Efficient Noisy Optimisation with the Multi-Sample and Sliding Window Compact Genetic Algorithms

Simon M. Lucas
Queen Mary University of London
London E1 4NS, United Kingdom
simon.lucas@qmul.ac.uk

Jialin Liu
Queen Mary University of London
London E1 4NS, United Kingdom
jialin.liu@qmul.ac.uk

Diego Pérez-Liébana
University of Essex
Colchester CO4 3SQ, United Kingdom
dperez@essex.ac.uk

***Abstract*—The compact genetic algorithm is an Estimation of Distribution Algorithm for binary optimisation problems. Unlike the standard Genetic Algorithm, no cross-over or mutation is involved. Instead, the compact Genetic Algorithm uses a virtual population represented as a probability distribution over the set of binary strings. At each optimisation iteration, exactly two individuals are generated by sampling from the distribution, and compared exactly once to determine a winner and a loser. The probability distribution is then adjusted to increase the likelihood of generating individuals similar to the winner.**

This paper introduces two straightforward variations of the compact Genetic Algorithm, each of which leads to a significant improvement in performance. The main idea is to make better use of each fitness evaluation, by ensuring that each evaluated individual is used in multiple win/loss comparisons. The first variation is to sample $n > 2$ individuals at each iteration to make $n(n-1)/2$ comparisons. The second variation only samples one individual at each iteration but keeps a sliding history window of previous individuals to compare with. We evaluate the methods on two noisy test problems and show that in each case they significantly outperform the compact Genetic Algorithm, while maintaining the simplicity of the algorithm.

***Index Terms*—Compact genetic algorithm, evolutionary algorithm, estimation of distribution, sliding window, binary optimisation, discrete optimisation.**

I. Introduction

In many optimisation applications there is a need to deal with noisy evaluation functions, and also to make best possible use of a limited evaluation budget. Traditional, population-based evolutionary algorithms most commonly cope with noise by re-evaluating each evaluated individual several times and/or increasing the size of the population. For a recent survey refer to Rakshit et al [1].

The motivation behind our work is to develop powerful optimisation algorithms that are well-suited to applications in Game AI. The applications include rolling-horizon planning algorithms used to control non-player characters or bots, and to automatic game design or automatic game tuning. Both of these applications have many forms of uncertainty which introduce significant noise into the fitness evaluation function. Furthermore, they each operate under a limited time budget, so there is a strong need for algorithms that make the best possible use of a limited number of fitness (objective function) evaluations.

The algorithms developed in this paper are already showing promise at the initial testing phase on exactly these types of problems, including General Video Game AI [2], [3], but in this paper we focus on describing the algorithms and providing results on some simple benchmark problems.

The rest of this paper is structured as follows. The next section gives a very brief overview of the most relevant background work. Section III describes two variations of the compact Genetic Algorithm (cGA): the Multi-Sample version and the Sliding Window version. Both improve significantly on the standard cGA, with the sliding window version (a novel algorithm) providing the best results in our experiments. Section IV describes the test problems used and Section V presents the results. Section VI concludes and also discusses ongoing and future work.

II. Background

The cGA [4] is an Estimation of Distribution Algorithm (EDA) for binary optimisation problems. In contrast to the standard Genetic Algorithm (GA) [5], no cross-over or mutation is involved. Instead, the cGA uses a virtual population represented as a probability distribution over the set of binary strings. At each optimisation iteration, exactly two individuals are sampled from the distribution and evaluated for fitness to pick a winner and loser. The probability distribution is then adjusted to increase the probability of generating the winning vector. The iterations continue until either the evaluation budget has been exhausted or the distribution has converged (such that the probability of producing a ‘1’ in each position is either zero or one).

The cGA performs well in noisy conditions, as clearly shown in Friedrich et al [6]. The main contribution of this paper is to develop two new versions of the cGA that make even more efficient use of the available evaluation budget. Both algorithms are efficient and easy to implement.

The standard cGA is a type of Univariate Estimation of Distribution Algorithm, since each dimension is considered independently of all others. Harik emphasises that the algorithm can be sensitive to the probability model used to model the virtual population and that "the choice of a good distribution is equivalent to linkage learning" [7]. Regarding this, the extended cGA (ECGA) using a set of probability models known as Marginal Product Models (MPMs) is proposed [7]. An MPM can represent a probability distribution over one bit or a tuple of bits, taking into account the dependency among the bits, and hence modelling higher-order effects. A similar idea has been used in a so-called N-Tuple system by Kunanusont et al. [8].

The multiple sample variations of the cGA introduced in this paper should also work with higher-order probability models, but this has not yet been implemented or tested.

A. Compact Genetic Algorithm (cGA)

The standard cGA[4] models a population using a d-dimensional vector of probabilities, with an element for each bit of the solution. The algorithm has one parameter k which refers to the virtual population size and determines the learning rate, $\frac{1}{k}$.

Each element of the vector is initialised to 0.5 and represents the probability that each corresponding value in the solution string should be a 1. At each iteration two random binary vectors are produced by sampling from the probability vector, and the fitness of each one is evaluated. The fitness values are compared to determine a winner and loser and the probability distribution is updated: no update occurs if the two vectors have the same value. The algorithm then iterates over each dimension, comparing each candidate bit-by-bit. Updates to the probability vector only occur when the corresponding bits in the winner and loser differ. If the winning bit is 1, then the probability of producing a 1 in that position is increased by $\frac{1}{k}$, otherwise (i.e. if the winning bit is a 0) the probability is decreased by $\frac{1}{k}$.

