The benefits and limitations of voting mechanisms in evolutionary optimisation

Rowe, Jon; Aishwaryaprajna

The Benefits and Limitations of Voting Mechanisms in Evolutionary Optimisation

Jonathan E. Rowe
School of Computer Science,
University of Birmingham
Birmingham, UK
The Alan Turing Institute,
London, UK
j.e.rowe@cs.bham.ac.uk

Aishwaryaprajna
School of Computer Science,
University of Birmingham
Birmingham, UK
axx784@cs.bham.ac.uk

ABSTRACT

We study the use of voting mechanisms in populations, and introduce a new Voting algorithm which can solve OneMax and Jump in $O(n \log n)$, even for gaps as large as $O(n)$. More significantly, the algorithm solves OneMax with added posterior noise in $O(n \log n)$, when the variance of the noise distribution is $\sigma^2 = O(n)$ and in $O(\sigma^2 \log n)$ when the noise variance is greater than this. We assume only that the noise distribution has finite mean and variance and (for the larger noise case) that it is unimodal. We also examine the performance on arbitrary linear and monotonic functions. The Voting algorithm fails on LeadingOnes but we give a variant which can solve the problem in $O(n \log n)$. We empirically study the use of voting in population based algorithms (UMDA, PCEA and cGA) and show that this can be effective for large population sizes.

CCS CONCEPTS

• **Computing methodologies** → **Search methodologies**; **Randomized search**; *Artificial intelligence*;

KEYWORDS

voting, crossover, noise

1 INTRODUCTION

In a recent paper, Whitley et al. [13] analysed the efficacy of employing *voting crossover* in optimising the Jump function, defined (for bit strings of length n) as follows:

$$\textsc{Jump}(x) = \begin{array}{ll} n+m & \text{if } |x|_1 \leq n-m \text{ or } |x|_1 = n \\ n-|x|_1 & \text{otherwise} \end{array}$$

where m is the size of the gap that must be jumped in order to find the optimum, and $|x|_1$ is the count of the number of ones in string x.

Voting crossover is a form of crossover involving multiple parents, in which the value of each bit in the offspring is determined by a simple majority vote of each of the selected parents. Whitley et al. show that their algorithm, which involves a phase of hill-climbing, followed by a phase of voting crossover, solves the Jump problem, when $m = O(\log n)$, in linear time.

In this paper, we investigate this further, proving simpler and better results (section 2), and then look at the use of other algorithms based on voting. We examine an algorithm in which a population is generated using tournament selection, followed by a round of voting, and show this is able to solve OneMax and Jump in $O(n \log n)$, for gaps even as large as $O(n)$ (section 3). More significantly, it can solve the noisy OneMax problem, with posterior noise, in time $O(n \log n)$, when the noise variance is less than a constant times n, with any noise distribution having finite mean and variance. We also show that for larger values of of the variance, σ^2, the runtime is $O(\sigma^2 \log n)$, where we additionally assume the noise distribution is unimodal. This is the best algorithm that we are aware of for this problem (section 4). We then look at the behaviour of the algorithm on monotone functions (section 5) where we see a deterioration in perfroamnce, for which we provide a bound. A modification of the voting algorithm is presented in section 6, which gives a variant that solves LeadingOnes in $O(n \log n)$.

We then look at the idea of incorporating a voting mechanism into the other algorithms, including UMDA, PCEA and cGA. We study this approach empirically (section 7), and show that it leads to significant speed-ups for OneMax (with and without noise) and Jump, for large populations. For the noisy OneMax problem, the performance beats the basic voting algorithm in our experiments.

2 VOTING CROSSOVER ON JUMP

The algorithm proposed by Whitley et al. [13] for the Jump problem proceeds in two phases. In the first phase, a hill climber is used to get to a string containing $n - m$ ones and m zeros. They use the *next ascent bit climber* [4]. This works by choosing a random permutation of $\{1, \ldots, n\}$, and then mutating bits in that order, keeping changes that lead to improvements. After one round of this (which take n steps) we will have arrived at a string with $n-m$ ones. We mention is passing, that this approach would also solve OneMax in n steps. This process is repeated three times, starting each time from a fresh random string. The three resulting strings then perform the voting crossover — the offspring takes bit values given by a majority vote for each bit position. The whole process is repeated until the optimum is found. Whitley et al. show that only a constant number of repeats is needed if the gap $m = O(\log n)$. However, we can improve on this (and simplify the proof) as follows. Since each of the three strings has begun from a fresh random initial string, the location of the m zeros in the resulting strings are independent, for each string. The vote will go wrong, in a single bit position, if either all three strings contain a zero, or only one of them contains a one at that position. The probability that this will happen is

$$\left(\frac{m}{n}\right)^3 + 3\left(\frac{m}{n}\right)^2\left(1 - \frac{m}{n}\right) \leq 3\left(\frac{m}{n}\right)^2$$

Thus, by the union bound, the probability that the vote goes wrong in at least one bit position, it less than $3m^2/n$. So if $m < a\sqrt{n}$, where $a < 1/\sqrt{3}$ is a constant, then the vote fails with probability at most $3a^2$, meaning that we need to repeat the process an expected number of $1/(1-3a^2)$ times before the solution is found. Thus, the algorithm solves the Jump problem for gap sizes $m < \sqrt{n/3}$ in linear time.

A more typical evolutionary algorithm employing voting crossover is analysed in [6], and shown to solve Jump in $O(n \log n)$.

