Hyper-parameterized Dialectic Search for Non-linear Box-Constrained Optimization with Heterogenous Variable Types

Meinolf Sellmann[1] and Kevin Tierney[2(✉)]

[1] General Electric, Niskayuna, NY, USA
meinolf@ge.com
[2] Bielefeld University, Bielefeld, Germany
kevin.tierney@uni-bielefeld.de

Abstract. We consider the dialectic search paradigm for box-constrained, non-linear optimization with heterogeneous variable types. In particular, we devise an implementation that can handle any computable objective function, including non-linear, non-convex, non-differentiable, non-continuous, non-separable and multi-modal functions. The variable types we consider are bounded continuous and integer, as well as categorical variables with explicitly enumerated domains. Extensive experimental results show the effectiveness of the new local search solver for these types of problems.

1 Introduction

Box-constrained optimization problems are ubiquitous. They appear when optimizing designs where objective functions estimate key performance characteristics of the design, when optimizing control parameters in simulated environments in simulation-based optimization, when optimizing function parameters to fit training samples in machine learning, when searching for new sample points over surrogate functions in Bayesian optimization, and when searching for optimal strategies in decision aiding. In many cases, these target functions are highly non-linear, non-convex, even non-differentiable, non-separable, or non-continuous. Moreover, frequently decision variables are discrete integer variables or even categorical variables that can only take specific values.

We tackle this problem using the dialectic search paradigm [8] in a hyper-parameterized setting [4]. The objective of this work is to show that a tuned, hyper-parameterized dialectic search program (that is implemented in C++ with less than 1,500 lines of code) can effectively tackle these problems. All that a user needs to provide is a short piece of code that computes the objective function given a variable assignment, as well as an incremental objective evaluation when changing two variable assignments from the last assignment. The latter can obviously be performed easily simply by reducing to the first evaluation function,

I. S. Kotsireas and P. M. Pardalos (Eds.): LION 14 2020, LNCS 12096, pp. 102–116, 2020.
https://doi.org/10.1007/978-3-030-53552-0_12

meaning our approach supports full black-box settings. However, in many cases, objectives can be evaluated more quickly when exploiting incrementality, giving the user the opportunity to gain efficiency in this way.

This paper makes the following contributions:

1. We extend the dialectic search paradigm to a general variable setting (continuous, integer and categorical).
2. We increase dialectic search's performance through a convex search procedure for its *synthesis* procedure, which can be seen as an alternative to path relinking.
3. We apply the hyper-parameterization paradigm from [4] to give this general-purpose tool the ability to be tuned for specific benchmarks and to adapt key search parameters dynamically during search.

In the following, we first review the concepts of dialectic search and hyper parameterization. Then, we apply these concepts to devise an algorithm implementation for solving box-constrained non-linear optimization problems with heterogeneous variable types. Following this detailed description, we then evaluate our implementation on various benchmarks to establish the efficacy of our new approach.

2 Dialectic Search

2.1 A Philosophical Framework as Metaheuristic

Dialectic search was first introduced in [8] as a metaheuristic for local search. The approach exists as a means to overcome the difficulties when applying a local search metaheuristic framework to a specific problem domain by enforcing a strict separation of intensification via greedy search and search space exploration via randomization. In [8], it is shown that dialectic search does not require any sophisticated tailoring for the concrete application domain at hand. At the same time, it significantly outperforms other metaheuristics, such as Tabu Search, Simulated Annealing, and Genetic Algorithms, on multiple benchmarks from constraint satisfaction and combinatorial and continuous optimization.

Dialectic search works as follows:

1. **Initialization:** We construct an assignment of variables to random values in their respective domains.
2. **Thesis:** We call the current assignment the "thesis" and greedily improve it until we find a local optimum. We can conduct a full (where we consider all potential moves before choosing the best) or a first-improvement (where we take the first possible move that improves the objective) greedy search. A search parameter determines the percentage of variables we consider in each greedy step.

3. **Antithesis:** We randomly modify the thesis by changing select variables' values, which can be seen as a type of perturbation like in Iterated Local Search [9]. This new assignment is called the "antithesis." A search parameter determines the percentage of variables to modify. Another search parameter determines the probability with which we greedily improve the antithesis. Again, we can conduct a full or a first-improvement greedy search. A separate search parameter determines how many variables to consider in each greedy step.
4. **Synthesis:** Next we search the space between thesis and antithesis by conducting a limited local search in the space that thesis and antithesis span, i.e., variables to whom both the thesis and antithesis assign the same value cannot be changed. The exact way how this nested local search is to be conducted is left open by dialectic search. For example, we could conduct a greedy path relinking [7] between the thesis and antithesis. Alternatively, we could start a more elaborate nested local search to find a favorable recombination of thesis and antithesis. In this paper, we will introduce yet another procedure, where we search over the space of convex combinations of thesis and antithesis. No matter how this step is implemented, we call the resulting assignment the "synthesis."
5. **Iteration and Restarts:** We next decide if we want to restart and return to Step 1. A search parameter determines the probability with which we restart. If we do not restart, we next test if the synthesis improves over the thesis. If so, the synthesis becomes the new thesis, and we continue with Step 2. If not, we continue with Step 3.

