mathbf{10^2}$	$(4.0 \pm 0.5) \times 10^2$	$(4.3 \pm 0.6) \times 10^2$	$(4.1 \pm 0.7) \times 10^2$	503 ± 31	$(\boldsymbol{1.2} \pm \boldsymbol{0.4}) \times \boldsymbol{10^2}$	$(5.5 \pm 0.4) \times 10^2$	$(3.9 \pm 0.4) \times 10^2$	$\mathbf{360} \pm \mathbf{16}$
100	$(\mathbf{6.0} \pm \mathbf{1.8}) \times \mathbf{10^2}$	$(6.9 \pm 1.3) \times 10^2$	$(7.0 \pm 1.5) \times 10^2$	$(7.2 \pm 1.3) \times 10^2$	$(1.12 \pm 0.04) \times 10^3$	$(\boldsymbol{3.0} \pm \boldsymbol{0.9}) \times \boldsymbol{10^2}$	$(1.11 \pm 0.06) \times 10^3$	$(1.00 \pm 0.05) \times 10^3$	$\mathbf{605} \pm \mathbf{26}$

F.3 Graphs for results of Section 4.1

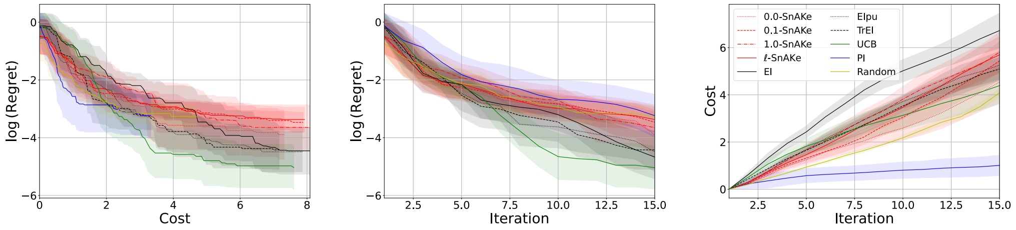
We include the full graphs of the asynchronous Bayesian Optimization experiments. Each row represents a different budget. The left column shows the evolution of regret against the cost used, the middle column shows the evolution of regret with iterations, and the right columns show the evolution of the 2-norm cost. The results encompass Figures 17 to 28. The caption in each figure tells us the benchmark function being evaluated, and the time-delay for getting observations back. Each experiment is the mean ± half the standard deviation of 10 different runs.

F.4 Graphs for results of SnAr Benchmark (section 4.2)

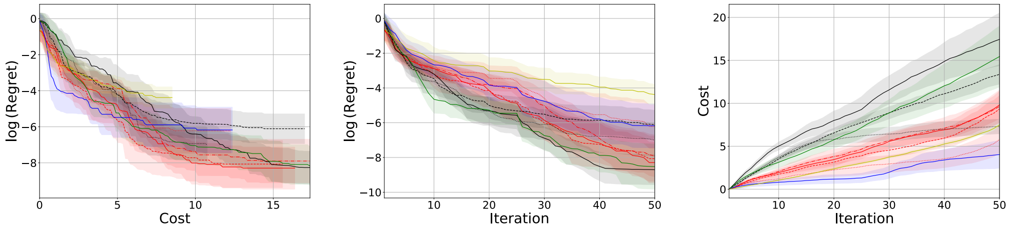
Figure 29 includes the whole set of results of the SnAr benchmark in the asynchronous setting. Figure 30 includes the results for the synchronous setting. Each experiment is the mean ± half the standard deviation of 10 different runs.

Table 10: Comparison of $\log(\text{regret})$ for SnAr benchmark (synchronous) for different budgets. The best three performances are shown in bold, and the best one in italics. SnAKe achieves regret comparable with Bayesian Optimization methods. EIpu achieves the worst non-random performance in every instance.

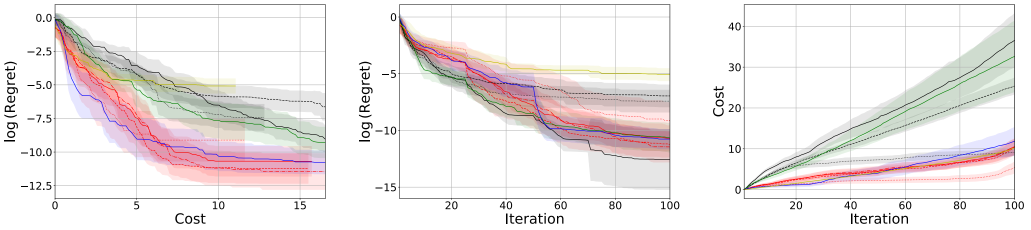
Budget	**0.0-SnAKe**	**0.1-SnAKe**	**1.0-SnAKe**	**ℓ-SnAKe**	**EI**	**EIpu**	**UCB**	**PI**	**Random**
10	-1.9 ± 0.9	-2.0 ± 0.9	$-\mathbf{2.2} \pm \mathbf{1.1}$	-1.6 ± 0.8	$-\boldsymbol{2.2} \pm \boldsymbol{1.0}$	-1.0 ± 0.8	$-\mathbf{2.0} \pm \mathbf{1.0}$	-1.0 ± 1.0	-0.29 ± 0.26
25	$-\mathbf{3.4} \pm \mathbf{0.8}$	-3.3 ± 0.7	-3.3 ± 1.1	-2.9 ± 0.8	$-\mathbf{3.6} \pm \mathbf{0.7}$	-2.9 ± 0.9	-3.0 ± 0.8	$-\boldsymbol{3.8} \pm \boldsymbol{1.3}$	-0.51 ± 0.27
50	-4.5 ± 1.1	$-\mathbf{4.5} \pm \mathbf{1.0}$	-4.1 ± 1.2	-4.3 ± 1.0	$-\boldsymbol{4.9} \pm \boldsymbol{0.8}$	-3.7 ± 1.0	-4.2 ± 0.9	$-\mathbf{4.9} \pm \mathbf{0.7}$	-0.9 ± 0.5
100	-5.3 ± 1.3	-5.8 ± 1.1	$-\mathbf{5.9} \pm \mathbf{1.2}$	-5.6 ± 1.0	$-\mathbf{6.0} \pm \mathbf{0.8}$	-4.9 ± 1.0	$-\boldsymbol{6.0} \pm \boldsymbol{0.5}$	-5.8 ± 0.7	-1.0 ± 0.4