3 THE VOTING ALGORITHM

We now introduce a new algorithm based on applying a bitwise vote to a population that has been produced by selection. In this version we use binary tournament selection. We generate two random strings, choose the best of the two, and add it to the population. When we have enough strings in the population, we take a bitwise vote (see Algorithm 1).

```
Let p = (0, ..., 0);
repeat
    Let x ∈ {0,1}^n be a random string;
    Let y ∈ {0,1}^n be a random string;
    if f(x)>f(y) then
        p = p+x;
    else
        p = p+y;
    end
until μ times;
for 1 ≤ i ≤ n do
    z_i = [p_i > μ/2];
end
Return z;
```

Algorithm 1: The Voting Algorithm

We first analyse the performance of this algorithm on OneMax.

$$\textsc{OneMax}(x) = |x|_1$$

To do this, we need the following result

Lemma 3.1. *Let $x, y \in \{0,1\}^n$ be random strings and let the tournament winner, z, be decided according to the OneMax function. For any $k \in \{1, \dots n\}$,*

$$\Pr(z_k = 1) \geq \frac{1}{2} + \frac{1}{8\sqrt{n}}$$

Proof. The probability that the winner of the tournament has a one in position k is

$$\begin{aligned}\Pr(z_k = 1) = &\Pr(x_k = 1 \mid x \text{ wins}) \Pr(x \text{ wins}) + \\ &\Pr(y_k = 1 \mid y \text{ wins}) \Pr(y \text{ wins})\end{aligned}$$

and so by symmetry

$$\Pr(z_k = 1) = \Pr(x_k = 1 \mid x \text{ wins}).$$

By Bayes' Theorem

$$\Pr(x_k = 1 \mid x \text{ wins}) = \frac{\Pr(x_k = 1)}{\Pr(x \text{ wins})} \Pr(x \text{ wins} \mid x_k = 1)$$

and so

$$\Pr(x_k = 1 \mid x \text{ wins}) = \Pr(x \text{ wins} \mid x_k = 1)$$

since $\Pr(x_k = 1) = \Pr(x \text{ wins}) = 1/2$. Then, by the law of total probability,

$$\begin{aligned}&\Pr(x_k = 1 \mid x \text{ wins}) \\ = \;& \frac{1}{2}\Pr(x \text{ wins} \mid x_k = 1, y_k = 1) + \frac{1}{2}\Pr(x \text{ wins} \mid x_k = 1, y_k = 0) \\ \geq \;& \frac{1}{4} + \frac{1}{2}\Pr\left(\sum_{i\neq k} x_i + 1 > \sum_{i\neq k} y_i\right) \\ = \;& \frac{1}{4} + \frac{1}{2}\Pr\left(\sum_{i\neq k} y_i - x_i < 1\right) \\ = \;& \frac{1}{4} + \frac{1}{2}\left(\Pr\left(\sum_{i\neq k} y_i - x_i < 0\right) + \Pr\left(\sum_{i\neq k} y_i - x_i = 0\right)\right)\end{aligned}$$

By symmetry, we have

$$\Pr\left(\sum_{i\neq k} y_i - x_i < 0\right) = \frac{1}{2}\left(1 - \Pr\left(\sum_{i\neq k} y_i - x_i = 0\right)\right)$$

so that

$$\begin{aligned}&\Pr(x_k = 1 \mid x \text{ wins}) \\ = \;& \frac{1}{2} + \frac{1}{4}\left(\Pr\left(\sum_{i\neq k} y_i - x_i = 0\right)\right) \\ = \;& \frac{1}{2} + \frac{1}{4}\left(\sum_{j=0}^{n-1} \Pr\left(\sum_{i\neq k} y_i = j\right)\Pr\left(\sum_{i\neq k} x_i = j\right)\right) \\ = \;& \frac{1}{2} + \frac{1}{2^{2n}}\sum_{j=0}^{n-1}\binom{n-1}{j}^2 \\ = \;& \frac{1}{2} + \frac{1}{2^{2n}}\binom{2n-2}{n-1} \\ \geq \;& \frac{1}{2} + \frac{1}{8\sqrt{n}}\end{aligned}$$

where the final inequality derives from [11]. □

We can now show that the voting algorithm solves OneMax (with high probability) if $\mu = \Omega(n \log n)$.

Theorem 3.2. *If $\mu = 32(c+1)n \log n$, then the Voting algorithm correctly solves OneMax with probability greater than $1 - 1/n^c$.*

Proof. For any one bit position, k, the probability that the vote is incorrect is

$$\Pr(p_k \leq \mu/2) \leq \exp(-2\mu/64n) = \frac{1}{n^{c+1}}$$

by Hoeffding's inequality. So by the union bound, the probability that at least one bit gets the incorrect vote is at most $1/n^c$. □

It is clear that the Voting algorithm only samples strings which have a number of ones close to $n/2$. Indeed by Hoeffding's inequality it is exponentially unlikely to sample strings with greater than $n/2 + \alpha n$ ones, for any constant $1/2 < \alpha < 1$. Thus we immediately have:

Corollary 3.3. *If $\mu = 32(c+1)n\log n$, then the Voting algorithm correctly solves Jump with probability greater than $1 - 1/n^c$, for any gap size $m < (1-\alpha)n$ where α is a constant in the range $1/2 < \alpha < 1$.*

4 NOISY ONEMAX

For the noisy OneMax problem, the fitness, at each evaluation, receives an addition of a random value drawn from some probability distribution.

One approach to solving this problem is to simply resample the fitness of each string enough times that the average gives a sufficiently good estimate. A result from [1] shows that if the running time of an algorithm on a problem with no noise is T, then the running time for the algorithm, with suitable re-sampling, on the same problem with added Gaussian noise is $O(\sigma^2 T \log T)$. The most efficient algorithm for OneMax requires $T = \Theta(n/\log n)$ string evaluations [2]. Adopting this strategy, with resampling, would give a runtime of $O(\sigma^2 n)$ on noisy OneMax with posterior Gaussian noise.