2.2 Hyper-parameterized Dialectic Search

While dialectic search was shown to give very good results even without sophisticated tuning, in [4] dialectic search was *hyper-parameterized* to tackle the MaxSAT problem. The core idea of hyper-parameterization is to enable a search heuristic like dialectic search to self-adapt search parameters *during* runtime based on runtime statistics. The parameters we adjust include the probability to restart, the size of the antithesis, the number of variables to open for the greedy heuristic, etc. Furthermore, the runtime statistics are, for example, the estimated number of local search moves until time limit, time since last overall improvement, time since last improvement in current restart, and the total time in the current restart. Hyper-parameterization has also been successfully applied to tabu search [3].

To hyper-parameterize dialectic search, [4] suggests to use a logistic regression function to transform normalized runtime statistics into probabilities (e.g., for restart probabilities) or percentages (e.g., for the antithesis size). These logistic regression functions in turn introduce new "hyper-parameters."

$$P_k = \frac{1}{1 + e^{\sum_i h_i^k s_i + h_0^k}} \tag{1}$$

In Eq. (1), the probability (for example for a restart in Step 5), or the percentage (for example to determine the percentage of variables to be modified in Step 3) P_k for the kth search parameter is derived from runtime statistics s. The key is that this transformation of current runtime statistics into search parameter values takes place *each time when the search parameter is needed*. For example, whenever we need to decide if we are going to restart the search in Step 5, we gather the current runtime statistics s and enter them into the logistic function using (static) hyper-parameters h^k to compute the probability of a restart. Then, we flip a coin, and with the computed probability we restart the search. So, to compute this probability/percentage, the current runtime statistics s are determined, the inner product with the search-parameter-specific hyper-parameters h^k is computed, the constant h_0^k added, and the result handed to the exponential function. This produces a value between minus infinity and infinity, which the logistic function transforms into a value between 0 and 1.

In contrast to regular parameterization, in which we would keep the restart probability and the percentage of variables we modify in the antithesis constant, hyper-parameters merely determine how the respective search parameters are derived from *current* runtime statistics. In [4], it is further suggested to use an instance-specific parameter tuner (e.g., [2] or [1]) to "learn" good settings for the hyper-parameters. For the MaxSAT problem, this resulted in a dialectic search portfolio that won four out of nine categories at the 2016 MaxSAT Evaluation [5].

3 Dialectic Search for Box-Constrained Non-Linear Optimization Problems with Continuous, Integer, and Categorical Variables

Having reviewed the general framework of hyper-parameterized dialectic search, we now introduce our new program for box-constrained non-linear optimization. In particular, our goal is to develop an implementation of dialectic search that works with any computable objective function and for mixtures of bounded continuous, integer, and explicitly enumerated categorical variables. In the following, we devise such a program that, in order to apply it to a new problem domain, only requires the implementation of an evaluation function of the objective to be minimized[1] for a given assignment of variables. For reasons of efficiency, we also require a second objective evaluation function that returns the objective value for the same assignment as was given for the last evaluation when two variables are changed to new values. This can be implemented easily by altering the last assignment and evaluating the objective from scratch using the first evaluation function. However, for many objectives it is possible to evaluate a new assignment incrementally and much faster than by re-evaluating the new assignment from scratch.

To instantiate dialectic search, we need to define each of the five steps (Initialization, Thesis, Antithesis, Synthesis, and Restarting) of dialectic search, as

[1] Note that the latter easily allows maximization as well, simply by having the function return the negative of the actual objective value.

well as determine how to hyper-parameterize the resulting algorithm. In the following, we describe in detail how each of these aspects is implemented.

3.1 Initialization

The dialectic search receives three parameters: 1. The number of variables n. 2. A vector of length n of a composite class that describes each variable's type, as well as its lower and upper bound or, in case of categorical variable, and explicit enumeration of the values that the variable can take. 3. A pointer to an evaluation class that has two evaluation functions, one that takes an entire assignment vector of length n as input and returns a rational number as output, the second taking two variable indices and two new values for the respective variables as input and returning a rational number as output. To initialize the local search we simply assign a random value within each variable's domain to the respective variable.

