Efficiency Through Procrastination: Approximately Optimal Algorithm Configuration with Runtime Guarantees

Robert Kleinberg
Dept. of Computer Science
Cornell University
rdk@cs.cornell.edu

Kevin Leyton-Brown
Dept. of Computer Science
University of British Columbia
kevinlb@cs.ubc.ca

Brendan Lucier
Microsoft Research
brlucier@microsoft.com

Abstract

Algorithm configuration methods have achieved much practical success, but to date have not been backed by meaningful performance guarantees. We address this gap with a new algorithm configuration framework, *Structured Procrastination*. With high probability and nearly as quickly as possible in the worst case, our framework finds an algorithm configuration that provably achieves near optimal performance. Further, its running time requirements asymptotically dominate those of existing methods.

1 Introduction

Algorithm configuration techniques have received considerable study in artificial intelligence over the past decade. General-purpose procedures include ParamILS [Hutter *et al.*, 2007; 2009], GGA [Ansótegui *et al.*, 2009; 2015], irace [Birattari *et al.*, 2002; López-Ibáñez *et al.*, 2011] and SMAC [Hutter *et al.*, 2011b; 2011a]. All of these methods have demonstrated significant successes in practice. However, they are also all heuristic: they cannot assess how close they have come to finding an optimal algorithm configuration, and offer no theoretical guarantees about the running time they would require to find such a configuration.

We focus on such guarantees, and thus consider the worst-case expected performance of algorithm configuration methods. Specifically, we assume that an adversary causes every deterministic choice to prolong runtime as much as possible, while observations of random variables are unbiased samples from the underlying distribution. Under this analytic framework, gradient-following methods like ParamILS and GGA can perform very poorly: in the worst case, gradients will lead the search astray and optimal configurations will not be surrounded by large basins of attraction. (Of course, such methods can nevertheless perform well in practice, as extensive experimental evidence shows that they often do; our concern in this paper is exclusively with worst-case analysis rather than such empirical investigations.) Similarly, Bayesian optimization methods like SMAC can learn misleading models that will drive the search to investigate unpromising regions of the configuration space. However, as a hedge against a misleading model, SMAC guarantees that it spends half of its time examining randomly sampled configurations. Such samples are immune to the manipulations of an adversary. Indeed, SMAC is defined as an extended version of a model-free method, ROAR, that works entirely by such random sampling.

This paper proposes an algorithm configuration method, *Structured Procrastination*, that is similarly based on random sampling, but that is accompanied by a nontrivial runtime guarantee. All random-sampling-based methods like SMAC and ROAR eventually encounter the optimal configuration; the crucial question, therefore, is how long it takes for a method to do so. We call a method *incumbent driven* if it only ever sets timeouts for algorithm runs either at a fixed maximum timeout or at a predetermined fraction of the best runtime that some "incumbent" configuration has achieved on any problem instance. (This argument is made more formally in Section 3.) We demonstrate that our approach takes arbitrarily less time to find a configuration close to optimal than any incumbent-driven algorithm configurator, where this arbitrarily large gap depends on the ratio between the running time of any candidate incumbent on its best-case instance and the running time of the optimal configuration averaged across instances. All algorithm configuration methods of which we are aware are incumbent diven (notably, ParamILS, GGA, irace, SMAC, and ROAR);[1] thus, to our knowledge our method provides the best worst-case runtime guarantees.

The literature on multi-armed bandits provides another important source of related work. Indeed, our method is structurally similar to various methods from that literature, leveraging the influential "optimism in the face of uncertainty" paradigm [Auer *et al.*, 2002; Bubeck *et al.*, 2012], and we employ similar analytic tools. Furthermore, there has already

[1]However, the public software releases of both SMAC and ParamILS support experimental command-line options, never described in any publication, that produce non-incumbent-driven behavior. In both cases, preliminary experiments did not provide strong evidence that these options yielded substantially improved performance. Nevertheless, we describe them here for completeness. SMAC (and hence ROAR) has an option `--init-mode` that affects the initialization phase that precedes the subsequent, incumbent-driven search. This option can be used to direct SMAC to iteratively increase captimes until a sufficient fraction of preselected initialization configurations complete. ParamILS has an option `-id` that causes it to execute in stages of increasing time budgets and doubling captimes, where the incumbent from stage k is used as the initial configuration for stage $k+1$. Our subsequent discussion of SMAC, ROAR, and ParamILS focuses only on the algorithms as described in publications.

been considerable contact between the algorithm configuration and bandits communities, largely in the sphere of hyperparameter optimization [Bergstra *et al.*, 2011; Thornton *et al.*, 2013; Li *et al.*, 2016] and in the literature on bandits with correlated arms that are able to scale to large experimental design settings [Kleinberg, 2006; Kleinberg *et al.*, 2008; Chaudhuri *et al.*, 2009; Bubeck *et al.*, 2011; Srinivas *et al.*, 2012; Cesa-Bianchi and Lugosi, 2012; Munos, 2014]; see also the survey by Shahriari *et al.* [2016]. Nevertheless, the runtime minimization objective of algorithm configuration is crucially different from the more general objective functions targeted in most of the bandits literature. First, our cost of pulling an arm is measured in the same units as the objective function to be minimized; we aim to find the arm with the lowest expected cost (ranging over both instances and random seeds). Second, we have the freedom to set a maximum amount x we are willing to pay in pulling an arm; if the true cost exceeds x, we pay only x but learn only that the true cost was higher. In contrast, in a general bandits setting, pulling arms is an all-or-nothing operation. In most of the literature, all arms simply involve the same, fixed cost. An alternative body of work has variable costs and a fixed overall budget [Guha and Munagala, 2007; Tran-Thanh *et al.*, 2012; Badanidiyuru *et al.*, 2013]; here, however, one always pays the full cost associated with that arm (whether fixed or variable) and observes a sample from the underlying distribution. Because all methods with this property fail to avail themselves of the option to cap long runs before they terminate, these methods are again incumbent driven, and hence our separation theorem applies.

A few methods depart from this paradigm. In some cases, the algorithm can *specify* a maximum cost to be paid when pulling an arm, but never pays less than that specified budget [Kandasamy *et al.*, 2016]. Our setting is different (a run that does not time out costs less than its runtime budget to evaluate), and in the most natural way of translating such algorithms to our setting, they are again incumbent driven. Work by Ganchev *et al.* [2010] is similar because it considers bandits where observations are censored if they exceed a given budget. Nevertheless, this setting is different enough that there is no natural way to map its algorithm onto the algorithm configuration domain: Ganchev *et al.* consider a maximization (rather than minimization) objective; they also choose a vector of "allocations" across many arms at each time step that sum to an exogenously specified amount, rather than choosing only one arm to run for an endogenously specified amount of time.