There have been a number of evolutionary algorithms whose performance on noisy OneMax has been analysed. The Paired Crossover Evolutionary Algorithm [10], for example, finds the optimal string in $O(n(\log n)^2)$ function evaluations, when the noise distribution is Gaussian and $\sigma^2 = n$. This is already faster than the resampling method described above. The same paper also shows that the (1 + 1)EA is not able to solve noisy OneMax efficiently.

Dang and Lehre [3] show that a population-based algorithm, with mutation but no crossover can solve noisy OneMax with Gaussian noise in $O(\sigma^7 n \log n \log\log n)$.

Friedrich et al. [7] prove that the compact Genetic Algorithm (cGA) solves noisy OneMax with Gaussian noise, with the runtime of $O(K\sigma^2\sqrt{n}\log(Kn)$ where the parameter K must be considered to be $\omega(\sigma^2\sqrt{n}\log n)$. This gives an upper bound in an excess of $\sigma^4 n(\log\sigma^2)(\log n)$.

We will show the voting algorithm has superior runtime on noisy OneMax for arbitrary noise distributions. We will assume only that the noise distribution has finite mean and variance σ^2.

In the proof above for OneMax with no noise, we used a bound on the central binomial coefficient. We now need a similar bound for binomial coefficients that are close to the centre.

Lemma 4.1. *For any integers $m > 0$ and $0 \le k \le \sqrt{m}$ we have*

$$\binom{2m}{m+k} \ge \left(\frac{2\sqrt{\pi}}{e^4}\right)\frac{2^{2m}}{\sqrt{m}}$$

Proof. We use the following inequalities associated with Stirling's approximation:

$$\sqrt{2\pi}n^{n+1/2}e^{-n} \le n! \le en^{n+1/2}e^{-n}$$

to give

$$\begin{aligned}
\binom{2m}{m+k} &= \frac{(2m)!}{(m+k)!(m-k)!} \\
&\ge \left(\frac{\sqrt{2\pi}}{e^2}\right)\frac{2^{2m+1/2}m^{2m+1/2}}{(m+k)^{m+k+1/2}(m-k)^{m-k+1/2}} \\
&= \left(\frac{2\sqrt{\pi}}{e^2}\right)\left(\frac{2^{2m}}{\sqrt{m}}\right)\frac{1}{(1+k/m)^{m+k+1/2}(1-k/m)^{m-k+1/2}} \\
&= \left(\frac{2\sqrt{\pi}}{e^2}\right)\left(\frac{2^{2m}}{\sqrt{m}}\right)\frac{1}{(1-k^2/m^2)^{m-k+1/2}(1+k/m)^{2k}} \\
&\ge \left(\frac{2\sqrt{\pi}}{e^2}\right)\left(\frac{2^{2m}}{\sqrt{m}}\right)\frac{1}{(1+k/m)^{2k}} \\
&\ge \left(\frac{2\sqrt{\pi}}{e^4}\right)\frac{2^{2m}}{\sqrt{m}}
\end{aligned}$$

□

We will also need the following.

Lemma 4.2. *Given two random binary strings, a, b, of length m, and any integer $0 \le s \le m$, we have*

$$\Pr(|a|_1 - |b|_1 = s) = \frac{1}{2^{2m}}\binom{2m}{m+s}$$

Proof. To achieve a difference of s requires picking $i \ge s$ ones in string a and then $i - s$ ones in string b. The probability this happens is

$$\frac{1}{2^{2m}}\sum_{i=s}^{m}\binom{m}{i}\binom{m}{i-s}$$

Now the number of ways of choosing $m + s$ items from $2m$ items can be described by the number of ways of choosing at least s items from the first m, and then the remainder from the other m items. That is

$$\binom{2m}{m+s} = \sum_{i=s}^{m}\binom{m}{i}\binom{m}{m+s-i} = \sum_{i=s}^{m}\binom{m}{i}\binom{m}{i-s}$$

by the symmetry of binomial coefficients. The result follows. □

Theorem 4.3. *The Voting algorithm correctly solves noisy OneMax with high probability, when the noise distribution has finite mean and variance $\sigma^2 \le 3n/8$, in $O(n\log n)$ function evaluations. If, in addition, the noise distribution is unimodal, then in the case $\sigma^2 \ge 3n/8$, the algorithm requires $O(\sigma^2\log n)$ function evaluations.*

Proof. As with the analysis of OneMax without noise, the probability that the winner of a tournament between two random strings x and y has a one in position k is equal to

$$\begin{aligned}
&\Pr(x_k = 1 \mid x \text{ wins}) \\
&= \frac{1}{2}\Pr(x \text{ wins} \mid x_k = 1, y_k = 1) + \frac{1}{2}\Pr(x \text{ wins} \mid x_k = 1, y_k = 0) \\
&\ge \frac{1}{4} + \frac{1}{2}\Pr\left(\sum_{i\ne k} x_i + 1 + U > \sum_{i\ne k} y_i + V\right) \\
&= \frac{1}{4} + \frac{1}{2}\Pr\left(\sum_{i\ne k} y_i - x_i < 1 + U - V\right)
\end{aligned}$$

where U and V are independent random samples from the noise distribution. We note that if U and V are two independent random values drawn from the noise distribution, then $U - V$ comes from a symmetric distribution with zero mean, and variance $2\sigma^2$.