The algorithm terminates when the probability vector has converged. We also stop the algorithm when the total evaluation budget has been consumed. The solution found by the algorithm is the argmax of the probability vector, i.e. a 0 whenever the corresponding probability is less than 0.5 and a 1 otherwise.

Note that this is a rank-based algorithm, in that the magnitude of the difference in fitness between winner and loser makes no difference.

The single parameter, virtual populations size k, has an important effect on the algorithm's behaviour. Setting k too low (and hence the learning rate too high) causes premature convergence of the probability vector, and results in very poor final solutions. Setting k too high causes slower than necessary convergence but does not harm solution quality so much. Friedrich et al [6] show how to set k optimally when the noise model is Gaussian-distributed and the noise variance is known. For this paper we compare a range of k values to the one used in [6].

III. Multiple-Sample cGA Variants

This section describes the novel contribution of the paper. The motivation for the new algorithm is to make better

Algorithm 1 Multiple-Sample compact Genetic Algorithm (MScGA) for a d-bit binary problem. Note that the standard cGA is the special case of MScGA with sample number $n = 2$.

```
Require: problem dimension $d$
Require: fitness function $fitness(\cdot) : \{0,1\}^d \mapsto \mathbb{R}$
Require: virtual population size $k$
Require: sample number $n$

function MScGA($d, k, n$)
    Initialise $p \leftarrow \{\frac{1}{2}\}^d$
    while ! isConverged($p$) do
        for $i \in \{1, \ldots, n\}$ do
            $x_i, y_i \leftarrow$ reproduceAndEvaluate($p$)
        $j_1, j_2, \ldots, j_n \leftarrow$ competeNSort($y_1, y_2, \ldots, y_n$)
        for $a \in \{1, \ldots, n-1\}$ do
            for $b \in \{a+1, \ldots, n\}$ do
                $x_{win} \leftarrow x_{j_a}$
                $x_{loss} \leftarrow x_{j_b}$
                $p \leftarrow$ update($p, x_{win}, x_{loss}$)
    $x^* \leftarrow$ recommend($p$)
    return $x^*$

function reproduceAndEvaluate($p$)
    $x \leftarrow \{0\}^{|p|}$
    for $i \in \{1, \ldots, d\}$ do
        if $rand < p(i)$ then
            $x(i) = 1$
    $y \leftarrow fitness(x)$
    return $x, y$

function competeNSort($y_1, y_2, \ldots, y_n$)
    Define $j_1, j_2, \ldots, j_n$ such that $y_{j_1} \geq y_{j_2} \geq \cdots \geq y_{j_n}$
    return $j_1, j_2, \ldots, j_n$

function update($p, x_{win}, x_{loss}$)
    for $i \in \{1, \ldots, d\}$ do
        if $x_{win}(i) \neq x_{loss}(i)$ then
            if $x_{win}(i) == 1$ then
                $p(i) = p(i) + \frac{1}{k}$
            else
                $p(i) = p(i) - \frac{1}{k}$
    return $p$

function recommend($p$)
    $x \leftarrow \{0\}^{|p|}$
    for $i \in \{1, \ldots, d\}$ do
        if $p(i) > \frac{1}{2}$ then
            $x(i) = 1$
    return $x$

function isConverged($p$)
    for $i \in \{1, \ldots, d\}$ do
        if $0 < p(i) < 1$ then
            return $false$
    return $true$
```

use of the fitness evaluations in order to find optimal or close to optimal solutions more quickly. Observe that in the standard cGA, at each iteration we draw two samples from the distribution and make one comparison and one update of the probability vector. This gives us an update per sample ratio of $1/2$. The question arises as to whether we can make more efficient use of the samples we evaluate (keeping in mind that for most practical applications, the main cost of an evolutionary algorithm is in performing fitness evaluations).

This observation raises the question as to whether we may make better use of the fitness evaluations if we make more comparisons and updates for each one. This leads us on to the following two algorithms. The first is a natural extension of the cGA to increase the number of individuals sampled at each iteration. Note that this algorithm was described in [9]. The second version only samples and evaluates a single candidate solution at each iteration, but then makes comparisons and updates with a number of previously evaluated vectors stored in a sliding history window.

A. Multiple-Sample per Iteration cGA

In the multi-sample version we now make n samples per iteration and present it in Alg. 1. The standard cGA is the case of $n = 2$ in Alg. 1.

Apart from this detail, the algorithm is very similar to the standard cGA. Since we are now making n samples and n evaluations, we now have $n(n-1)/2$ comparisons and updates to make. For instance, for $n = 10$ the ratio of updates per sample is now 4.5, nine times higher than the standard $n = 2$ case.

Note that an algorithm similar to this was described in [9] though the way the algorithm was listed did not separate the fitness evaluation from the comparison (which is necessary in order to make best use of the fitness evaluation budget), though this detail may have been considered to be a low-level implementation detail by the authors. More importantly, the results presented in [9] for the multi-sample case were not particular good, perhaps due to a poorly chosen k value. When making more updates per fitness evaluation, k needs to be set higher to avoid premature convergence.

This could be the reason why recent work on the cGA [6] has not mentioned the Multiple-Sample variant. We will show that when k is chosen well, the Multiple-Sample cGA (MScGA) greatly outperforms the standard cGA.

B. Sliding Window cGA

While the MScGA aims to provide more efficient use of the available fitness evaluations, it suffers from the fact that the probability vector is only updated after all the samples for an iteration have been drawn.