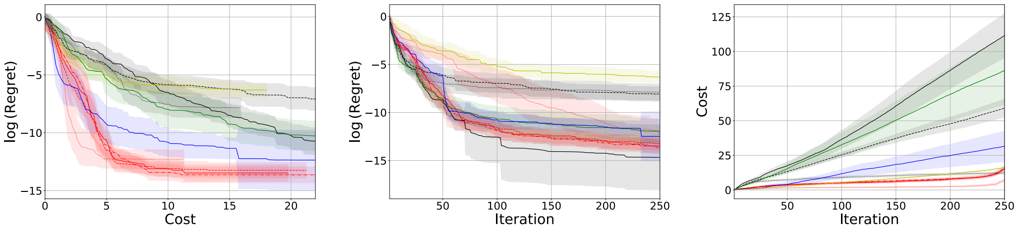
(a) $T = 15$



(b) $T = 50$

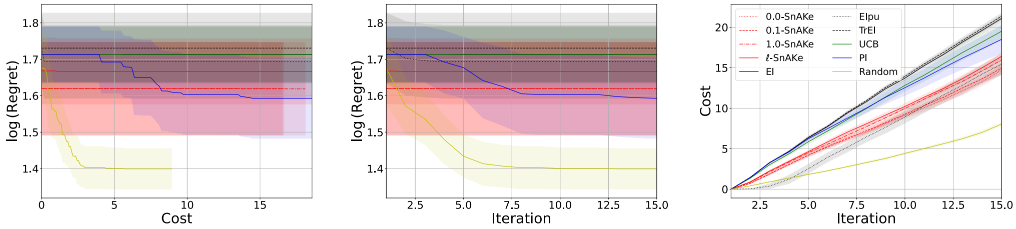


(c) $T = 100$

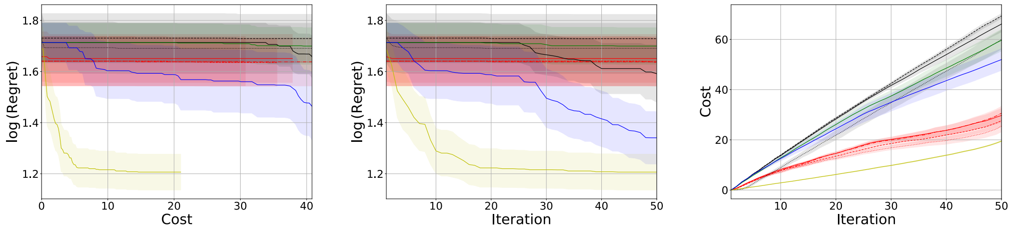


(d) $T = 250$

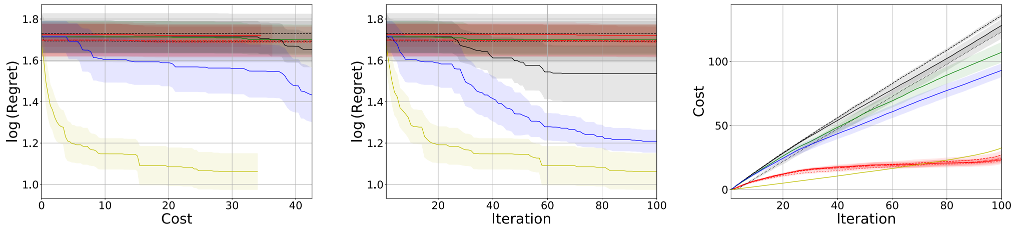
Figure 11: Branin2D. Each row represents a different budget. The left column shows the evolution of regret against the cost used. The middle column shows the evolution of regret with iterations, and the right columns show the evolution of the 2-norm cost. As we increase the budget, SnAKe outperforms two BO methods in regret, and outperforms all methods in cost. $\epsilon = 0$ gives the smallest cost of all at the expense of some regret.



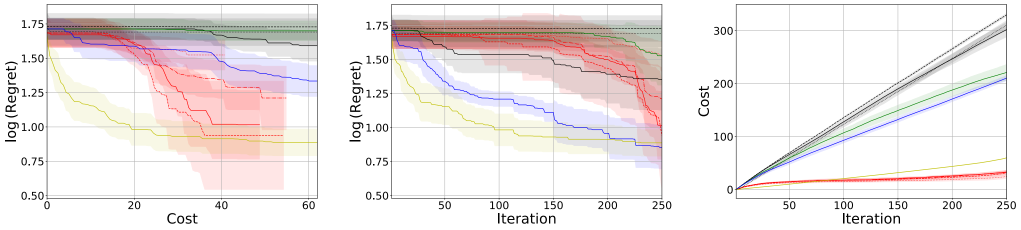
(a) $T = 15$



(b) $T = 50$

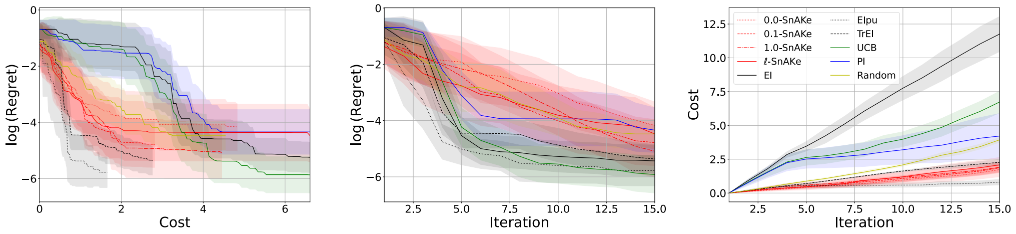


(c) $T = 100$

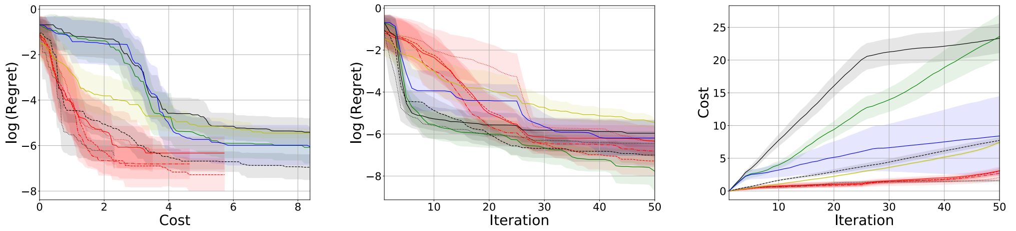


(d) $T = 250$

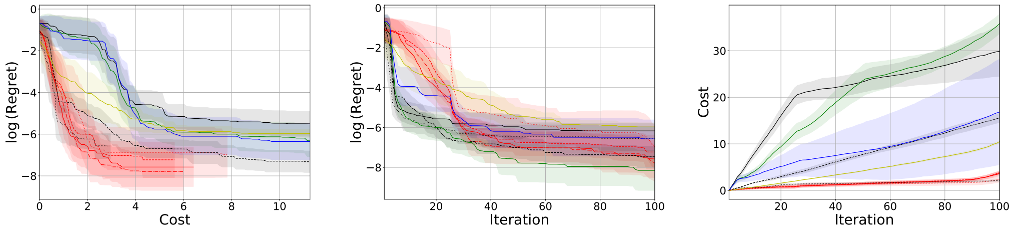
Figure 12: Ackley4D. Each row represents a different budget. The left column shows the evolution of regret against the cost used. The middle column shows the evolution of regret with iterations, and the right columns show the evolution of the 2-norm cost. SnAKe performs badly for smaller budgets, this may be because of the Thompson Sampling (see Figure 20, TS performs very badly in asynchronous Ackely). For the largest budget SnAKe recovers and performs comparably with PI in terms of regret, but achieves low cost.