Now

$$\Pr\left(\sum_{i\neq k} y_i - x_i < 1 + U - V\right)$$
$$= \sum_{s=-(n-1)}^{n-1} \Pr\left(\sum_{i\neq k} y_i - x_i = s\right) \Pr(U - V > s - 1)$$
$$= \Pr\left(\sum_{i\neq k} y_i - x_i = 0\right) \Pr(U - V > -1)$$
$$+ \sum_{s=1}^{n-1} \Pr\left(\sum_{i\neq k} y_i - x_i = s\right) (\Pr(s - 1 < U - V) + \Pr(U - V < s + 1))$$

(where we have used the fact that $U - V$ is symmetric)

$$= \Pr\left(\sum_{i\neq k} y_i - x_i = 0\right) \Pr(U - V > -1) +$$
$$\sum_{s=1}^{n-1} \Pr\left(\sum_{i\neq k} y_i - x_i = s\right) (1 + \Pr(s - 1 < U - V < s + 1))$$
$$= \Pr\left(\sum_{i\neq k} y_i - x_i = 0\right) \Pr(U - V > -1) + \frac{1}{2} - \frac{1}{2} \Pr\left(\sum_{i\neq k} y_i - x_i = 0\right)$$
$$+ \sum_{s=1}^{n-1} \Pr\left(\sum_{i\neq k} y_i - x_i = s\right) \Pr(s - 1 < U - V < s + 1)$$
$$= \frac{1}{2} + \Pr\left(\sum_{i\neq k} y_i - x_i = 0\right) \Pr(0 < U - V < 1)$$
$$+ \sum_{s=1}^{n-1} \Pr\left(\sum_{i\neq k} y_i - x_i = s\right) \Pr(s - 1 < U - V < s + 1)$$
$$\geq \frac{1}{2} + \Pr\left(\sum_{i\neq k} y_i - x_i = 0\right) \Pr(0 < U - V < 1)$$
$$+ \sum_{s=1}^{\lfloor\sqrt{n}\rfloor} \Pr\left(\sum_{i\neq k} y_i - x_i = s\right) \Pr(s - 1 < U - V < s + 1)$$
$$\geq \frac{1}{2} + \left(\frac{2\sqrt{\pi}}{e^4\sqrt{n}}\right) \Pr(|U - V| \leq \lfloor\sqrt{n}\rfloor - 1)$$

where we have used the preceding lemmas. Thus, the probability of a one appearing in bit position k in the tournament winner is at least

$$\frac{1}{2} + \left(\frac{\sqrt{\pi}}{e^4\sqrt{n}}\right) \Pr(|U - V| \leq \lfloor\sqrt{n}\rfloor - 1)$$

In the case where $\sigma^2 \leq 3n/8$, we can use Chebyshev's inequality:

$$\Pr(|U - V| \leq \lfloor\sqrt{n}\rfloor - 1) \geq 1 - \frac{2\sigma^2}{(\lfloor\sqrt{n}\rfloor - 1)^2}$$

to show that the probability of a one appearing in position k is at least

$$\frac{1}{2} + \left(\frac{\sqrt{\pi}}{100e^4\sqrt{n}}\right)$$

for sufficiently large n. Following the same argument as in the non-noisy case allows us to conclude that the runtime is $O(n \log n)$ with high probability.

When σ^2 is larger, we additionally assume that the noise distribution is unimodal, from which it follows that the distribution of $U - V$ is also unimodal [12]. We may then use the Camp-Meidell inequality:

$$\Pr(|U - V| \leq \lfloor\sqrt{n}\rfloor - 1) \geq \frac{\lfloor\sqrt{n}\rfloor - 1}{\sqrt{6}\sigma}$$

which shows the probability of a one in position k to be at least:

$$\frac{1}{2} + \frac{\sqrt{\pi}}{4\sqrt{3}e^4\sigma}$$

for sufficiently large n. The inequality is valid for $\sigma^2 > 3n/8$. The remainder of the proof follows as before. □

5 MONOTONE AND LINEAR FUNCTIONS

As the Voting algorithm samples strings with close to $n/2$ ones, it is unlikely to efficiently solve all linear functions. In the case of BinVal the selection will be dominated by the highest order bits, and the voting on the low order bits will essentially be random. Nevertheless, it is interesting to see for what linear functions the algorithm remains efficient (in the sense of having a polynomial runtime). We can do this by looking at the broader class of *monotonic* functions which have the property that the fitness always increases when a zero bit is changed to a one [5].

Denoting by $\mathbf{e_k}$ the binary string with a one in position k and zeros elsewhere, then we have for integer valued monotonic functions, f:

$$x_k = 1 \Longrightarrow f(x) \geq f(x \oplus \mathbf{e_k}) + 1$$

for all strings x.

Theorem 5.1. *Let f be a monotonic function. Then the Voting algorithm optimises f in $O(|\operatorname{Im} f|^2 \log n)$ function evaluations.*

Proof. As with the case of OneMax, we have the probability that the winner of a tournament between two random strings x and y has a one in position k is equal to

$$\Pr(x_k = 1 \mid x \text{ wins})$$
$$= \frac{1}{2} \Pr(x \text{ wins} \mid x_k = 1, y_k = 1) + \frac{1}{2} \Pr(x \text{ wins} \mid x_k = 1, y_k = 0)$$
$$\geq \frac{1}{4} + \frac{1}{2} \Pr(f(x) > f(y) \mid x_k = 1, y_k = 0)$$
$$\geq \frac{1}{4} + \frac{1}{2} \Pr(f(x \oplus \mathbf{e_k}) + 1 > f(y) \mid x_k = 1, y_k = 0)$$

(where we consider a worst case scenario that the bit in question gains only an increase of one in the fitness of x)

$$= \frac{1}{4} + \frac{1}{2} \Pr(f(y) - f(x \oplus \mathbf{e_k}) = 0 \mid x_k = 1, y_k = 0)$$
$$+ \frac{1}{2} \Pr(f(y) - f(x \oplus \mathbf{e_k}) < 0 \mid x_k = 1, y_k = 0)$$
$$= \frac{1}{2} + \frac{1}{4} \Pr(f(y) - f(x \oplus \mathbf{e_k}) = 0 \mid x_k = 1, y_k = 0)$$

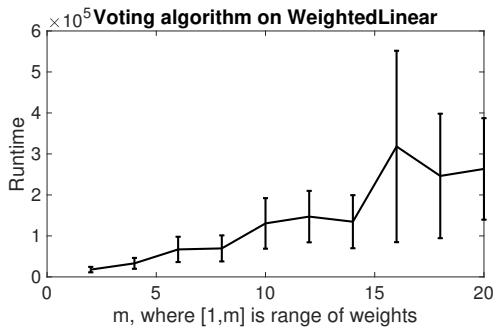

Figure 1: Runtime of the Voting algorithm on linear functions, with weights chosen randomly from the range $1, \ldots, m$.