3.2 Thesis

The new assignment generated is labeled as the *thesis*. To greedily improve the thesis, we proceed as follows:

1. In each greedy step, we first select a random subset of variables, How many is determined by a search parameter.
2. Next we consider the variables in random order. For categorical variables, we consider each possible value and evaluate the objective if we change the respective variable to the new value. For ordinal and continuous variables, we conduct a pseudo-convex optimization as follows. First, we choose a number $\alpha \in [0, 1]$ uniformly at random. Next we construct the interval $[x_1, x_4]$ where $x_1 = \max\{v - \alpha/2, L\}$ and $x_4 = \min\{v + \alpha/2, U\}]$, where v is the current value of the variable, and L, U are its lower and upper bound, respectively. We evaluate the objective when setting the variable to each end point of the interval, as well as at two intermediate points x_2 and x_3. The point x_3 is at $\frac{200}{\sqrt{(5)}+1}\% \simeq 61.8\%$ inside the interval and x_2 is at $100 - \frac{200}{\sqrt{(5)}+1}\% \simeq 38.2\%$ inside the interval. Note that for ordinal variables, we round to the nearest integer for evaluating the objective. However, the computation below continues using the actual fractional values.
   If the objective at x_1 is strictly less than at the other three points, or if the objective at x_2 is lower than at x_3, we reduce the interval to $[x_1, x_3]$. In this case, the point x_2 automatically finds itself at 61.8% of the new interval. Therefore, to continue, we only need to evaluate one new point at 38.2% of the new interval to iterate the search. On the other hand, if x_4 is strictly less than the other three points, or if x_3 results in a better objective than x_2, we continue the search in $[x_2, x_4]$. In this case, x_3 already finds itself at 38.2% of the new interval, so we only need to evaluate one more point at 61.8% to conduct the next iteration. The search stops when the interval length shrinks below a certain minimum length, which is set by a search parameter.

3. Having thus found an optimal or at least very favorable value for the respective variable in this way, we check if it improves the objective. We conduct a first-improvement greedy search. In this case, we next commit the variable to this value and continue with Step 1. If the best variable assignment found does not lead to an improvement of the objective, we continue with the next variable in the random selection, in random order.
4. The greedy search ends when no value can be found for any variable in the random selection that improves on the objective value.

3.3 Antithesis

To select an antithesis, we first choose a random subset of variables. How many variables is determined by a search parameter. Then, for each variable in the selection we choose a new value uniformly at random. Depending on another search parameter, we may conduct a greedy search to improve the antithesis (in the same manner as we just improved the thesis) before we move to the next step.

3.4 Synthesis

The purpose of this step is to opportunistically search the space between the thesis and antithesis. All variables to which the antithesis and thesis assign the same value are not changed. In a first step, we aim to recombine the variable settings between thesis and antithesis via path-relinking. We start at the thesis and consider all variables in turn to assess which variable, when set to the value given by the antithesis, would give the best objective value. This variable is then set to the antithesis value, regardless whether this leads to a worsening of the objective or not. We proceed in this way until all variables are set to the respective antithesis values. We next consider the assignment with the best objective value on this chain of assignments that "connect" thesis and antithesis. If the best assignment on this chain improves over the thesis, then this synthesis becomes the new thesis and we continue by jumping back to Step 2.

If the best recombination found does not improve over the current thesis, we next consider convex-combinations between thesis and anti-thesis. That is, we interpolate ordinal and continuous parameters between the respective values in the thesis and antithesis, and "round" categorical variables to the "nearest" thesis or antithesis value. For example, assume we have three parameters, the first is categorical (say it can take values red, blue and green), the second parameter is ordinal, and the third is continuous. Assume the thesis assignment is [green, -2, 0.2] and the antithesis is [red, 5, 0.7]. For the categorical parameter, we associate value 0 with the thesis value, and 1 with the antithesis value. Any interpolation value below 0.5 is then "rounded down" to the thesis value, all values greater or equal 0.5 get "rounded up" to the antithesis value. Similarly, for the ordinal parameters we round to the nearest integer. Then, the 0.6 interpolation between thesis and antithesis is, for example, [red, 2, 0.5]. Having thus defined how points on the "line" between thesis and antithesis map to assignments, we can conduct

a pseudo-convex optimization, exactly as we did earlier in the greedy improvement of ordinal and continuous variables in Step 2. If this procedure results in an assignment that improves over the thesis, this synthesis becomes the new thesis. If not, then we keep the old thesis and consider a new randomized antithesis in Step 3.

3.5 Restarting

At the end of each synthesis step, we flip a coin to determine if we restart the search instead of attempting to further improve the current thesis assignment. The probability of a restart is again given by a search parameter. When a restart is triggered, we apply a randomized modification of the current thesis before we jump back to Step 2. This modification works exactly like the way how an antithesis is constructed, whereby we use a different search parameter to determine how much the new starting point will differ from the current thesis. Finally, when the time-limit of the search is exceeded, we return the best assignment ever encountered during the search.

