Benchmarking discrete optimization heuristics: from building a sound experimental environment to algorithm configuration

Ye, F.

Chapter 5

Benchmarking Algorithms on IOHprofiler Problems

IOHprofiler provides a benchmark suite of pseudo-Boolean optimization, which allows us to investigate the performance of algorithms on a wide range of problems. In this chapter, we compare the performance of twelve different heuristics on the first twenty-three PBO problems to show how to apply IOHprofiler for such a benchmarking study. Moreover, we investigate how (or whether) crossover can be beneficial for the genetic algorithm by testing the $(\mu + \lambda)$ GA with different parameter settings on the PBO problems.

5.1 Benchmarking Heuristics

5.1.1 Background

As the discussion on the design of IOHprofiler, our goal is to make the platform as flexible as possible so that the user can easily test their algorithms on the problems and with respect to performance criteria of their choice (see Chapter 3 for the discussion). However, the original framework only provided the experimental setup, but did not fix any benchmark problems or reference algorithms. In this chapter, we present the results of 12 different heuristics for the original 23 PBO problems, which serves as the first baseline for the performance evaluation of user-defined heuristics. All performance data is available in our data repository and can be straightforwardly assessed through the web-based version of IOHanalyzer (http://iohprofiler.liacs.nl/), which is easily accessible for future comparative studies. An important by-product of our contribution is the identification of additional statistics, which are included within the IOHprofiler. This section provides extensive examples of assessing algorithms' performance over a set of problems concerning different perspectives (such as fixed-target result, fixed-budget result, and ECDF.)

5.1.2 Summary of Baseline Algorithms

High-level Description

We evaluate a total number of twelve different algorithms on the first 23 problems described in Section 2.5. We have chosen algorithms that may serve for future references, since they all have some known strengths and weaknesses that will become apparent in the following discussions. The selection therefore shows a clear bias towards algorithms for which theoretical analyses are available.

Note that most algorithms are parametrized, and we use here in this work only standard parametrizations (e.g., we use standard bit mutation with $1/n$ as mutation rates, etc.). Analyzing the effects of different parameter values as was done, for example in [32, 133], would be very interesting, related work on parameter tuning will be introduced in Section 6.

We also note that, except for the so-called vGA, our implementations (deliberately) deviate slightly from the text-book descriptions referenced below. Following the discussion in previous chapters, we enforce that offspring created by mutation are different from their parent and resample without further evaluation if needed. Likewise, we do not evaluate recombination offspring that are identical to one of their immediate parents.

All algorithms start with uniformly chosen initial solution candidates.

We list here the twelve implemented algorithms, and provide further details and pseudo-codes:

1. **gHC:** A (1+1) greedy hill climber, which goes through the string from left to right, flipping exactly one bit per each iteration, and accepting the offspring if it is at least as good as its parent.

2. **RLS:** Randomized Local Search, the elitist (1+1) strategy flipping one uniformly chosen bit in each iteration. That is, RLS and gHC differ only in the choice of the bit which is flipped. While RLS is unbiased in the sense of Section 2.5.1, gHC is not permutation-invariant and thus biased.

3. $(1+1)$ **EA:** The $(1+1)$ EA with static mutation rate $p = 1/n$. This algorithm differs from RLS in that the number of uniformly chosen, pairwise different bits to be flipped is sampled from the conditional binomial distribution $\text{Bin}_{>0}(n,p)$. Details refer to Algorithm 2 in Section 4.1.

4. **fGA:** The "fast GA" proposed in [50] with $\beta = 1.5$. Its *mutation strength* (i.e., the number of bits flipped in each iteration) follows a power-law distribution with exponent β. This results in a more frequent use of large mutation-strength, while maintaining the property that small mutation strengths are still sampled with reasonably large probability.

5. $(1+10)$ **EA:** The $(1+10)$ EA with static $p = 1/n$, which differs from the $(1+1)$ EA only in that 10 offspring are sampled (independently) per each iteration.

6. $(1+10)$ $\textbf{EA}_{r/2,2r}$**:** The two-rate EA with self-adjusting mutation rates suggested and analyzed in [46] (see Algorithm 3 in Section 4.2).

7. $(1+10)$ $\textbf{EA}_{\text{norm.}}$**:** a variant of the $(1+10)$ EA sampling the mutation strength from a normal distribution $N(pn, pn(1-p))$ with a self-adjusting choice of p (see Algorithm 5 in Section 4.2).

8. $(1+10)$ $\textbf{EA}_{\text{var.}}$**:** The $(1+10)$ $\text{EA}_{\text{norm.}}$ with an adaptive choice of the variance in the normal distribution from which the mutation strengths are sampled (see Algorithm 6 in Section 4.2).

9. $(1+10)$ $\textbf{EA}_{\text{log-n.}}$ The $(1+10)$ EA with log-normal self-adaptation of the mutation rate proposed in [7].

10. $(1+(\lambda,\lambda))$ **GA:** A binary (i.e., crossover-based) EA originally suggested in [43]. We use the variant with self-adjusting λ analyzed in [41].

11. **vGA:** A $(30,30)$ "vanilla" GA (following the so-called traditional GA, as described, for example, in [69, 5]).

12. **UMDA:** A univariate marginal distribution algorithm from the family of estimation of distribution algorithms (EDAs). UMDA was originally proposed in [120].

Detailed Description of the Algorithms

Detailed description of $(1+1)$ EA, $(1+10)$ EA, $(1+10)$ $\text{EA}_{r/2,2r}$, $(1+10)$ $\text{EA}_{\text{norm.}}$, and $(1+10)$ $\text{EA}_{\text{var.}}$ can be found in Section 4, and descriptions of the remaining algorithms follow. An operator frequently used in these descriptions is the $\text{flip}_\ell(\cdot)$ mutation operator, which flips the entries of ℓ pairwise different, uniformly at random chosen bit positions. Details can be found in Algorithm 1.

Again, to avoid useless evaluations of offspring that are identical to their parents, we make use of the conditional binomial distribution $\text{Bin}_{>0}(n, p)$, which assigns probability $\text{Bin}(n, p)(k)/(1-(1-p)^n)$ to each positive integer $k \in [n]$, and probability zero to all other values. Sampling from $\text{Bin}_{>0}(n, p)$ is identical to sampling from $\text{Bin}(n, p)$ until a positive value is returned ("resampling strategy").

Greedy Hill Climber The greedy hill climber (gHC, Algorithm 9) uses a deterministic mutation strength, and flips one bit in each iteration, going through the bit string from left to right, until being stuck in a local optimum, see Algorithm 9.

Algorithm 9: Greedy hill climber (gHC)

```
Initialization: Sample $x \in \{0,1\}^n$ uniformly at random and evaluate $f(x)$;
Optimization: for $t = 1, 2, 3, \dots$ do
    $x^* \leftarrow x$;
    Flip in $x^*$ the entry in position $1 + (t \mod n)$ and evaluate $f(x^*)$;
    if $f(x^*) \geq f(x)$ then $x \leftarrow x^*$;
```

Randomized Local Search RLS uses a deterministic mutation strength, and flips one randomly chosen bit in each iteration, see Algorithm 10.

Algorithm 10: Randomized local search (RLS)

```
Initialization: Sample $x \in \{0,1\}^n$ uniformly at random and evaluate $f(x)$;
Optimization: for $t = 1, 2, 3, \dots$ do
    create $x^* \leftarrow \text{flip}_1(x)$, and evaluate $f(x^*)$;
    if $f(x^*) \geq f(x)$ then $x \leftarrow x^*$;
```

Fast Genetic Algorithm The *fast Genetic Algorithm* (fGA) chooses the mutation length ℓ according to a power-law distribution $D^\beta_{n/2}$, which assigns to each integer $k \in [n/2]$ a probability of $\Pr[D^\beta_{n/2} = k] = \left(C^\beta_{n/2}\right)^{-1} k^{-\beta}$, where $C^\beta_{n/2} = \sum_{i=1}^{n/2} i^{-\beta}$. We use the (1+1) variant of this algorithm with $\beta = 1.5$.

Algorithm 11: Fast genetic algorithm (fGA) from [50]

```
Initialization: Sample $x \in \{0,1\}^n$ uniformly at random and evaluate $f(x)$;
Optimization: for $t = 1, 2, 3, \dots$ do
    for $i = 1, \dots, \lambda$ do
        Sample $\ell^{(i)} \sim D^{\beta}_{n/2}$;
        create $y^{(i)} \leftarrow \text{flip}_{\ell^{(i)}}(x)$, and evaluate $f(y^{(i)})$;
    $x^* \leftarrow \arg\max\{f(y^{(1)}), \dots, f(y^{(\lambda)})\}$ (ties broken by favoring the largest
      index);
    if $f(x^*) \geq f(x)$ then $x \leftarrow x^*$;
```

The EA with log-Normal self-adaptation of mutation rate The $(1+\lambda)$ $\text{EA}_{\text{log-n.}}$, Algorithm 12, uses a self-adaptive choice of the mutation rate.

