Comparison and modification of self-adjusting evolutionary algorithms

Mario Alejandro Hevia Fajardo
Department of Computer Science
University of Sheffield, UK
maheviafajardo1@sheffield.ac.uk

September 12, 2018

COM6912
Supervisor Dr Dirk Sudholt

This report is submitted in partial fulfilment of the requirement for the degree of MSc Data Analytics by Mario Alejandro Hevia Fajardo

Abstract

Evolutionary algorithms (EA) are population-based optimization algorithms that mimic evolution to search for an (near-)optimal solution. A key topic in EA research is the amount of evaluations or generations used to solve a problem. The research is generally focused on parameter optimization and most of the research focuses on static parameters, but lately started to focus on dynamic parameters that are success based and does not need information about the problem.

The aim of the project is to make empirical evaluations of some of these success based parameter optimizitations. The main algorithms that are tested in the project are the self-adjusting mutation rate $(1 + \lambda)$ EA (Doerr et al., 2017), self-adjusting offspring population size $(1 + \lambda)$ EA (Jansen et al., 2005) and the self-adjusting $(1 + (\lambda, \lambda))$ GA (Doerr and Doerr, 2015). They are compared with the toy problems OneMax, LeadingOnes, Makespan Scheduling and SufSamp. Afterwards several modifications are made to find insights and improvements on the original algorithms.

In this work is shown a comparison and discussion of the optimization times, and a proposed new metric to capture the parallel and sequential optimization time. From the modifications made a new selection scheme emerged that improves up to 20% in the number of evaluations needed by the self-adjusting $(1 + (\lambda, \lambda))$ GA on OneMax.

Acknowledgements

I would like to thank to my supervisor Dirk Sudholt for guiding and assisting me through the entire project.

Contents

List of Figures

List of Tables

1 Introduction

Evolutionary algorithms (EA) are population-based optimization algorithms that mimic evolution to search for an (near-)optimal solution. They are often used when a problem is hard to solve using other algorithms or a good approximation is sufficient (Dyer, 2008). Even though there exist a wide range of problems and applications with these characteristics, EAs have two requirements to work. Firstly there needs to be a way to encode the possible solutions of a problem (a common encoding used is a bit string) and secondly there need to be a way to evaluate these solutions (fitness function) and such evaluation should return a score.

For some real-world problems the evaluation of a solution is computationally-intensive, therefore a key topic in EA research is the amount of evaluations or generations used to solve a problem. The research is generally focused on parameter optimization (tuning) since it has been observed that this can improve the performance of EAs. Most of the research focuses on static parameters, but lately started to focus on dynamic parameters that depend directly on the fitness of the current best individual because it has been proved to lead to faster optimizations, but for this strategies to work a good understanding of the problem is needed in order to create a suitable dependence of the parameters. This is often not feasible, therefore other reaserchers have focused on dynamic parameters that are success based and does not need information about the problem.

The aim of the project is to make empirical evaluations of some of these success based parameter optimizitations in the $(1 + \lambda)$ EA and compare them on common toy problems. The main algorithms that are tested in the project are the self-adjusting mutation rate $(1 + \lambda)$ EA (Doerr et al., 2017), self-adjusting offspring population size $(1 + \lambda)$ EA (Jansen et al., 2005) and the $(1 + (\lambda, \lambda))$ GA (Doerr and Doerr, 2015). In the first part of the document a description of relevant research is given, afterwards a description of the optimization algorithms that are studied is shown, these include the algorithms mentioned above with modifications and combinations of them. Then they are compared and refined using the most simple toy problem (OneMax), since it is the most studied problem for all the algorithms, subsequently three other toy problems are compared (LeadingOnes and Makespan Scheduling). Finally a comparison with the SufSamp problem is made, which is a more complex problem; in this last comparison the algorithms are not only evaluated on the optimization time but in their ability to get to the best result possible too.

2 Literature Review

2.1 Evolutionary Algorithms

Evolutionary algorithms are a family of algorithms that are inspired by natural evolution. These algorithms are used to search in a space of solutions; solutions are initialized randomly to create a population and then evolutionary operations (usually recombination and mutation) are used to explore into the unknown space of solutions to finally select from these solutions based on a fitness value or rule. This process is repeated continuously until a termination criterion is fulfilled; each iteration is called a generation (Bartz-Beielstein et al., 2014). A well-rounded definition for EAs is given in Beyer et al. (2002) as follows:

"collective term for all variants of (probabilistic) optimization and approximation algorithms that are inspired by Darwinian evolution. Optimal states are approximated by successive improvements based on the variation-selection-paradigm. Thereby, the variation operators produce genetic diversity and the selection directs the evolutionary search."