(by symmetry, since both y and $x \oplus \mathbf{e_k}$ are conditioned to have a zero in bit k)

$$= \frac{1}{2} + \frac{1}{4}\Pr(f(y) = f(x \oplus \mathbf{e_k}) \mid x_k = 1, y_k = 0)$$
$$= \frac{1}{2} + \frac{1}{4}\sum_{\phi \in \Phi} \Pr(f(y) = \phi \mid y_k = 0)\Pr(f(x \oplus \mathbf{e_k}) = \phi \mid x_k = 1)$$

(where by Φ we denote the set of all values f can take on, conditioned on bit k of its argument having value zero)

$$= \frac{1}{2} + \frac{1}{4}\sum_{\phi \in \Phi} \Pr(f(y) = \phi \mid y_k = 0)^2$$
$$\geq \frac{1}{2} + \frac{1}{4|\Phi|}$$
$$\geq \frac{1}{2} + \frac{1}{4|\operatorname{Im} f|}$$

(using the Cauchy-Schwarz inequality). The result follows as before. □

In the case of linear functions in which there is a set of positive integer weights W, and

$$f(x) = \sum_{i=1}^{n} w_i x_i$$

we have $|\operatorname{Im} f| \leq 1 + \sum_{i=1}^{n} w_i$ to give a runtime bound of

$$O((\sum_{i=1}^{n} w_i)^2 \log n) = O(\overline{w}^2 n^2 \log n)$$

where $\overline{w}$ is the average of the weights. We see that the Voting algorithm can solve linear functions in polynomial time, as long as the average of the weights is polynomial.

We can see this is an over-estimate in the case of OneMax. This is due to the use of the Cauchy-Schwarz inequality, which avoids the need for detailed combinatorial analysis, but gives a weaker bound.

To test how the runtime depends on the weights, we ran experiments with $n = 100$, choosing weights uniformly at random from the range $1, \ldots, m$. The results, shown in figure 1, indicate that the dependency on the average weight is closer to linear than quadratic.

6 LEADINGONES

The observation that the Voting algorithm always samples strings with close to $n/2$ ones, makes it clear that it cannot efficiently solve the LeadingOnes problem.

$$\textsc{LeadingOnes}(x) = \sum_{i=1}^{n}\prod_{j=1}^{i} x_i$$

However, we can still use the voting idea to create a reasonably efficient algorithm. We do this by voting for one bit at a time, starting from the most significant bit. We also make use of truncation selection, rather than tournament selection. This variant is shown as Algorithm 2. The vector z keeps track of which bits have been determined. At each iteration, μ strings are generated using the predetermined bits recorded in z, and generating the remainder randomly. The best third of these are kept, and we determine which bit (that has not already been set) has the largest vote. The value of this one is then set by the vote and recorded in z.

```
Let z = (−1, ..., −1);
repeat
    Let pop be the empty population;
    repeat
        for 1 ≤ i ≤ n do
            if z_i = −1 then
                x_i = 0 or 1 uniformly at random ;
            else
                x_i = z_i;
            end
        end
        Add x to pop;
    until μ times;
    Rank pop by fitness, and keep only the best μ/3 strings;
    Let v be the sum of all strings in pop;
    Let k be the index for which z_k = −1 and |v_k − μ/6| is
      maximised;
    Let z_k = [v_k > μ/6]
until n times;
Return z;
```

Algorithm 2: The Significant Bit Voting Algorithm

This algorithm runs in μn time, and works by determining the bit values of the final solution one at a time. We will set $\mu = (c+2)\log n$. When applied to LeadingOnes a number of things could go wrong. Firstly, not enough strings in the top third of the population might have the next correct bit value. However, since the probability of generating the correct bit in the population is $1/2$, the probability that the top third does not contain only the correct next bit value is $O(1/n^{c+2})$. It might also happen by chance that one of the other bit positions also gets only one value represented in the top third of the population. Again, the probability that this happens is $O(1/n^{c+2})$. There are at most n things that could go wrong in each iteration, which means the probability of getting an incorrect bit in an iteration is $O(1/n^{c+1})$. Therefore, by the union bound, the probability of not returning the correct solution is $O(1/n^c)$. We have thus shown:

Theorem 6.1. *The Significant Bit Voting algorithm solves LeadingOnes (with high probability) in $O(n \log n)$ function evaluations.*

It should be noted that this result depends on the fine balance between population size and selection pressure, which needs to ensure we get the leading bit right, but is very unlikely to accidentally set an incorrect bit value.

7 EXPERIMENTS

We now consider the use of voting as a practical method to enhance the performance of population based algorithms. The idea is that the population may be "pointing" to the correct solution long before it actually converges on it. For example, for UMDA, it is possible that the bit frequencies are all in the correct direction before it has produced a single copy of the optimum. To explore this idea, we look at three different algorithms (UMDA, PCEA and cGA) and empirically study the vote of the population at each iteration. For the purposes of these experiments, the vote has no effect on the running of the algorithm; we simply report the fitness of the population vote at each iteration.