Algorithm 12: The $(1+\lambda)$ $\text{EA}_{\text{log-n.}}$ with log-Normal self-adaptation of the mutation rate

```
Initialization: Sample $x \in \{0,1\}^n$ uniformly at random and evaluate $f(x)$;
$p = 0.2$;
Optimization: for $t = 1, 2, 3, \dots$ do
    for $i = 1, \dots, \lambda$ do
        $p^{(i)} = \left(1 + \frac{1-p}{p} \cdot \exp(0.22 \cdot \mathcal{N}(0,1))\right)^{-1}$ ;
        Sample $\ell^{(i)} \sim \text{Bin}_{>0}(n, p^{(i)})$;
        create $y^{(i)} \leftarrow \text{flip}_{\ell^{(i)}}(x)$, and evaluate $f(y^{(i)})$;
    $i \leftarrow \min\left\{j \mid f(y^{(j)}) = \max\{f(y^{(k)}) \mid k \in [\lambda]\}\right\}$;
    $p \leftarrow p^{(i)}$;
    $x^* \leftarrow \arg\max\{f(y^{(1)}), \dots, f(y^{(\lambda)})\}$ (ties broken by favoring the smallest
      index);
    if $f(x^*) \geq f(x)$ then $x \leftarrow x^*$;
```

The Self-Adjusting $(1+(\lambda,\lambda))$ GA The self-adjusting $(1+(\lambda,\lambda))$ GA, Algorithm 13, was introduced in [43] and analyzed in [41]. The offspring population size λ is updated after each iteration, depending on whether or not an improving offspring could be generated. Since both the mutation rate and the crossover bias (see Algorithm 14 for the definition of the biased crossover operator cross) depend on λ, these two parameters also change during the run of the $(1+(\lambda,\lambda))$ GA. In our implementation we use update strength $F = 3/2$.

Algorithm 13: The self-adjusting $(1 + (\lambda, \lambda))$ GA

```
Initialization: Sample $x \in \{0,1\}^n$ uniformly at random and evaluate $f(x)$;
Optimization: for $t = 1, 2, 3, \dots$ do
    Mutation phase:
        Sample $\ell \sim \mathrm{Bin}_{>0}(n, \lambda/n)$;
        for $i = 1, \dots, \lambda$ do  create $y^{(i)} \leftarrow \mathrm{flip}_\ell(x)$, and evaluate $f(y^{(i)})$;
        $x^* \leftarrow \arg\max\{f(y^{(1)}), \dots, f(y^{(\lambda)})\}$ (ties broken by favoring the largest
          index);
    Crossover phase:
        for $i = 1, \dots, \lambda$ do  create $y^{(i)} \leftarrow \mathrm{cross}_c(x, x^*)$, and evaluate $f(y^{(i)})$;
        $y^* \leftarrow \arg\max\{f(y^{(1)}), \dots, f(y^{(\lambda)})\}$ (ties broken by favoring the largest
          index);
    Selection phase:
        if $f(y^*) > f(x)$ then $x \leftarrow y^*$; $\lambda \leftarrow \max\{\lambda/F, 1\}$;
        if $f(y^*) = f(x)$ then $x \leftarrow y^*$; $\lambda \leftarrow \min\{\lambda F^{1/4}, n\}$;
        if $f(y^*) < f(x)$ then $\lambda \leftarrow \min\{\lambda F^{1/4}, n\}$;
```

Algorithm 14: Crossover operation $\mathrm{cross}_c(x, x^*)$ with crossover bias c

```
$y \leftarrow x$;
Sample $\ell \sim \mathrm{Bin}_{>0}(n, c)$;
Select $\ell$ different positions $\{i_1, \dots, i_l\} \in [n]$;
for $j = 1, 2, \dots, \ell$ do  $y_{i_j} \leftarrow x^*_{i_j}$;
```

The "Vanilla" GA The vanilla GA (vGA, Algorithm 16) constitutes a textbook realization of the so-called Traditional GA [5, 69]. The algorithm holds a parental population of size μ. It employs the Roulette-Wheel-Selection (RWS, that is, probabilistic fitness-proportionate selection which permits an individual to appear multiple times) as the sexual selection operator to form $\mu/2$ pairs of individuals that generate the offspring population. 1-point crossover (Algorithm 15) is applied to every pair with a fixed probability of $p_c = 0.37$. A mutation operator is then applied to every individual, flipping every bit with a fixed probability of $p_m = 2/n$. This completes a single cycle.

Algorithm 15: 1-Point crossover of two parents $x^{(1)}$ and $x^{(2)}$

```
Sample $\ell \in [n]$ uniformly at random;
for $i = 1, 2, \ldots, \ell$ do Set $y_i^{(1)} \leftarrow x_i^{(1)}$ and $y_i^{(2)} \leftarrow x_i^{(2)}$;
for $i = \ell + 1, \ldots, n$ do Set $y_i^{(1)} \leftarrow x_i^{(2)}$ and $y_i^{(2)} \leftarrow x_i^{(1)}$;
```

Algorithm 16: The (μ, μ)-"Vanilla-GA" with mutation rate p_m and crossover probability p_c

```
Initialization:
    for $i = 1, \ldots, \mu$ do sample $x^{(i)} \in \{0,1\}^n$ uniformly at random and
     evaluate $f(x^{(i)})$;
Optimization:   for $t = 1, 2, 3, \ldots$ do
    Parent selection phase: Apply roulette-wheel selection to $\{x^{(1)}, \ldots, x^{\mu}\}$
     to select $\mu$ parent individuals $y^{(1)}, \ldots, y^{(\mu)}$;
    Crossover phase:
        for $i = 1, \ldots, \mu/2$ do with probability $p_c$ replace $y^{(i)}$ and $y^{(2i)}$ by the
         two offspring that result from a 1-point crossover of these two
         parents, for a randomly chosen crossover point $j \in [n]$;
    Mutation phase:
        for $i = 1, \ldots, \mu$ do Sample $\ell^{(i)} \sim \mathrm{Bin}(n, p_m)$, set $y^{(i)} \leftarrow \mathrm{flip}_{\ell^{(i)}}(y^{(i)})$,
         and evaluate $f(y^{(i)})$;
    Replacement:
        for $i = 1, \ldots, \mu$ do Replace $x^{(i)}$ by $y^{(i)}$;
```

The Univariate Marginal Distribution Algorithm The univariate marginal distribution algorithm (UMDA, Algorithm 17) is one of the simplest representatives of the family of so-called estimation of distribution algorithms (EDAs). The algorithm maintains a population of size s (we use $s = 50$ in our experiments) and uses the best $s/2$ of these to estimate the marginal distribution of each decision variable, by simply counting the relative frequency of ones in the corresponding position. These frequencies are capped at $1/n$ and $1 - 1/n$, respectively. In the t-th iteration, a new population is created by sampling from these marginal distributions. Building upon previous work made in [119], the UMDA was introduced in [120]. Theoretical results for this algorithm are summarized in [100].

Algorithm 17: The Univariate Marginal Distribution Algorithm (UMDA), representing the family of EDAs

```
Initialization:
    for $i = 1, \ldots, s$ do sample $x^{(0,i)} \in \{0,1\}^n$ uniformly at random and evaluate $f(x^{(0,i)})$;
    Let $\boldsymbol{P_0}$ be the collection of the best $s/2$ of these search points, ties broken uniformly at random (u.a.r.);
Optimization:
    for $t = 1, 2, 3, \ldots$ do
        for $j = 1, \ldots, n$ do
            $p_j \leftarrow 2|\{x \in \boldsymbol{P_{t-1}} \mid x_j = 1\}|/s$;
            if $p_j < 1/n$ then $p_j = 1/n$;
            if $p_j > 1 - 1/n$ then $p_j = 1 - 1/n$;
        for $i = 1, \ldots, s$ do  sample $x^{(t,i)} \in \{0,1\}^n$ by setting, independently for all $j \in [n]$, $x_j^{(t,i)} = 1$ with probability $p_j$ and setting $x_j^{(t,i)} = 0$ otherwise. Evaluate $f(x^{(t,i)})$;
        Let $\boldsymbol{P_t}$ be the collection of the best $s/2$ of the points $x^{(t,1)}, x^{(t,2)}, ..., x^{(t,s)}$, ties broken u.a.r.;
```

5.1.3 Experimental Results

Experimental Setup

Our experimental setup can be summarized as follows:

- 23 test-functions F1-F23, described in Section 2.5
- Each function is assessed over the four problem dimensions $n \in \{16, 64, 100, 625\}$
- Each algorithm is run on 11 different instances of each of these 92 (F, n) pairs, yielding a total number of 1,012 different runs per each algorithm. Each run is granted a budget of $100n^2$ function evaluations for dimensions $n \in \{16, 64, 100\}$ and a budget of $5n^2$ function evaluations for $n = 625$. More precisely, each algorithm performs one run on each of the instances $1-6$ and $51-55$ described in Section 2.5.1.