However, it may be beneficial to update the probability vector more frequently, ideally after every new sample has been drawn and evaluated. This is exactly what the Sliding Window cGA (SWcGA) achieves. In addition to the parameter k, this algorithm adds the parameter w for the size of the window.

Algorithm 2 Sliding Window compact Genetic Algorithm (SWcGA) for a d-bit binary problem.

```
Require: problem dimension $d$
Require: fitness function $fitness(\cdot) : \{0,1\}^d \mapsto \mathbb{R}$
Require: virtual population size $k$
Require: sliding window width $w$

1:  function CGA($d, k, w$)
2:      Initialise $p \leftarrow \{\frac{1}{2}\}^d$
3:      Initialise empty queue $QX$ to save individuals
4:      Initialise empty queue $QY$ to save fitness values
5:      while ! ISCONVERGED($p$) do
6:          $x, y \leftarrow$ REPRODUCEANDEVALUATE($p$)
7:          for $h \in \{1, \dots, length(QX)\}$ do
8:              $x_{win}, x_{loss} \leftarrow$ COMPETE($x, y, QX_h, QY_h$)
9:              $p \leftarrow$ UPDATE($p, x_{win}, x_{loss}$)
10:         $QueueX \leftarrow$ FIFO($x, QX, w$)
11:         $QueueY \leftarrow$ FIFO($y, QY, w$)
12:     $x^* \leftarrow$ RECOMMEND($p$)
13:     return $x^*$

14: function COMPETE($x, y, x', y'$)
15:     if $y > y'$ then
16:         $x_{win} \leftarrow x$
17:         $x_{loss} \leftarrow x'$
18:     else
19:         $x_{win} \leftarrow x'$
20:         $x_{loss} \leftarrow x$
        return $x_{win}, x_{loss}$

21: function FIFO($e, Q, w$)
22:     if $length(Q) == w$ then
23:         Dequeue the first element of $Queue$
24:     Enqueue $e$ to the tail of $Queue$
25:     return $Queue$
```

Again, the algorithm is similar to the standard cGA, except that now every time a sample is drawn from the probability vector, the fitness is evaluated and then the scored vector is compared with every other one in the window, and for each comparison the probability vector is updated. Note that each sample only has its fitness evaluated once and stored together with the sample in the sliding window (which can be implemented as a circular buffer or a FIFO queue). See algorithm 2 for the listing. For each new candidate sampled, assuming steady state when the buffer is already full, we make comparisons and updates with the w previously evaluated samples. Hence, the ratio of comparisons and updates to fitness evaluations is w. After the comparisons and updates have been made, the new scored sample is added to the sliding window buffer, replacing the oldest one if the buffer is already full (i.e. already has w scored samples in it).

IV. Test Problems

We considered two binary optimisation problems based on bit strings: the OneMax problem corrupted by additive Gaussian noise, namely noisy OneMax, and the noisy PMax problem.

A. *Noisy OneMax*

The OneMax problem aims at maximising the number of 1 bits in a binary string. Let $\mathcal{N}(\mu, \sigma^2)$ denote Gaussian noise with mean μ and variance σ^2, and $\mathbf{x}$ is an d-bit binary string. The d-bit OneMax problem with additive Gaussian noise is formalised as $f(\mathbf{x}) = \sum_{i=1}^{d} \mathbf{x}_i + \mathcal{N}(0, 1)$. Friedrich et al [6] have proven that with high probability, the standard cGA with $k = \omega(\sigma 2\sqrt{d} \log d)$ converges to the optimal distribution p^* after $O(k\sigma^2\sqrt{d}\log kd)$ iterations when optimising a noisy OneMax with variance $\sigma^2 > 0$, where $k = 7\sigma^2\sqrt{d}(\ln d)^2$ is used. Thus, in the case considered in this paper ($\sigma^2 = 1$, $d = 100$), the k should be $7 * \sqrt{d}(\ln d)^2 \approx 15d$ to guarantee the convergence. This setting is compared to as baselines in our experiments.

B. *Noisy PMax*

The Noisy PMax problem is proposed by Lucas et al. [10] to represent an artificial game outcome optimisation problem. In this artificial model, $\mathbf{x}$ is treated as an d-bit binary number, and the true winning rate of $\mathbf{x}$ is defined as $\mathcal{P}_{win}(\mathbf{x}) = \frac{Value(\mathbf{x})}{2^d - 1}$, where $Value(\mathbf{x})$ denotes the numeric value of $\mathbf{x}$ located between 0 and $(2^d - 1)$. Thus, the outcome of a game is either win (1) with probability $\mathcal{P}_{win}(\mathbf{x})$ or loss, otherwise.

This problem formulation is also relevant to learning playout control parameters for Monte Carlo Tree Search (MCTS) [11], where the parameters control the biases for a stochastic playout policy. The efficient cGA variants described in this paper should be able to improve on the simple evolutionary algorithms used in [12], [13] but this has not yet been tried. The relevance is due to three factors: the extreme noise when evaluating stochastic playout policies, and the requirement for rapid adaptation (a feature of the MScGA algorithms), and the expectation that different parameters have very different levels of importance in controlling the playouts.