For each experiment, we fixed $n = 200$ and examined the runtime (in terms of number o ffunction evaluations) of the algorithms for different population sizes. For UMDA [9], we used truncation selection to pick the best $\lambda/2$ strings at each iteration. For PCEA we used tourmanent selection between each pair of generated offspring as described in [10]. For cGA, the parameter K represents the population size, although only two individuals are created at each iteration [8]. The vote is performed by looking at the bit probabilities.

In each plot, we show error bars of one standard-deviation. Each relevant comparison has been tested using the Mann-Whitney test, and found to be significant at the 95% level.

(1) Set $t \leftarrow 0$.
(2) $p(0) = (0.5, \ldots, 0.5)$
(3) Sample $p(t)$ λ times to form population.
(4) Set $t \leftarrow t + 1$.
(5) Let $q^1, q^2, \ldots, q^\lambda$ be the population sorted according to fitness.
(6) For each $i = 1, \ldots, n$ do the following:
(7) Let $X_i = \sum_{j=1}^{\mu} q_i^j$
(8) $p(t)_i = \frac{X_i}{\mu}$.
(9) If $p(t)_i < 1/n$, set $p(t)_i = 1/n$.
(10) If $p(t)_i > 1 - 1/n$, set $p(t)_i = 1 - 1/n$.
(11) If $p(t)_i = 0.5$ then set v_i to be 0 or 1 at random.
(12) If $p(t)_i \neq 0.5$, then set $v_i = [p(t) > 0.5]$.
(13) Report vote v.
(14) Continue at 3.

Algorithm 3: Voting UMDA

7.1 Experiments on OneMax

The results on OneMax (without any added noise) for $n = 200$ are shown for UMDA in figure 2; for PCEA in figure 3; and for cGA in figure 4. It can be seen in all cases, the vote improves

(1) Initialise a random population of λ strings.
(2) Choose parents X^α and X^β uniformly from population.
(3) Generate a random vector $a = \{a_1, a_2, ..., a_n\} \in \{0, 1\}^n$.
(4) Create complementary children X^μ and $\overline{X^\mu}$ by uniform crossover such that
$$X_i^\mu = a_i X_i^\alpha + (1 - a_i) X_i^\beta$$
$$\overline{X_i^\mu} = (1 - a_i) X_i^\alpha + a_i X_i^\beta$$
(5) The better of the offspring goes to the next generation.
(6) Repeat steps 2 to 5 times to create all the members of the next population $q^1, \ldots, q^\lambda$.
(7) Let $X_i = \sum_{j=1}^{\lambda} q_i^j$
(8) $p(t)_i = \frac{X_i}{\lambda}$.
(9) If $p(t)_i = 0.5$ then set v_i to be 0 or 1 at random.
(10) If $p(t)_i \neq 0.5$, then set $v_i = [p(t) > 0.5]$.
(11) Report vote v.
(12) Repeat from (2) until termination condition reached.

Algorithm 4: Voting PCEA

(1) Set $t = 0$.
(2) $p(0) = (0.5, \ldots, 0.5)$
(3) For all i, set $x_i = 1$ with probability $p(t)_i$ and $x_i = 0$ with probability $1 - p(t)_i$.
(4) For all i, set $y_i = 1$ with probability $p(t)_i$ and $y_i = 0$ with probability $1 - p(t)_i$.
(5) If $f(x) < f(y)$, swap x and y.
(6) For all $i \in \{1, 2, \ldots, n\}$
  (a) If $x_i > y_i$, set $p(t+1)_i = p(t)_i + \frac{1}{K}$.
  (b) If $x_i < y_i$, set $p(t+1)_i = p(t)_i - \frac{1}{K}$.
  (c) If $x_i = y_i$, set $p(t+1)_i = p(t)_i$.
(7) If $p(t)_i = 0.5$ then set v_i to be 0 or 1 at random.
(8) If $p(t)_i \neq 0.5$, then set $v_i = [p(t) > 0.5]$.
(9) Report vote v.
(10) Set $t = t + 1$ and continue at 3.

Algorithm 5: Voting cGA

the performance for larger population sizes, with the effect being significant for all the considered algorithms. This is because for these algorithms, when the population is small, it often happens that some of the bit probabilities will go the wrong value, where they meet the lower margin (set to $1/n$ in all cases). By the time these have recovered, the rest of the bit values are at the upper margin (set to $1 - 1/n$). When all except one or two bits are at the upper margin, and the remaining bits just below $1/2$, it is likely that the optimum solution will be produced, even though the vote will be incorrect. When the population sizes are larger, this does not happen, and each bit probability quickly exceeds $1/2$, giving the correct vote.

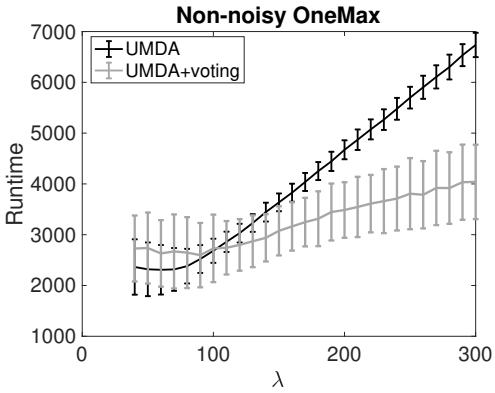

Figure 2: Comparison of UMDA and UMDA+voting while solving the (non-noisy) OneMax function