  Most of the tested algorithms are unbiased and comparison-based. For these algorithms all 11 instances look the same, i.e., performing one run each is equivalent to 11 independent runs on instance 1, which is the “pure” problem instance without fitness scaling nor any other transformation applied to it. However, in order to understand how the transformations impact the behavior of vGA and

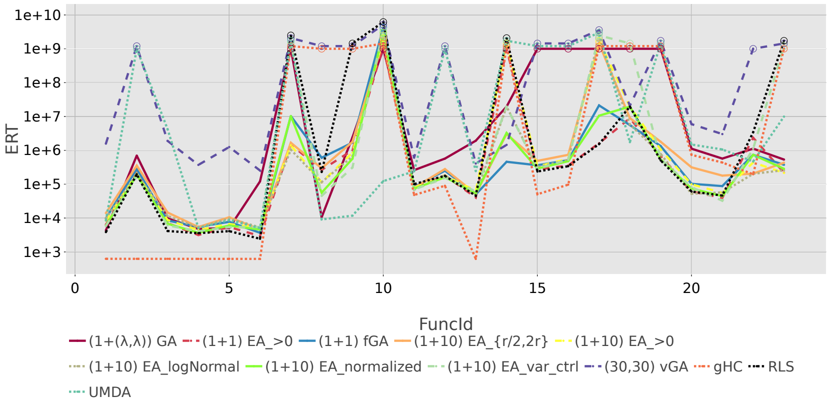

Figure 5.1: ERT values of the twelve baseline algorithms for the 625-dimensional test suite, with respect to the best solution quality found by any of the algorithms in any of the eleven runs. These target values can be found in Table 5.2.

gHC, we also performed 11 independent runs of each algorithm on instance 1 of each (F, n) pair, yielding another $1,012$ runs per each algorithm.

- For each run we store the current and the best-so-far function value at each evaluation. This setup allows very detailed analyses, since we can zoom into each range of fixed budgets and/or fixed-targets of choice, and obtain our anytime performance statistics in terms of quantiles, averages, probabilities of success, ECDF curves, etc.

  For some of the algorithms we also store information about the self-adjusting parameters, for example the value of λ in the $(1 + (\lambda, \lambda))$ GA and the mutation rates for the (1+10) EA$_{r/2,2r}$, the (1+10) EA$_{\text{var.}}$, and the (1+10) EA$_{\text{norm.}}$. From this data we can derive how the parameters evolve with respect to the time elapsed and with respect to the quality of the best-so-far solutions.

Concerning the number of repetitions, we note that with 11 runs we already get a good understanding of the key differences between the algorithms. 11 runs can be enough to get statistical significance, if the differences in performance are substantial. We refer the interested reader to the tutorial [76], which argues that for a first experiment a small number of experiments can suffice.

Function-wise Raw Observations Across Dimensions Figures 5.1 and 5.2 depict the ERT of the baseline algorithms on the 625-dimensional and the 64-dimensional

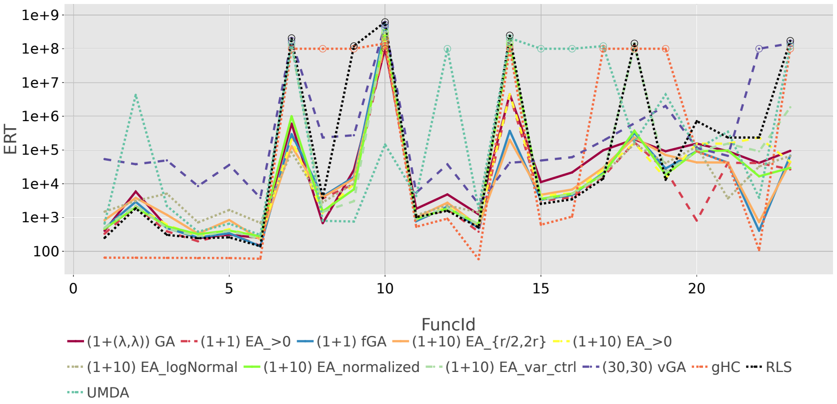

Figure 5.2: ERT values of the twelve baseline algorithms for the 64-dimensional test suite, with respect to the best solution quality found by any of the algorithms in any of the eleven runs. These target values can be found in Table 5.1.

funcId	$F1$	$F2$	$F3$	$F4$	$F5$	$F6$	$F7$	$F8$	$F9$	$F10$	$F11$	$F12$
$n = 64$	64	64	2, 080	32	57	21	64	33	64	63.2	32	57
$n = 625$	625	625	195, 625	312	562	208	576.4	314	625	625	312	562

funcId	$F13$	$F14$	$F15$	$F16$	$F17$	$F18$	$F19$	$F20$	$F21$	$F22$	$F23$
$n = 64$	21	43.8	33	64	64	3.981492	64	128	192	28	8
$n = 625$	208	36.6	314	625	625	4.2655266	621	1, 200	1, 775	268.4	24

Table 5.1: Target values for which the ERT curves in Figures 5.1 and 5.2 are computed.

functions, respectively, when considering the best function value found by any of the algorithms in any of the runs. These target values are summarized in Table 5.1.

We summarize a few basic observations for each function.

F1: This baseline OneMax problem is easily solved, having the gHC winning (it solves each n-dimensional OneMax instance in at most $n + 1$ queries), the majority of the algorithms clustered with a practically-equivalent performance, the (1+10)-EA$_{r/2,2r}$ lagging behind, and the vGA outperformed by far. All algorithms locate the global optimum eventually. Figure 5.3 presents the average fixed-target performance of the algorithms on F1 at $n = 625$, in terms of ERT. Evidently, the vGA and the UMDA obtain a clear advantage in the beginning of the optimization process, although the vGA eventually uses the largest number of evaluations, by far, to locate the optimum. We also see here that, as expected, the performances of the unbiased algorithms (i.e., all algorithms except the vGA) are identical for the 11 runs on instance 1 and the 1 run on 11 different instances. For the vGA this is clearly not the case, the fixed-target performances of these two settings

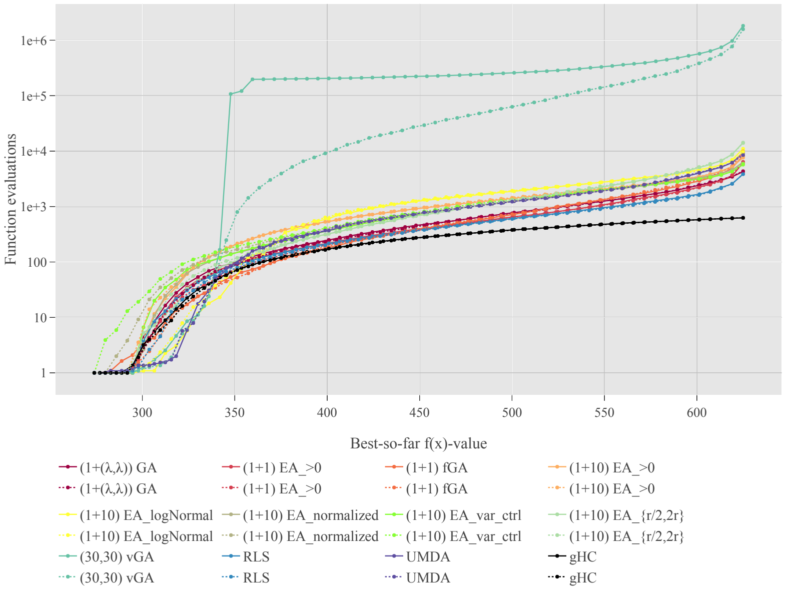

Figure 5.3: ERT values for F1 (OneMax) at dimension $n = 625$ in a fixed-target perspective. The dashed lines are the average running times of 11 independent runs on instance 1, while the solid lines are average running times for one run on each of the eleven different instances 1-6 and 51-55. Note that this figure is not generated by IOHanalyzer.

differ substantially.