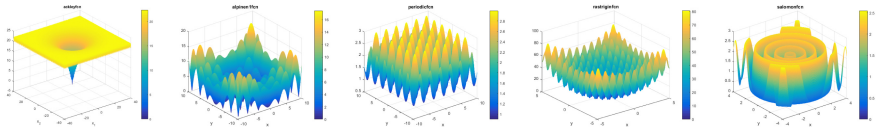
3.6 Hyper-parameterization

The outline of our dialectic search instantiation shows that there are numerous search parameters influencing how the search progresses. For example, if we use a very high restart probability, we will primarily perform randomly restarted greedy searches. Alternatively, if we consider a very high percentage of variables to construct the antithesis, the bulk of the search effort will consist in synthesis steps (Fig. 1).

We list the search parameters within our implementation below:

g: The size of the greedy candidate set as percentage of all variables in the problem.
a_l, a_u: A lower and upper bound on the percentage of variables to be changed to construct an antithesis. The exact size of the change is then chosen uniformly at random in the interval given whenever a new antithesis is generated.
p_a: The probability of greedily improving the antithesis.
p_r: The probability of restarting the search.
r_l, r_u: A lower and upper bound on the percentage of variables to be changed to construct a new starting point when a restart is triggered. The exact size of the change is then chosen uniformly at random in the interval $[r_l, r_u]$.

As reviewed earlier, [4] proposes not to assign static values to these search parameters, but to allow them to dynamically adapt to the way in which the search progresses. Equation (1) governs how we derive probabilities and percentages from runtime statistics. Therefore, the only information needed at this point is which runtime statistics s we will track. In the following, we list the statistics we track during the search:



(a) Ackley (b) Alpine (c) Periodic (d) Rastrigin (e) Salomon

Fig. 1. Optimization functions over two continuous variables. Images from http://benchmarkfcns.xyz/.

1. Time elapsed as percentage of total time before timeout.
2. Number of restarts conducted as a percentage of total restarts expected to be completed within the time limit.
3. Number of moves as a percentage of the total moves expected to be completed within the time limit.
4. Number of steps as a percentage of the total steps expected to be completed within the time limit.
5. Total number of improving syntheses found over the total number of dialectic moves expected to be completed within the time limit.
6. Number of moves in the current restart over the total number of dialectic moves expected to be completed within the time limit.
7. Number of moves since the current best known solution was found over the total number of dialectic moves expected to be completed within the time limit.
8. Number of moves since the last thesis update in the current restart over the total number of dialectic moves expected to be completed within the time limit.
9. Number of steps in the current restart over the total number of steps expected to be completed within the time limit.
10. Number of steps since the current best known solution was found over the total number of steps expected to be completed within the time limit.
11. Number of steps since the last thesis update in the current restart over the total number of steps expected to be completed within the time limit.

The resulting dialectic search with seven search parameters is now *hyper-configurable* with 84 (7 times 12) hyper-parameters. To tune these parameters for specific instance distributions, we employ the parameter tuner GGA++ [1].

4 Experimental Results

We now evaluate our approach on various benchmarks to assess its effectiveness. We consider the following function classes:

– **Ackley Instances:** This class has n continuous decision variables $X_i \in [-500, 500]$, m integer decision variables $Y_j \in [-500, 500]$, n dependent modeling variables $A_i = X_i - t_i$ for n given values $t_i \in [-500, 500]$ and m dependent modeling variables $B_j = Y_j - s_j$ for m given values $s_j \in [-500, 500]$. The objective is to minimize $b + f(n + m) -$

$f(n+m)b^{-\sum_i \frac{A_i^2}{10n} - \sum_j \frac{B_j^2}{10m}} - b^{\sum_i \frac{\cos(2\pi A_i^2)}{2n} + \sum_j \frac{\cos(2\pi B_i^2)}{2m}}$ for given parameters $f, b \in \mathbb{N}$. An instance to this problem is therefore fully described by the vector $(n, t_1, \ldots, t_n, m, s_1, \ldots, s_m, f, b)$.