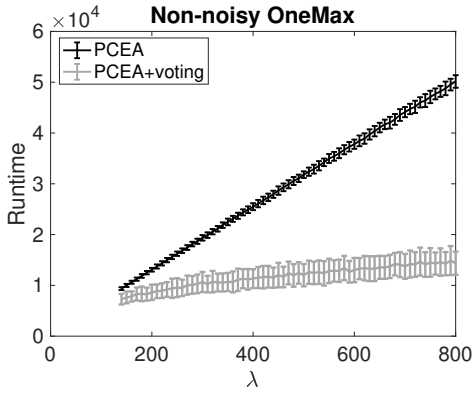

Figure 3: Comparison of PCEA and PCEA+voting while solving the (non-noisy) OneMax function

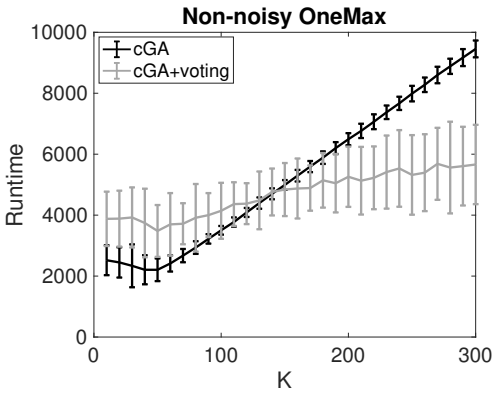

Figure 4: Comparison of cGA and cGA+voting while solving the (non-noisy) OneMax function

The runtime is analysed for UMDA and PCEA while population size λ is varied. Similarly, the parameter K in cGA is varied to analyse how it affects the runtime. The voting algorithm solves the non-noisy OneMax problem on an average over 100 runs, in 13110 function evaluations.

7.2 Experiments on noisy OneMax

For the experiments with noisy OneMax for $n = 200$, we use Gaussian noise with $\sigma = 5$. The results are shown for UMDA in figure 5; for PCEA in figure 6; and for cGA in figure 7. Again, we see that the voting improves the algorithms for large population sizes. For UMDA with a small population, there is little difference with and without voting.

The voting algorithm solves the noisy OneMax with $\sigma = 5$ problem on an average over 100 runs, in 17943 function evaluations.

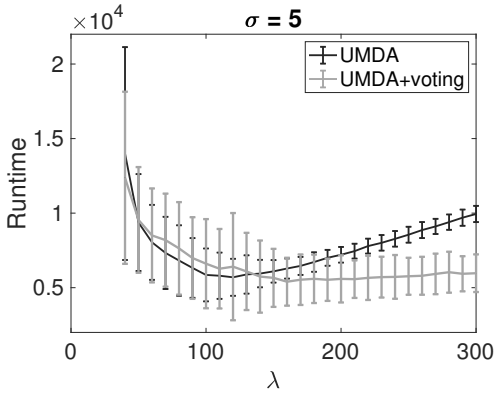

Figure 5: Comparison of UMDA and UMDA+voting while solving the noisy OneMax function

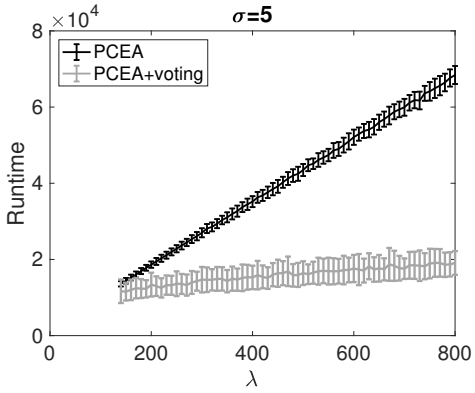

Figure 6: Comparison of PCEA and PCEA+voting while solving the noisy OneMax function

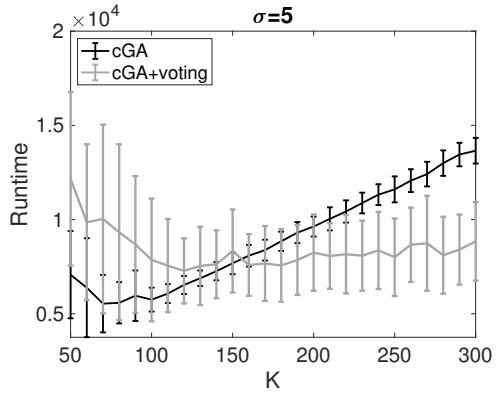

Figure 7: Comparison of cGA and cGA+voting while solving the noisy OneMax function

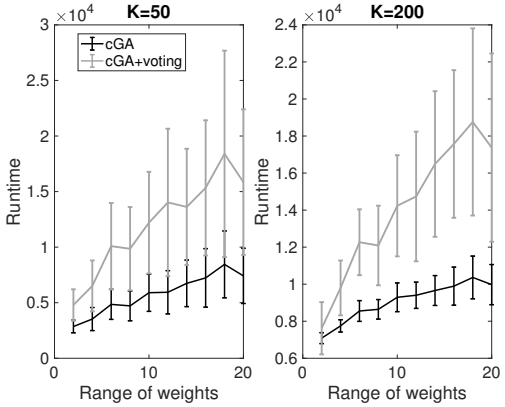

Figure 9: Comparison of cGA and cGA+voting while solving the non-noisy WeightedLinear function

7.3 Experiments on non-noisy WeightedLinear

For the non-noisy WeightedLinear problem for $n = 200$, the results are illustrated in Figure 8 for UMDA; Figure 9 for cGA and Figure 10 for PCEA. Random problem instances are chosen with weights ranging from $1, \ldots, m$ with m varying from 1 to 20.

For UMDA, population sizes $\lambda = 50$ and 200 are chosen, representing the regimes where voting does not and does help in OneMax respectively. Similarly, for cGA the different regimes are analysed by considering $K = 50$ and 200. The algorithms are able to find the optimum in each cases.However, we see that the voting mechanism does not help as the weights get larger, which is consistent with our analysis of the Voting algorithm on such problems.