V. Experimental results and discussion

A. *Experimental setting*

We consider two baseline algorithms, the standard cGA and the Random Mutation Hill Climber (RMHC) on each of the tested problems. For each experimental run each algorithm was given a fixed maximum budget of $1,000$ fitness evaluations. Thus, the cGA and its variants stopped when the stopping condition defined in Alg. 1 was met, or the (noisy) fitness function had been evaluated $1,000$ times. Note that we did not use first hitting time as a measure, since this has been shown to give misleading results for noisy optimisation problems [10]. Since each algorithm under test is able to return its best guess (by RECOMMEND(p) used by cGA and its variants) or best solution found so far (RMHC) at any iteration, we plotted the true (noise-free) fitness of the current solutions of each algorithm at each iteration. Hence, in addition to the final fitness found we may also observe how fitness evolves over time.

First, we optimise separately the problems using cGA with different virtual population size k and using RMHC with different resampling number r, then choose the k and r with the best performance, respectively. More study on the optimal resampling using RMHC on the OneMax with additive Gaussian noise can by found in [14].

B. *Noisy OneMax*

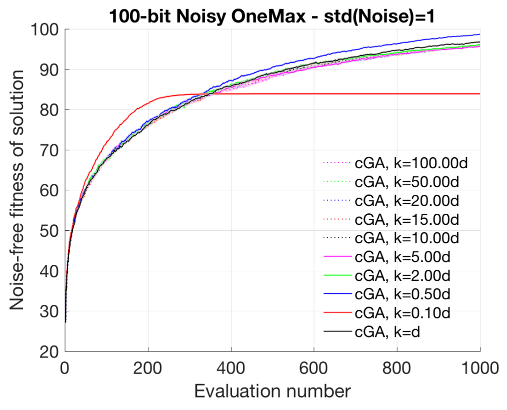
The performance of the standard cGA with different virtual population size k and RMHC with different resampling number r on the 100-bit noisy OneMax problem is illustrated in Fig. 1. Fig. 1a shows that the cGA with virtual population size $k = d/10$ performs the best when the maximal fitness evaluation number is smaller than 300. It is notable that using small virtual population size, the cGA converges quickly to a good solution at the early stage of optimisation then never finds the optimum. As the RMHC with variant resampling numbers (see Fig. 1b) does not outperform the standard cGA with best $k = d/2$ (blue curve in Fig. 1a), our algorithms are directly compared to the standard cGA with $k = d/2$.

Fig. 2 compares the performance of MScGA instances using different virtual population size k and sampling number n and Fig. 3 compares the performance of SWcGA instances using different virtual population size k and sliding window width w on the identical noisy OneMax problem. When k is close to d, the more samples there are, the worse the solution recommended by MScGA at each iteration is; the wider the sliding window is, the worse the solution recommended by SWcGA at each iteration is. When k is large, larger sample number leads to better performance of MScGA and MScGA significantly outperforms the best standard cGA, but the difference led by using different sample number is minor; wider window leads to better performance of SWcGA, and the overall performance is better than MScGA. However, very big k will weaken the performance of both MScGA and SWcGA. MScGA and SWcGA with optimal parameter setting have similar performance, but MScGA is less sensitive to its parameter, sample number n.

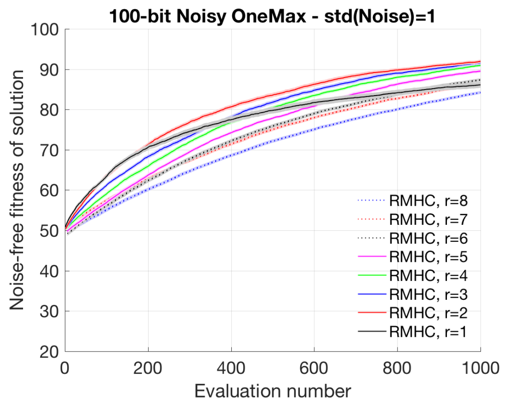
The best parameter settings of each of the algorithms are listed and compared in Figs. 7a and 7b, as well as the averaged final probability vector p over 100 trials. Though MScGA converges faster than SWcGA, it did not stop with better solutions than SWcGA.

C. *PMax*

The performance of the standard cGA with different virtual population size k and RMHC with different resampling number r on the PMax problem is illustrated in Fig. 4. As the RMHC with variant resampling numbers does not outperform the standard cGA with best setting $k = 5d$ (pink curve in Fig. 4a), our algorithms are directly compared to the standard cGA with $k = 5d$.



(a) Quality of solution recommended by cGA with different learning rate k using different budget. The performance of cGA is less sensitive to the tested k value on the tested problem.



(b) Quality of solution recommended by RMHC with r resamplings using different budget. The performance of RMHC without resampling (case $r = 1$) is poor as predicted.

Fig. 1: Results of cGA and RMHC on the noisy OneMax problem. Each curve is an average of 100 trials, with $d = 100$. The standard error is also given as a faded area around the average.