- **Alpine Instances:** This class has n continuous decision variables $X_i \in [-10, 10]$ and n dependent modeling variables $A_i = X_i - t_i$ for n given values in $[-10, 10]$. The objective is to minimize $\sum_i |A_i \sin A_i + 0.1 A_i|$. An instance to this problem is therefore fully described by the vector $(n, t_1, \ldots, t_n)$.
- **Griewank Instances:** This class has n continuous decision variables $X_i \in [-600, 600]$ and n dependent modeling variables $A_i = X_i - t_i$ for n given values in $[-600, 600]$. The objective is to minimize $1 + \sum_i \frac{A_i^2}{4000} - \prod_i \cos(\frac{A_i}{\sqrt{i}})$. An instance to this problem is therefore fully described by the vector $(n, t_1, \ldots, t_n)$.
- **Periodic Instances:** This class has n continuous decision variables $X_i \in [-10, 10]$ and n dependent modeling variables $A_i = X_i - t_i$ for n given values in $[-10, 10]$. The objective is to minimize $1 + 10 \sum_i \sin^2(A_i) - e^{-\sum_i A_i^2}$. An instance to this problem is therefore fully described by the vector $(n, t_1, \ldots, t_n)$.
- **Rastrigin Instances:** This class has n continuous decision variables $X_i \in [-5.12, 5.12]$ and n dependent modeling variables $A_i = X_i - t_i$ for n given values in $[-5.12, 5.12]$. The objective is to minimize $10n + \sum_i A_i^2 - 10 \cos(2\pi A_i)$. An instance to this problem is therefore fully described by the vector $(n, t_1, \ldots, t_n)$.
- **Salomon Instances:** This class has n continuous decision variables $X_i \in [-100, 100]$ and $n+1$ dependent modeling variables $A_i = X_i - t_i$ for n given values in $[-100, 100]$ and $Z = \sqrt{\sum_i A_i^2})$. The objective is to minimize $1 - \cos(2\pi Z) + 0.1Z$. An instance to this problem is therefore fully described by the vector $(n, t_1, \ldots, t_n)$.
- **Mixed Instances:** This class has n continuous decision variables $X_i \in [-1, 1]$ for $i \in \{0, \ldots, n-1\}$, m integer decision variables $Y_j \in [-25, 5]$ for $j \in \{0, \ldots, m-1\}$, and o categorical decision variables $Z_k \in \{2, 3, 4\}$ for $k \in \{0, \ldots, o-1\}$. For each instance, we are given n continuous target values $t_i \in [0, 1]$, m integer target values $s_j \in [-25, 5]$, and o categorical targets $r_k \in \{2, 3, 4\}$. We use $n + m + o$ continuous modeling constants $a_l = \frac{t_{l\%n} t_{(l+1)\%n} + s_{(l-1)\%m} - s_{(l+1)\%m}}{r_{l\%o}}$, whereby % symbolizes the modulo operation. The objective is to

$$\text{Minimize} \sum_{l=0}^{n+m+o-1} \left(\frac{X_{l\%n} X_{(l+1)\%n} + Y_{(l-1)\%m} - Y_{(l+1)\%m}}{Z_{l\%o}} - a_l\right)^2.$$

An instance to this problem is therefore fully described by the vector

$$(n, t_0, \ldots, t_{n-1}, m, s_0, \ldots, s_{m-1}, o, r_0, \ldots, r_{o-1}).$$

Note that, in the above, we use modeling variables A, B to simplify the definition of these problems. In the actual implementation, we immediately substitute these variables with the respective expression over the decision variables in the objective function.

Table 1. Comparison of the new dialectic search approach (DS) with the dialectic search from [8] (DS-old) and simulated annealing approaches from [10] (SA-0.98 and SA-0.99)

Function	Dims	DS		DS-old		SA-0.98		SA-0.99	
		Value	Evals	Value	Evals	Value	Evals	Value	Evals
Alpine	20	10^{-3}	**21K**	10^{-3}	86K	10^{-3}	1M	10^{-3}	2M
	50	10^{-3}	**364K**	10^{-3}	458K	10^{-3}	2.9M	10^{-3}	5.8M
Rastrigin	20	10^{-3}	**18K**	10^{-3}	208K	24.4	3.4M	22.4	6.8M
	50	10^{-3}	**138K**	10^{-3}	818K	87.3	8.3M	86.8	9.9M

4.1 Minimizing the Number of Function Evaluations

As our approach works with any computable objective function, it is ideally suited for black-box optimization. The main objective is to minimize the number of black-box function evaluations that are needed to reach a certain approximation level of the global optimum. We use randomly generated Rastrigin and Alpine instances for this purpose.

In Table 1, we compare against the results published in [8]. There, a dialectic search (DS-old) algorithm was introduced for continuous optimization problems, and compared with a simulated annealing approach using two different cooling factors (SA-0.98 and SA-0.99) [10]. While the old dialectic search already gave very competitive results, we clearly see the benefits of the new approach, that has three main improvements over the old dialectic search approach: 1. The line search used as greedy step in DS-old is replaced with the pseudo-convex search approach we outlined in Step 2 in the Section "Thesis." 2. The new DS approach conducts interpolation searches on top of recombination searches to produce an improving synthesis, as outlined in Section "Synthesis." And 3. The new approach is hyper-parameterized, which not only allows the approach to adjust otherwise static parameters during the search, but also to tailor the search behavior for the respective benchmark – fully automatically, thanks to parameter tuners like GGA++ [1].

Overall, on average the new approach lowers the number of function evaluations by a factor 3. On 20-dimensional Rastrigin functions, the improvement is a whole order of magnitude, while for 50-dimensional Alpine instances we "only" save 20% of function evaluations, compared to the old DS approach. When compared to the better of the SA approaches, the hyper-reactive dialectic search approach only requires 3.5% of the function evaluations required by SA-0.98.