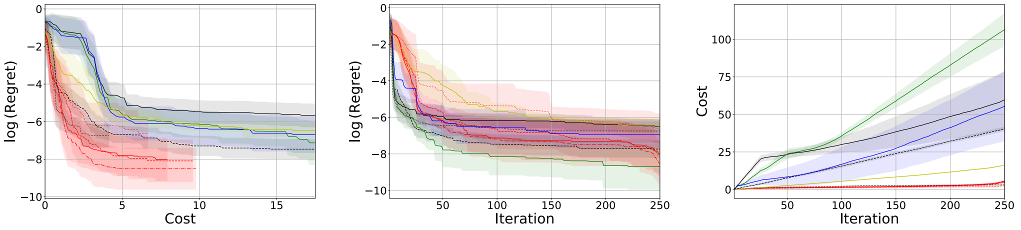
(a) $T = 15$



(b) $T = 50$

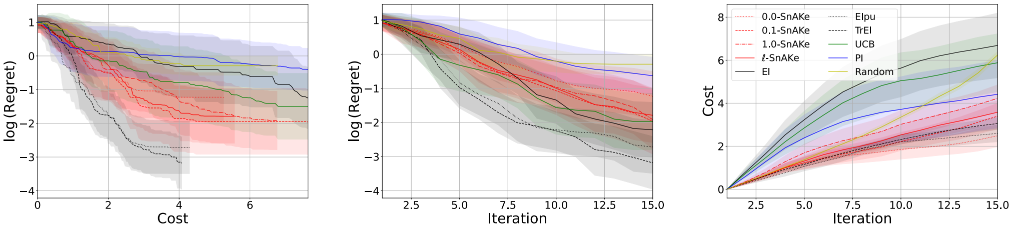


(c) $T = 100$

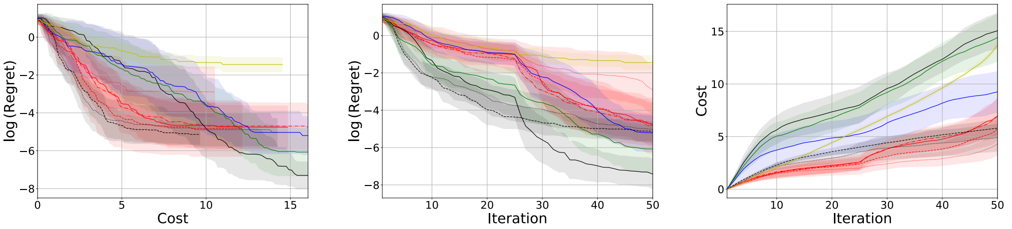


(d) $T = 250$

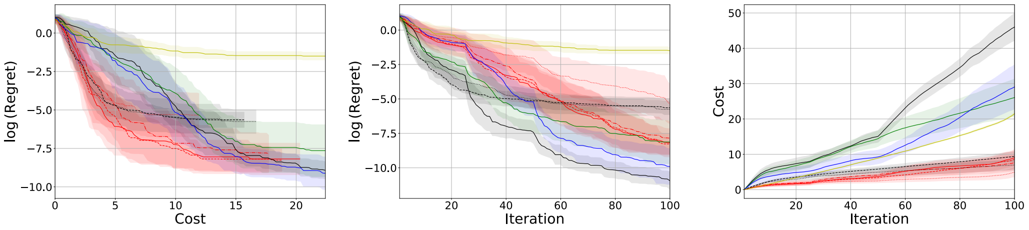
Figure 13: Michaelwicz2D. Each row represents a different budget. The left column shows the evolution of regret against the cost used. The middle column shows the evolution of regret with iterations, and the right columns show the evolution of the 2-norm cost. SnAKe has regret comparable with other methods for all budgets (UCB outperforms the rest for larger ones). SnAKe achieves significantly less cost at all budgets, this may be due to SnAKe exploring the many local optimums carefully. The first column shows that SnAKe achieves by far the best regret for low cost. In this example, SnAKe and EIpu have similar performance.



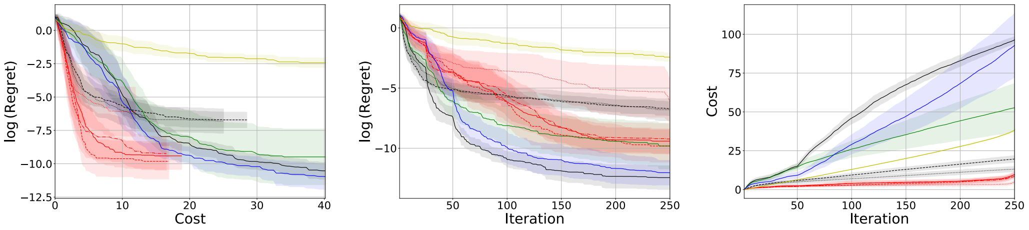
(a) $T = 15$



(b) $T = 50$

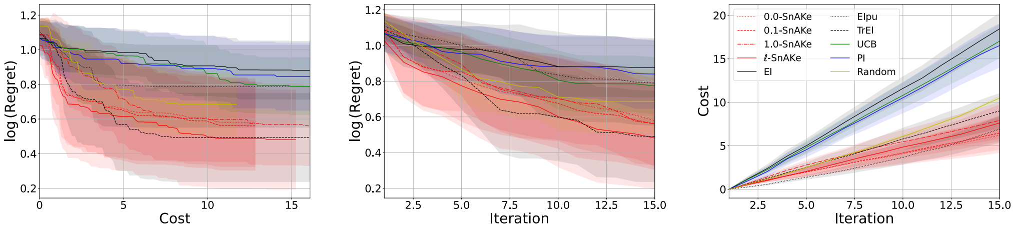


(c) $T = 100$

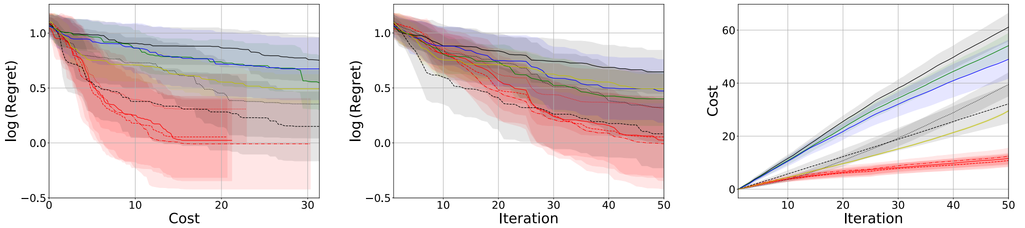


(d) $T = 250$

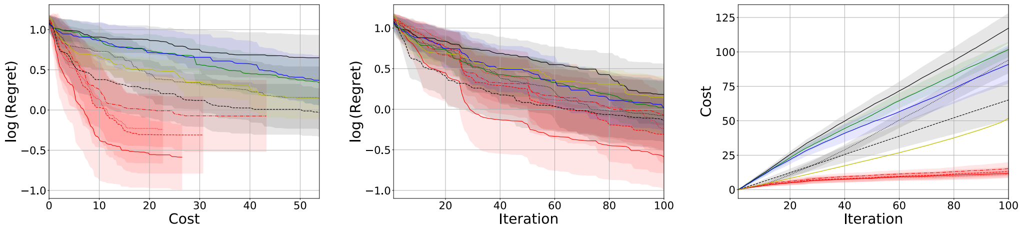
Figure 14: Hartmann3D. Each row represents a different budget. The left column shows the evolution of regret against the cost used. The middle column shows the evolution of regret with iterations, and the right columns show the evolution of the 2-norm cost. Again, SnAKe achieves the best regret at low cost for all budgets. $\epsilon = 0$ struggles in this benchmark, showcasing the impact that Point Deletion can have. EIpu and TrEI achieve higher cost and worse regret than SnAKe.