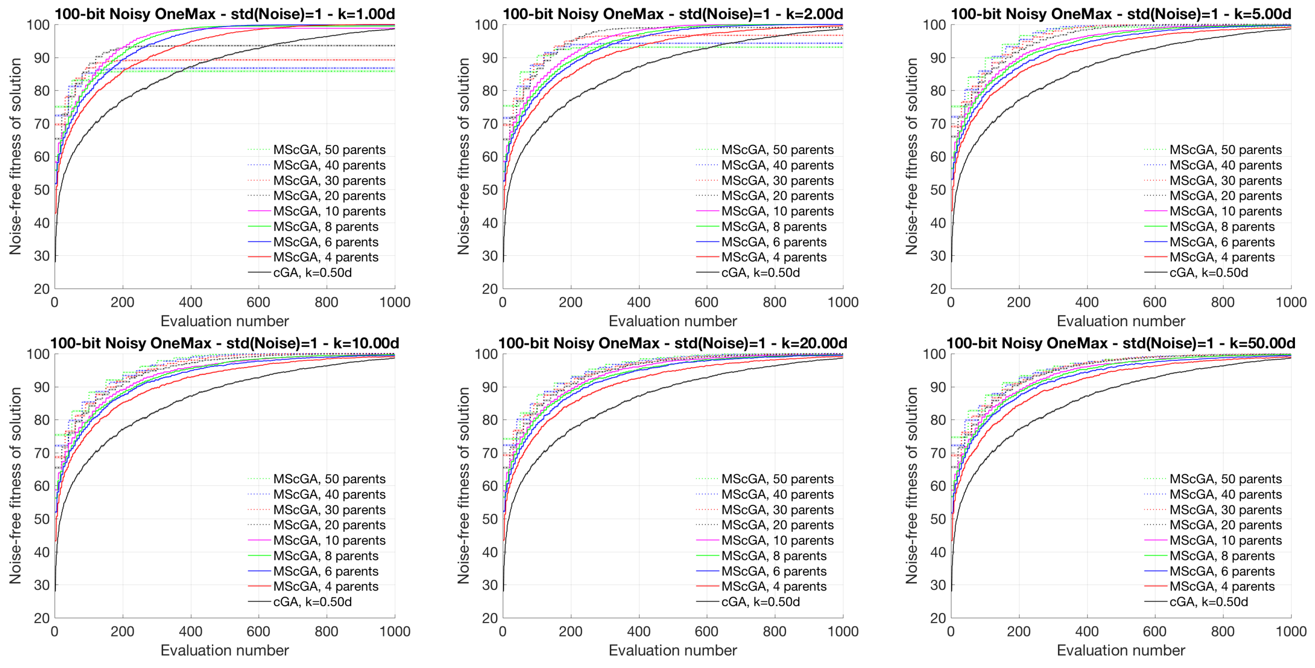


Fig. 2: Results of the noisy OneMax problem optimised by MScGA. Each curve is an average of 100 trials, with $d = 100$. The standard error is also given as a faded area around the average. The results using other values of k are not shown as they are similar or worse than the ones shown.

The performance of MScGA and SWcGA with different parameter settings are compared to cGA with $k = 5d$ (black curves) in Figs. 5 and 6, respectively. The best parameter settings of each of the algorithms are listed and compared in Figs. 7c and 7d, as well as the averaged final probability vector p over 100 trials. The SWcGA slightly outperforms the MScGA.

VI. Conclusion and Future Work

This paper introduced a simple but important principle to improve the performance of the compact Genetic Algorithm: to make best possible use of each fitness evaluation by reusing the result in multiple comparisons, and hence in multiple updates of the probability distribution.

This principle was used to develop two variations of the algorithm: the first made multiple samples, comparisons and updates at each iteration, while the second one made just one sample at each iteration, but then performed multiple comparisons and updates by accessing a sliding window of previously evaluated candidates (samples).

Both algorithms significantly outperformed the standard

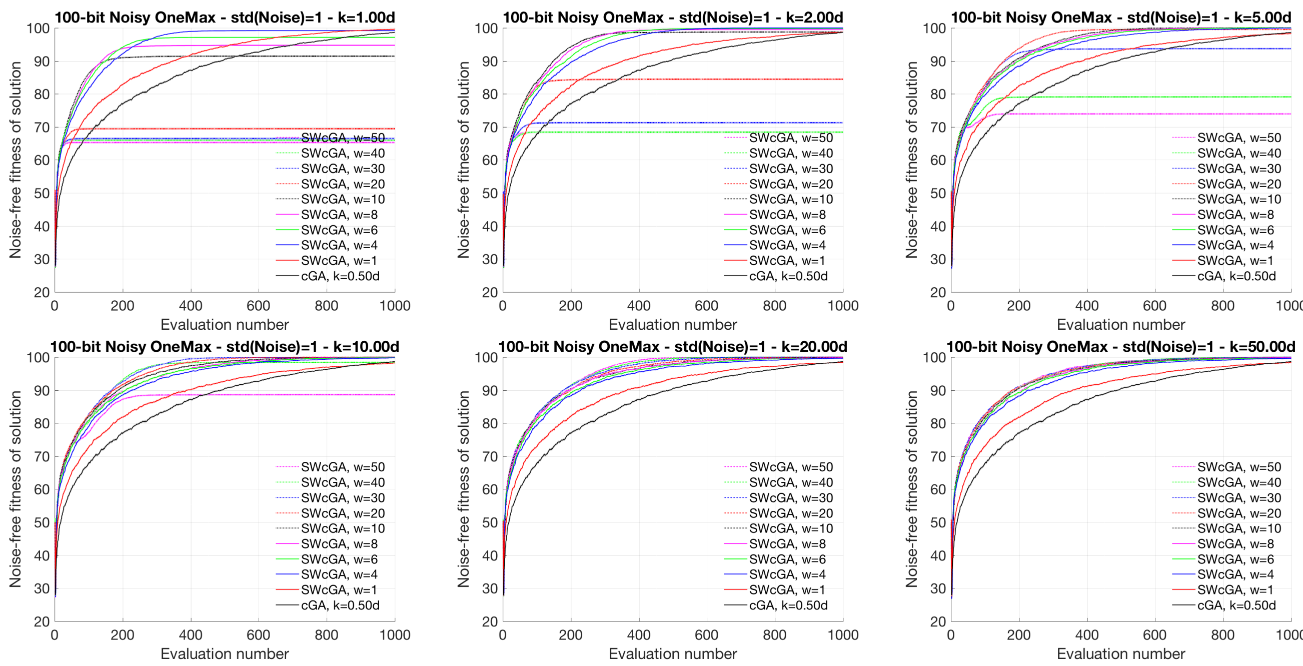
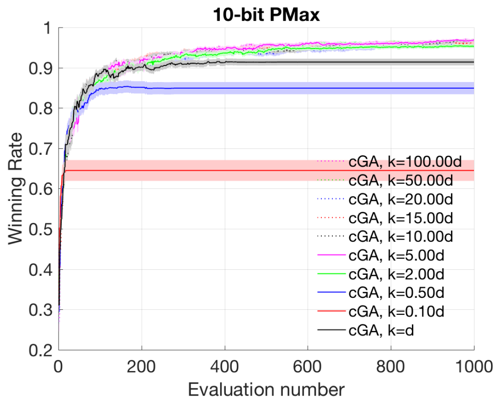
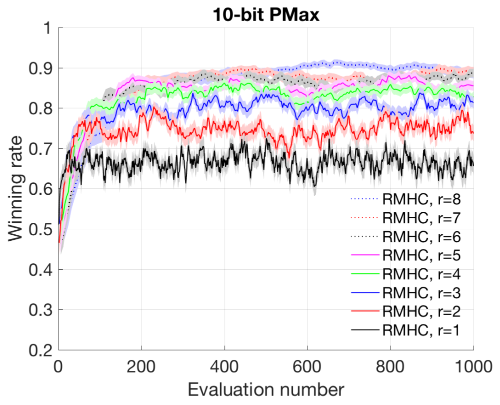