4.2 Optimizing Functions Within a Given Time Limit

We now change the setting to the more realistic scenario where we have been given a timelimit and have to find the best solution possible in the given time. We first examine the performance of DS versus a well-known continuous black-box optimizer, LSHADE [11], which we use thanks to its performance and its

Table 2. Comparison of DS to LSHADE on the Alpine and Rastrigin functions with 50 instances for each function/dimension pair.

Function	Dims	# ≤ 1e-3		Mean		Geo. Mean	
		LSHADE	DS	LSHADE	DS	LSHADE	DS
Alpine	20	49	50	0.021	0.001	0.001	0.001
	50	49	50	0.021	0.001	0.001	0.001
	200	4	50	3.500	0.001	1.652	0.001
	500	0	50	57.140	0.001	56.369	0.001
Rastrigin	20	33	50	0.836	0.001	0.013	0.001
	50	30	50	1.228	0.001	0.021	0.001
	200	0	50	416.282	0.001	410.934	0.001
	500	0	50	3010.779	0.001	2457.697	0.001

freely available source code. For this comparison, we again consider the Alpine and Rastrigin benchmarks. Later in this section, we will expand our analysis to the other benchmarks, including ones with discrete parameters, and include the optimizer LocalSolver [6] for comparison. In all experiments, we impose a 60 second CPU timelimit.

Comparison with LSHADE. Table 2 provides a comparison of dialectic search (DS) and LSHADE. We note that LSHADE is generally used on lower dimensional problems. For 20 and 50 dimensional instances, it performs reasonably well, even though it does not manage to find a solution below 1e-3 for all instances. However, many real-world optimization problems contain thousands of variables, thus we also examine LSHADE on larger instances with 200 and 500 decision variables. Here, LSHADE requires all 60 seconds of CPU time to find solutions far from the optimal. In contrast, DS requires on average 3.6 seconds for even the hardest benchmark in the table, the Rastrigin function with 500 dimensions. Given these encouraging results, we move on from conventional continuous, black-box approaches to compete on harder problems.

Comparison with LocalSolver. LocalSolver is a general purpose heuristic solver that supports essentially any white-box optimization problem. It offers an interface for problem modelling similar to those of mixed-integer programming solvers. We compare to LocalSolver[2] on randomly generated Ackley, Alpine, Griewank, Mixed, Periodic, Rastrigin, and Salomon functions.

In Tables 3 and 4 we present our results on the set of instances used to train the hyper-parameters of DS trained on each target problem individually, and

[2] We note that we are unable to tune LocalSolver's parameters with GGA due to LocalSolver's license restrictions, meaning our results should only be seen as a lower bound on performance.

Table 3. LocalSolver (LS) versus Dialectic Search (DS) on continuous and mixed continuous functions evaluated on our training set. DS's hyperparameters are trained specifically on the function it is being tested on, whereas DS-A is tuned on all functions.

	# ≤ 1e−3			Mean			Geo. Mean			Stdev.		
	LS	DS	DS-A	LS	DS	DS-A	LS	DS	DS-A	LS	DS	DS-A
Ackley	**97**	1	1	**0.001**	0.066	0.080	**0.001**	0.032	0.039	0.000	0.072	0.079
Alpine	41	**68**	**68**	0.002	0.003	**0.001**	0.002	**0.001**	**0.001**	0.001	0.009	0.001
Griewank	23	45	**85**	**0.002**	0.055	0.002	**0.001**	0.007	**0.001**	0.003	0.004	0.004
Mixed	7	7	7	6.823	**1.605**	1.611	2.108	**0.649**	0.650	6.167	1.176	1.185
Periodic	**4**	0	0	**0.934**	1.000	1.000	**0.699**	1.000	1.000	0.244	0.000	0.000
Rastrigin	45	**46**	41	73.712	**0.002**	**0.002**	0.160	**0.001**	0.002	143.144	0.001	0.001
Salomon	**84**	0	0	0.215	**0.100**	0.507	**0.003**	0.100	0.390	0.748	0.000	0.319
⌀ Total	**43**	23.86	28.86	11.67	**0.40**	0.46	**0.0203**	0.0283	0.0296	59.829	0.744	0.747

Table 4. LocalSolver (LS) versus Dialectic Search (DS) on continuous and mixed continuous functions evaluated on our test set. DS's hyperparameters are trained specifically on the function it is being tested on, whereas DS-A is tuned on all functions.