Finally, the Hyperband algorithm of Li *et al.* [2016] is perhaps the closest to our work. They reason about a setting in which the quality of a hyperparameter configuration can be approximated with increasing fidelity as more time is devoted to assessing it. They propose an approach in which a large number n of configurations are assessed, each with a time budget B/n; the worst half of configurations are discarded; the remaining configurations are given doubled captimes ($B/0.5n$); and the process continues until only one configuration remains. This whole process runs inside an outer loop that performs a grid search trading off the number of initial configurations n against the initial captime B/n (i.e., holding B constant). Replacing the term "hyperparameter configuration" with "algorithm configuration", this approach would seem applicable to our setting. Indeed, both Hyperband and our own approach take inspiration from the bandits literature, so this similarity is more than superficial. However, it turns out to be highly nontrivial to fill in the details. First, given the decision to evaluate a configuration for time B/n, how should this time budget be allocated across the infinite stream of available (instance, seed) pairs? Second, after the allocated runs have completed for configuration i, how should the measured runtimes—some of which may represent capped runs—be aggregated to form an estimate of i's expected runtime across (instance, seed) pairs? (Li *et al.* require simply that these estimates converge to the correct answer as the time allocated to configuration i goes to infinity.) Third, when i is reconsidered with a longer time budget, how should the choices of which runs to preform and which performance estimates to return depend on previous runs? (Li *et al.* do not explicitly propose such dependence, since they consider arbitrary performance functions that do not decompose across (instance, seed) pairs; however, doing better than restarting from scratch at each iteration is critical in our domain, and indeed our core idea of "procrastination" addresses this issue.) We are not aware of any way to answer these questions within the Hyperband framework without obtaining a worst-case bound that is asymptotically worse than ours, and we suspect that it is not possible to do so. Nevertheless, we see our paper's main technical contribution as the adaptation of existing bandits approaches such as Hyperband to the algorithm configuration setting.

In what follows, we define the problem we address more formally (Section 2) and provide a lower bound on the runtime of incumbent-driven algorithm configuration approaches (Section 3). We then present our own approach, Structured Procrastination. We begin by considering the case of a small set of configurations among which we would like to identify the best (Section 4). We then consider large or continuous spaces of configurations, and aim to identify a configuration with performance in the top $1/\gamma$-quantile across the set of configurations, where γ shrinks as the algorithm runs (Section 5). In both cases, we show that our procedure approximately optimizes the desired objective with high probability. Its runtime is asymptotically better than the lower bound for incumbent-driven procedures developed earlier; in Section 6 we also show that it is optimal up to a logarithmic factor. Section 7 shows how Structured Procrastination can be extended to work with Bayesian optimization approaches such as SMAC and how it can achieve linear speedups when parallelized.

2 Problem Statement

We define an algorithm configuration problem by the 4-tuple (N, Γ, R, κ_0), where these elements are defined as follows. N is a family of (potentially randomized) algorithms, which we call *configurations* to suggest that a single piece of code instantiates each algorithm under a different parameter setting. We do not assume that different configurations exhibit any sort of performance correlations, and can so capture the case of n distinct algorithms by imagining a "master algorithm" with a single, n-valued categorical parameter. Parameters are allowed to take continuous values: $|N|$ can be uncountable. We typically use i to index configurations. Γ is a probability distri-

bution over input instances. When the instance distribution is given implicitly by a finite benchmark set, let Γ be the uniform distribution over this set. We typically use j to index (input instance, random seed) pairs, to which we will hereafter refer simply as instances. $R(i,j)$ is the execution time when configuration $i \in N$ is run on input instance j. Given some value of $\theta > 0$, we define $R(i,j,\theta) = \min\{R(i,j),\theta\}$, the runtime capped at θ. $\kappa_0 > 0$ is a constant such that $R(i,j) \geq \kappa_0$ for all configurations i and inputs j.

Informally, our goal is to find a configuration from N with the minimum average running time over the input space Γ and over its own random seeds. We are allowed to execute any configuration on any input and adaptively terminate any such execution at any time. More precisely, for any timeout threshold θ, let $R_\theta(i) = \mathrm{E}_{j\sim\Gamma}[R(i,j,\theta)]$ denote the average running time of configuration i, over distribution Γ of input instances. Fixing some running time $\bar{\kappa} = 2^\beta\kappa_0$ that we will never be willing to exceed, the quantity $R_{\bar{\kappa}}(i)$ corresponds to the expected running time of configuration i and will be denoted simply by $R(i)$. Given $\epsilon > 0$, a basic problem is to find $i^* \in N$ such that $R(i^*) \leq (1+\epsilon)\min_i\{R(i)\}$. It will turn out, from the standpoint of worst-case guarantees, that this goal is too ambitious because it is too hard to estimate $R(i)$ when the running-time distribution of i is very heavy-tailed. If the average running time is driven by a small set of bad inputs that occur very rarely, but induce configuration i to run for an astronomical number of steps when they occur, then the only way to accurately estimate the average running time of configuration i is to run it on enough inputs that we see enough of these "black swans." This pathological scenario leads to worst-case bounds that scale linearly with $\bar{\kappa}$ even when $\min_i\{R(i)\} \ll \bar{\kappa}$; see Section 6. To avoid an analysis aimed at such pathologies, we relax our objective by allowing the running time of i^* to be *capped* at some threshold value θ for some small fraction of (instance, seed) pairs δ.

Definition 2.1 *A configuration i^* is* (ϵ,δ)-optimal *if there exists some threshold θ such that $R_\theta(i^*) \leq (1+\epsilon)\min_i\{R(i)\}$, and $\Pr_{j\sim\Gamma}\left(R(i^*,j) > \theta\right) \leq \delta$. Otherwise, we say i^* is* (ϵ,δ)-suboptimal.

Example 2.2 *Consider* 3 *deterministic configurations, $N = \{C_1, C_2, C_3\}$, with Γ as the uniform distribution over* 1000 *input instances. Runtime is measured in ms, with $\kappa_0 = 1$ and timeout $\bar{\kappa} = 2^{20}$. Configuration C_1 has runtime* 10 *for all instances and hence $R(C_1) = 10$. Configuration C_2 has runtime* 1000 *for 10 out of the 1000 instances and runtime* 11 *for the remaining instances; i.e., $R(C_2) = 20.89$. Configuration C_3 has runtime* 1000 *for 100 input instances, runtime* 100 *for 100 input instances, and runtime 5 for the remaining instances; i.e., $R(C_3) = 114$. C_1 is optimal. Since $R_{11}(C_2) = 11$ and $\Pr[R(C_2,j) > 11] = 0.01$, C_2 is $(0.1, 0.01)$-optimal. Since $R_5(C_3) = 5 \leq 10$ and $\Pr[R(C_3,j) > 5] = 0.2$, C_3 is $(0, 0.2)$-optimal. Additionally, $R_{100}(C_3) = 24$ and $\Pr[R(C_3,j) > 100] = 0.1$, so C_3 is also $(1.4, 0.1)$-optimal.*

3 Runtime of Incumbent-Driven Procedures

In this section we develop a lower bound on the runtime of what we call *incumbent-driven* approaches, which as discussed above include most widely used algorithm configuration procedures. We describe algorithm configurators' queries of target algorithm runtimes via the subroutine $\textsc{Run}(i,j,\theta)$, which executes configuration i on instance j and caps the run if it does not complete after θ seconds. This subroutine takes time $R(i,j,\theta)$, and returns the value $R(i,j,\theta)$.

