Golden Parameter Search

Exploiting Structure to Quickly Configure Parameters in Parallel

Yasha Pushak
Department of Computer Science
The University of British Columbia
Vancouver, Canada
ypushak@cs.ubc.ca

Holger H. Hoos
LIACS, Universiteit Leiden
Leiden, The Netherlands
hh@liacs.nl

ABSTRACT

Automated algorithm configuration procedures such as SMAC, GGA++ and `irace` can often find parameter configurations that substantially improve the performance of state-of-the-art algorithms for difficult problems – *e.g.*, a three-fold speedup in the running time required by EAX, a genetic algorithm, to find optimal solutions to a set of widely studied TSP instances. However, it is usually recommended to provide these methods with running time budgets of one or two days of wall-clock time as well as dozens of CPU cores. Most general-purpose algorithm configuration methods are based on powerful meta-heuristics that are designed for challenging and complex search landscapes; however, recent work has shown that many algorithms appear to have parameter configuration landscapes with a relatively simple structure. We introduce the golden parameter search (GPS) algorithm, an automatic configuration procedure designed to exploit this structure while optimizing each parameter semi-independently in parallel. We compare GPS to several state-of-the-art algorithm configurators and show that it often finds similar or better parameter configurations using a fraction of the computing time budget across a broad range of scenarios spanning TSP, SAT and MIP.

CCS CONCEPTS

• **Computing methodologies → Artificial intelligence**; **Search methodologies**; *Machine learning*; Parallel computing methodologies;

KEYWORDS

Automated Algorithm Configuration, Parallel Algorithm Configuration, Parameter Turning, Hyper-Parameter Optimization

ACM Reference Format:
Yasha Pushak and Holger H. Hoos. 2020. Golden Parameter Search: Exploiting Structure to Quickly Configure Parameters in Parallel. In *Genetic and Evolutionary Computation Conference (GECCO '20), July 8–12, 2020, Cancún, Mexico.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3377930.3390211

1 INTRODUCTION

Automated algorithm configuration procedures seek to find parameter settings that optimize the performance of an algorithm on a particular distribution of problem instances. Several existing state-of-the-art methods have been shown to obtain substantial speedups over default parameter settings (see, *e.g.*, Ansótegui et al. [1, 2], Balaprakash et al. [3], Cáceres et al. [8], Hutter et al. [13, 16, 17], López-Ibáñez et al. [21]). However, it is typically recommended to provide algorithm configuration procedures with time budgets of one or two wall-clock days [18] and dozens of CPU cores. Many prominent algorithm configuration procedures are based on powerful meta-heuristics, *e.g.*, ParamILS [16], an iterated local search procedure; GGA[2], a gender-based genetic algorithm; or SMAC [13], a Bayesian optimization procedure. However, we recently showed that many prominent algorithms appear to have much simpler configuration landscapes than suggested by the use of these sophisticated methods [24]. In particular, all but one of the parameters we studied appeared to have uni-modal responses when varied individually. We introduce the golden parameter search algorithm (GPS), which is designed to exploit this recent discovery. GPS combines (and in some cases improves upon) several of the essential ingredients of existing algorithm configurators, such as intensification mechanisms [15], adaptive capping [8, 16] and racing procedures [6, 26], with a novel, parallel version of golden section search [19], a simple bracketing procedure with optimal performance for one-dimensional uni-modal functions [23].

We demonstrate that GPS is often able to find better configurations when provided with smaller running time budgets than several other state-of-the-art algorithm configuration procedures (see Section 5), even though it is based on two strong assumptions: First, that numerical parameters are uni-modal, and second, that most parameters do not strongly interact and can therefore be optimized semi-independently in parallel. After formally defining the algorithm configuration problem and introducing some notation (Section 2), we introduce GPS (Section 3). We describe how we evaluated GPS (Section 4) and report the results (Section 5). Finally, after discussing related work (Section 6), we provide some concluding remarks and briefly outline possible future improvements to GPS (Section 7).

2 PRELIMINARIES

Formally, let P be the set of configurable parameters for target algorithm A and let m_I be the performance metric for which we wish to optimize A on a set I of instances; we will write m rather then m_I in some cases where I is clear in a given context. Let $p \in P$ be a parameter with values $v \in V_p$. We denote by $c = (v_1, ..., v_n) \in$

$C \subseteq V_{p_1} \times \ldots \times V_{p_n}$ a configuration for algorithm A. The goal is to find $c^* \in C$ such that $m_I(c^*) \leq m_I(c)$ for all $c \in C$. Furthermore, we will use the following notation related to configurations and parameter settings: $c[p]$ denotes the value of parameter p in configuration c; $c[p] := v$ modifies c by setting parameter p to value v; and $c|_{p=v}$ denotes configuration c with parameter p (temporarily) set to v.

3 GOLDEN PARAMETER SEARCH

GPS conducts an efficient search process on each parameter semi-independently. It begins with a bracket B_p for each parameter $p \in P$, where a bracket corresponds to a set of parameter values believed to contain the optimum parameter value. Each bracket is evaluated in parallel and shrunk around the optimum value. A bracket may be expanded if there is evidence that it no longer contains the optimum (due to, *e.g.*, parameter interactions). For numerical parameters, this search procedure is based on the *golden section search algorithm* [19], which has the optimal worst-case bound for one dimensional, uni-modal functions [23]. To save on computational resources, GPS uses a racing procedure based on a permutation test. Once a better value for a parameter is found, the search procedure for each of the other parameters is updated to use this value. Since parameter interactions may cause old target algorithm runs to become stale, these old runs are slowly forgotten (down-weighted), and then eventually re-run. When GPS terminates, the incumbents for each parameter value are returned as the final incumbent configuration.

Many algorithms contain conditional parameters, whose values are only used when their "parent" parameters are set to certain values. GPS optimizes the values of all such child parameters in parallel by appropriately modifying their ancestors' parameter values to enable the child parameters when they are being evaluated. This avoids incorrectly discarding parent parameter values because their children were sub-optimally configured.

To avoid wasting time evaluating poor configurations on a large number of instances, GPS uses an intensification mechanism that determines how many runs are performed for each value, and slowly introduces more instances into the set $I' \subseteq I$ used to evaluate parameter values. A crucial component of many algorithm configuration procedures are capping mechanisms [8, 16], which limit the running time cutoffs for each target algorithm run. GPS employs a novel capping mechanism that avoids being overly aggressive.