	# ≤ 1e−3			Mean			Geo. Mean			Stdev.		
	LS	DS	DS-A	LS	DS	DS-A	LS	DS	DS-A	LS	DS	DS-A
Ackley	**99**	3	0	**0.001**	0.070	0.085	**0.001**	0.036	0.044	0.000	0.084	0.089
Alpine	39	**62**	61	**0.002**	0.004	0.003	0.002	0.002	0.001	0.001	0.009	0.008
Griewank	20	**37**	21	**0.003**	0.063	0.255	**0.002**	0.011	0.030	0.008	0.003	0.402
Mixed	**12**	10	10	7.177	1.680	**1.679**	1.340	0.591	**0.475**	6.696	1.300	1.300
Periodic	**1**	0	0	**0.964**	1.000	1.000	**0.875**	1.000	1.000	0.178	0.000	0.000
Rastrigin	48	**51**	49	92.320	**0.002**	**0.002**	0.167	**0.002**	**0.002**	155.001	0.001	0.001
Salomon	**92**	0	0	**0.055**	0.100	0.462	**0.001**	0.100	0.368	0.408	0.00	0.276
⌀ Total	**44.43**	23.29	20.04	14.36	**0.26**	0.50	**0.0186**	0.0369	0.0464	66.764	0.790	0.784

DS-A, which is DS trained on a subset of training instances from all functions but Ackley. We also provide results on a test set of instances that were not used for the development or training of the new approach. For each benchmark function, we generate 200 instances, 100 for training and another 100 for testing. The dimensionality for each instance is chosen uniformly at random from the following ranges: $[250, 1000]$ (for both continuous and ordinal variables) for Ackley, $[1500, 5000]$ for Alpine, $[500, 5000]$ for Griewank, $[10, 50]$ for each variable type for Mixed, $[125, 1000]$ for Periodic, $[500, 5000]$ for Rastrigin and $[25, 200]$ for Salomon.

The tables give us multiple insights. First, we see that both DS (trained on each individual benchmark) and DS-A (trained on a mix of instances from all benchmarks, with the exception of Ackley instances) generalize well to the formerly unseen test instances. Second, we can see here that DS-A also does a reasonable job on the Ackley benchmark it was not trained for. In fact, it achieves almost the same performance as DS. Finally, we observe that there is