We say that a search procedure is *incumbent-driven* if, whenever a query $\textsc{Run}(i,j,\theta)$ is made, it must be that either $\theta = \bar{\kappa}$ or $\theta \geq \textsc{Run}(i',j',\theta')$ for some query $\textsc{Run}(i',j',\theta')$ that was executed previously. Such procedures can be thought of as either performing unconditional algorithm runs (capping only at the maximum runtime $\bar{\kappa}$) or maintaining an 'incumbent' best algorithm and using previously observed runtimes to bound the amount of time needed to run an alternative 'challenger' algorithm before determining which is preferable (as is done, e.g., by the "adaptive capping" mechanisms in ParamILS, ROAR, and SMAC).

Example 3.1 *Consider a discrete set N of n configurations, indexed so that $R(1) < R(2) < \ldots < R(n)$. Each configuration's runtime is tightly concentrated around its expectation, so that i has runtime nearly $R(i)$ on every input instance. Moreover, R_2/R_1 is large, meaning that configuration* 1 *is significantly faster than the others. Any incumbent-driven search procedure must begin by choosing a configuration and executing it until completion. Since the configurations are initially indistinguishable, the choice of which to execute is essentially random, so this takes time at least R_2, with probability at least $1 - 1/n$ (the probability that algorithm* 1 *is not chosen).*

In comparison to the example above, our search procedure takes time rougly proportional to nR_1. If n is smaller than the ratio R_2/R_1, this can be a significant improvement. The same idea can be formalized to show that any incumbent-driven search procedure can experience arbitrarily poor runtime performance with arbitrarily high probability. Here we handle the more realistic setting of a continuum of candidate configurations, by speaking about the fraction of configurations with favorable runtimes; see Section 5 for details.

Theorem 3.2 *For any incumbent-driven search procedure, any $\gamma > 0$, and any runtimes $R_1 < R_2$, there is an algorithm configuration problem in which a γ fraction of all possible configurations have average runtime no greater than R_1, but with probability at least $(1-\gamma)$ the search procedure will require time at least R_2.*

4 The Case of Few Configurations

We now describe our proposed algorithm configuration method. We begin by considering a case where the family N of configurations is finite and relatively small; we consider the alternative in Section 5. Let $|N| = n$ and write $OPT = \min_i R(i)$. Algorithm configuration in this case boils down to selecting the configuration with smallest expected runtime from a fixed set of alternatives. This problem may seem straightforward: we could simply run every configuration on every instance (capping at $\bar{\kappa}$). The catch is that we care about how long our procedure takes to run. To speed things up, we could sample instances at random from Γ, run every configuration on the sampled instances, and use Chernoff bounds

Algorithm 1: Structured Procrastination (few configs)

require : Set N of n algorithm configurations
require : Precision parameter $\epsilon \in (0, \frac{1}{3})$
require : Failure probability parameter $\zeta \in (0, 1)$
require : Lower and upper runtime bounds, κ_0 and $\bar{\kappa}$
require : Sequence $j_1, j_2, \ldots$ of (instance, seed) pairs

// Initializations
$\beta = \log_2(\bar{\kappa}/\kappa_0)$
for $i \in N$ **do**
  $k_i := 0$
  $\ell_i := \lceil 12\epsilon^{-2} \ln(3\beta n/\zeta) \rceil$
  $Q_i :=$ empty double-ended queue
  for $\ell = 1, \ldots, \ell_i$ **do**
    $R_{i\ell} := 0$
    Insert (ℓ, κ_0) at tail of Q_i

// Main loop. Run until interrupted.
repeat
  $i := \arg\min_{i \in N} \left\{ \frac{1}{k_i} \sum_{\ell=1}^{k_i} R_{i\ell} \right\}$
  Remove (ℓ, θ) from head of Q_i
  if $R_{i\ell} = 0$ **then** *// j_ℓ is a fresh instance*
    $k_i := k_i + 1$
    $q_i := \lceil 12\epsilon^{-2} \ln(3\beta n(k_i)^2/\zeta) \rceil$
  if RUN(i, j_ℓ, θ) *terminates in time* $t \leq \theta$ **then**
    $R_{i\ell} := t$
  else
    $R_{i\ell} = \theta$
    Insert $(\ell, 2\theta)$ at tail of Q_i
  while $|Q_i| < q_i$ **do** *// Replenish queue*
    $\ell_i := \ell_i + 1$
    $R_{i,\ell_i} := 0$
    Insert (ℓ_i, θ) at head of Q_i
until *anytime search is interrupted*
return $i^* = \arg\max_{i \in N} \left\{ \sum_{\ell=1}^{k_i} R_{i\ell} \right\}$, $\delta = \frac{\sqrt{1+\epsilon}\, q_{i^*}}{k_{i^*}}$

to bound the probability that we select an configuration with performance within some given constant factor of optimal. Or we could adopt a racing procedure such as F-Race [Birattari *et al.*, 2002], evaluating configurations round robin and discarding those that statistically sufficient evidence shows are suboptimal (F-Race uses the nonparametric Friedman test). Observe that such racing procedures are incumbent driven.

This work shows that we can do even better, introducing the first algorithm configuration method with worst-case running time guarantees that are superior to all of these procedures. We call our method *Structured Procrastination*, in homage to the eponymous time management technique due to Stanford philosopher John Perry [1996] (for which he received the 2011 Ig Nobel Prize in Literature [Improbable Research, 2011]). In essence, Perry proposes maintaining a set of daunting and apparently important tasks that one procrastinates to avoid, thereby accomplishing other tasks. Eventually, each daunting task is superseded by a new task that is even more daunting, and is completed in its turn.

Similarly, our approach maintains lists of tasks (for each configuration i, a bounded-length queue Q_i of (instance, captime) pairs describing runs that i should perform, initialized with instances constructed by randomly sampling from Γ and randomly sampling a seed, and with aggressive captimes of κ_0), and procrastinates when these tasks prove daunting (need to be retried with doubled captimes). We maintain lower bounds on the expected runtime achieved by every configuration i, assigning expected runtime of zero to configurations on which no runs have yet been performed and otherwise averaging observed runtimes, treating capped runs as though they completed at their captimes. We then choose the task that we forecast will be easiest: the (instance, captime) pair at the head of the queue corresponding to the configuration i for which the lower bound on expected runtime is smallest. If the run does not complete within the specified captime, we will double the captime and try again. But doing so is a harder problem, so we procrastinate, adding this task to the tail of queue Q_i, and instead choosing the easiest available task. The key way this approach differs from past work—and the reason it is not incumbent based—is its reliance on the idea of procrastination: we only ever spend a long time running a given configuration on a given instance after having failed to find any different instance that could be evaluated more quickly.