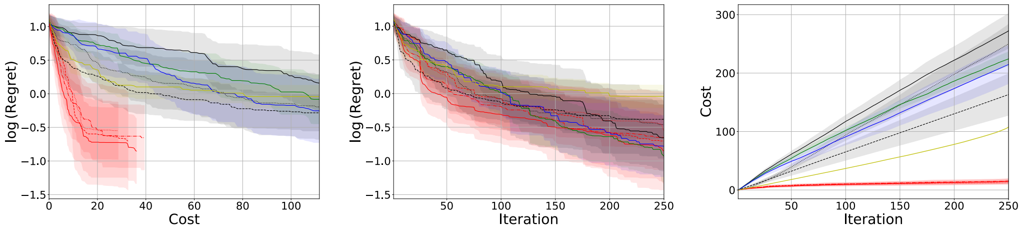
(a) $T = 15$



(b) $T = 50$

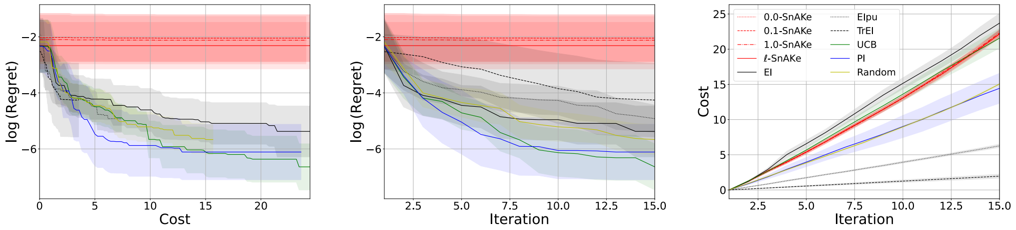


(c) $T = 100$

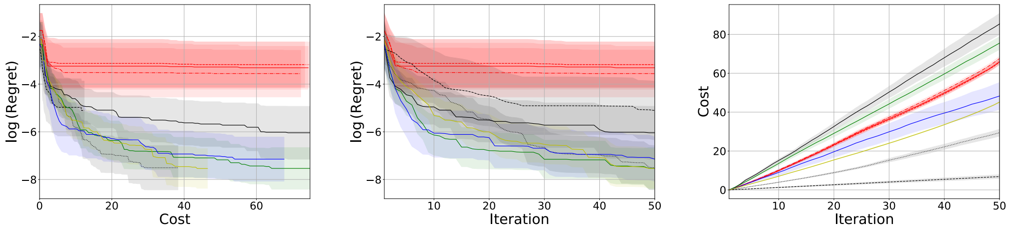


(d) $T = 250$

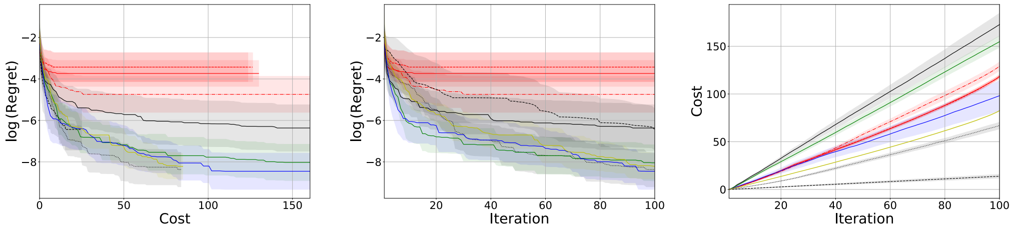
Figure 15: Hartmann6D. Each row represents a different budget. The left column shows the evolution of regret against the cost used. The middle column shows the evolution of regret with iterations, and the right columns show the evolution of the 2-norm cost. A high-dimensional example where SnAKe performs exceedingly well, giving the best regret at low costs for all budgets except $T = 15$. The final cost is considerably lower for SnAKe than any other method.



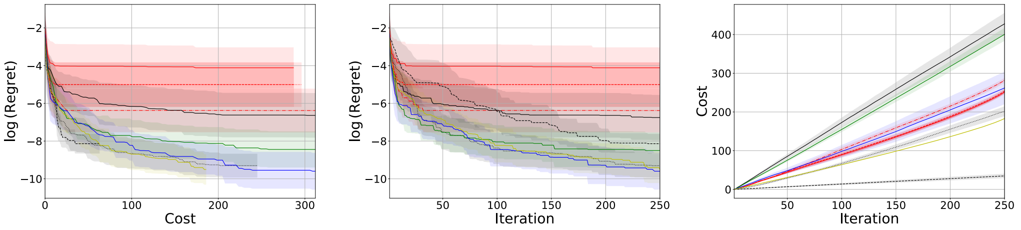
(a) $T = 15$



(b) $T = 50$

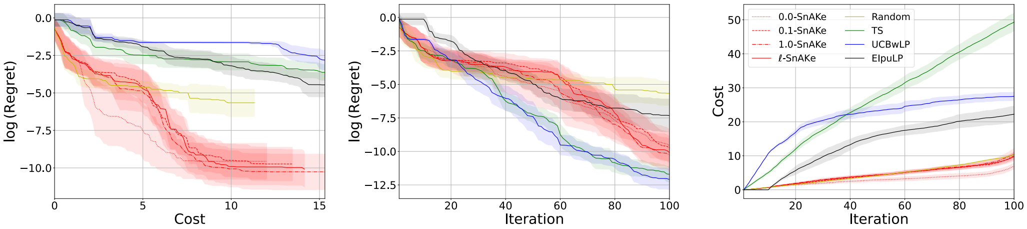


(c) $T = 100$

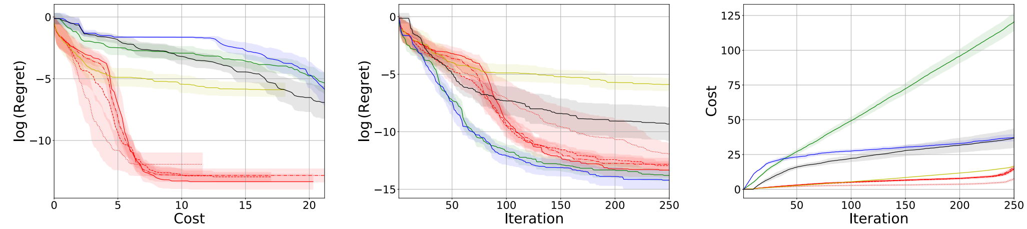


(d) $T = 250$

Figure 16: Perm10D. Each row represents a different budget. The left column shows the evolution of regret against the cost used. The middle column shows the evolution of regret with iterations, and the right columns show the evolution of the 2-norm cost. SnAKe struggles in this benchmark, however, EI also struggles. As an interesting observation, if we *did not* update the model, we would achieve a much better performance (as it would be equivalent to Random). We observe this behavior in the asynchronous case, where having a time-delay helps the method perform better (see asynchronous Ackley, Figure 20). EIpu and TrEI perform well in this example, we conjecture this is because they are doing far more localized searches while SnAKe is trying to cover all the space available (which is very difficult in higher dimensions).