However, for such small values of population sizes, the PCEA cannot solve the WeightedLinear problem. According to the theoretical result in [10], we choose the population size $\lambda = 10 \times \sqrt{n} \times \log n$. Considering this choice of λ, both PCEA with and without voting solves the problems.

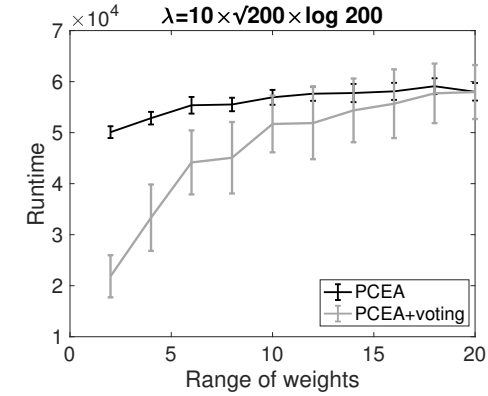

Figure 10: Comparison of PCEA and PCEA+voting while solving the non-noisy WeightedLinear function

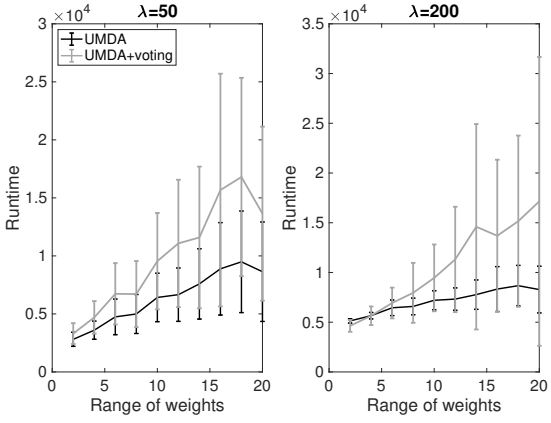

Figure 8: Comparison of UMDA and UMDA+voting while solving the non-noisy WeightedLinear function

8 CONCLUSION

We have studied the use of voting as a heuristic method. It is particularly effective for the noisy OneMax problem, where we have proved an upper bound on the runtime better than any other algorithm we are aware of. A variant of the voting idea which works one bit at a time is reasonably efficient for LeadingOnes. The approach works less well for general linear problems, and we have investigated this effect, in fact showing an upper bound for general monotonic functions. Finally, we have empirically studied the idea of incorporating voting into a population-based algorithm and conclude that this may be effective for large population sizes.

ACKNOWLEDGMENTS

The computations described in this paper were performed using the University of Birmingham's BlueBEAR HPC service, which provides a High Performance Computing service to the University's research community.

REFERENCES

[1] Youhei Akimoto, Sandra Astete-Morales, and Olivier Teytaud. 2015. Analysis of runtime of optimization algorithms for noisy functions over discrete codomains. *Theoretical Computer Science* 605 (2015), 42 – 50. https://doi.org/10.1016/j.tcs.2015.04.008

[2] Gautham Anil and R Paul Wiegand. 2009. Black-box search by elimination of fitness functions. In *Proceedings of the tenth ACM SIGEVO workshop on Foundations of genetic algorithms*. ACM, 67–78.

[3] Duc-Cuong Dang and Per Kristian Lehre. 2015. Efficient optimisation of noisy fitness functions with population-based evolutionary algorithms. In *Proceedings of the 2015 ACM Conference on Foundations of Genetic Algorithms XIII*. ACM, 62–68.

[4] Lawrence Davis. 1991. Bit-climbing, representational bias, and test suit design. In *Proc. Intl. Conf. Genetic Algorithm, 1991*. 18–23.

[5] Benjamin Doerr, Thomas Jansen, Dirk Sudholt, Carola Winzen, and Christine Zarges. 2013. Mutation rate matters even when optimizing monotonic functions. *Evolutionary computation* 21, 1 (2013), 1–27.

[6] Tobias Friedrich, Timo Kötzing, Martin S. Krejca, Samadhi Nallaperuma, Frank Neumann, and Martin Schirneck. 2016. Fast Building Block Assembly by Majority Vote Crossover. In *Proceedings of the ACM Genetic and Evolutionary Computation Conference (GECCO)*. ACM Press, 661–668.

[7] Tobias Friedrich, Timo Kötzing, Martin S Krejca, and Andrew M Sutton. 2017. The compact genetic algorithm is efficient under extreme gaussian noise. *IEEE Transactions on Evolutionary Computation* 21, 3 (2017), 477–490.

[8] Georges R Harik, Fernando G Lobo, and David E Goldberg. 1999. The compact genetic algorithm. *IEEE transactions on evolutionary computation* 3, 4 (1999), 287–297.

[9] Heinz Mühlenbein. 1997. The equation for response to selection and its use for prediction. *Evolutionary Computation* 5, 3 (1997), 303–346.

[10] Adam Prugel-Bennett, Jonathan Rowe, and Jonathan Shapiro. 2015. Run-time analysis of population-based evolutionary algorithm in noisy environments. In *Proceedings of the 2015 ACM Conference on Foundations of Genetic Algorithms XIII*. ACM, 69–75.

[11] Pantelimon Stanica. 2001. Good lower and upper bounds on binomial coefficients. *Journal of Inequalities in Pure and Applied Mathematics* 2, 3 (2001), 30.

[12] H Vogt. 1983. Unimodality of differences. *Metrika* 30, 1 (1983), 165–170.

[13] Darrell Whitley, Swetha Varadarajan, Rachel Hirsch, and Anirban Mukhopadhyay. 2018. Exploration and Exploitation Without Mutation: Solving the Jump Function in $\Theta(n)$ Time. In *International Conference on Parallel Problem Solving from Nature*. Springer, 55–66.