Fig. 3: Results of the noisy OneMax problem optimised by SWcGA. Each curve is an average of 100 trials. The standard error is also given as a faded area around the average. The results using other values of k are not shown as they are similar or worse than the one shown.



(a) Quality of solution recommended by cGA with different virtual population size k using different budget. The performance of cGA is less sensitive to the tested k values.



(b) Quality of solution recommended by RMHC with resampling r resamplings using different budget. The performance of RMHC without resampling (case $r = 1$) is poor as predicted.

Fig. 4: Results of cGA and RMHC on the PMax problem. Each curve is an average of 100 trials. The standard error is also given as a faded area around the average.

cGA, with the sliding window version performing best. The sliding window version is therefore the one we are focusing on in on-going work. In addition to offering the best performance at the end of the each run, it also consistently offered better recommendations at nearly every stage of each run, making it a better choice as an anytime algorithm for use in real-time game AI. The sliding window variant is better as an anytime algorithm as it adds only a single candidate solution per iteration, meaning that the update of the recommendation happens more frequently. Another interesting observation is the ability of cGA, MScGA and SWcGA to correctly *recommend* the optimal solution without having actually sampled it. This is illustrated in Appendix A of [15].

We are currently extending the work in two ways. The first is to allow multi-valued strings, since binary is an unnatural way to represent many problems. The second is to explore alternative ways to model the probability distribution. Both of these are already yielding positive results and will be the focus of future research. Also relevant is our recent work on bandit-based optimisation [8], [16], which explicitly balances