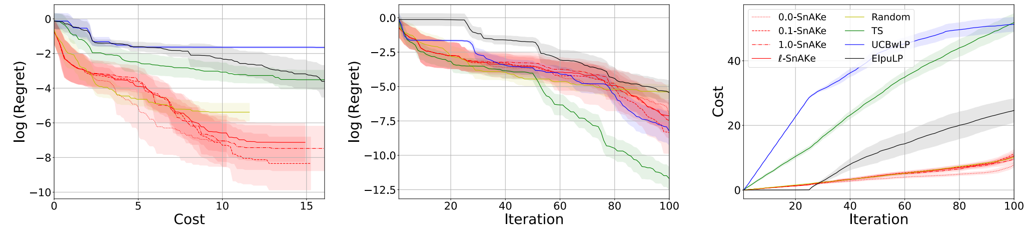


(a) $T = 100$

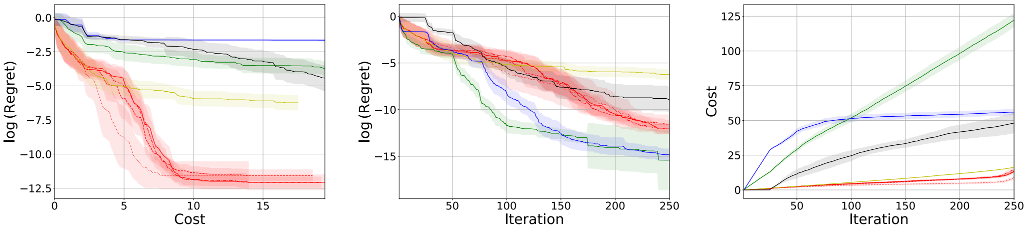


(b) $T = 250$

Figure 17: Branin2D (Asynchronous), $t_{delay} = 10$. Each row represents a different budget. The left column shows the evolution of regret against the cost used. The middle column shows the evolution of regret with iterations, and the right columns show the evolution of the 2-norm cost. SnAKe achieves significantly better regret than all other methods at low costs. The final regret of other BO methods is slightly better, but this comes at the expense of much larger cost. EIpuLP performs poorly, as it seems Local Penalization is over-powering the cost term.

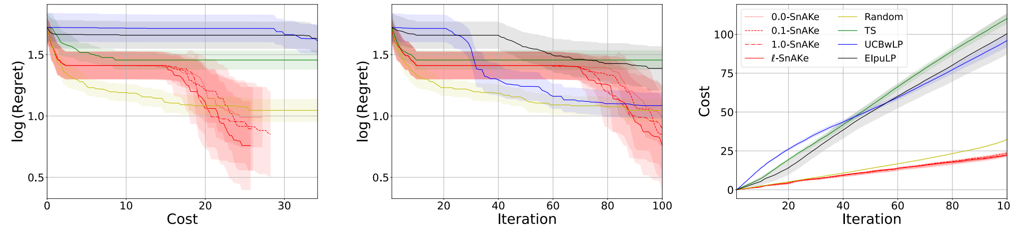


(a) $T = 100$

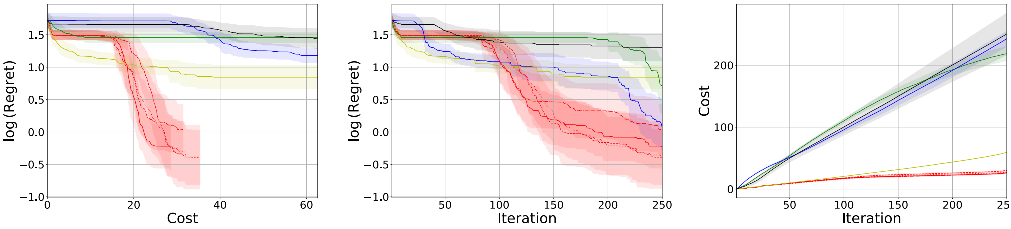


(b) $T = 250$

Figure 18: Branin2D (Asynchronous), $t_{delay} = 25$. Each row represents a different budget. The left column shows the evolution of regret against the cost used. The middle column shows the evolution of regret with iterations, and the right columns show the evolution of the 2-norm cost. The results are similar to the shorter delay seen in Figure 17.

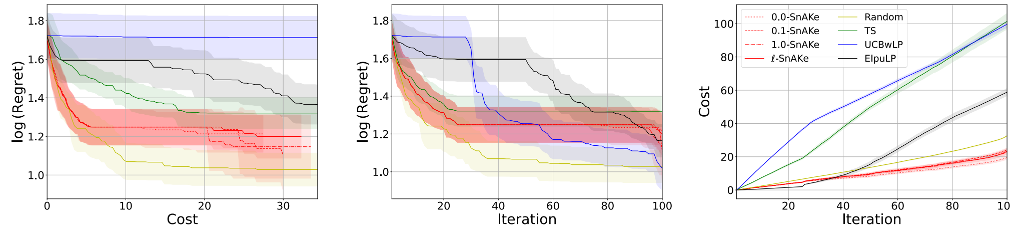


(a) $T = 100$

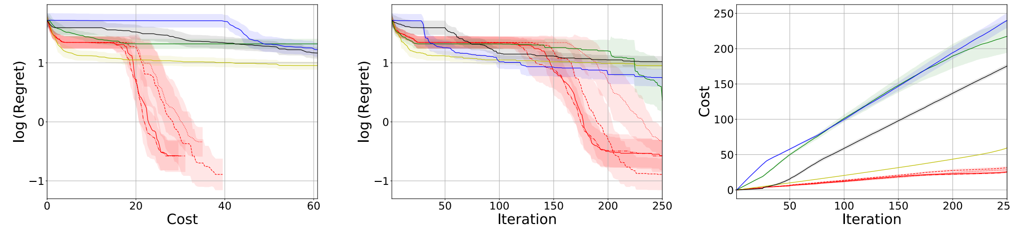


(b) $T = 250$

Figure 19: Ackley4D (Asynchronous), $t_{delay} = 10$. Each row represents a different budget. The left column shows the evolution of regret against the cost used. The middle column shows the evolution of regret with iterations, and the right columns show the evolution of the 2-norm cost. For the larger budget, SnAKe outperforms all other methods in both regret and cost. Interestingly, the performance of SnAKe *improves* when adding delay (see Figure 12 for synchronous results). EIpuLP performs poorly, as it seems Local Penalization is over-powering the cost term.

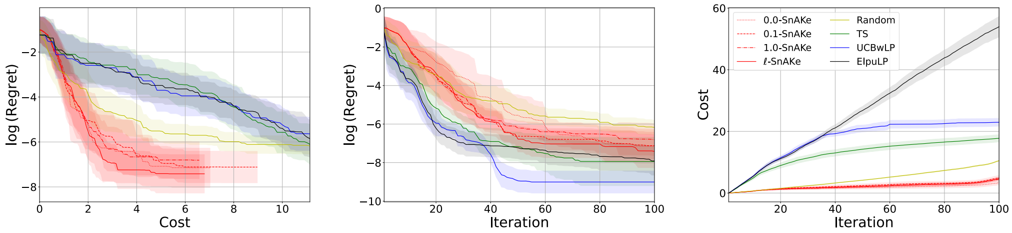


(a) $T = 100$

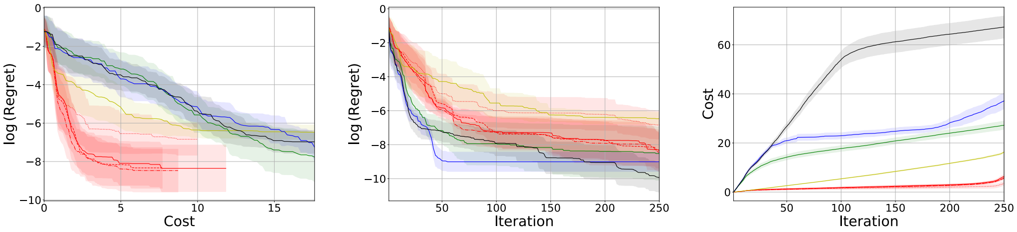


(b) $T = 250$

Figure 20: Ackley4D (Asynchronous), $t_{delay} = 25$. Each row represents a different budget. The left column shows the evolution of regret against the cost used. The middle column shows the evolution of regret with iterations, and the right columns show the evolution of the 2-norm cost. Results are similar to the case when $t_{delay} = 10$, see Figure 19.