The primary bottleneck in algorithm configuration is running many configurations on sets of problem instances; therefore, GPS uses a master-worker process design, whereby the master process loops through each parameter $p \in P$ to check for newly completed target algorithm runs, updates the bracket B_p and incumbent value $c^*[p]$, and then queues new runs. We show high-level pseudo-code for the master process in Algorithm 1. The worker processes repeatedly query a queue in a database to obtain new runs to perform. GPS queues an exponentially increasing number of instances on which each parameter will be evaluated each time new instances are queued for a given parameter. For some highly-parameterized algorithms, only a small fraction of parameters affect the performance of the algorithm [10, 24]. GPS therefore uses a multi-armed bandit procedure to prioritize parameters believed to be important.

For some scenarios, GPS may be unable to keep the target algorithm run queue sufficiently populated. Hence, GPS dynamically adjusts an instance increment parameter, *instIncr*, such that the intensification and queuing mechanisms operate on batches of *instIncr* instances (target algorithm runs).

Algorithm 1 The main algorithm for Golden Parameter Search.

```
input
    $A$, the algorithm to be optimized
    $I$, the training instance set
    $m$, the metric with respect to which $A$ is optimized
    $P$, the set of configurable parameters of $A$
    $C$, the configuration space (ranges and constraints)
    $c_0$, the default configuration
    numInitInst, an integer in $[1, \infty)$ (default: 1)
    decayRate, a real number in [0,1] (default: 0.2)
    $\alpha$, a real number in $(0, 1)$ (default: 0.05)
    instIncr, a positive Fibonacci number (default: 1).
output
    $c^*$, the best configuration found so far
procedure GPS
    # Initialization
    Initialize the incumbent $c^* := c_0$
    for each parameter $p \in P$, do
        Initialize a bracket $B_p$
        Initialize $I_p$ with numInitInst random instances
        Queue a run for the default value $c_0[p]$
    Initialize empty arrays $R$ and $Q$
    # Main Procedure
    while budget not exhausted, do
        # See Section 3.9 for the bandit queue
        Sample a parameter $p \in P$ using the bandit queue
        # $R$ and $Q$ are used to update the instIncr to help
        # make use of all workers (See Section 3.10)
        Append values to $R$ and $Q$
        if sufficient time has elapsed, then
            Update instIncr, if needed
            Re-initialize $R$ and $Q$ as empty arrays
        # Compare performances (see Section 3.1)
        for each pair $v_1, v_2 \in B_p$, do
            Perform permutation test with significance level $\alpha$
        # Incumbent update (see Section 3.6)
        if there exists $v$ such that $m(c^*|_{p=v}) \prec_\alpha m(c^*)$, then
            Update the incumbent $c^*[p] := v$
        # Bracket Update (see Section 3.2)
        if there is sufficient evidence for improvement, then
            Expand/shrink the bracket, $B_p$
            # Intensification (see Section 3.8)
            Add instIncr random instances to $I_p$
        else if each $v \in B_p$ has been run on each $i \in I_p$, then
            Add instIncr random instances to $I_p$
        Append values to $R$ and $Q$
        # Offload tasks to the workers (see Section 3.8)
        Queue new target algorithm runs as needed
    return $c^*$
```

3.1 Test for Significance (Permutation Test)

Algorithm performance measures are typically noisy, randomized objective functions [15, 24]. Therefore, a core component of GPS is a permutation test, which is used to determine, in a distribution-free way, whether or not there is sufficient evidence at a given significance level α to conclude that one parameter value is worse than another. Let $m(c^*|_{p=v_1}) \prec_\alpha m(c^*|_{p=v_2})$ denote a significant difference, let $m(c^*|_{p=v_1}) \approx_\alpha m(c^*|_{p=v_2})$ denote a statistical tie, and let $m(c^*|_{p=v_1}) \preceq_\alpha m(c^*|_{p=v_2})$ denote that $m(c^*|_{p=v_1}) \prec_\alpha m(c^*|_{p=v_2})$ or $m(c^*|_{p=v_1}) \approx_\alpha m(c^*|_{p=v_2})$. Note that $m(c^*|_{p=v})$ is an estimate of the performance on a subset of I. If the intersection of completed runs (technically: run equivalents – see Section 3.4) for $m(c^*|_{p=v_1})$ and $m(c^*|_{p=v_2})$ is less than $numInitInst$, we skip the test and record $m(c^*|_{p=v_1}) \approx_\alpha m(c^*|_{p=v_2})$.

3.2 Expanding/Shrinking the Bracket

For a parameter $p \in P$, the bracket B_p is simply a set of values that are believed to contain or bound the optimal value for p.

Real-Valued Parameters. For real-valued parameters, the bracket updating procedure is inspired by the *golden section search algorithm* [19], which provides optimal worst-case performance for one-dimensional, uni-modal functions (an assumption which we assume is true for most algorithms' parameters [24]). The bracket contains two end points v_a and v_b (believed to bound the optimal value) and two interior points v_c and v_d, such that $v_a < v_c < v_d < v_b$ and $\frac{v_b-v_a}{v_d-v_a} = \frac{v_b-v_a}{v_b-v_c} = \frac{v_d-v_a}{v_c-v_a} = \frac{v_b-v_c}{v_b-v_d} = \frac{\sqrt{5}+1}{2} \equiv \phi$, the golden ratio. Both an expand and shrink operation corresponds to replacing one parameter value in the set $\{v_a, v_b, v_c, v_d\}$ with a new value. For example, we expand the bracket in the direction of v_a if we observe $m(c^*|_{p=v_a}) \prec_\alpha m(c^*|_{p=v_c}) \preceq_\alpha m(c^*|_{p=v_d}) \preceq_\alpha m(c^*|_{p=v_b})$, where an expand operation corresponds to updating $v_d := v_c$ and $v_c := v_a$ followed by $v_a := v_c \cdot (\phi + 1) - v_d \cdot \phi$. The value v_b remains unchanged. Similarly, we shrink the bracket around v_c if we observe $m(c^*|_{p=v_a}) \succeq_\alpha m(c^*|_{p=v_c}) \prec_\alpha m(c^*|_{p=v_d}) \preceq_\alpha m(c^*|_{p=v_b})$, where a shrink operation corresponds to updating $v_b := v_d$ and $v_d := v_c$ followed by $v_c := v_b - \frac{v_b-v_a}{\phi}$. Note that GPS does not require that parameter values stay within the maximum and minimum values specified in the parameter configuration space file. While this may lead GPS to evaluate configurations that cause the algorithm to crash, it also allows GPS to recover from situations where the user chose a poor maximum or minimum value.