F2: The LeadingOnes problem introduces more difficulty when compared to F1, with the ERT consistently shifting upward, but it is still easily solved. The gHC wins, the vGA loses, and the majority of the algorithms are again clustered, but now the $(1+(\lambda,\lambda))$-GA lags behind. The UMDA fails to find the optimum within the given time budget, for all tested dimensions except for $n = 16$. An example of the evolution of the parameter λ in the $(1+(\lambda,\lambda))$ GA is visualized in Figure 5.4. We observe – as expected – that larger function values are evidently correlated with larger population sizes (and, thus, larger mutation rates).

F3: The behavior on this problem, the linear function with harmonic weights, is similar to F1 for most algorithms. Exceptions are the vGA, for which it is slightly easier, and the UMDA, which shows worse performance on F3 than on F1.

F4: This problem, OneMax with 50% dummy variables, is the most easily-solved

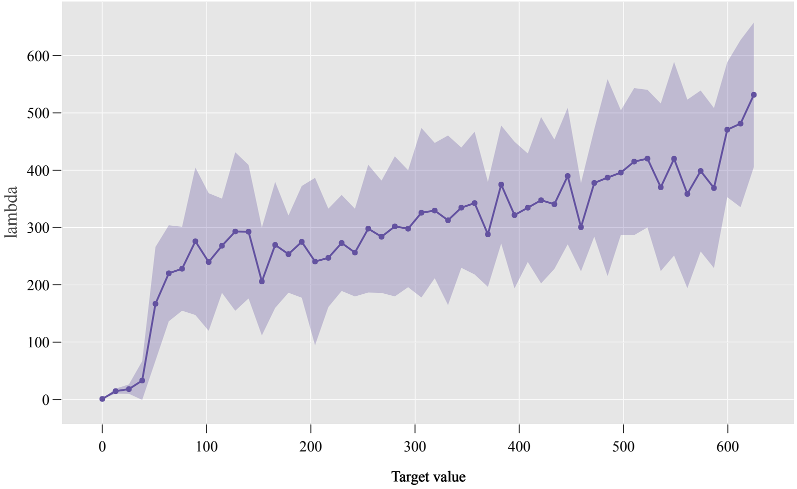

Figure 5.4: Evolution of the population size λ of the $(1+(\lambda,\lambda))$ GA on the LeadingOnes problem F2 at dimension $n = 625$, correlated to the best-so-far objective function values (horizontal axis). The line shows the average value of λ for iterations starting with a best-so-far solution of the value indicated by the x-axis. The shade represents the standard deviation.

problem in the suite, with an even simpler performance classification – the gHC performs at the top, the vGA at the bottom, and the rest are tightly clustered. Given the consistent correlation with the F1 performance profiles, across all twelve algorithms, it seems debatable whether or not to keep this function in a benchmark suite, since it seems to offer only limited additional insights, which could be of a rather specialized interest, e.g., for theoretical investigations addressing precise running times of the algorithms.

F5: Solving this problem, OneMax with 10% dummy variables, exhibits equivalent behavior to F1. Similarly to F4, we suggest to ignore this setup for future benchmarking activities. Note, however, that the exclusion of F4 and F5 does *not* imply that the dummy variables do not play an interesting role – in an ongoing evaluation of the W-model, we are currently investigating their impact when combined with other W-model transformations.

F6: The neutrality (“majority vote”) transformation apparently introduces difficulty to the $(1+(\lambda,\lambda))$ GA, which exhibits deteriorated performance compared to F1. The vGA, despite a slightly better performance compared to F1, is the worst among the twelve algorithms. At the same time, the (1+10)-EA$_{\text{log-n.}}$ lags behind its competitors in the beginning, but it eventually shows a competitive result in the later optimization process, ending up with an overall fine ERT value. The gHC outperforms all other algorithms also on this function.

F7: The introduction of local permutations to OneMax, within the current problem, introduced difficulties to all the algorithms. The ability to locate the global optimum within the designated budget deteriorated for all of them, except for the (1+10)-EA$_{r/2,2r}$ on "low-dimensional" scales ($n \in \{16, 64, 100\}$). Figure 5.5 depicts the ERT values of the algorithms on F7 at $n = 625$, where it is evident that they all failed to locate the global optimum. Note that this figure encompasses results for both instantiations (a single instance or 11 instances). The twelve algorithms' performances are clustered in two groups that are associated with two fitness regions (and likely two basins of attraction) - the first around an objective function value of 500 (including the UMDA, with the gHC being the fastest to approach it and get stuck), and the other below 600. It seems that the latter cluster could use additional budget to further improve the results.

F8: Being OneMax with the small fitness plateaus induced by the ruggedness function r_1, the UMDA performs best on this problem, with the $(1 + (\lambda, \lambda))$ GA following very closely. It seems to introduce medium difficulty to all the algorithms, except for the gHC, whose performance is dramatically hampered and becomes worse than the vGA. Interestingly, the ERT values are distributed sparsely compared to other OneMax variants.

F9: The UMDA also performs best for this problem, with the (1+10) EA$_{\text{var.}}$ being the runner-up. Generally, the behavior on this problem, OneMax with small fitness perturbations, is close to F8, but with certain differences. F9 is evidently harder, as the algorithms meet larger ERT values. Importantly, unlike F8, RLS always fails on F9 (since it gets stuck in local optima), and "joins" the gHC and vGA at the bottom of the performance table. The $(1 + (\lambda, \lambda))$ GA also shows worse performance on F9 than on F8.

F10: This problem, OneMax with fitness perturbations of block size five, presents a dramatic difficulty to all the algorithms, including the UMDA, which, however, clearly outperforms all other algorithms. It is evidently the hardest OneMax variant for all the tested algorithms, among the eight variants studied in this work. For $n = 625$ the UMDA finds the optimum after an average of $141,243$ evaluations, while none of the other algorithms finds a solution better than 575.

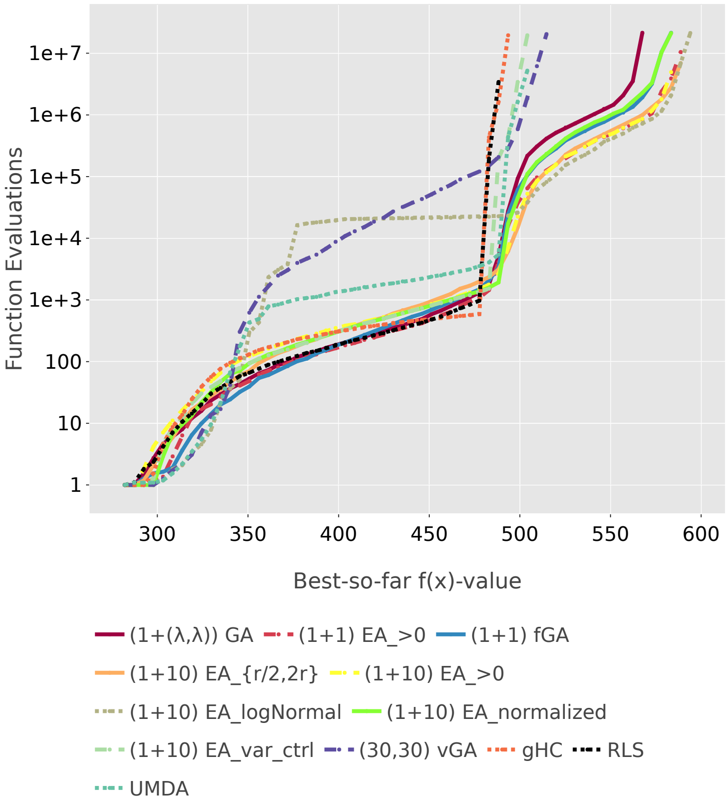

Figure 5.5: ERT values for F7 at dimension $n = 625$ in a fixed-target perspective.

F11: The gHC performs strongly on this problem, namely the LeadingOnes with 50% dummy variables, consistently with its winning behavior on F2. The vGA performs poorly, and the UMDA is also at the bottom of the table. Notably, the problem should become easier compared to F2, since the effective number of variables is reduced. RLS, however, which generally performs well on LeadingOnes, only ranks third from the bottom on ERT values when solving this problem.