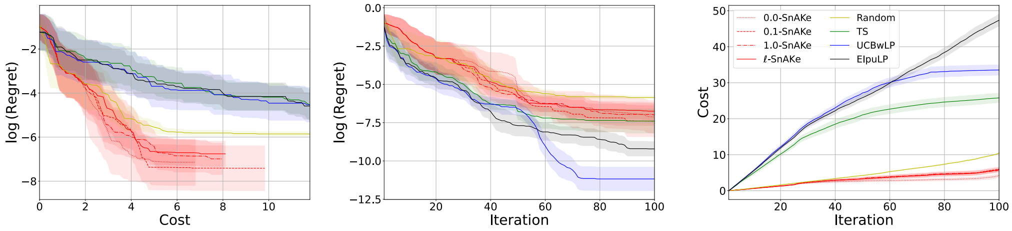


(a) $T = 100$

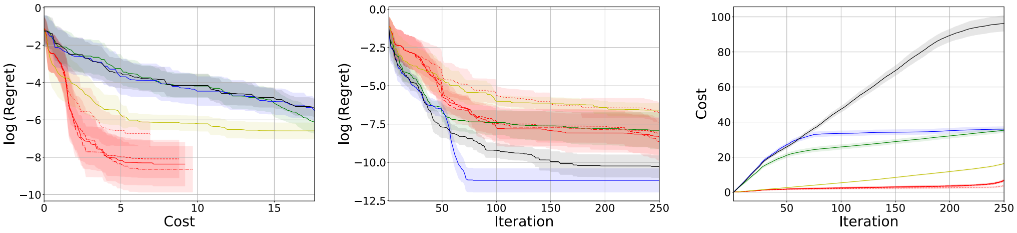


(b) $T = 250$

Figure 21: Michaelwicz2D (Asynchronous), $t_{delay} = 10$. Each row represents a different budget. The left column shows the evolution of regret against the cost used. The middle column shows the evolution of regret with iterations, and the right columns show the evolution of the 2-norm cost. Surprisingly, EIpuLP achieves the best regret, but at considerable cost - suggesting local penalization is over-powering any cost-awareness. For low cost, SnAKe achieves much better regret.

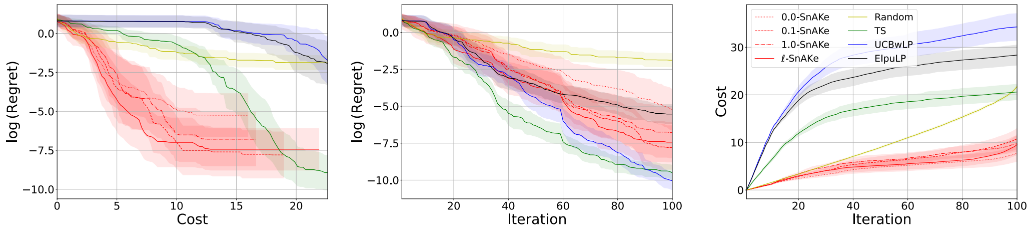


(a) $T = 100$

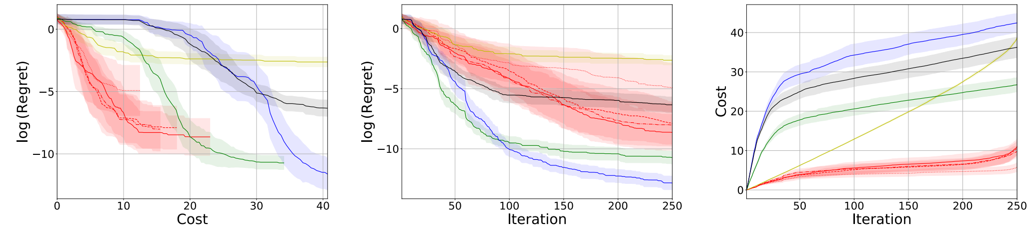


(b) $T = 250$

Figure 22: Michaelwicz2D (Asynchronous), $t_{delay} = 25$. Each row represents a different budget. The left column shows the evolution of regret against the cost used. The middle column shows the evolution of regret with iterations, and the right columns show the evolution of the 2-norm cost. Similar results to shorter delay, see Figure 21.

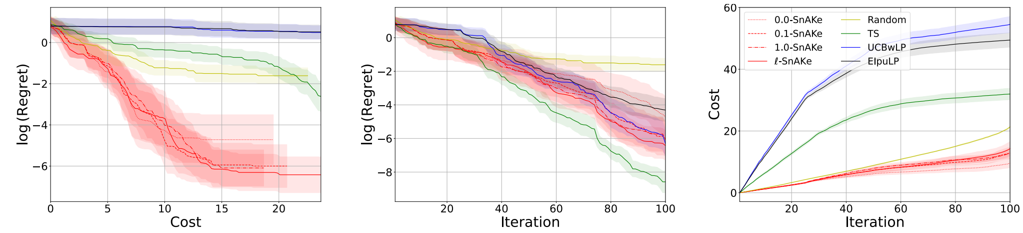


(a) $T = 100$

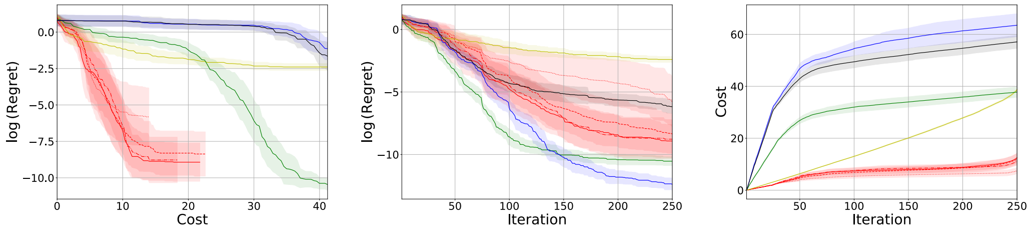


(b) $T = 250$

Figure 23: Hartmann3D (Asynchronous), $t_{delay} = 10$. Each row represents a different budget. The left column shows the evolution of regret against the cost used. The middle column shows the evolution of regret with iterations, and the right columns show the evolution of the 2-norm cost. SnAKe achieves the best regret for low cost, with Thompson Sampling also giving a good performance. For the full optimization, UCBwLP achieves the best regret, at the expense of four times the cost of SnAKe. EIpuLP performs poorly, again, it seems Local Penalization is over-powering the cost term.

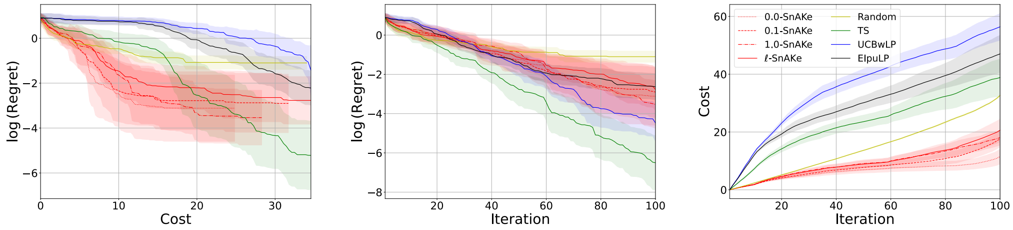


(a) $T = 100$

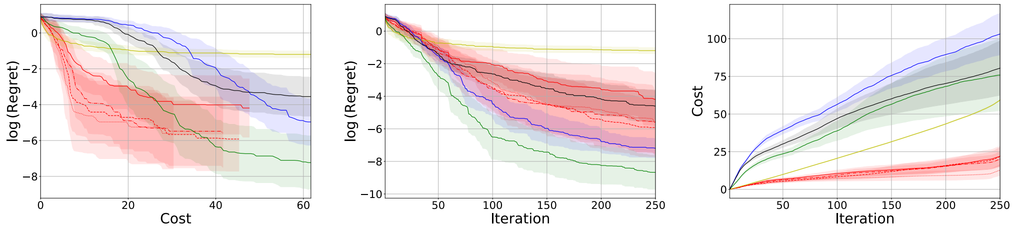


(b) $T = 250$

Figure 24: Hartmann3D (Asynchronous), $t_{delay} = 25$. Each row represents a different budget. The left column shows the evolution of regret against the cost used. The middle column shows the evolution of regret with iterations, and the right columns show the evolution of the 2-norm cost. Similar results to the case with smaller delay, see Figure 23.