Integer-Valued Parameters. We use the same procedure as for real-valued parameters; however, parameter values are rounded to the nearest integer value not already contained in the bracket. Technically, this is similar to the *Fibonacci search algorithm* [19] with a custom bracket initialization strategy (see Section 3.3).

Categorical Parameters. The bracket B_p may contain any subset of the parameter values available for p. When a bracket is shrunk, a value is simply removed from the set. We perform this update as soon as we have seen sufficient evidence that one value $v_{bad} \in B_p$ performs worse than the current incumbent (*i.e.*, $m(c^*) \prec_\alpha m(c^*|_{p=v_{bad}})$). Parameter values may be re-added to the bracket if we lose confidence in the target algorithm runs upon which we based this decision due to parameter interactions (see Section 3.4).

3.3 Bracket Initialization

Real-Valued Parameters. Existing algorithm configurators [1, 8, 13, 16] require that a default, maximum and minimum value are provided. Our bracket initialization procedure guarantees that at least one of v_a, v_b, v_c and v_d are the default value, while making the bracket as large as possible, without exceeding the maximum and minimum range. We further require that the golden ratio properties between v_a, v_b, v_c and v_d are satisfied (see Section 3.2).

Integer-Valued Parameters. These brackets are initialized using the same procedure as for real-valued parameters, with rounding.

Categorical Parameters. All parameter values are initially added to the bracket, *i.e.* $B_p = V_p$.

3.4 Parameter Interactions (Decaying Memory)

Each time one of the concurrent search processes updates its incumbent parameter value, the algorithm now contains stale performance measurements for the other parameters. Instead of completely trusting or discarding this information, GPS uses a heuristic to slowly forget stale information. For a target algorithm run performed on instance i with configuration c_i, we compute a weight, $w_i \in [0, 1]$, which encodes the amount of trust we have in the corresponding performance measurement. Let c^* be the current incumbent configuration and let c_i be the configuration that was the incumbent when the target algorithm run on i was performed. Then,

$$w_i = decayRate^{\Delta(c_i, c^*)}, \tag{1}$$

where $\Delta(c_i, c^*)$ denotes the difference between the previous incumbent c_i and the current incumbent c^*, and $decayRate \in [0, 1]$ is a parameter that controls how quickly information is forgotten. Specifically,

$$\Delta(c_i, c^*) = \sqrt{\sum_{p \in P_i \cap P^*} \delta(c_i[p], c^*[p])^2}, \tag{2}$$

where P_i and P^* denote the set of active parameters (see Section 3.5) in c_i and c^*, respectively and where $\delta(c_i[p], c^*[p])$ measures the difference in parameter values $c_i[p]$ and $c^*[p]$. For numerical parameters,

$$\delta(c_i[p], c^*[p]) = \frac{|c_i[p] - c^*[p]|}{|v_{\max} - v_{\min}|}, \tag{3}$$

where $v_{\max}$ and $v_{\min}$ correspond to the maximum and minimum values for parameter p, respectively. For categorical parameters,

$$\delta(c_i[p], c^*[p]) = \begin{cases} 0 & \text{if } c_i[p] = c^*[p], \\ 1 & \text{otherwise.} \end{cases} \tag{4}$$

When performing permutation tests, we use multiplication to combine the weights w_i and w'_i for the target algorithm runs that approximately measure $m_{\{i\}}(c^*|_{p=v})$ and $m_{\{i\}}(c^*|_{p=v'})$; this intuitively corresponds to adding the uncertainties associated with the respective performance estimates.

Throughout the following, we refer to the number of *run equivalents* performed for a particular parameter value, which is simply the sum of the weights for the target algorithm runs performed for that parameter value. Intuitively, this approximates the total number of reliable target algorithm runs for that parameter value.

3.5 Evaluating Conditional Parameters

Some parameters, known as conditional (or child) parameters, are only active if their parent parameter is set to a certain value or set of values. With GPS, we only allow parameters that have a single parent value that activates them (this only affected one of the algorithms we studied, see Section 4). To optimize a conditional parameter p_c, with parent (or ancestor) p_a, the value of p_a must be set to a specific value v_{on}. If $c^*[p_a] \neq v_{\text{on}}$, then GPS will temporarily set $c^*[p_a]$ to v_{on} when evaluating values of p_c. An improvement to $c^*[p_c]$ will not immediately improve $m(c^*)$ when $c^*[p_a] \neq v_{\text{on}}$. However, it will decrease the confidence in the old target algorithm runs for $m(c^*|_{p_a=v_{\text{on}}})$ (see Section 3.4), which may eventually lead to setting $c^*[p_a] := v_{\text{on}}$.

3.6 Updating the Incumbent

GPS uses a conservative incumbent updating procedure, whereby an incumbent parameter value is only updated if it is believed it *improves* the performance of the incumbent configuration. This helps avoid over-fitting, losing confidence in old target algorithm runs (see Section 3.4) and breaking things (on unseen instances) that do not need fixing (on seen instances). GPS picks the incumbent for each parameter using a 7-step process. Steps 1–2 make sure that each candidate has been run on a sufficient instance set and 3 checks to see if any challengers are better than the incumbent. Steps 4–7 are a series of tie-breakers. The process begins with all of the values in the bracket as candidates, and in each, a filter is used to reject some of the candidates. Unless otherwise specified, if no candidate remains at the end of a round, the previous incumbent is returned, and if only one candidate remains, it becomes the new incumbent. Specifically, the steps are as follows:

(1) Admit candidates v with $\geq$ *numInitInst* run equivalents.
(2) Admit candidates v that have been run on a (non-strict) super-set of the instances upon which the last incumbent was run when it was chosen as the incumbent.
(3) Admit candidates v with statistically sufficient evidence of improved performance compared to the previous incumbent, according to a permutation test, *i.e.*, $m(c^*|_{p=v}) \prec_\alpha m(c^*)$.
(4) Admit candidates v that are not worse than any of the other candidates according to the permutation test. If every candidate is eliminated (*e.g.*, a triangle where $m(c^*|_{p=v_a}) \prec_\alpha m(c^*|_{p=v_b}) \prec_\alpha m(c^*|_{p=v_c}) \prec_\alpha m(c^*|_{p=v_a})$), then skip this filter.
(5) Admit candidates v with the best performance $m(c^*|_{p=v})$.
(6) Admit candidates v with the most run equivalents.
(7) Select one of the remaining candidates uniformly at random.