F12: The behavior of the algorithms on this problem, LeadingOnes with 10% dummy variables, is very similar to F11, with excellent performance of the gHC. However, one major difference is the dramatic deterioration of UMDA and vGA, which fail to find the optimum with given time budget for $n = 625$ (see Figure 5.1). UMDA performs better than vGA for $n = 64$, but still obtains clear disadvantage comparing to other algorithms (see Figure 5.2).

F13: The introduction of neutrality on LeadingOnes makes this problem easier in practice (that is, by observing ERT decrease compared to F2). The gHC wins, while the vGA, $(1 + (\lambda, \lambda))$ GA as well as the UMDA lag behind the other methods. The poor performance of these three algorithms is consistent with their performance on LeadingOnes.

F14: Being LeadingOnes with epistasis, this problem introduces high difficulty. For high dimensions, $n \in \{64, 100, 625\}$, none of the algorithms was capable of locating an optimal solution within the allocated budget. The vGA tops the ERT values on this problem, followed by the (1+1)-fGA, the (1+10)-EA$_{r/2,2r}$, and the (1+10)-EA$_{\text{norm.}}$. On the other hand, three algorithms, namely the gHC, the UMDA, and the RLS, seem to get trapped with low objective function values.

F15: The introduction of fitness perturbations to LeadingOnes makes this problem difficult. The UMDA exhibits the worst performance among the competing methods. The remainder of the algorithms, except for the vGA and the $(1+(\lambda, \lambda))$ GA, are still able to hit the optimum of this problem, but with significantly larger ERT values. The gHC performs best, and the first runner-up is the RLS.

F16: The obtained ranks of algorithms, with respect to the ERT values, are similar to those of F15, but generally exhibit higher ERT values. Notably, the UMDA is still the worst performer.

F17: As expected, the rugged LeadingOnes function is the second-hardest among the LeadingOnes variants, following F14. Only the RLS and the (1+1)-EA are able to hit the optimum in dimension 625, while the gHC has a diminished performance on this problem. This can be explained by the fact that the gHC has a very high probability of getting stuck in a local traps, while the RLS is capable of performing random walks about local optima, until eventually escaping them (e.g., by flipping the right bit when all the consecutive four bits are also identical to the target string). This is of course a rare event, and the ERT values are therefore significantly worse than all other LeadingOnes variants, except F14. As on the previous two functions, the UMDA performs poorly, with similar ERT values as the gHC.

Comparing to F10, the effect of the fitness permutation r_3 on LeadingOnes is not as significant as on OneMax, which can be explained by the ability of most of the algorithms to perform random walks to deal with local traps, through which the four first bits of the tail are eventually set correctly, at which point flipping the significant bit (i.e., the bit in position $\mathrm{LO}(x)+1$) results in a LeadingOnes fitness increase of at least five, and consequently a fitness increase of at least one for the problem $r_3 \circ$ LeadingOnes. This candidate solution is thus accepted by all of our algorithms, and the next phase of optimizing the following consecutive five bits begins.

F18: The LABS problem is the most complex problem in our assessment. For the higher dimensions, $n \in \{64, 100, 625\}$, none of the algorithms obtained the maximally attainable values, or got fitness values close to those of the best-known sequences (see, e.g., [125]). Additionally, a couple of algorithms (e.g., the gHC and the RLS) did not succeed to escape low-quality "local traps" on most dimensions. Surprisingly, the vGA was superior to the other algorithms at $n = 16$ but, as expected, over the higher dimensions presented weaker performance. Notably, the UMDA outperforms the other methods at $n = 625$.

F19: The simplest problem among the Ising instances. Most of the algorithms exhibited similar performance, except for the vGA, the UMDA and the gHC, which obtained weak results. The latter preformed worst among all algorithms, and obtained the lowest objective function values across all the dimensions for the given time budget. As a demonstration of the performance statistics that IOHprofiler provides, average fixed-target and fixed-budget running times are provided in Figure 5.6. This figure illustrates that ERT values tell only one side of the story: the performance of UMDA is comparable to that of the other algorithms for all targets up to around 85; only then it starts to perform considerably worse.

F20: In contrast to its poor performance on the 1D-Ising (F19), the gHC outperformed the other algorithms on the 2D-Ising for target values up to around $1,136$ ($d = 625$), after which its performance becomes worse than most of the other algorithms, except for the vGA, which is consistently the worst except for a few initial target values.For $d = 625$, however, the (1+10) $\mathrm{EA}_{\text{log-n.}}$ achieves the best ERT value for the target recorded in Table 5.2, followed by the (1+1) EA and RLS.

F21: As expected, the most complex among the Ising model instances. The observed performances resemble the observations on F20. For $d = 625$, the best ERT is obtained by the (1+10)-$\mathrm{EA}_{\text{var.}}$.

F22: None of the algorithms succeeded in locating the global optimum across all dimensions of this problem. This is explained by the existence of a local optimum with a strong basin of attraction. The gHC and the vGA exhibited inferior performance compared to the other algorithms.

F23: Some algorithms failed to locate the global optimum of the N-Queens problem in high dimensions, yet the vGA, the gHC and the UMDA constantly possessed the worst ERT values. Fine performance was observed for the (1+10)-EA$_{>0}$ and the (1+10) EA$_{\text{log-n.}}$.

Grouping of Functions and Algorithms In the following we are aiming to recognize patterns and identify classes within (i) the set of all functions, and (ii) the set of all algorithms.

Functions' Empirical Grouping: It is evident that problems F1-F6, F11-F13 and F15-F16 are treated relatively easily by the majority of the algorithms, with those functions based on LeadingOnes (i.e., F2, F11-13, F15, F16) being more challenging within this group. On the other extreme, F7, F9-F10, F14, F18-F19 and F22 evidently constitute a class of hard problems, on which all algorithms consistently exhibit difficulties (except for $n = 16$); the LABS function (F18) seems the most difficult among them. F8, the instances of the Ising model (F19-F21), as well as the NQP (F23), constitute a class of moderate level of difficulty.

Algorithms' Observed Trends: The gHC and the vGA usually exhibited extreme performance with respect to the other algorithms. The vGA consistently suffers from poor performance over all functions, while the gHC either leads the performance on certain functions or performs very poorly on others. The gHC's behavior is to be expected, since it is correlated with the existence of local traps (by construction) – for instance, it consistently performs very well on F1-F6, while having difficulties on F7-F10. Clearly, RLS also gets trapped by the deceptive functions, while at the same time it shows fine performance on most of the non-deceptive problems. The UMDA's performance stands out. Evidently, it performs well on the OneMax-based problems, but fails to optimize the LeadingOnes function F2 and its derivatives F11-F17, with the exception of F11 and F13 – a behavior that might be interesting to analyze further in future work. Otherwise, we observe one primary class of algorithms exhibiting equivalent performance over all problems in all dimensions: The seven algorithms $(1 + (\lambda, \lambda))$-GA, (1+1)-EA, (1+10)-EA$_{\text{var.}}$, (1+10)-EA, (1+10)-EA$_{\text{norm.}}$, (1+10)-EA$_{r/2,2r}$, and (1+1)-fGA behave consistently, typically exhibiting fine performance. In terms of ERT values, the (1+10)-EA$_{\text{log-n.}}$ could also be grouped into this class of seven algorithms, but it behaves quite differently during the optimization process, often showing an opposite trend of convergence speed at the early stages of the

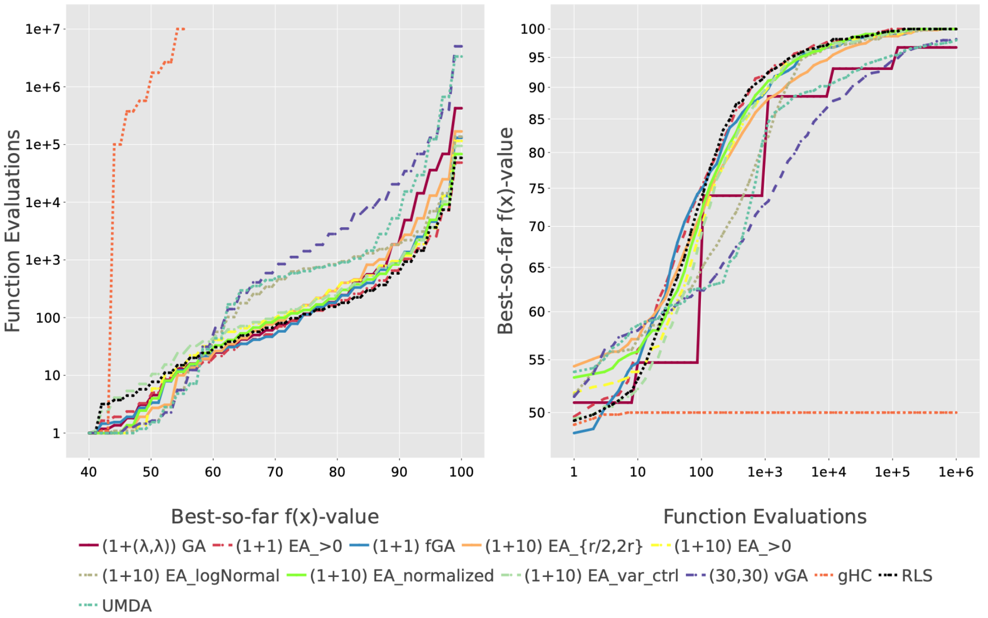

Figure 5.6: Demonstration of the basic performance plots for F19 at dimension $n = 100$: **Left:** best obtained values as a function of evaluations calls ("fixed-target perspective"), versus **Right:** evaluations calls as a function of best obtained values ("fixed-budget perspective"). For F19, these patterns of relative behavior are observed across all dimensions.

optimization procedure.