To run our method, the user must specify a precision parameter ϵ (how far solutions can be from optimal), a failure probability ζ (the maximum probability with which our guarantees can fail to hold), a maximum amount of time $\bar{\kappa}$ that any configuration could ever be run, and a minimum amount of time κ_0 below which configuration running times can be considered identical. The failures captured by ζ are due to the unlikely events that sampled instances are atypical; in our proofs below we bound this failure probability with specially-tailored statistical tests based on Chernoff and union bounds. As discussed above, we will identify (ϵ, δ)-optimal configurations; hence, we must specify δ, the fraction of "outlying" instances on which the returned configuration's running time may be capped. Unlike the other parameters, we do not believe that this is one that a user will know how to set in advance. Structured Procrastination thus runs in an anytime manner, gradually reducing δ; when it is stopped, it returns the δ for which its guarantee holds. We note that our algorithm does not return the configuration with the best empirical estimate, as might seem most intuitive. Observe that our algorithm could spend most of its time testing one configuration, but right before anytime search is interrupted, a few outliers could inflate its estimate beyond that of a second configuration that received little scrutiny. We could not return the latter configuration and maintain strong statistical guarantees. Instead, our approach amounts to choosing the configuration that was most often the empirically best.

In more detail, for each configuration i there is a FIFO queue Q_i, containing pairs (ℓ, θ) where ℓ is the index of an element j_ℓ in an infinite sequence of i.i.d. samples from the input distribution Γ and θ represents twice the timeout threshold that was applied the last time we attempted to run configuration i on instance j_ℓ. (The same sequence is used for every configuration, producing a blocking design [Dean and Voss, 1999].) When j_ℓ reaches the head of the queue we will attempt to run

it for θ steps. In this way, the timeout threshold of any input instance j progressively doubles until we have successfully completed j. When an input instance j completes, we select the next "fresh" instance from the infinite sequence of random instances and insert it into the back of the queue, matching its timeout threshold with that of the element ahead of it. (Occasionally more than one such element is inserted because the desired queue length, q_i, has increased.) In this way, we ensure three invariants. First, at any point in time, when the head of the queue is a pair (ℓ, θ), then all of the elements in the queue have a timeout threshold of either θ or 2θ. Second, each input instance that configuration i has already finished solving was completed in θ or fewer steps. Third, the queue's contents are arranged in non-decreasing order of timeout threshold.

Every instance j that has ever been inserted into Q_i is either *fresh, completed, or deferred*, depending on whether the algorithm has never attempted to run configuration i on instance j, has run i on j until termination, or has attempted run i on j at least once but all such runs have timed out. The variable k_i stores the number of completed or deferred instances, whereas ℓ_i stores the number of fresh, completed, or deferred instances. The value $R_{i\ell}$ stores the amount of time taken by the most recent call to $\textsc{Run}(i, j, \theta)$ for the instance $j = j_\ell$: it equals the capped running time $R(i, j, \theta)$ if j is completed or deferred, or 0 if j is fresh. The quantity $\frac{1}{k_i}\sum_{\ell=1}^{k_i} R_{i\ell}$ therefore represents the average θ-capped running time of all the completed and deferred instances of configuration i, a quantity which we shall denote by $R_\theta^{\text{emp}}(i)$ henceforth. In each iteration of the main loop, the structured procrastination algorithm selects the configuration with the minimum average capped running time and processes the next element in that configuration's queue.

Example 4.1 *Consider our algorithm's execution on Example 2.2, say with $\epsilon = 0.2$ and failure probability $\zeta = 0.1$. There are 3 queues, Q_1, Q_2, Q_3, corresponding to configurations C_1, C_2, and C_3. Each queue is initialized with ℓ_i input instances, which in this example is approximately 2248. Note that this is more than the total number of distinct input instances in our example; while we know that the number of inputs is limited, the search procedure does not know this* a priori *and its statistical tests suggest a minimum of this many inputs to yield the desired guarantees.*

As instances are drawn from the queues (in round-robin fashion, since the configurations appear identical at first), their runtime caps are doubled until we reach threshold 8. At this point, assuming events follow expectations, roughly 80% of instances in Q_3 terminate with a runtime of 5 while all other instances time out at 8. This causes C_3 to have lower estimated average runtime than C_1 or C_2, so Algorithm 1 begins testing instances only from Q_3. As instances are completed from C_3's queue, they are replaced with fresh instances, some of which will also complete with a runtime of 5. Any instance that times out is sent to the back of the queue. If the search procedure is terminated at this point, C_3 will be returned, and the corresponding δ will be greater than 0.2.

Eventually, Q_3 becomes filled with instances that timed out at $\theta = 8$. These instances must then be executed with higher thresholds, which slowly increases the runtime estimate for C_3. This estimate grows and eventually overtakes the estimates for C_1 and C_2. Algorithm 1 then resumes testing instances from Q_1 and Q_2. Eventually we evaluate instances from Q_1 with timeout $\theta = 16$. At this point, all instances from Q_1 terminate with runtime 10, and eventually the runtime estimate for C_1 drops below that of C_2 and C_3. The search procedure will return C_1 if terminated after this point. Note that Algorithm 1 continues to evaluate instances from Q_1 until termination. This can be interpreted as searching for outlier instances on which C_1 has very poor runtime (which, in this example, do not exist). The longer the search continues, the more rare we conclude such outliers (if any) must be, and correspondingly the value of δ returned becomes smaller and smaller.

To justify this greedy rule of always choosing the configuration with minimum average capped runtime, we will show that, with high probability, we spend only a limited amount of time testing any individual (ϵ, δ)-suboptimal configuration. Consider any configuration i that satisfies $R(i) > (1+\epsilon)\text{OPT}$. By Lebesgue's Monotone Convergence Theorem, we know that $\lim_{\theta\to\infty} R_\theta(i) = R(i)$, so let $\theta(i)$ be the smallest timeout threshold such that $R_{\theta(i)} > (1+\epsilon)\text{OPT}$ and $\theta(i)/\kappa_0$ is a power of 2. We will show that, with high probability, the Structured Procrastination algorithm will never attempt to run configuration i with timeout threshold $\theta(i)$. This will, in turn, imply an upper bound on the total running time devoted to experimenting with configuration i. In what follows, given a configuration i we will tend to write θ for the largest value such that each deferred instance in Q_i has been executed with a threshold of at least θ.