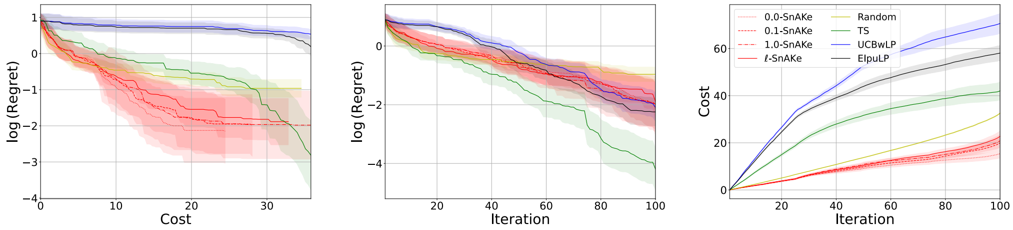


(a) $T = 100$

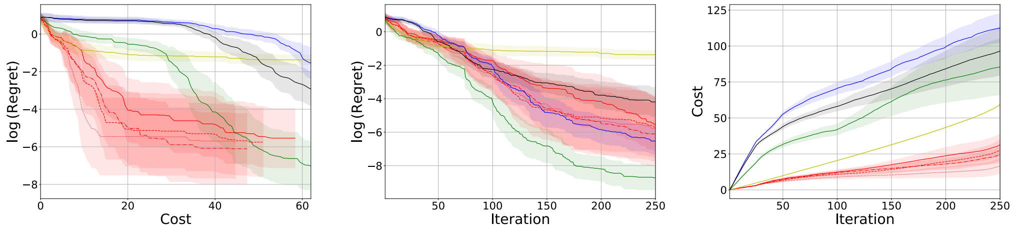


(b) $T = 250$

Figure 25: Hartmann4D (Asynchronous), $t_{delay} = 10$. Each row represents a different budget. The left column shows the evolution of regret against the cost used. The middle column shows the evolution of regret with iterations, and the right columns show the evolution of the 2-norm cost. Similar results to other Hartmann benchmarks, see Figure 23.

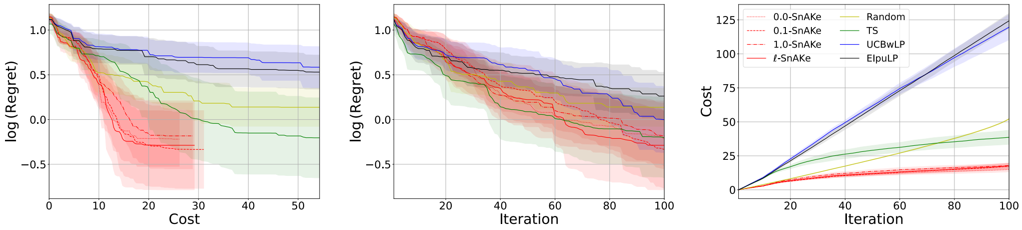


(a) $T = 100$

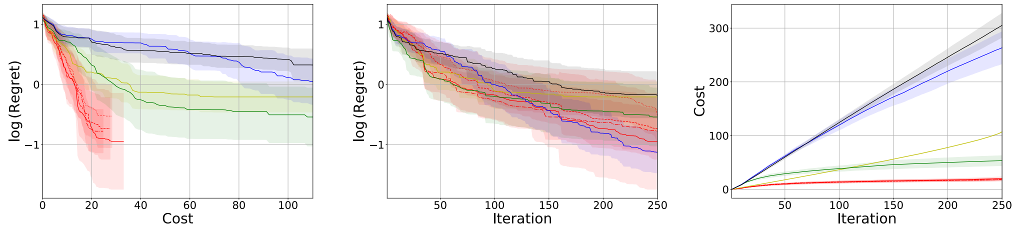


(b) $T = 250$

Figure 26: Hartmann4D (Asynchronous), $t_{delay} = 25$. Each row represents a different budget. The left column shows the evolution of regret against the cost used. The middle column shows the evolution of regret with iterations, and the right columns show the evolution of the 2-norm cost. Similar results to other Hartmann benchmarks, see Figure 23.

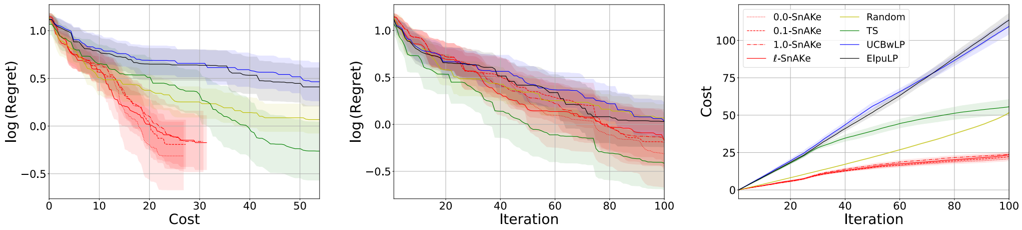


(a) $T = 100$

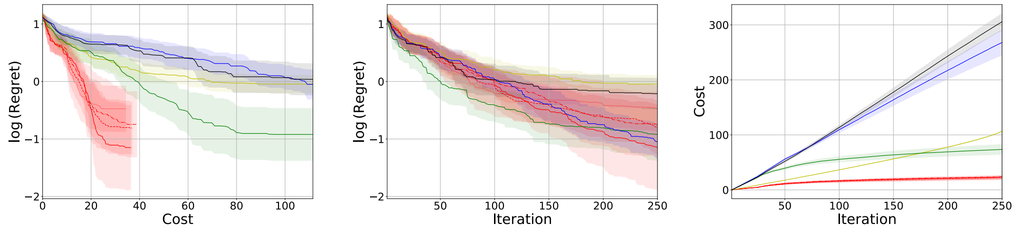


(b) $T = 250$

Figure 27: Hartmann6D (Asynchronous), $t_{delay} = 10$. Each row represents a different budget. The left column shows the evolution of regret against the cost used. The middle column shows the evolution of regret with iterations, and the right columns show the evolution of the 2-norm cost. Similar results to other Hartmann benchmarks, see Figure 23.