Ranking: We also examined the overall number of runs per test-function in which an algorithm successfully located the best recorded value – the so-called *hitting number*. We then grouped those hitting numbers by dimension, and ranked the algorithms per each dimension. The (1+10)-EA$_{r/2,2r}$ consistently leads the grouped hitting numbers on the "low-dimensional" functions ($n \in \{16,\ 64\}$), with (1+1)-fGA and (1+10)-EA$_{\text{norm.}}$ being together the first runner-up. The (1+10)-EA also exhibits high ranking across all dimensions. (1+10)-EA$_{\text{norm.}}$ leads the grouped hitting numbers on $n = 100$, whereas the (1+1)-EA leads the hitting numbers on the "high-dimensional" functions at $n = 625$, with (1+10)-EA being the runner-up. Across all dimensions, UMDA, gHC and vGA are with the lowest rankings.

Visual Analytics: As a demonstration of the performance statistics offered by IOHprofiler, we provide snapshots of visual analytics that supported our examination. Figure 5.6 depicts basic performance plots for F19 at dimension $n = 100$, in

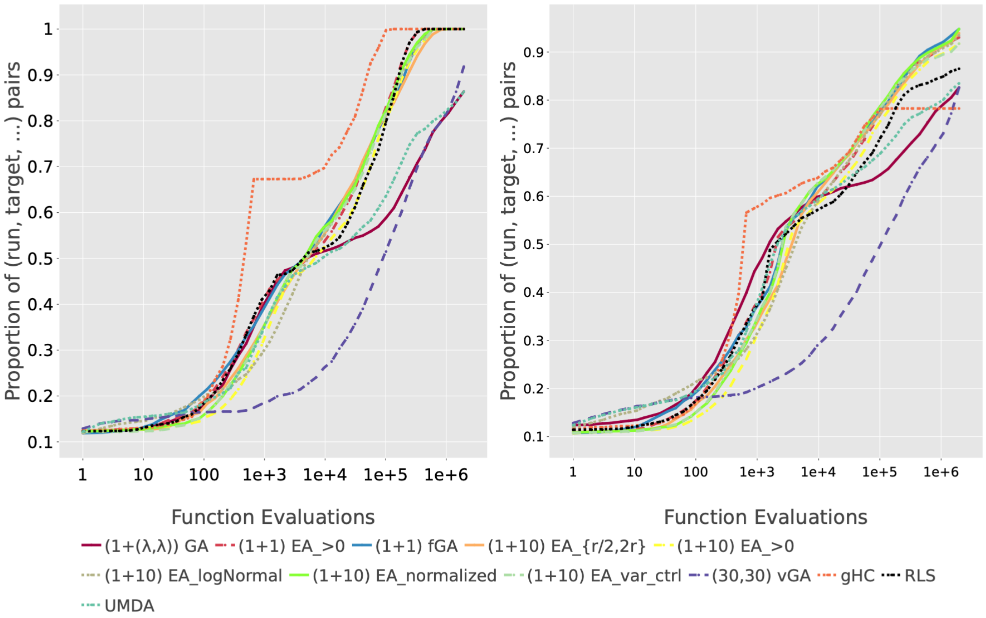

Figure 5.7: ECDF curve for the class of “easily-solved” functions in dimension $n = 625$: F1-F6, F11-F13, and F15-F16 [LEFT] and of all 23 functions [RIGHT], with respect to equally spaced target values.

so-called *fixed-target* and *fixed-budget* perspectives. For clarity of the plots we only show the ERT values and the average function values achieved per each budget, respectively. Standard deviations as well as the $2, 5, 10, 15, 50, 75, 90, 95, 98\%$ quantiles are available on http://iohprofiler.liacs.nl/.

In Figure 5.7 we provide two plots obtained from our new module which computes ECDF curves for user-specified target values. The plot on the left depicts an ECDF curve for the “easily-solved” functions identified above (i.e., F1-F6, F11-F13, and F15-F16) in dimension $n = 625$. The one on the right shows the ECDF curves across all 23 benchmark functions. For both figures we have chosen ten equally spaced target values per each function, with the largest value being again the best function value identified by any of the algorithms in any of the runs. Since the number of “easy” problems dominates our overall assessment the curves on the right are to a large extent dominated by the performances depicted on the left. This indicates once again the need for a thorough revision of our benchmark selection.

Unbiasedness: Following our experimental planning to test the hypothesized

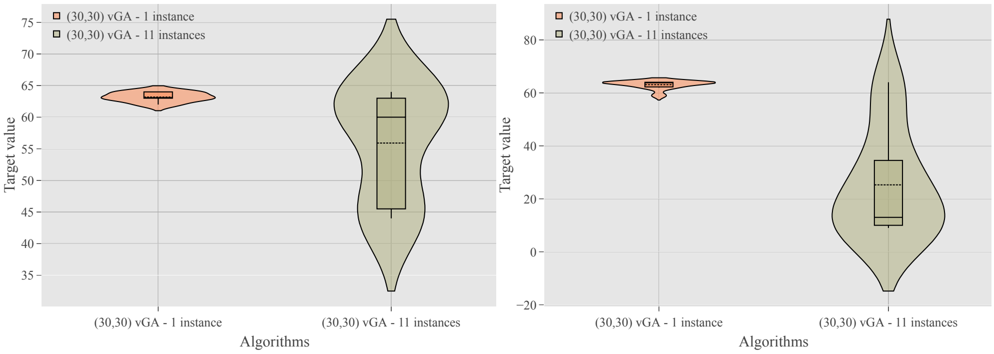

Figure 5.8: Statistical box-plots for vGA's attained function values on instance one alone versus on the eleven different instances 1-6 and 51-55, after exploiting the entire budget (namely, $409,600$ function evaluations): F1 [LEFT] and F2 [RIGHT]. Both plots are for $n = 64$.

"biasedness" effect for the vGA, we compared its averaged performance on instance 1 versus on all the other instances (1-6 and 51-55) altogether. Figure 5.8 depicts a comparison of attained objective function values, by means of box-plots, on F1 and on F2 for $n = 64$. Performance deterioration is indeed evident on the permuted instances; that is, instances 51-55, for which the base functions are composed with a σ-transformation of the bit strings, as described in Section 2.5.1. The box-plots in Figure 5.8 show very clearly that the vGA treats the plain F1 and F2 much better, in terms of attained target values, than their transformed variants. The plots are for $n = 64$ and after exhausting the full budget of $100n^2$ function evaluations.

5.1.4 Summary

This section presented results of the 12 heuristics on the first 23 PBO problems, which contributes a baseline for future comparative studies. This work has inspired many directions for IOHprofiler, and some of them are already under development.

Additional Performance Measures: While this section presents a very detailed assessment of algorithms' performance, we are continuously strengthening the statistical repertoire of IOHanalyzer by introducing new performance measures and by devising better procedures. Currently, IOHanalyzer also supports pairwise Kolmogorov-Smirnov test, Glicko2-based ranking, the Deep Statistical Comparison (DSC) analysis [59], and Contribution to portfolio (Shapley-values).

Combinations of W-model Transformations: As discussed in Section 2.5.6, the transformations of the W-model can be combined with each other. To analyze the individual effects of each transformation, and in order to keep the size of the experimental setup reasonable, we have not considered such combinations in this work. A critical consideration of adding such combinations, and of extending the base transformations (e.g., with respect to the fitness transformation, but also the size of the neutrality transformation, etc.) forms another research line that we are currently addressing in a parallel work stream.