3.7 *"Adaptive"* Adaptive Capping

An important component of many algorithm configurators, such as SMAC [13], ParamILS [16] and `irace` [8], is a so-called "adaptive capping mechanism" that selects running time cutoffs for individual target algorithm runs. This mechanism is designed to avoid running a very poor configuration for a very long time on a single instance. All three previously mentioned configurators use a simple heuristic for their adaptive capping mechanism, which is based on the performance of (one of) the best-known configuration(s) found so far. This mechanism can be easily modified for use in GPS; *e.g.*, if $m(\cdot) = \text{PAR10}(\cdot)$, then the adaptive cap is calculated as

$$\begin{aligned} AC = {} & \text{PAR10}_{(I^* \cap I_{\text{new}}) \cup \{i'\}}(c^*) \cdot (|I^* \cap I_{\text{new}}| + 1) \cdot BM \\ & - \text{PAR10}_{I^* \cap I_{\text{new}}}(c^*|_{p=v_{\text{new}}}) \cdot |I^* \cap I_{\text{new}}|, \end{aligned} \tag{5}$$

where $|I^*|$ and $|I_{\text{new}}|$ denote the number of instances upon which c^* and $c^*|_{p=v_{\text{new}}}$ have each been evaluated so far, respectively; BM (the bound multiplier) is a constant (typically set to 2); and where $\text{PAR10}(\cdot)$ denotes mean running time with censored runs replaced by 10 times their running time cutoff.

Unfortunately, this is overly aggressive, which causes high-quality challengers to be rejected before the permutation test used by GPS (see Section 3.6). In fact, it often takes only a single unlucky target algorithm run for a challenger to be prematurely rejected with this cap. Since GPS eliminates entire regions of the configuration space after observing a poorly performing parameter value, such a mistake can lead to a very large performance penalty for GPS's final incumbent (preliminary experiments indicated that slow-down factors of up to 10 were not uncommon). We therefore chose to modify the capping mechanism to make it even more adaptive, by using a bound multiplier that depends on $|I^* \cap I_{\text{new}}|$. Specifically,

$$BM(x) = \max(\exp(7.21 \cdot x^{-0.63}), 2). \tag{6}$$

We chose this function for $BM(x)$ by simulating the effects of varying x when comparing two hypothetical algorithm configurations simulated using identical exponential distributions.

To incorporate the effects of the decaying memory (see Section 3.4), everywhere in Equation 5 we use the number of run equivalents instead of the size of the instance sets (where we multiply the weights from $m(c^*)$ and $m(c^*|_{p=v_{\text{new}}})$ to obtain a combined level of trust, as for permutation tests).

Finally, as soon as a run with a parameter value has exhausted an adaptive cap for a single instance, we stop evaluating that value on new instances. Furthermore, for the purposes of the permutation test (see Section 3.1), it is considered worse than all other parameter values in its bracket which have not been capped.

3.8 Intensification & Queuing Runs

One of the most important components of algorithm configuration procedures are intensification mechanisms [13, 15], which control the number of target algorithm runs performed to evaluate each candidate configuration. Poorly performing configurations can typically be eliminated using only a small number of runs, whereas high-quality configurations require many more. It is also advantageous to use only a small number of benchmark instances when comparing configurations at early stages of the process (when the goal is to quickly distinguish between good and very bad regions of the configuration space), and to increase this number slowly as the search progresses. This is especially important in a parallel setting, where a large number of target algorithm runs could otherwise be queued all at once for an extremely bad configuration.

GPS addresses this with a two-part intensification mechanism: First, for each parameter p, GPS starts with a very small set of *numInitInst* random instances, I_p. Each time B_p is updated, or when each $v \in B_p$ has been evaluated on each $i \in I_p$, the mechanism adds one randomly chosen instance (without replacement) to I_p. If there are no more new instances to add to I_p, then GPS starts adding instances it has already used with new random seeds.

The second component of the intensification mechanism is captured by how many target algorithm runs GPS queues to be performed for each candidate. GPS only queues target algorithm runs in powers of 2; that is, for parameter value $v \in B_p$, we find the largest power q for which $c^*|_{p=v}$ has been run on at least 2^q instances, and then we queue all of the first 2^{q+1} instances in I_p that have not yet been queued. If an old target algorithm run has become stale ($w \leq 0.05$), then we also re-queue that run. GPS does not queue any target algorithm runs for parameter values for which there is sufficient information to reject those values as worse than the current incumbent (unless it detects that the response for a given parameter is not uni-modal, in which case it assumes that it does not yet have enough data to correctly determine the ordering of the values and continues to queue runs for all of them). Sets of target algorithm runs are pushed into the queue in a random order.

3.9 Bandit Queue

Some evidence suggests that only a small fraction of parameters for highly-parameterized algorithms affect the algorithm's performance [10, 24]. To avoid spending a large amount of time evaluating parameters that are unimportant, GPS uses a multi-armed-bandit-style mechanism to determine the order in which parameters are considered for queuing runs. GPS approximates the relative importance of each parameter p by counting the number of times k_p that p has been updated in c^*. Since most parameters are unlikely to be updated more than a small handful of times, we need a mechanism that increases the probability of choosing a parameter very quickly (*e.g.*, exponentially). We therefore start with a set $P' = P$, and when we are picking a new parameter to evaluate, we sample parameter p_{next} with probability

$$\frac{\text{Fib}(k_{p_{\text{next}}})}{\sum_{p \in P'} \text{Fib}(k_{p_{\text{next}}})}, \tag{7}$$

where Fib(k) is the k^{th} Fibonacci number. If we sample a parameter that has not been updated, then we remove it from P' and pick a new sample. Once a parameter has been picked that has been updated, or once P' is empty, we reinitialize $P' = P$.

3.10 Instance Increment

When algorithm runs can be performed very quickly (*e.g.* $\leq$ 10 CPU seconds per run) or when there are lots of parallel resources available, then GPS may be unable to keep the queue sufficiently populated to keep all of the workers busy. To compensate, GPS dynamically updates the value for an instance increment, *instIncr*. This is a multiplier used to make the operations in the intensification mechanism (see Section 3.8) operate on batches of I instances.