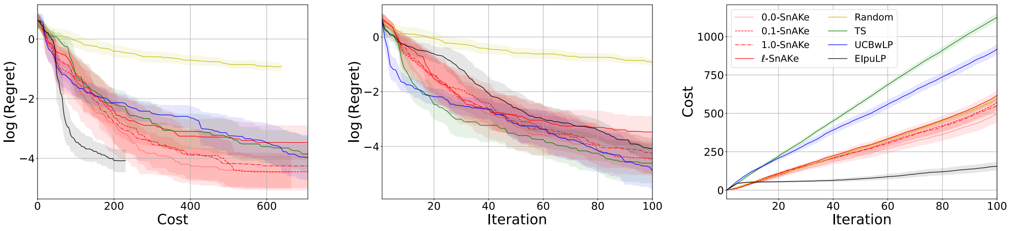


(a) $T = 100$

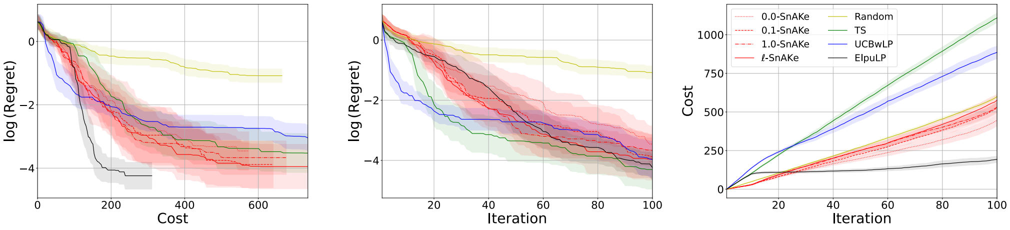


(b) $T = 250$

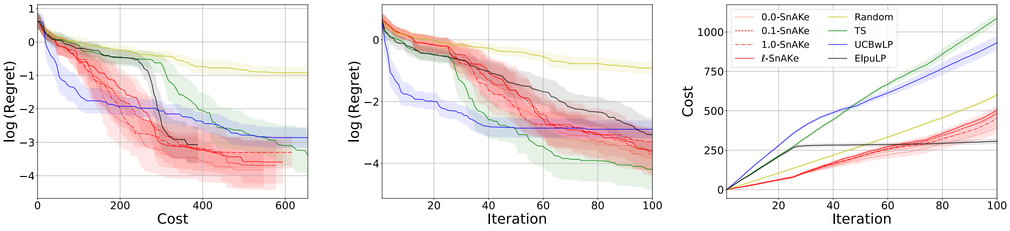
Figure 28: Hartmann6D (Asynchronous), $t_{delay} = 25$. Each row represents a different budget. The left column shows the evolution of regret against the cost used. The middle column shows the evolution of regret with iterations, and the right columns show the evolution of the 2-norm cost. Similar results to other Hartmann benchmarks, see Figure 23.



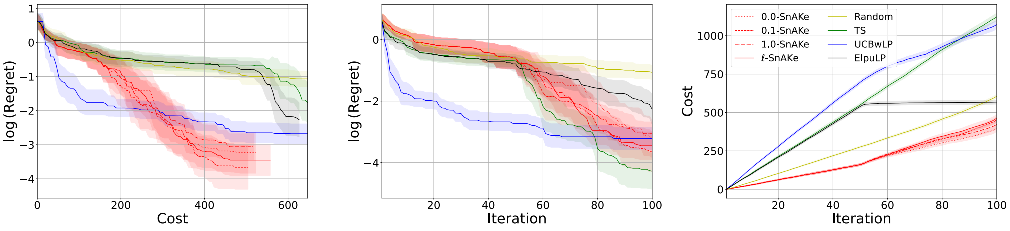
(a) $t_{delay} = 5$



(b) $t_{delay} = 10$

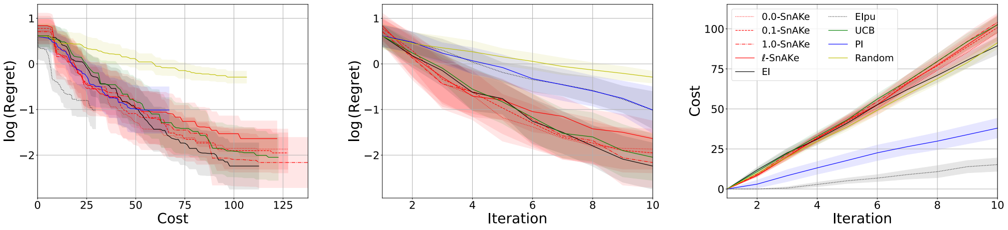


(c) $t_{delay} = 25$

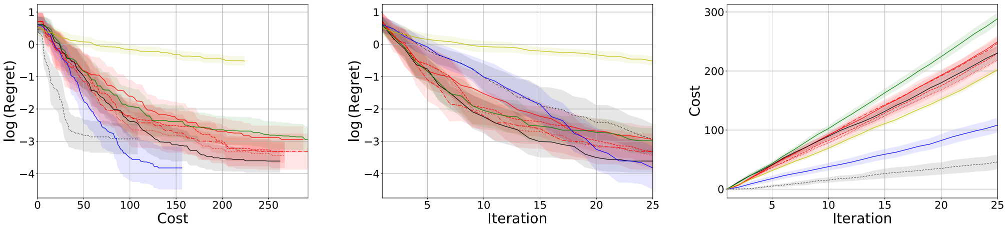


(d) $t_{delay} = 50$

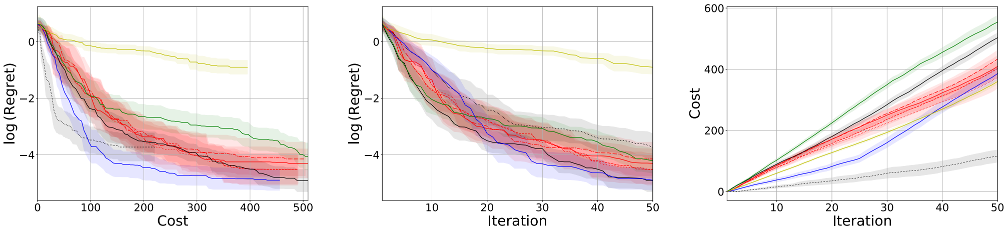
Figure 29: SnAr benchmark (Asynchronous) with $T = 100$. Each row represents a different t_{delay}. The left column shows the evolution of regret against the cost used. The middle column shows the evolution of regret with iterations, and the right columns show the evolution of the cost as defined in Section 4.2. SnAKe achieves the better regret than classical BO algorithms at low cost for all budgets. EIpuLP performs well for small delays, but poorly for larger delays.



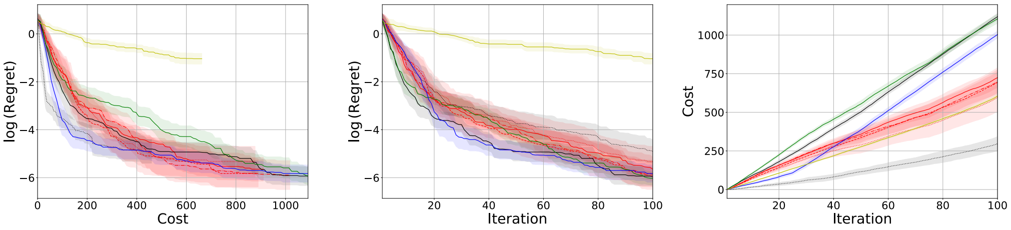
(a) $T = 10$



(b) $T = 25$



(c) $T = 50$



(d) $T = 100$

Figure 30: SnAr benchmark (synchronous, $t_{delay} = 1$). Each row shows a different budget. The left column shows the evolution of regret against the cost used. The middle column shows the evolution of regret with iterations, and the right columns show the evolution of the cost as defined in Section 4.2. SnAKe is the only method achieving low regret and low cost especially for larger budgets. EIpu generally achieves low cost but poor regret.