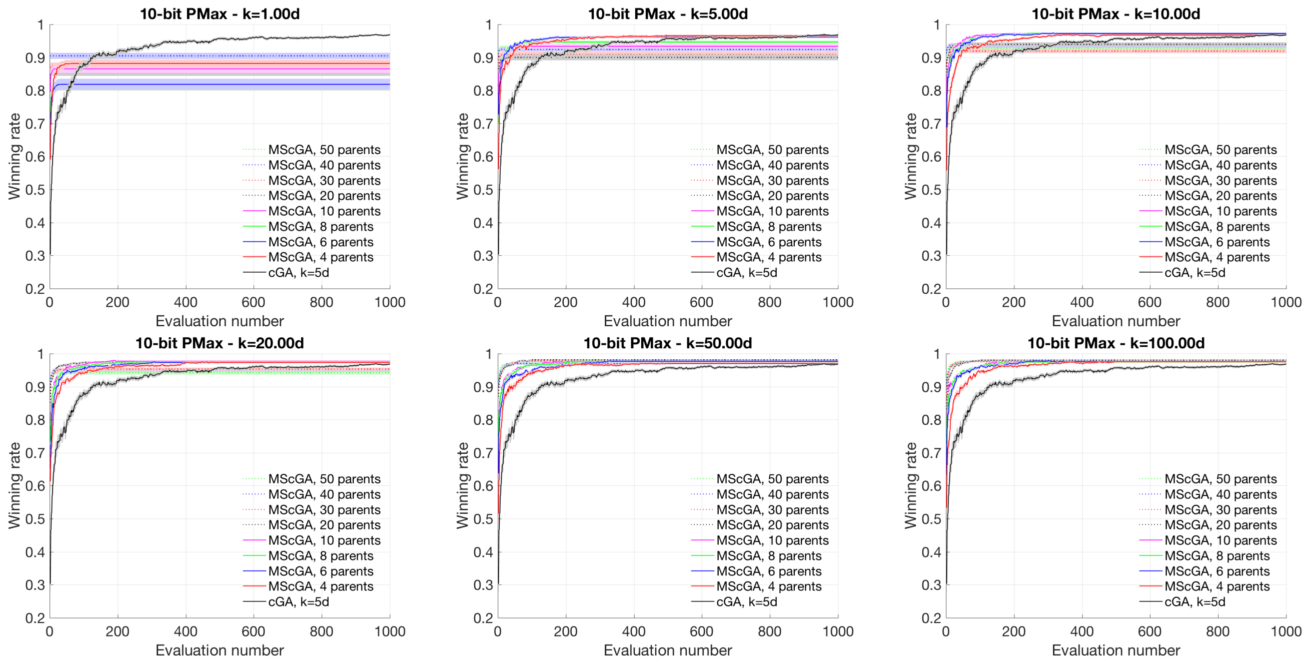


Fig. 5: Results of the PMax problem optimised by MScGA. Each curve is an average of 100 trials. The standard error is also given as a faded area around the average. The difference of performance of MScGA using different parameter settings is tiny, and all MScGA instances significantly outperform the standard cGA with $k = 5d$. The results using other values of k are not shown as they are similar or worse than the one shown.

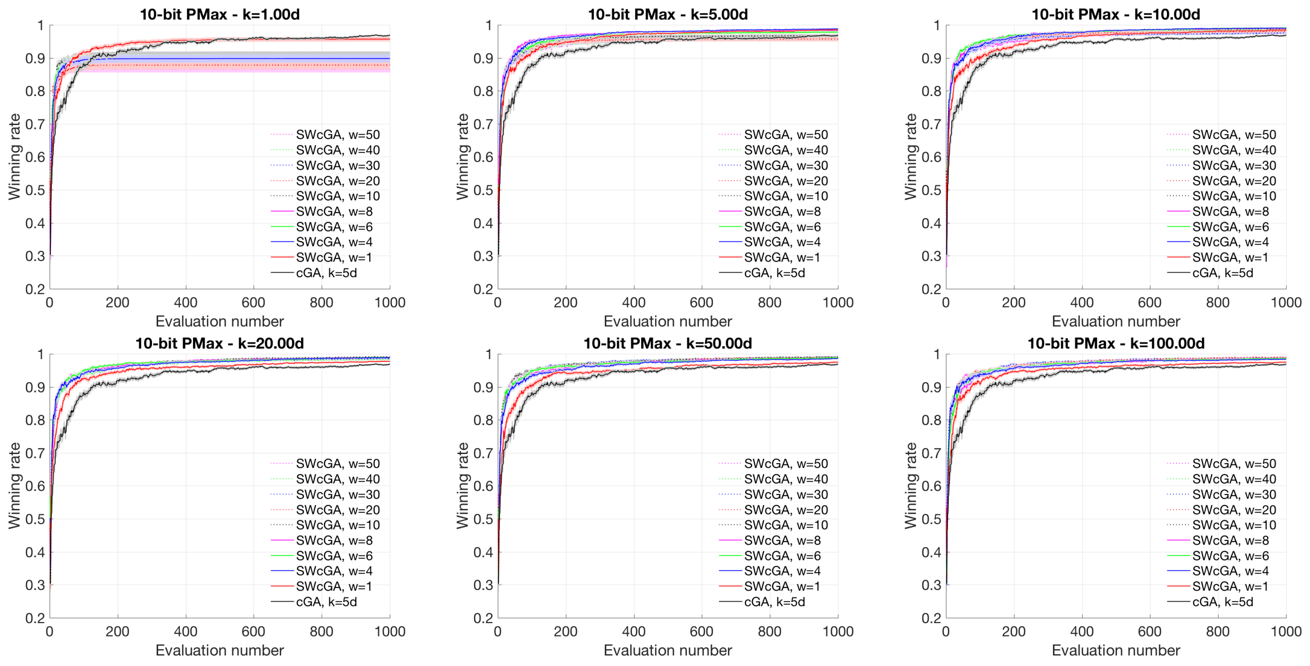
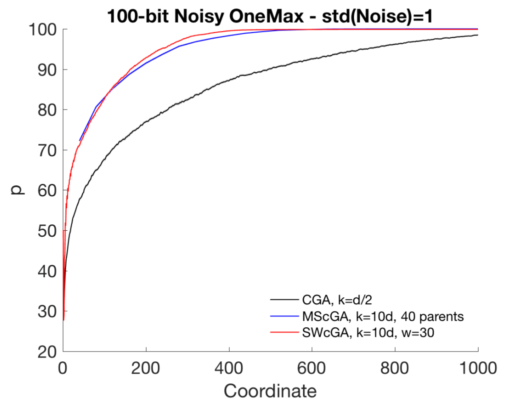
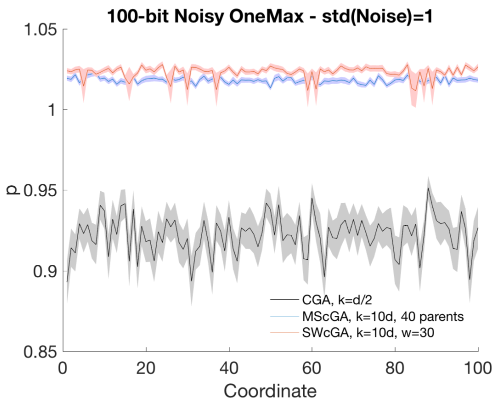


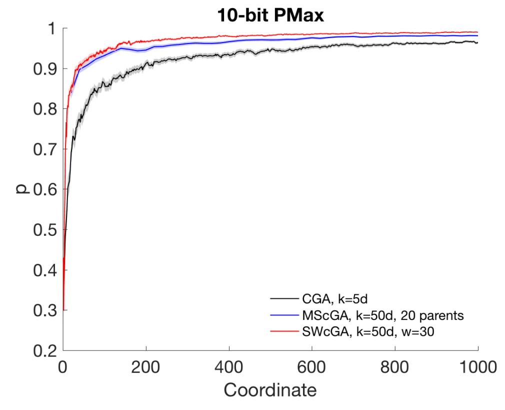
Fig. 6: Results of the PMax problem optimised by SWcGA. Each curve is an average of 100 trials. The standard error is also given as a faded area around the average. The difference of performance of SWcGA using different parameter settings is tiny, and all SWcGA instances significantly outperform the standard cGA with $k = 5d$. The results using other values of k are not shown as they are similar or worse than the one shown.