To dynamically update *instIncr*, GPS periodically records the number of worker processes, r, that are currently performing target algorithm runs and the number of tasks, q, that are waiting in the queue. Let R and Q denote the sequences of these recently recorded values. If $\text{median}(Q) < \frac{\max(R)}{2}$ or $\text{median}(Q) \leq 4$, then *instIncr* is increased. If $\text{median}(Q) \geq 2 \cdot \max(R)$, then *instIncr* is decreased. Otherwise, *instIncr* is left unchanged. To allow *instIncr* to quickly respond to changes, the value is always set to a positive Fibonacci number. We use $\max(R)$ to approximate the number of available workers, since GPS can operate with a dynamically changing amount of parallel resources.

3.11 The Worker Process

The worker processes query a database for new target algorithm runs (*i.e.*, a parameter p, a value v and an instance i). For each run, the worker calculates an adaptive cap (see Section 3.7), creates a temporary entry in the database that is set to expire in twice that run's cutoff time, and then begins the target algorithm run to obtain $m_{\{i\}}(c^*|_{p=v})$. Once the run is complete, the worker pushes the result to the database and removes the temporary entry. The temporary entry is a back-up designed to prevent GPS from stalling in the event of an unexpected crash occurring for a worker process. Once the temporary entry expires, if the result has not been pushed to the database, the master process will re-queue the run.

4 EXPERIMENTAL SETUP

We compared GPS to the latest versions of three other state-of-the-art general-purpose algorithm configurators: SMAC 3.0[1] [13], GGA++ [1] and `irace` 3.3 [8]. Due to time constraints, we did not compare GPS to earlier versions of these configurators, nor to ParamILS, since the authors of ParamILS and SMAC recommend to use SMAC, unless there is sufficient budget available to use both.[2] GPS, `irace` and GGA++ all support parallel execution; for each run of these configurators we used 8 CPU cores. Unfortunately, to the best of our knowledge, the parallel version of SMAC [14] was never made publicly available; however, the quality of the configurations found between independent runs of SMAC can vary substantially, and it is typically recommended to exploit this via multiple independent parallel runs of SMAC [14, 26]. The so-called *standard protocol* is to evaluate all of the final incumbents on the entire training set and return the one with the best performance [26]. We applied SMAC in precisely this way by performing 8 independent runs of SMAC per scenario.

However, one may wonder whether the quality of the configurations obtained by GPS also varies substantially, and if the standard protocol might also benefit GPS. To answer this question, we performed 5 independent runs of each parallel algorithm configurator (and 40 independent runs of SMAC) for each scenario. From this single set of results, we then performed an analysis that simulates two experiments: One *small parallel budget* experiment, where each parallel configurator was run only once and SMAC was run 8 times in parallel; and a second *large parallel budget* experiment, where each parallel configurator was run 3 times independently in parallel (using a total of 24 cores each) and SMAC was run 24 times in parallel. We then used bootstrap sub-sampling to simulate 1 001 independent trials of each type of experiment to obtain median speedups and 95% bootstrap percentile confidence intervals.

We evaluated the performance of the configuration procedures on six ACLib [18] scenarios, which we summarize in Table 1. Four of these scenarios (LKH [11] and EAX [22] on TSP RUE 1000-3000 instances, CaDiCaL [5] on circuit fuzz SAT instances and CPLEX [9] on Regions200 MIP instances) were chosen, because they were

[1]We chose to use SMAC 3 after checking with the authors and in light of the fact that all recent publications regarding SMAC also use SMAC 3.
[2]Personal communication with the authors.

Table 1: The ACLib benchmark scenarios studied.

Problem	Algorithm	Instance Set	# of Params.	
			Num.	Cat.
TSP	LKH	TSP RUE 1000-3000	12	11
	EAX		2	0
SAT	CaDiCaL	Circuit Fuzz	40	22
	probSAT	7SAT90	4	5
MIP	CPLEX	Regions200 RCW2	22	63

among the scenarios that we identified as having "interesting" parameter responses in the landscape analysis [24] that inspired GPS. However, since we found that the EAX scenario had only a very minor possible speedup available over the default configuration, we modified the default parameter values to make this scenario more interesting. In particular, we changed the default for the population size from 100 to 418 and the number of children from 30 to 36. In this landscape work we also found that one of the parameters for LKH, the number of backbone trials, had a non-unimodal response; we hypothesize that this was because the value 0 had a special meaning (that a particular heuristic should be disabled). Between the scenarios, there were 10 other parameters (1 other for LKH, 9 for CPLEX) where the algorithm's documentation indicated that a similar special meaning was encoded in a particular value of a numerical parameter. GPS makes strong assumptions about the structure of algorithm configuration landscapes, therefore, when using GPS, parameters should not be encoded in this way. We therefore modified these two scenarios to introduce additional categorical parent parameters that control whether or not the special value for the numerical child parameter is used, or if the child parameter is otherwise used as normal. Finally, since GPS is also only able to handle conditional parameters that are active when their parents take on a single value (rather than a set or range of values), we further modified two of the parameters of CPLEX by creating multiple copies of the child parameter (this causes the number of parameters for LKH and CPLEX in Table 1 to differ from the respective AClib scenarios [18]).

To compare the performance of each configurator for different overall configuration budgets, we performed the previously described analysis for the anytime incumbent configurations after 30 minutes, 1 hour, 3 hours, 6 hours, 12 hours and 24 hours. We note that for most of these scenarios, ACLib recommends to use 48 hours for each configurator run; however, we shortened this to 24 hours to keep the total CPU time required for our experiments within our available budget. Even with this reduced budget, for all but the final scenario (CPLEX on RCW2), at least one of the configurators was still able to find a configuration with a significant speedup over the default with high probability. However, for CPLEX on RCW2, only SMAC and GPS returned configurations better than the default, and most of these were found within ca. 3 minutes, *i.e.*, when the configurators had not yet evaluated the configurations on enough instances to have any kind of statistical confidence of an improvement. Since this scenario was more challenging than the others, we turned it into a stretch test for GPS by increasing the total degree of parallelism from 8 cores per run to 32 cores per run for the parallel configurators (and increased the total SMAC runs from 40 to 160). In the following, we report the results for this more challenging setting.

All experiments were run on a cluster of 20 nodes, each equipped with 32 2.10 GHz Intel Xeon E5-2683 v4 CPUs with 40 960 KB cache and 96 GB RAM each, running openSUSE Leap 42.1 (x86_64). We allotted 3 GB of RAM to each core used by an algorithm configurator, and further restricted each target algorithm run to a maximum of 3 GB of RAM. We measured the performance of each target algorithm using penalized average running time 10 (PAR10), where censored runs (runs unable to complete within their running time cutoff or within the 3 GB RAM limit) are replaced with 10 times their running time cutoff. When evaluating the configurations on the test set, we ran each configuration on each instance in random order, to ensure that background environmental noise effects (*e.g.*, cache effects or dynamic CPU clock speed changes, which are designed to avoid over-heating) were independently and identically distributed.