Definition 4.2 *An execution of the Structured Procrastination algorithm is called* clean *if it satisfies the following properties for every configuration $i \in N$, at all times during its execution.*

$$(1+\epsilon)^{-1/2} \Pr_{j\sim\Gamma}\big(R(i,j) > \theta\big) \le \frac{q_i}{k_i} \qquad (1)$$

$$(1+\epsilon)^{-1/2} R_\theta(i) \le R_\theta^{emp}(i) \le (1+\epsilon)^{1/2} R_\theta(i) \qquad (2)$$

Lemma 4.3 *An execution of the Structured Procrastination algorithm is clean with probability at least $1-\zeta$.*

Proof Sketch: We use Chernoff bounds to verify that empirical averages of i.i.d. random variables are unlikely to deviate from their expected values by a factor greater than $(1+\epsilon)^{\pm 1/2}$. We adjust the constants in these bounds to account for the fact that the multiplicative error factor is expressed as $(1+\epsilon)^{\pm 1/2}$ instead of $1 \pm \epsilon$. For (1) we define the random variables by $X_\ell = 1$ if $R(i, j_\ell) > \theta$ and $X_\ell = 0$ otherwise. For (1) we define the random variables by $X_\ell = \frac{1}{\theta} R(i, j_\ell, \theta)$. ∎

During a clean execution, if an iteration of the main loop selects configuration i in Line 13, then Lemma 4.3 implies

$$\begin{aligned} R_\theta(i) \le (1+\epsilon)^{1/2} R_\theta^{\text{emp}}(i) &\le (1+\epsilon)^{1/2} R_\theta^{\text{emp}}(i^{\text{opt}}) \\ &\le (1+\epsilon) R_\theta(i^{\text{opt}}) \le (1+\epsilon)\text{OPT}. \end{aligned}$$

This implies that $\theta \le \theta(i)$. Consequently, if we let $\delta(i)$ denote the fraction of input instances j (under distribution Γ) that satisfy $R(i,j) \ge \theta(i)$, then $\Pr_{j\sim\Gamma}(R(i,j) \ge \theta) \ge \delta(i)$.

Lemma 4.3 now implies that

$$\frac{q_i}{k_i} \;\geq\; (1+\epsilon)^{-1/2} \Pr_{j\sim\Gamma}\big(R(i,j) \geq \theta\big) \;\geq\; \frac{\delta(i)}{(1+\epsilon)^{1/2}}$$

$$k_i \;\leq\; \frac{(1+\epsilon)^{1/2} q_i}{\delta(i)} \;\leq\; \frac{12(1+\epsilon)^{1/2}}{\delta(i)\epsilon^2} \ln\left(\frac{3\beta n k_i^2}{\zeta}\right). \quad (3)$$

The equation implies an upper bound on k_i, but it requires a tedious calculation to extract the upper bound because the right side depends on k_i. Assuming the failure probability parameter ζ is less than $\beta n/600$, this calculation implies

$$k_i < \frac{30(1+\epsilon)^{1/2}}{\delta(i)\epsilon^2} \ln\left(\frac{3\beta n}{\zeta\delta(i)\epsilon^2}\right). \quad (4)$$

Our next theorem shows that for every δ, there exists a bounded amount of time after which our algorithm outputs an (ϵ,δ)-optimal configuration with high probability.

Theorem 4.4 *For any $\delta > 0$, if an execution of the Structured Procrastination algorithm is halted at any time*

$$T \geq \frac{104}{\epsilon^2} \ln\left(\frac{3\beta n}{\zeta\delta\epsilon^2}\right) \sum_{i\in N} \min\left\{\frac{1}{\delta},\, \frac{1}{\delta(i)}\right\} \mathit{OPT}$$

then it outputs an (ϵ,δ)-optimal configuration with probability at least $1-\zeta$.

Proof: With probability $1-\zeta$ the execution is clean. We will show that a clean execution running for T or more steps can never output an (ϵ,δ)-suboptimal configuration. If i is (ϵ,δ)-suboptimal then $\delta(i) \geq \delta$. Above we have argued that over the entire course of a clean execution, k_i satisfies the upper bound in (4). The total running time invested in configuration i up to time T is $\sum_{\ell=1}^{k_i} R_{i\ell}$, which is at most $3k_i R_\theta^{\text{emp}}$, where θ denotes the largest value such that each deferred instance in Q_i has been executed with a threshold of at least θ. Owing to the greedy rule that was used to select configuration i at that time, we know that the inequality $R_\theta^{\text{emp}}(i) \leq R_\theta^{\text{emp}}(i^{\text{opt}})$ held at that time, and hence $R_\theta^{\text{emp}}(i) < (1+\epsilon)^{1/2}\text{OPT}$ by Lemma 4.3. Multiplying this inequality by (4), and using the fact that $\epsilon < \frac{1}{3}$, we find that the total time spent running configuration i is bounded by

$$k_i \cdot R_\theta^{\text{emp}}(i) < \frac{104}{\delta(i)\epsilon^2} \ln\left(\frac{3\beta n}{\zeta\delta\epsilon^2}\right) \text{OPT}, \quad (5)$$

leveraging our assumption that $\epsilon < \frac{1}{3}$.

Let $N(\epsilon,\delta) = \{\text{all } (\epsilon,\delta)\text{-suboptimal configurations}\}$. Subtracting the right side of (5) for each $i \in N(\epsilon,\delta)$ from T, the remainder is at least $|N \setminus N(\epsilon,\delta)| \cdot \frac{104}{\delta\epsilon^2} \ln\left(\frac{3\beta n}{\zeta\delta\epsilon^2}\right)$ OPT. By the pigeonhole principle, at least one of the $|N \setminus N(\epsilon,\delta)|$ configurations that is (ϵ,δ)-optimal has run for at least $\frac{104}{\delta\epsilon^2} \ln\left(\frac{3\beta n}{\zeta\delta\epsilon^2}\right)$ time steps before time T. This is greater than the amount of time that has been invested in running any of the (ϵ,δ)-suboptimal configurations, by (5) and the fact that $\delta(i) > \delta$ for every (ϵ,δ)-suboptimal configuration i. Therefore, when the Structured Procrastination algorithm is stopped at time T and it outputs the i^* which has run for the greatest total amount of time thus far, if the execution is clean then it will choose an (ϵ,δ)-optimal configuration. ∎

5 The Case of Many Configurations

Our method of choosing among a relatively small set of configurations, while interesting (and useful for what follows), is far from a general algorithm configuration method. The catch is that our runtime guarantee is linear in n, the number of configurations. This is unavoidable when each configuration must be considered explicitly, since if a configuration is not tried there is no way of being sure that it is not dramatically faster than all others. This is not too high a price to pay when n is small. In a general algorithm configuration application, however, it is common for parameters to number in the dozens or even hundreds, and furthermore for some parameters to have continuous domains. Thus, an approach based on explicitly testing each configuration cannot work.

We must therefore relax the requirement that we identify a configuration with performance close to that of the very best one—this best configuration could be arbitrarily better than others, and hidden by an adversary in the last part of the search space that we check, making it a 'needle in a haystack' that is effectively impossible to find. Instead, our relaxed objective will be to find a configuration with performance in the top $\lfloor 1/\gamma \rfloor$-quantile. (Alternately, we will seek a configuration with performance close to the best one that remains after we exclude the γ fraction of fastest configurations from N, treating rare but exceptionally good configurations as outliers.) Our approach to achieving this relaxed goal is to use a variant of Algorithm 1 to find the (approximately) best configuration from among a set of configurations sampled from the population. This random sample serves as a net with which we search the configuration space. Our search procedure will incrementally tighten this net by sampling additional configurations as necessary. While the Structured Procrastination method from Section 4 improves (i.e., reduces) the parameter δ over time, our modified algorithm will gradually reduce both δ and the fraction γ of configurations treated as outliers.