Integration of Algorithm Design Software: IOHs are to a large extent modular algorithms, whose components can be exchanged and executed in various different ways. This has letted the community to develop software which enables an easier algorithmic design. Examples for such software are ParadisEO [24] for single-objective and multi-objective optimization and jMetal [58] for multi-objective algorithms. Building or integrating such software could allow much more comprehensive algorithm benchmarking, and could eventually automate the detection of promising algorithmic variants. We are glad to see that IOHprofiler has contributed to this domain, for example, by being integrated with other frameworks for the work of large scale automated algorithm design [3, 153, 168, 170].

5.2 Benchmarking a $(\mu + \lambda)$ Genetic Algorithm with Configurable Crossover Probability

5.2.1 Background

Classic evolutionary computation methods build on two main variation operators: *mutation* and *crossover*. While the former can be mathematically defined as unary operators (i.e., families of probability distributions that depend on a single argument), crossover operators sample from distributions of higher arity, with the goal to "recombine" information from two or more arguments.

There is a long debate in evolutionary computation about the (dis-)advantages of these operators, and about how they interplay with each other [118, 140]. In lack of generally accepted recommendations, the use of these operators still remains a rather subjective decision, which in practice is mostly driven by users' experience. Little guidance is available on which operator(s) to use for which situation, and how to most efficiently interleave them. The question how crossover can be useful can therefore be seen as far from being solved.

Of course, significant research efforts are spent to shed light on this question, which is one of the most fundamental ones that evolutionary computation has to offer. While in the early years of evolutionary computation (see, for example, the classic works [5, 34, 69]) crossover seems to have been widely accepted as an integral part of an evolutionary algorithm, we observe today two diverging trends. Local search algorithms such as GSAT [136] for solving Boolean satisfiability problems, or such as the general-purpose Simulated Annealing [96] heuristic, are clearly very popular optimization methods in practice – both in academic and in industrial applications. These purely mutation-based heuristics are nowadays more commonly studied under the term *stochastic local search*, which forms a very active area of research. Opposed to this is a trend to reduce the use of mutation operators, and to fully base the iterative optimization procedure on recombination operators; see [152] and references therein. However, despite the different recommendations, these opposing positions find their roots in the same problem: we hardly know how to successfully dovetail mutation and crossover.

In addition to large bodies of empirical works aiming to identify useful combinations of crossover and mutation [34, 63, 85, 121], the question how (or whether) crossover can be beneficial has also always been one of the most prominent problems in *runtime analysis*, the research stream aiming at studying evolutionary algorithms by mathematical means [25, 29, 30, 31, 43, 47, 48, 88, 89, 98, 107, 123, 141, 143, 157, 162], most of these results focus on very particular algorithms or problems, and are not (or at least not easily) generalizable to more complex optimization tasks.

In this section, we study a simple variant of the $(\mu+\lambda)$ GA mentioned in Section 4.3, which allows us to conveniently scale the relevance of crossover and mutation, respectively, via a single parameter. More precisely, our algorithm is parameterized by a crossover probability p_c, which is the probability that we generate in the reproduction step an offspring by means of crossover. The offspring is generated by mutation otherwise, so that $p_c = 0$ corresponds to the mutation-only $(\mu + \lambda)$ EA, whereas for $p_c = 1$ the algorithm is entirely based on crossover. Note here that we *either* use crossover *or* mutation, so as to better separate the influence of the two operators.

We study the performance of different configurations of the $(\mu + \lambda)$ GA on 25 IOHprofiler problems. We observe that the algorithms using crossover perform significantly better on some simple functions as OneMax (F1) and LeadingOnes (F2), but also on some problems that are considered hard, e.g., the 1-D Ising model (F19).

$F1$	$F2$	$F3$	$F4$	$F5$	$F6$	$F7$	$F8$	$F9$	$F10$	$F11$	$F12$	$F13$	$F14$	$F15$	$F16$	$F17$	$F18$	$F19$	$F20$	$F21$	$F22$	$F23$	$F24$	$F25$
100	100	$5,050$	50	90	33	100	51	100	100	50	90	33	7	51	100	100	4.215852	98	180	260	42	9	17.196	-0.2965711

Table 5.2: Target values used for computing the ERT value in Figure 5.9.

5.2.2 Experimental Results

Experiment setup In order to probe into the empirical performance of the $(\mu + \lambda)$ GA, we test it on the 25 problems mentioned in Section 2.5, with a total budget of $100n^2$ function evaluations. We perform 100 independent runs of each algorithm on each problem. For the $(\mu+\lambda)$ GA (see Algorithm 8 in Section 4.3), we study three different crossover operators, *one-point crossover*, *two-point crossover*, and *uniform crossover*, and two different mutation operators, *standard bit mutation* and the *fast mutation* scheme suggested in [50]. These variation operators are briefly described as follows.

- *One-point crossover*: a crossover point is chosen from $[1..n]$ u.a.r. and an offspring is created by copying the bits from one parent until the crossover point and then copying from the other parent for the remaining positions.

- *Two-point crossover*: similarly, two different crossover points are chosen u.a.r. and the copy process alternates between two parents at each crossover point.

- *Uniform crossover* creates an offspring by copying for each position from the first or from the second parent, chosen independently and u.a.r.

- *Standard bit mutation:* a mutation strength ℓ is sampled from the conditional binomial distribution $\text{Bin}_{>0}(n, p)$, which assigns to each k a probability of $\binom{n}{k}p^k(1-p)^{n-k}/(1-(1-p)^n)$ [25]. Thereafter, ℓ distinct positions are chosen u.a.r. and the offspring is created by first copying the parent and then flipping the bits in these ℓ positions. Still, we restrict our experiments to the standard mutation rate $p = 1/n$.

- *Fast mutation* [50]: operates similarly to standard bit mutation except that the mutation strength ℓ is drawn from a power-law distribution: $\Pr[L = \ell] = (C^{\beta}_{n/2})^{-1}\ell^{-\beta}$ with $\beta = 1.5$ and $C^{\beta}_{n/2} = \sum_{i=1}^{n/2} i^{-\beta}$.

Moreover, we test the algorithm with $\mu \in \{10, 50, 100\}$ and $\lambda \in \{1, \lceil \mu/2 \rceil, \mu\}$. Detailed results for the different configurations of the $(\mu + \lambda)$ GA are available in our data repository at [171].

Results on IOHprofiler problems In Figure 5.9, we highlight a few basic results of this experimentation for $n = 100$, where the mutation operator is fixed to the

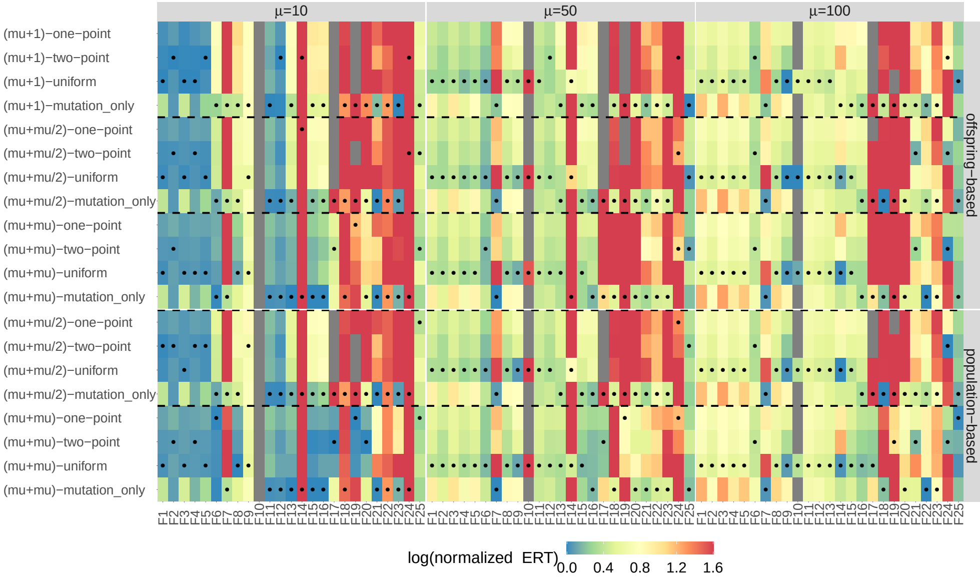

Figure 5.9: Heat map of normalized ERT values of the $(\mu+\lambda)$ GA with offspring-based (top part) and population-based (bottom part) variator choice for the 100-dimensional benchmark problems, computed based on the target values specified in Table 5.2. The crossover probability p_c is set to 0.5 for all algorithms except the mutation-only ones (which use $p_c = 0$). The displayed values are the quotient of the ERT and ERT_{best}, the ERT achieved by the best of all displayed algorithms. These quotients are capped at 40 to increase interpretability of the color gradient in the most interesting region. The three algorithm groups – the $(\mu + 1)$, the $(\mu + \lceil\mu/2\rceil)$, and the $(\mu + \mu)$ GAs – are separated by dashed lines. A dot indicates the best algorithm of each group of four. A grey tile indicates that the $(\mu + \lambda)$ GA configuration failed, in all runs, to find the target value within the given budget.

standard bit mutation. More precisely, we plot in this figure the normalized expected running time (ERT), where the normalization is with respect to the best ERT achieved by any of the algorithms for the same problem. Table 5.2 provides the target values for which we computed the ERT values. For each problem and each algorithm, we first calculated the 2% percentile of the best function values. We then selected the largest of these percentiles (over all algorithms) as target value.