5 RESULTS

We encountered varying degrees of difficulty with each of the configurators we used as baselines. For SMAC, 2.5% of the runs we performed terminated early after raising an uncaught exception. We treated these runs as if the final incumbent found remained unchanged for all time budgets after the runs crashed. For all of GGA++'s runs on the CPLEX scenarios, we provided 52 hour job allocations on our cluster (which should provide ample slack for the 24-hour configuration budget). However, GGA++ was unable to complete any of these runs within the 52 hours. Since GGA++ does not output any information about its anytime configurations until the very end of its runs, we were unable evaluate its performance on these scenarios. We suspect GGA++ did not respect its configuration budgets on these scenarios because the running time cutoffs for CPLEX are 10 000 seconds, which is much larger than all the other scenarios and it does not use adaptive capping or intensification mechanisms. When we gave `irace` a wall-clock budget of 24 hours we quickly observed that it used only a small fraction of the available budget before terminating. Initially, `irace` estimates the performance of the default configuration and uses this to convert the remaining time budget into a target algorithm run budget, a number which it consistently under-estimated. Since we still had most of `irace`'s computational budget remaining, we repeated all of these runs with a configuration budget of 48 hours of wall-clock time and report the anytime results for these instead. Nevertheless, 80% of `irace`'s runs terminated after using between 25-75% of the available budget (and the rest used less – as little as 3%); however, we had exhausted the computational resources we had available to evaluate `irace`, therefore, we treat these runs similarly to SMAC's crashed runs.

In Table 2 we show selected results using the large parallel budget analysis method as described in Section 4. In particular, we compare the 24 hour configuration budget results (which is on the small end of a typical algorithm configuration budget) with the 6 hour budget results (a time for which the best configurator for each scenario has typically found a high-quality solution). While

Table 2: Large parallel budget analysis speedups (medians and 95% confidence intervals). Median speedups not worse than the best speedup for each time budget according to a permutation test with a 5% signifance level are shown in boldface.

	TSP		**SAT**		**MIP**	
	LKH	**EAX**	**CaDiCaL**	**probSAT**	**CPLEX**	
	TSP RUE 1000-3000		**Circuit Fuzz**	**7SAT90**	**Regions200**	**RCW2**
Configuration budget (excluding validation) = 6.0 wall-clock hours						
GPS	**1.20** [0.95, 1.20]	**3.05** [2.82, 3.21]	**1.41** [1.12, 1.41]	**3.76** [1.93, 4.73]	**1.16** [1.00, 1.31]	1.01 [1.00, 1.05]
SMAC3.0	1.00 [1.00, 1.00]	**2.62** [2.53, 3.55]	1.25 [1.09, 1.36]	**4.90** [4.28, 13.41]	0.96 [0.77, 1.18]	**1.26** [1.22, 1.27]
irace3.3	0.94 [0.92, 1.11]	2.48 [1.85, 3.38]	0.87 [0.83, 0.95]	**4.90** [4.90, 13.06]	0.01 [0.01, 1.00]	1.00 [1.00, 1.00]
GGA++	0.66 [0.58, 1.05]	0.61 [0.61, 0.61]	0.92 [0.89, 1.02]	**7.14** [5.38, 8.58]	–	–
Configuration budget (excluding validation) = 24.0 wall-clock hours						
GPS	**1.21** [1.18, 1.28]	**3.22** [2.36, 3.46]	**1.44** [1.16, 1.55]	3.03 [1.93, 5.52]	0.68 [0.01, 1.12]	**1.41** [1.09, 1.41]
SMAC3.0	1.00 [1.00, 1.00]	2.73 [2.62, 3.08]	**1.36** [1.16, 1.60]	**5.76** [4.28, 13.41]	**1.18** [0.77, 1.23]	1.26 [1.22, 1.27]
irace3.3	1.03 [0.79, 1.14]	2.72 [2.33, 3.00]	0.90 [0.84, 1.01]	**5.86** [4.97, 12.92]	0.01 [0.01, 1.00]	1.00 [1.00, 1.00]
GGA++	1.02 [0.77, 1.11]	0.61 [0.61, 0.61]	1.02 [0.84, 1.02]	**8.87** [6.73, 8.89]	–	–

the absolute and relative performance of each configurator varies a little between configuration budgets, the results presented here nevertheless reflect the main trends for each scenario. The remainder of the results are available as supplementary material. We mark the median speedups in boldface if they are not statistically worse than the best speedup within each configuration budget according to a permutation test with a 5% significance level.

When we compare the small parallel budget analysis results (available in the supplementary material) to the large parallel budget analysis results, we can see that all procedures tend to perform better using the large budget analysis and that GPS performs best (relative to the other configurators) in the large budget results. We therefore refer the reader to the supplementary material for the complete results, and discuss the 6 and 24 hour large budget analysis results in the remainder of this section.

Overall, we can see that GPS consistently obtained the greatest median speedup for three of the five scenarios with 8 cores per configurator run: LKH and EAX on TSP RUE 1000-3000 instances, and CaDiCaL on circuit fuzz instances, with SMAC finding similar-performing configurations for EAX at 6 hours and for CaDiCaL at 24 hours. The only 8 core scenario for which GPS did not perform competitively with all (in fact any) of the other configurators was probSAT on the 7SAT90 instance set. GPS did find configurations with speedups greater than 3 for both time budgets; however, the other procedures obtained even greater speedups. For CPLEX on Regions200, GPS initially obtains the best speedup, but then suffers from some performance degradation, so that SMAC ends up winning for the 24-hour configuration budget.

For CPLEX on the RCW2 instance set (the stretch test, for which we provided the parallel configurators with 32 cores per run instead of 8), we see that SMAC initially finds the greatest speedup, but that GPS’s median speedup eventually surpasses SMAC. Surprisingly, we actually see that SMAC obtains exactly the same speedups for all configuration budgets (see supplementary material). In fact, only 17.5% of the runs of SMAC updated the incumbent after the first three minutes; however, 100% of the runs updated the incumbent within the first three minutes. This indicates that SMAC may be getting lucky by finding better-than-default configurations prior to having any statistical confidence of their improvement. Therefore, these improvements are likely due to the standard protocol when applied to mostly-random configurations.