Our procedure, given as Algorithm 2, builds on Algorithm 1. It runs in phases, with additional configurations sampled at the end of each phase. Each phase runs approximately twice as long as the last. N_p is the set of configurations being tested in phase $p \geq 1$. When the search procedure is terminated, it returns the configuration that had the longest total execution time in the most recently completed phase. It is parameterized by $\omega > 0$, which controls the rate at which δ is improved relative to the number of configurations searched: the algorithm will return a configuration that is $(\epsilon, 1/n^\omega)$-optimal with respect to the best configuration in a sample of size n, where n grows over time.

We are now ready to state the performance guarantee of this modified Structured Procrastination method. Given a set N' of configurations, write $\text{OPT}(N') = \min_{i\in N'} R(i)$.

Theorem 5.1 *For any $n \geq n_0$ there exists some $n' \geq n$ such that if Algorithm 2 is halted at any time*

$$T \geq \frac{208 n^{1+\omega}}{\epsilon^2} \ln\left(\frac{3n^{1+\omega}}{\zeta^2\epsilon^2}\right) \mathit{OPT}(N')$$

where $N' = \{i_1, \ldots, i_{n'}\}$ (from Algorithm 2), then it outputs a configuration that is $(\epsilon, n^{-\omega})$-optimal with respect to N' with probability at least $1-\zeta$.

Algorithm 2: Structured Procrastination (many configs)

require : Minimum number of configurations n_0
require : Tradeoff parameter $\omega > 0$
require : Precision parameter $\epsilon \in (0, \frac{1}{3})$
require : Failure probability parameter $\zeta \in (0, 1)$
require : Lower and upper runtime bounds, κ_0 and $\bar{\kappa}$
require : Sequence $i_1, i_2, \ldots$ of configurations
require : Sequence $j_1, j_2, \ldots$ of (instance, seed) pairs

// Initializations
Define function $\mathcal{T}(n) := 40n^{1+\omega}\epsilon^{-2}\ln(3n^{1+\omega}/\zeta^2\epsilon^2)$
$n := n_0$, $N_0 := \emptyset$, T:=0, $\beta := \log_2(\bar{\kappa}/\kappa_0)$

// Main loop. Run until interrupted.
for $p = 1, 2, 3, \ldots$ **do**
  // Ensure that we have n initialized queues
  $N_p := \{i_1, \ldots, i_n\}$, $T_{ip} := 0$ for all $i = 1, \ldots, n$
  for $i \in N_p \backslash N_{p-1}$ **do**
    $k_i := 0$
    $\ell_i := \lceil 12\epsilon^{-2}\ln(3\beta n/\zeta)\rceil$
    $Q_i :=$ empty double-ended queue
    for $\ell = 1, \ldots, \ell_i$ **do**
      $R_{i\ell} := 0$
      Insert (ℓ, κ_0) at tail of Q_i
  repeat
    $i := \arg\min_{i \in N} \left\{ \frac{1}{k_i} \sum_{\ell=1}^{k_i} R_{i\ell} \right\}$
    Remove (ℓ, θ) from head of Q_i
    if $R_{i\ell} = 0$ **then** *// j_ℓ is a fresh instance*
      $k_i := k_i + 1$
    $q_i := \lceil 12\epsilon^{-2}\ln(3\beta n(k_i)^2/\zeta)\rceil$
    if RUN(i, j_ℓ, θ) *terminates in time* $t \le \theta$ **then**
      $R_{i\ell} := t$
    else
      $R_{i\ell} := \theta$
      Insert $(\ell, 2\theta)$ at tail of Q_i
    $T := T + R_{i\ell}$ *// Total execution time*
    $T_{i,p} := T_{i,p} + R_{i\ell}$ *// T, restricted to i and p*
    while $|Q_i| < q_i$ **do** *// Replenish queue*
      $\ell_i := \ell_i + 1$
      $R_{i,\ell_i} := 0$
      Insert (ℓ_i, θ) at head of Q_i
  until $T > \mathcal{T}(n)$
  $n := 2^{1/(1+\omega)}n$ *// Increase n for next phase*
$p^* := p - 1$ *// Return config that was run most last phase*
return $i^* = \arg\max_{i \in N_{p^*}} \{T_{i,p^*}\}$, $\delta = \frac{\sqrt{1+\epsilon}\, q_{i^*}}{k_{i^*}}$

Proof: With probability $1 - \zeta$ the execution is clean, as described in Definition 4.2. We claim that in a clean execution, after the most recently-completed phase p prior to termination at time T, we have $|N_p| \ge n$ and the configuration returned by the procedure is $(\epsilon, 1/n^\omega)$-optimal with respect to the set N_p. The argument closely follows the proof of Theorem 4.4. The main difference is to consider only execution steps taken in phase p, when at least n configurations were under consideration. As in Theorem 4.4 one can bound the number of steps taken by any $(\epsilon, 1/n^\omega)$-suboptimal configuration in phase p, and thereby argue that the configuration that executes for the most steps in phase p must be $(\epsilon, 1/n^\omega)$-optimal. ∎

We can interpret the sample of configurations as a net that searches for potentially rare configurations with good performance. To that end, define N_γ to be all configurations except the γ fraction of fastest ones, and write $\text{OPT}_\gamma = \text{OPT}(N_\gamma)$ for the runtime of the best configuration in N_γ.

Corollary 5.2 *For any $n \ge n_0$ and any $\gamma \in (0, 1)$, if the Structured Procrastination algorithm is run using a random sequence of configurations, and it is halted at any time*

$$T \ge \frac{208n^{1+\omega}}{\epsilon^2} \ln\left(\frac{3n^{1+\omega}}{\zeta^2\epsilon^2}\right) OPT_\gamma$$

then with probability at least $1 - \zeta - e^{-\gamma n}$ it outputs a configuration that is $(\epsilon, n^{-\omega})$-optimal with respect to N_γ

Proof: We apply Theorem 5.1 and note that the probability that none of the γ-fraction of fastest configurations appear in a uniform sample of n configurations is at most $e^{-n\gamma}$. ∎

6 Near-Optimality of Runtime

In this section, we show that the worst-case guarantees for Structured Procrastination, Theorem 4.4 and Corollary 5.2, are optimal up to logarithmic factors.