On the OneMax-based problems F1, F4, and F5, the $(\mu+\lambda)$ GA outperforms the mutation-only GA, regardless of the variator choice scheme, the crossover operator, and the setting of λ. When looking at problem F6, we find that when $\mu = 10$ the mutation-only GA surpasses most of $(\mu+\lambda)$ GA variants except the population-based $(\mu+\mu)$ GA with one-point crossover. On F8-10, the $(\mu+\lambda)$ GA takes the lead in general,

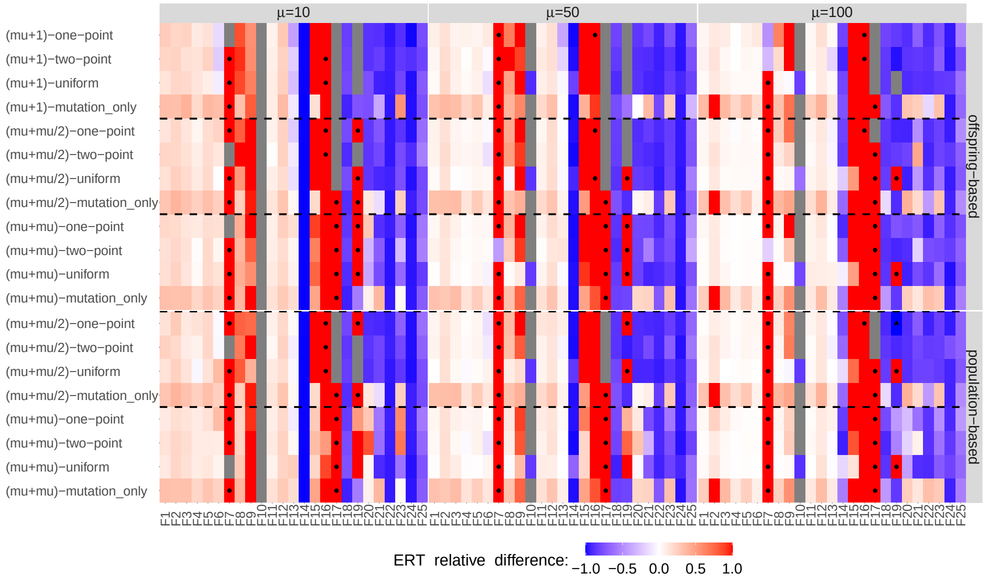

Figure 5.10: Heat map comparing the $(\mu+\lambda)$ GAs using the standard bit mutation (sbm) with the $(\mu+\lambda)$ GAs using the fast mutation on the 25 problems from Sec. 2.5 in dimensions $n = 100$. Plotted values are $(\text{ERT}_{\text{fast}} - \text{ERT}_{\text{sbm}})/\text{ERT}_{\text{sbm}}$, for ERTs computed wrt the target values specified in Table 5.2. p_c is set to 0.5 for all crossover-based algorithms. Values are bounded in $[-1, 1]$ to increase visibility of the color gradient in the most interesting region. A black dot indicates that the $(\mu+\lambda)$ GA with fast mutation failed in all runs to find the target with the given budget; the black triangle signals failure of standard bit mutation, and a gray tile is chosen for settings in which the $(\mu+\lambda)$ GA failed for both mutation operators.

whereas it cannot rival the mutation-only GA on F7. Also, only the configuration with uniform crossover can hit the optimum of F10 within the given budget.

On the linear function F3 we observe a similar behavior as on OneMax. On LeadingOnes (F2), the $(\mu+\lambda)$ GA outperforms the mutation-only GA again for $\mu \in \{50, 100\}$ while for $\mu = 10$ the mutation-only GA becomes superior with one-point and uniform crossovers. On F11-13 and F15-16 (the W-model extensions of LeadingOnes), the mutation-only GA shows a better performance than the $(\mu+\lambda)$ GA with one-point and uniform crossovers and this advantage becomes more significant when $\mu = 10$. On problem F14, that is created from LeadingOnes using the same transformation as in F7, the mutation-only GA is inferior to the $(\mu+\lambda)$ GA with uniform crossover.

On problems F18 and F23, the mutation-only GA outperforms the $(\mu+\lambda)$ GA for most parameter settings. On F21, the $(\mu + \lambda)$ GA with two-point crossover yields a better result when the population size is larger (i.e., $\mu = 100$) while the mutation-only GA takes the lead for $\mu = 10$. On problems F19 and F20, the $(\mu + \mu)$ GA with the population-based variator choice significantly outperforms all other algorithms, whereas it is substantially worse for the other parameter settings. On problem F24, the $(\mu+\mu/2)$ GA with two-point crossover achieves the best ERT value when $\mu = 100$. None of the tested algorithms manages to solve F24 with the given budget. The target value used in Figure 5.9 is 17.196, which is below the optimum 20. On problem F25, the mutation-only GA and the $(\mu + \lambda)$ GA are fairly comparable when $\mu \in \{10, 50\}$. Also, we observe that the population-based $(\mu+\mu)$ GA outperforms the mutation-only GA when $\mu = 100$.

In general, we have made the following observations: (1) on problems F1-6, F8-9, and F11-13, all algorithms obtain better ERT values with $\mu = 10$. On problems F7, F14, and F21-25, the $(\mu + \lambda)$ GA benefits from larger population sizes, i.e., $\mu = 100$; (2) The $(\mu + \mu)$ GA with uniform crossover and the mutation-only GA outperform the $(\mu + \lceil\mu/2\rceil)$ GA across all three settings of μ on most of the problems, except F10, F14, F18, and F22. For the population-based variator choice scheme, increasing λ from one to μ improves the performance remarkably on problems F17-24. Such an improvement becomes negligible for the offspring-based scheme; (3) Among all three crossover operators, the uniform crossover often surpasses the other two on OneMax, LeadingOnes, and the W-model extensions thereof.

To investigate the impact of mutation operators on GA, we plot in Figure 5.10 the relative ERT difference between the $(\mu+\lambda)$ GA configurations using fast and standard bit mutation, respectively. As expected, fast mutation performs slightly worse on F1-6, F8, and F11-13. On problems F7, F9, and F15-17, however, fast mutation becomes detrimental to the ERT value for most parameter settings. On problems F10, F14, F18, and F21-25, fast mutation outperforms standard bit mutation, suggesting a potential benefit of pairing the fast mutation with crossover operators to solve more difficult problems. Interestingly, with an increasing μ, the relative ERT of the $(\mu + \lambda)$ GA quickly shrinks to zero, most notably on F1-6, F8, F9, F11-13.

Interestingly, in [117], an empirical study has shown that on a randomly generated maximum flow test generation problem, fast mutation is significantly outperformed by standard bit mutation when combined with uniform crossover. Such an observation seems contrary to our findings on F10, F14, F18, and F21-25. However, it is made on a standard $(100 + 70)$ GA in which both crossover and mutation are applied to the parent in order to generate offspring.

5.2.3 Summary

In this section, we have analyzed the performance of a family of $(\mu + \lambda)$ GAs, in which offspring are either generated by crossover (with probability p_c) or by mutation (probability $1 - p_c$). On the PBO problem set, it has been shown that this random choice mechanism reduces the expecting running time on OneMax, LeadingOnes, and many W-model extensions of those two problems.

It would certainly also be interesting to extend the study to a $(\mu + \lambda)$ GA variant using (dynamic) tuned values for the relevant parameters μ, λ, crossover probability p_c, and mutation rate p. Therefore, based on the results in this section, we will introduce our work on *algorithm configuration* in Chapter 6 and *dynamic algorithm selection* in Chapter 7.