Overall, GPS appears to be the top-performing configuration procedure, followed by SMAC. However, one advantage shared by GPS, GGA++ and irace is that they are all able to make use of parallel resources within individual configurator runs, whereas SMAC must be run multiple times independently in parallel. The time to validate all of SMAC’s configurations is sometimes very large. The most extreme example is for the 6-hour configuration budget for CPLEX on Regions200, where GPS required 1.64 hours for validation and SMAC required 139.79 hours. While there are also some less extreme examples – *e.g.*, the 0.08 *vs* 0.63 hours required to validate the 24-hour configuration budget runs on LKH – SMAC still always required more validation time overall. This becomes even more pronounced for scenarios with large and difficult instance sets. An alternative is to perform fewer independent runs of SMAC; this gives it less total parallel resources, but a fair chance during validation. However, for challenging scenarios, this typically yields worse performance for SMAC.

6 RELATED WORK

Birattari et al. [6] proposed racing procedures for general purpose algorithm configuration. The key idea is to begin with an initial set of candidate configurations and then iteratively evaluate them on new instances interleaved with statistical tests to determine as quickly as possible when challengers can be discarded with confidence. Specifically, Birattari et al. [6] recommended to use the Friedman test. However, the initial version of F-Race is often prohibitively expensive to apply in practice, since they began the race by evaluating all possible configurations of an algorithm. A family of methods known as iterated racing procedures overcomes this limitation, whereby racing procedures are iteratively applied to a set of randomly generated configurations. In these methods, the winners of each race are used to bias the random sampling procedure towards high-quality regions of the configuration space. Balaprakash et al. [3] introduce such a method for numerical parameters, which Birattari et al. [7] later extended to handle categorical and conditional parameters. By their nature, racing procedures are an embarrassingly parallel method. This fact was first exploited in an implementation by López-Ibáñez et al. [21], which was later improved to include an adaptive capping procedure [8] (the variant we studied here). In a separate line of work, Styles and Hoos [25] proposed a permutation test to determine the outcome of individual races, arguing that the rank-based F-Test only indirectly optimizes mean running time. It was this work that inspired the permutation test used in GPS (see Sections 3.1, 3.2 and 3.6).

Bartz-Beielstein et al. [4] introduce using Bayesian Optimization concepts for automated algorithm configuration. Hutter et al. [15] argued that the large variance in the running time distributions of many algorithms posed a unique challenge to algorithm configuration procedures (a key insight that informed many components crucial to the design of GPS, see, *e.g.*, Sections 3.1, 3.7 and 3.8). They proposed to address this in two ways: First, with a projected process model that incorporated this uncertainty; and second, via an intensification mechanism. However, all of this early work was limited to optimizing only numerical parameters of algorithms applied to individual instances rather than instance sets. Hutter et al. [13] later proposed a configuration procedure capable of working with instance sets called SMAC, which uses a random forest model, thereby extending the method to categorical parameters.

In a second line of work Hutter et al. [17] introduced ParamILS, an iterated local search procedure for automated algorithm configuration. ParamILS was later improved with a novel adaptive capping mechanism [16]. However, since then SMAC has continued to be refined [10, 12, 20] and is typically recommended as the method of choice when choosing between the two[3].

While neither SMAC nor ParamILS are parallel algorithms, it is common practice to perform 10-25 independent runs of the methods in parallel and then validate the resulting configurations on the entire training set and return the best one (see, *e.g.*, Styles et al. [26]). However, a variant of SMAC was proposed by Hutter et al. [14] that evaluated multiple candidate configurations in parallel. Unfortunately, to the best of our knowledge this version of SMAC has never been made publicly available and so we were only able to compare GPS to SMAC when using the standard protocol with multiple independent parallel runs of SMAC.

Another approach for automated algorithm configuration is the use of a gender-based genetic algorithm, GGA, proposed by Ansótegui et al. [2]. GGA applies a different selection pressure to each gender of the population: For one gender only the top X% of the population are kept, whereas the other gender is not subjected to pressure and is instead used merely to store diversity in the population. While this avoids the computation time needed to evaluate half of the population, it is also a departure from classical gender-based genetic algorithms where both populations are normally evaluated according to a fitness criteria. Ansótegui et al. [1] later propose a modified method, GGA++, which uses a tailor-purposed random forest model trained on the competitive gender to predict the performance of the configurations in the non-competitive gender. These predictions are then used to sample from the non-competitive population when selecting mates. This random forest model is also used when determining which genes to keep from each parent when mating. Surprisingly, neither GGA nor GGA++ makes use of several of the crucial components commonly used by state-of-the-art algorithm configurators, for example an adaptive capping mechanism [8, 16], an intensification mechanism [15] or a racing mechanism [6], which may explain why GGA++ performed so poorly on the CPLEX scenarios (see Section 5). However, both methods are implemented in such a way that the evaluation of candidate configurations can be performed in parallel.

7 CONCLUSIONS AND FUTURE WORK

We introduced a powerful automated algorithm configurator, GPS, designed to operate in parallel and exploit recent insights on algorithm configuration landscapes [24]. GPS combines essential ingredients of state-of-the-art algorithm configuration procedures [8, 13, 16] with a simple one-dimensional bracketing procedure [19] applied to each parameter semi-independently in parallel. Despite strong assumptions made by GPS about algorithm configuration landscapes – *i.e.*, that parameter responses are uni-modal and that most parameters do not strongly interact – GPS found the best configurations in 5 out of 6 scenarios, often at a fraction of the time budget required by other methods. For example, for CaDiCaL on the circuit fuzz SAT instance set, GPS achieved a median speedup of 1.41 after a 6-wall-clock-hour configuration budget and 0.51 wall-clock hours for validation, compared to SMAC, which obtained a speedup of 1.25 after a 6-wall-clock-hour configuration budget and 4.79 wall-clock hours for validation. (Note that SMAC actually obtained a slightly better speedup of 1.31 within a 1-hour configuration budget; this phenomenon might be caused by SMAC's incumbent updating procedure, which does not make use of statistical tests for performance improvements.)