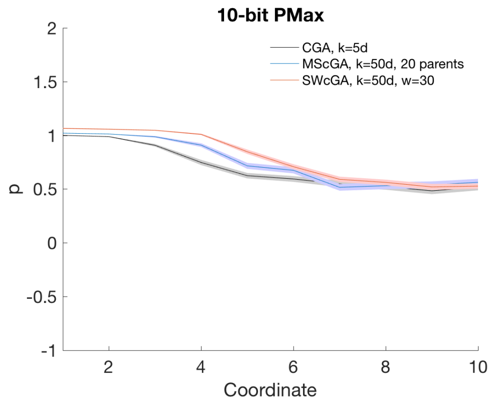
(a) Quality of recommendations.



(b) Averaged final probability vector p.



(c) Quality of recommendations.



(d) Averaged final probability vector p.

Fig. 7: Results of the noisy OneMax and PMax problem optimised by the cGA, MScGA and WScGA with best tested parameter settings. Each curve is an average of 100 trials. The standard error is also given as a faded area around the average.

exploration versus exploitation, but has not yet been combined with the sliding window approach developed here. There is reason to believe that such a combination will be beneficial. Another interesting direction for future work is choosing adaptively the parameters for both variants.

References

[1] P. Rakshit, A. Konar, and S. Das, "Noisy Evolutionary Optimization Algorithms-A Comprehensive Survey," *Swarm and Evolutionary Computation*, 2016.

[2] D. Perez-Liebana, S. Samothrakis, J. Togelius, T. Schaul, S. M. Lucas, A. Couëtoux, J. Lee, C.-U. Lim, and T. Thompson, "The 2014 general video game playing competition," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 8, no. 3, pp. 229–243, 2016.

[3] J. Liu, J. Togelius, D. Pérez-Liébana, and S. M. Lucas, "Evolving game skill-depth using general video game ai agents," 2017.

[4] G. R. Harik, F. G. Lobo, and D. E. Goldberg, "The Compact Genetic Algorithm," *IEEE transactions on evolutionary computation*, vol. 3, no. 4, pp. 287–297, 1999.

[5] M. Mitchell, *An Introduction to Genetic Algorithms*. MIT press, 1998.

[6] T. Friedrich, T. Ktzing, M. S. Krejca, and A. M. Sutton, "The Compact Genetic Algorithm is Efficient Under Extreme Gaussian Noise," *IEEE Transactions on Evolutionary Computation*, vol. 21, no. 3, pp. 477–490, June 2017.

[7] G. Harik, "Linkage Learning via Probabilistic Modeling in the ECGA," *Urbana*, vol. 51, no. 61, p. 801, 1999.

[8] K. Kunanusont, R. D. Gaina, J. Liu, D. Perez-Liebana, and S. M. Lucas, "The N-Tuple Bandit Evolutionary Algorithm for Automatic Game Improvement," in *2017 IEEE Congress on Evolutionary Computation (CEC)*, 2017.

[9] G. R. Harik, F. G. Lobo, and D. E. Goldberg, "The Compact Genetic Algorithm," *Urbana*, vol. 51, p. 61801, 1997.

[10] S. M. Lucas, J. Liu, and D. Pérez-Liébana, "Evaluating Noisy Optimisation Algorithms: First Hitting Time is Problematic," *https://arxiv.org/abs/1706.05086*, 2017.

[11] T. Cazenave, "Playout policy adaptation with move features," *Theoretical Computer Science*, vol. 644, pp. 43–52, 2016.

[12] S. M. Lucas, S. Samothrakis, and D. Perez-Liebana, "Fast evolutionary adaptation for Monte Carlo Tree Search," in *European Conference on the Applications of Evolutionary Computation*, 2014, pp. 349 – 360.

[13] D. Perez-Liebana, S. Samothrakis, and S. M. Lucas, "Knowledge-based fast evolutionary MCTS for general video game playing," in *IEEE Conference on Computational Intelligence and Games*, 2014.

[14] J. Liu, M. Fairbank, D. Pérez-Liébana, and S. M. Lucas, "Optimal Resampling for the Noisy OneMax Problem," *arXiv preprint arXiv:1607.06641*, 2016.

[15] S. M. Lucas, J. Liu, and D. Pérez-Liébana, "Efficient Noisy Optimisation with the Sliding Window Compact Genetic Algorithm," *arXiv preprint arXiv:1971567*, 2017. [Online]. Available: http://www.liujialin.tech/publications/efficient-noisy-optimisation.pdf

[16] J. Liu, D. Pérez-Liébana, and S. M. Lucas, "Bandit-based random mutation hill-climbing," in *Evolutionary Computation (CEC), 2017 IEEE Congress on*. IEEE, 2017, pp. 2145–2151.