Theorem 6.1 *Suppose an algorithm configuration procedure is guaranteed to select an (ϵ, δ)-optimal configuration with probability at least $\frac{1}{2}$. In the setting with a finite number $n = |N|$ of configurations, the worst-case expected running time of the procedure must be at least $\Omega\big(\frac{n}{\delta\epsilon^2} OPT\big)$. In the setting with many configurations, letting $\delta = n^{-\omega}$ and $\gamma = \frac{\ln 2}{n}$ (so that $1 - e^{-\gamma n} = \frac{1}{2}$), the worst-case expected running time of the procedure must be at least $\Omega\big(\frac{n^{1+\omega}}{\epsilon^2} OPT_\gamma\big)$.*

Proof: Suppose every time an algorithm solves an input instance, it is either a "safe run" that deterministically takes time κ or a "risky run" that either takes κ or κ/δ, depending on the configuration's random seed. For all configurations, the probability of a risky run is 4δ. The probability that $R(i, j) = \kappa/\delta$, conditional on the run being risky, is $\frac{1}{2}$ for most configurations (the "standard" configurations), but for a $\frac{1}{n}$ fraction of configurations (the "special" configurations), this conditional probability is $\frac{1}{2} - \epsilon$. The special configurations will then be the only ones that are (ϵ, δ)-optimal. We will show that in expectation, the search procedure executes at least $\Omega(\frac{n}{\delta\epsilon^2})$ operations of the form RUN(i, j, θ) where $\theta > \kappa$. Since each such operation takes time at least κ, and OPT $= O(\kappa)$, this will imply the stated lower bound for n configurations. The extension to many configurations follows by supposing we have $n^{1+\omega}$ configurations in total, and n^ω of them are special, so that with constant probability there is exactly one special configuration among the n sampled ones.

Before embarking on the proof that the expected number of calls to RUN(i, j, θ) with $\theta > \kappa$ is $\Omega(\frac{n}{\delta\epsilon^2})$, let us justify this

bound in intuitive terms. Selecting an (ϵ, δ)-optimal configuration requires **(a)** searching through enough configurations to find a special one (which explains the factor of n in the lower bound), **(b)** running each of the configurations enough times to observe some slow runs (which explains the $1/\delta$), and **(c)** observing enough slow runs to estimate their occurrence probability to within a $1 - 2\epsilon$ factor (which explains the $1/\epsilon^2$).

To formalize this reasoning, we define the *history* of an execution of the procedure to consist of the sequence of $\text{RUN}(i, j, \theta)$ operations the procedure invoked, along with the running times of those operations. The length-t initial history consists of the first t elements in this sequence. Fixing a search procedure, let Q_0 (resp., Q_0^t) denote the distribution over histories (resp., length-t initial histories) in a "null model" where all configurations are standard. Now for $1 \leq i \leq n$ consider a model in which all configurations are standard except for configuration i, which is special. Let Q_i and Q_i^t denote the induced distributions over histories and length-t initial histories, respectively.

For each configuration i let p_i denote the probability under distribution Q_0 that the search procedure selects configuration i. Since $p_1 + \cdots + p_n = 1$, if $n > 2$ then there must be at least one configuration i such that $p_i \leq \frac{1}{3}$. Under distribution Q_i, configuration i is the only one that is (ϵ, δ)-optimal, so the search procedure must select it with probability at least $\frac{1}{2}$. Hence the total variation distance between Q_0 and Q_i must be at least $\frac{1}{6}$. By Pinsker's Inequality, their KL-divergence is at least $\frac{1}{18}$. Using the fact that the sequence $D_{KL}(Q_0^t \| Q_i^t)$ converges monotonically to $D_{KL}(Q_0 \| Q_i)$ as $t \to \infty$, we obtain $\sum_{t=1}^{\infty} D_{KL}(Q_0^t \| Q_i^t) - D_{KL}(Q_0^{t-1} \| Q_i^{t-1}) \geq \frac{1}{18}$. By the chain rule for KL-divergence, $D_{KL}(Q_0^t \| Q_i^t) - D_{KL}(Q_0^{t-1} \| Q_i^{t-1})$ is simply equal to the conditional KL-divergence of the t^{th}-step of the history, given the first $t-1$ steps. If step t runs configuration $i' \neq i$, or if it is a safe run of configuration i, then the running time distribution in step t is the same under distributions Q_0^t and Q_i^t. Thus, the conditional KL-divergence equals the conditional probability (under Q_0) of a risky run of configuration i in step t, times the conditional KL-divergence of the running times in case of a risky run, which is $\frac{1}{2}\ln(\frac{1}{1-2\epsilon}) + \frac{1}{2}\ln(\frac{1}{1+2\epsilon}) < 4\epsilon^2$. Summing over $t \geq 1$, we find that the expected number of risky runs of configuration i, under distribution Q_0, must be at least $\frac{1}{72}\epsilon^{-2}$. Summing over configurations, the expected total number of risky runs is at least $\frac{n}{72}\epsilon^{-2}$. Since a run is only risky if the timeout threshold exceeds κ, and then it is only risky with probability 2δ, the expected number of times any configuration is run with timeout threshold greater than κ must be at least $\frac{n}{144\delta\epsilon^2}$, which completes the proof. ∎

7 Extensions for Practical Performance

To this point, we have limited ourselves to (uniform) random sampling of the parameter space. While even incumbent-driven random sampling has shown surprising success for algorithm configuration—and, hence, our approach is likely to work even better—considerable evidence in the literature (e.g., contrasting SMAC and ROAR) indicates that it is a good idea in practice to combine sampling with Bayesian optimization (as in SMAC). There are straightforward ways that our approach could be extended to do this. For example, in the Structured Procrastination method described in Section 5, whenever configurations are added to the pool of samples, one could draw half uniformly and generate the other half using an arbitrary model fit using all previous samples. This does not help in the worst case, of course, but nor does it degrade our worst-case bounds by very much. In effect, we pay no statistical cost for the fact that our model is conditioned on previous observations and obtain a bound as though the random samples were the only configurations considered.

Theorem 7.1 *Suppose that half of the algorithms sampled in Structured Procrastination are generated according to an arbitrary structural model, which in turn depends on previous observations. Then Theorem 5.1 continues to hold with n replaced by $2n$ in the upper bound on T.*

A second practical extension is from sequential to parallel processing. Structured Procrastination is inherently parallelizable: the main loop repeatedly draws and executes a simulation from a queue. Using the queues as a shared data structure, each parallel processor could repeatedly draw an instance from the queue for a configuration with minimal empirical runtime estimate—locking that (configuration, instance) pair to prevent other processors from considering it in the meantime—and then update that estimate once the execution completes. This can result in a suboptimal configuration being queried more than necessary due to stale empirical runtime estimates. However, this can only result in each element of a configuration's queue being executed one more time than necessary, which at most doubles the total time spent processing any given configuration. This intuition can be leveraged to show that using p processors results in a linear speedup, as long as the procedure runs for long enough that there are more input instances enqueued than there are processors.