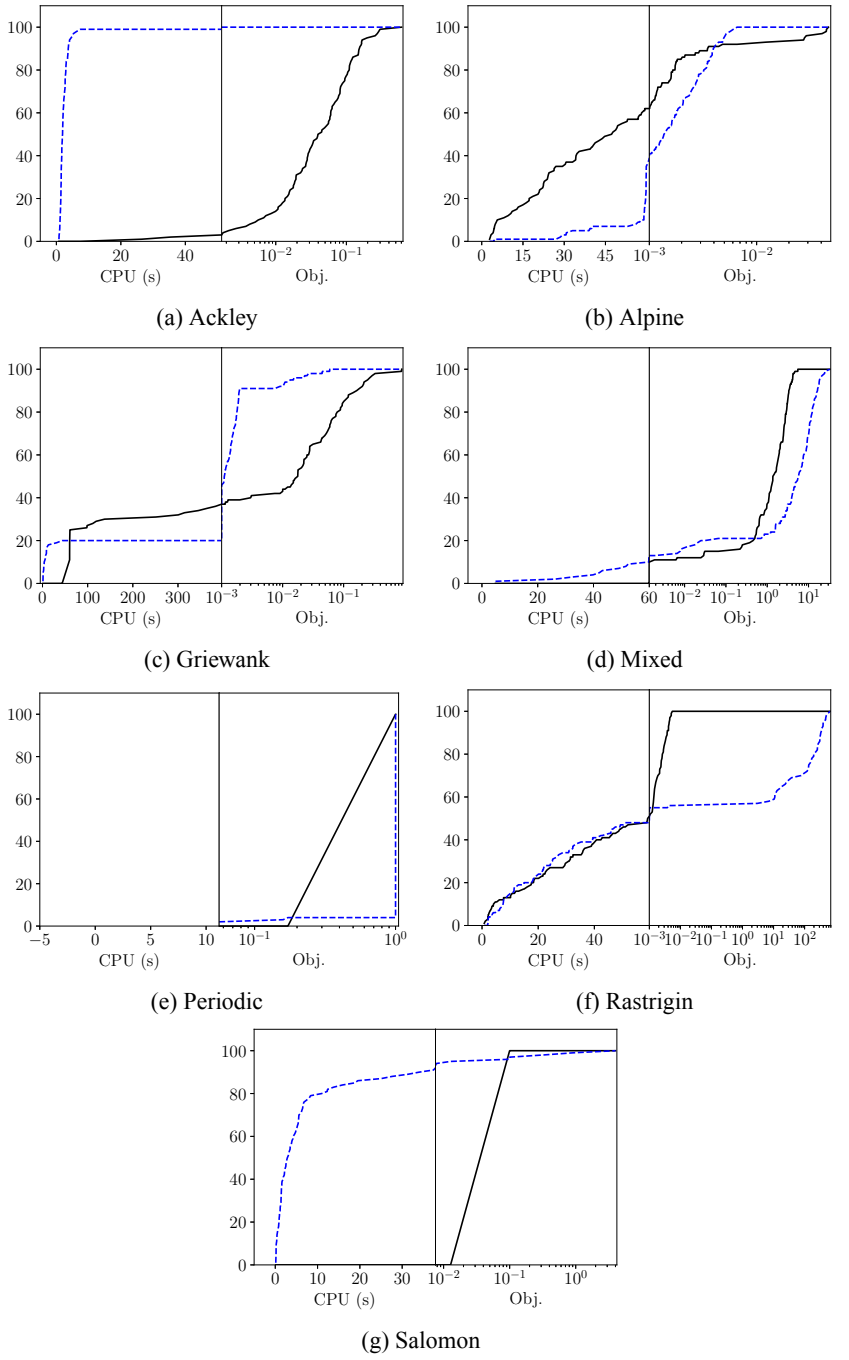


Fig. 2. Number of instances solved to 1e-3 in a given time limit (left) and the number of instances with an objective function better than the given value (right). LS is shown as a blue, dashed line and DS as a black, solid line.

a reasonable benefit of tuning DS for each individual benchmark. However, this is clearly not the main determining factor for DS' performance.

With respect to LocalSolver, we see that it solves more instances close to optimality than DS and DS-A. However, when looking at the mean gap over all instances, LS performs massively worse. We conclude that DS performs with less variance.

Figure 2 gives a detailed overview of how long it takes to solve instances on each function, and the quality of the solutions on those instances that aren't solved. Clearly, arguments can be made for both DS and LS depending on the function being solved and the amount of time available for solving. On Griewank and Alpine, DS is able to quickly get to 1e-3 on many instances, but has trouble on others, whereas the performance of LS remains mostly constant throughout all instances. In contrast, on the Rastrigin function, DS and LS solve roughly the same amount of instances, but the unsolved instances from DS are of significantly higher quality than those of LS.

5 Conclusion

We presented a novel dialectic search procedure for box-constrained, non-linear optimization problems with heterogeneous variable types. Our approach introduces a convex search procedure for *synthesizing* the thesis and antithesis of the search procedure, allowing it to highly effectively move through the search space. Moreover, we hyper-parameterized the resulting approach, allowing the meta-heuristics to adapt key search parameters during search based on runtime statistics characterizing the progress of the search. We compared our approach to three state-of-the-art procedures, the previous version of dialectic search, LSHADE and LocalSolver, and showed that the new dialectic search is able to compete, or even outperform these approaches, on occasion by very large margins.

Acknowledgements. The authors would like to thank the Paderborn Center for Parallel Computation (PC^2) for the use of the OCuLUS cluster.

References

1. Ansotegui, C., Malitsky, Y., Samulowitz, H., Sellmann, M., Tierney, K.: Model-based genetic algorithms for algorithm configuration. In: IJCAI, pp. 733–739 (2015)
2. Ansotegui, C., Sellmann, M., Tierney, K.: A gender-based genetic algorithm for the automatic configuration of algorithms. In: CP, pp. 142–157 (2009)
3. Ansótegui, C., Heymann, B., Pon, J., Sellmann, M., Tierney, K.: Hyper-reactive tabu search for MaxSAT. In: Battiti, R., Brunato, M., Kotsireas, I., Pardalos, P.M. (eds.) LION 12 2018. LNCS, vol. 11353, pp. 309–325. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-05348-2_27
4. Ansótegui, C., Pon, J., Sellmann, M., Tierney, K.: Reactive dialectic search portfolios for MaxSAT. In: AAAI Conference on Artificial Intelligence (2017)

5. Argelich, J., Li, C., Manyà, F., Planes, J.: MaxSAT Evaluation (2016). www.maxsat.udl.cat
6. Benoist, T., Estellon, B., Gardi, F., Megel, R., Nouioua, K.: 4or. Localsolver 1. x: a black-box local-search solver for 0–1 **9**(3), 299 (2011)
7. Glover, F., Laguna, M., Marti, R.: Fundamentals of scatter search and path relinking. Control Cybern. **39**, 653–684 (2000)
8. Kadioglu, S., Sellmann, M.: Dialectic search. In: CP, pp. 486–500 (2009)
9. Lourenço, H., Martin, O., Stützle, T.: Iterated local search. In: Glover, F., Kochenberger, G.A. (eds.) Handbook of Metaheuristics, vol. 57, pp. 320–353. Springer, Boston (2003). https://doi.org/10.1007/0-306-48056-5_11
10. SIMANN: Fortran simulated annealing code (2004)
11. Tanabe, R., Fukunaga, A.S.: Improving the search performance of shade using linear population size reduction. In: 2014 IEEE Congress on Evolutionary Computation (CEC), pp. 1658–1665. IEEE (2014)