In two cases, GPS exhibited undesirable behaviour, *i.e.*, premature stagnation for probSAT on the 7SAT90 instances, and returning substantially worse configurations after more configuration time for CPLEX on Regions200. We suspect that parameter interactions may have caused this behaviour, since GPS can occasionally return incumbents which have never been evaluated on any instances. Increasing the decay rate and using the standard protocol can help safeguard against such effects. Future work could combine multiple independent parallel runs of GPS with a racing procedure that evaluates the anytime incumbents of each GPS run as these runs are performed. This could also be combined with a mechanism to detect stagnation and initiate random restarts. We believe that this could not only improve the performance of GPS, but also remove the need for post-configuration validation. Future work should also be done for solution quality optimization scenarios, in particular from automated machine learning, and for configuration scenarios involving substantially increased parallel computing resources. Finally, it may be advantageous to extend GPS to allow for both soft and hard bounds on numerical parameter values, since exceeding some bounds may lead GPS to finding incumbents that produce incorrect output.

[3]Personal communication with the authors.

REFERENCES

[1] Carlos Ansótegui, Yuri Malitsky, Horst Samulowitz, Meinolf Sellmann, and Kevin Tierney. 2015. Model-Based Genetic Algorithms for Algorithm Configuration. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI 2015)*. 733–739.

[2] Carlos Ansótegui, Meinolf Sellmann, and Kevin Tierney. 2009. A Gender-Based Genetic Algorithm for the Automatic Configuration of Algorithms. In *Proceedings of the Fifteenth International Conference on Principles and Practice of Constraint Programming (CP 2009)*. 142–157.

[3] Prasanna Balaprakash, Mauro Birattari, and Thomas Stützle. 2007. Improvement strategies for the F-Race algorithm: Sampling design and iterative refinement. In *International workshop on hybrid metaheuristics*. Springer, 108–122.

[4] Thomas Bartz-Beielstein, Christian WG Lasarczyk, and Mike Preuss. 2005. Sequential parameter optimization. In *2005 IEEE congress on evolutionary computation*, Vol. 1. IEEE, 773–780.

[5] Armin Biere. 2017. CaDiCaL, Lingeling, Plingeling, Treengeling and YalSAT Entering the SAT Competition 2017. In *Proceedings of SAT Competition 2017: Solver and Benchmark Descriptions*. 14–15.

[6] Mauro Birattari, Thomas Stützle, Luis Paquete, and Klaus Varrentrapp. 2002. A Racing Algorithm for Configuring Metaheuristics. In *Proceedings of the Third Genetic and Evolutionary Computation Conference (GECCO 2002)*. 11–18.

[7] Mauro Birattari, Zhi Yuan, Prasanna Balaprakash, and Thomas Stützle. 2010. F-Race and iterated F-Race: An overview. In *Experimental methods for the analysis of optimization algorithms*. Springer, 311–336.

[8] Leslie Pérez Cáceres, Manuel López-Ibáñez, Holger H. Hoos, and Thomas Stützle. 2017. An experimental study of adaptive capping in irace. In *Proceedings of the Eleventh International Conference on Learning and Intelligent Optimization (LION 2017)*. Springer, 235–250.

[9] IBM Corp. 2020. IBM ILOG CPLEX Optimizer. https://www.ibm.com/analytics/data-science/prescriptive-analytics/cplex-optimizer, Last accessed on 2020-02-03. (2020).

[10] Stefan Falkner, Marius Lindauer, and Frank Hutter. 2015. SpySMAC: Automated configuration and performance analysis of SAT solvers. In *International Conference on Theory and Applications of Satisfiability Testing (SAT 2015)*. 215–222.

[11] Keld Helsgaun. 2000. An effective implementation of the Lin-Kernighan traveling salesman heuristic. *European Journal of Operational Research* 126 (2000), 106–130.

[12] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2011. Bayesian optimization with censored response data. In *Procedings of 2011 NeurIPS workshop on Bayesian Optimization, Experimental Design, and Bandits*.

[13] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2011. Sequential model-based optimization for general algorithm configuration. In *Proceedings of the Fifth International Conference on Learning and Intelligent Optimization (LION 2011)*. 507–523.

[14] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2012. Parallel Algorithm Configuration. In *Proceedings of Sixth International Conference on Learning and Intelligent Optimization (LION 2012)*. 55–70.

[15] Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Kevin Murphy. 2010. Time-bounded sequential parameter optimization. In *Proceedings of the Fourth International Conference on Learning and Intelligent Optimization (LION 2010)*. Springer, 281–298.

[16] Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Thomas Stützle. 2009. ParamILS: an automatic algorithm configuration framework. *Journal of Artificitial Intelligence Research* 36 (2009), 267–306. Issue 1.

[17] Frank Hutter, Holger H. Hoos, and Thomas Stützle. 2007. Automatic algorithm configuration based on local search. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence (AAAI 2007)*, Vol. 7. 1152–1157.

[18] Frank Hutter, Manuel López-Ibáñez, Chris Fawcett, Marius Lindauer, Holger H. Hoos, Kevin Leyton-Brown, and Thomas" Stützle. 2014. AClib: A Benchmark Library for Algorithm Configuration. In *Proceedings of the Fourteenth International Conference on Learning and Intelligent Optimization (LION 2014)*. 36–40.

[19] Jack Kiefer. 1953. Sequential minimax search for a maximum. *Proceedings of the American mathematical society* 4, 3 (1953), 502–506.

[20] Marius Lindauer and Frank Hutter. 2018. Warmstarting of model-based algorithm configuration. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI 2018)*. 1355–1362.

[21] Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie P. Cáceres, Thomas Stützle, and Mauro Birattari. 2016. The irace package: Iterated Racing for Automatic Algorithm Configuration. *Operations Research Perspectives* 3 (2016), 43–58.

[22] Yuichi Nagata and Shigenobu Kobayashi. 2013. A Powerful Genetic Algorithm Using Edge Assembly Crossover for the Traveling Salesman Problem. *Institute for Operations Research and the Management Sciences Journal on Computing* 25, 2 (2013), 346–363.

[23] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. 1992. Golden section search in one dimension. *Numerical Recipes in C: The Art of Scientific Computing* (1992).

[24] Yasha Pushak and Holger H. Hoos. 2018. Algorithm configuration landscapes: More benign than expected?. In *Proceedings of the Fifteenth International Conference on Parallel Problem Solving from Nature (PPSN 2018)*. Springer, 271–283.

[25] James Styles and Holger H. Hoos. 2013. Ordered Racing Protocols for Automatically Configuring Algorithms for Scaling Performance. In *Proceedings of the Fifteenth Conference on Genetic and Evolutionary Computation (GECCO 2013)*. ACM, 551–558.

[26] James Styles, Holger H. Hoos, and Martin Müller. 2012. Automatically Configuring Algorithms for Scaling Performance. In *Proceedings of the Sixth International Conference on Learning and Intelligent Optimization (LION 2012)*. 205–219.