Theorem 7.2 *Suppose that Structured Procrastination is executed by p processors running in parallel. Then for n sufficiently large and n/n_0 a power of 2, there is a random sample N_0 of n configurations such that if the elapsed wall clock time is at least $T \geq \frac{160n^{1+\omega}}{p\epsilon^2} \ln\left(\frac{3n^{1+\omega}}{\zeta^2\epsilon^2}\right) \textit{OPT}(N_0)$ then it outputs a configuration that is $(\epsilon, n^{-\omega})$-optimal with respect to N_0 with probability at least $1 - \zeta$.*

8 Conclusions

We have introduced Structured Procrastination, a new method for automated algorithm configuration. Crucially, it depends on the idea of *procrastinating* in the face of potentially hard inputs, rather than solving them to completion when first encountered. Our method is guaranteed to find an approximately optimal algorithm configuration in time that worst-case dominates that of any existing algorithm configuration technique.

Our approach in this paper has been focused purely on the worst case. This is justified by the fact that we obtain a positive result. However, the strong performance of heuristic methods suggests that realistic algorithm configuration settings are not as adversarial as our analysis has assumed. A critical direction for future work is empirically evaluating our techniques and comparing them to existing algorithm configuration methods.

References

[Ansótegui *et al.*, 2009] C. Ansótegui, M. Sellmann, and K. Tierney. A gender-based genetic algorithm for automatic configuration of algorithms. In *Principles and Practice of Constraint Programming (CP)*, pages 142–157, 2009.

[Ansótegui *et al.*, 2015] C. Ansótegui, Y. Malitsky, M. Sellmann, and K. Tierney. Model-based genetic algorithms for algorithm configuration. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 733–739, 2015.

[Auer *et al.*, 2002] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235–256, 2002.

[Badanidiyuru *et al.*, 2013] A. Badanidiyuru, R. Kleinberg, and A. Slivkins. Bandits with knapsacks. In *Foundations of Computer Science (FOCS)*, pages 207–216, 2013.

[Bergstra *et al.*, 2011] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. Algorithms for hyper-parameter optimization. In *Advances in Neural Information Processing Systems (NIPS)*, pages 2546–2554, 2011.

[Birattari *et al.*, 2002] M. Birattari, T. Sttzle, L. Paquete, and K. Varrentrapp. A racing algorithm for configuring metaheuristics. In *Genetic and Evolutionary Computation Conference (GECCO)*, pages 11–18, 2002.

[Bubeck *et al.*, 2011] S. Bubeck, R. Munos, G. Stoltz, and C. Szepesvári. X-armed bandits. *Journal of Machine Learning Research*, 12(May):1655–1695, 2011.

[Bubeck *et al.*, 2012] S. Bubeck, N. Cesa-Bianchi, et al. Regret analysis of stochastic and nonstochastic multi-armed bandit problems. *Foundations and Trends in Machine Learning*, 5(1):1–122, 2012.

[Cesa-Bianchi and Lugosi, 2012] N. Cesa-Bianchi and G. Lugosi. Combinatorial bandits. *Journal of Computer and System Sciences*, 78(5):1404–1422, 2012.

[Chaudhuri *et al.*, 2009] K. Chaudhuri, Y. Freund, and D. J. Hsu. A parameter-free hedging algorithm. In *Advances in Neural Information Processing Systems (NIPS)*, pages 297–305, 2009.

[Dean and Voss, 1999] A. Dean and D. Voss. *Design and analysis of experiments*. Springer-Verlag NY, 1999.

[Ganchev *et al.*, 2010] K. Ganchev, Y. Nevmyvaka, M. Kearns, and J. W. Vaughan. Censored exploration and the dark pool problem. *Communications of the ACM*, 53(5):99–107, 2010.

[Guha and Munagala, 2007] S. Guha and K. Munagala. Approximation algorithms for budgeted learning problems. In *ACM Symposium on Theory of Computing (STOC)*, pages 104–113, 2007.

[Hutter *et al.*, 2007] F. Hutter, H. Hoos, and T. Stützle. Automatic algorithm configuration based on local search. In *AAAI Conference on Artificial Intelligence*, pages 1152–1157, 2007.

[Hutter *et al.*, 2009] F. Hutter, H. Hoos, K. Leyton-Brown, and T. Stützle. ParamILS: An automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36:267–306, 2009.

[Hutter *et al.*, 2011a] F. Hutter, H. Hoos, and K. Leyton-Brown. Bayesian optimization with censored response data. In *NIPS workshop on Bayesian Optimization, Sequential Experimental Design, and Bandits (BayesOpt'11)*, 2011.

[Hutter *et al.*, 2011b] F. Hutter, H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Conference on Learning and Intelligent Optimization (LION)*, pages 507–523, 2011.

[Improbable Research, 2011] Improbable Research. The 2011 Ig Nobel prize winners. `http://www.improbable.com/ig/winners/#ig2011`, 2011.

[Kandasamy *et al.*, 2016] K. Kandasamy, G. Dasarathy, B. Poczos, and J. Schneider. The multi-fidelity multi-armed bandit. In *Advances in Neural Information Processing Systems (NIPS)*, pages 1777–1785, 2016.

[Kleinberg *et al.*, 2008] R. Kleinberg, A. Slivkins, and E. Upfal. Multi-armed bandits in metric spaces. In *ACM Symposium on Theory of Computing*, pages 681–690, 2008.

[Kleinberg, 2006] R. Kleinberg. Anytime algorithms for multi-armed bandit problems. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 928–936, 2006.

[Li *et al.*, 2016] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *arXiv preprint arXiv:1603.06560*, 2016.

[López-Ibáñez *et al.*, 2011] M. López-Ibáñez, J. Dubois-Lacoste, T. Stützle, and M. Birattari. The irace package, iterated race for automatic algorithm configuration. Technical report, IRIDIA, Université Libre de Bruxelles, 2011.

[Munos, 2014] R. Munos. From bandits to Monte-Carlo tree search: The optimistic principle applied to optimization and planning. *Foundations and Trends in Machine Learning*, 7(1):1–129, 2014.

[Perry, 1996] J. Perry. How to procrastinate and still get things done. `http://www.chronicle.com/article/How-to-ProcrastinateStill/93959`, 1996.

[Shahriari *et al.*, 2016] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. de Freitas. Taking the human out of the loop: A review of Bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2016.

[Srinivas *et al.*, 2012] N. Srinivas, A. Krause, S. M. Kakade, and M. W. Seeger. Information-theoretic regret bounds for Gaussian process optimization in the bandit setting. *IEEE Transactions on Information Theory*, 58(5):3250–3265, 2012.

[Thornton *et al.*, 2013] C. Thornton, F. Hutter, H. Hoos, and K. Leyton-Brown. Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms. In *Conference on Knowledge Discovery and Data mining (KDD)*, pages 847–855, 2013.

[Tran-Thanh *et al.*, 2012] L. Tran-Thanh, A. Chapman, A. Rogers, and N. R. Jennings. Knapsack based optimal policies for budget–limited multi–armed bandits. In *AAAI Conference on Artificial Intelligence*, 2012.