From this family of algorithms there are several strategies used depending on the operations used and the population size. One of the most simple is the $(1+1)$ EA which uses one parent, mutates it to generate one offspring and then selects the next parent comparing and selecting the fittest. A more general strategy and the focus of this study is the $(1+\lambda)$ EA that uses only one parent and λ offspring, every generation chooses the new parent based on an elitist rule, a pseudocode representation of this EA is represented in Algorithm 1.

Algorithm 1: The $(1+\lambda)$ EA with offspring population size λ and mutation probability p.

```
1 Initialization: Choose $x \in \{0,1\}^n$ uniformly at random.
2 Optimization: for $t \in \{1,2,\dots\}$ do
3     Mutation: for each $i \in \{1,\dots,\lambda\}$ do
4         Create $x'_i \in \{0,1\}^n$ by copying $x$ and, independently for each bit, flip this
          bit with probability $p$.
5     Selection: if $\max\{f(x'_1),\dots,f(x'_\lambda)\} \geq f(x)$ then
6         Replace $x$ by some randomly chosen $x'_i$ with maximal $f$-value.
```

Given a certain problem different strategies are better than others and even between the same strategy, parameters such as mutation rate, play an important role in the obtained results. This has created a research area around the tuning and optimization of these parameters of EAs.

2.2 Evaluation of EAs

There are two main approaches to evaluate an EA, the first one and the most obvious is to evaluate it with their ability to achieve the optimum within a certain number of function evaluations, this can be evaluated as a percentage of runs that find the optimum (Success Rate) or the average of the fitness obtained by the fittest individual in all runs (Mean Best Fitness). The second approach is to evaluate the average of function evaluations or generations necessary to solve the problem. Even though for real world problems the first type would be the most important criterion, in research and in this project the main evaluation used is the second.

The number of function evaluations is selected as a metric in this work because, as mentioned before, some problem evaluations are computationally expensive, therefore is important that the algorithms use the least possible evaluations. Is worth noticing that in algorithms such as $(1+\lambda)$ EA the number of generations is important too, since the evaluations can be made via parallel computing to improve the overall sequential time (real-time processing) and can even work when using adaptive populations with the model shown by Lässig and Sudholt (2011)

2.3 The (1 + λ) EA

In the literature there are several studies on the selection of the best parameters for the (1 + λ) EA. Between these studies the most studied paramater has been the mutation rate.

2.3.1 Mutation rate in (1 + λ) EA

The mutation probability p is described as the independent probability of fliping each bit, and its formula is $p = c/n$ with mutation rate c and number of bits n. Traditionally the mutation probability of $p = 1/n$ has been used in (1 + 1) EA as a rule of thumb, which is a special case of (1 + λ) EA where $\lambda = 1$, but this was not formally proven to be the best choice. The first to prove a tight (up to lower order terms) bound of $(1 \pm o(1))\frac{e^c}{c} n \ln n$ for all linear functions was Witt (2013); given that this bound has the factor $\frac{e^c}{c}$ that depends on the mutation rate, it can be derived that the mutation probability $p = c/n$ with $c = 1$ is optimal to minimize the specified bound.

This optimal probability was later generalized for the (1 + λ) EA on the OneMax function by Gießen and Witt (2015) were they found that the expected parallel time (number of generations) is equal to

$$(1 \pm o(1)) \left(\frac{e^c}{c} \cdot \frac{n \ln n}{\lambda} + \frac{1}{2} \cdot \frac{n \ln \ln \lambda}{\ln \lambda} \right)$$

from this we can infer that if the value of λ is smaller than a certain threshold, the first part of the function is predominant and the optimal mutation rate is still $c = 1$, and after that threshold the value of c does not change the expected runtime; the threshold value of λ is $\lambda = \ln(n) \ln(\ln(n)) / \ln(\ln(\ln(n)))$.

Even though these results help us choosing parameters, they do not imply that for every problem and every algorithm the best choice of mutation rate is $c = 1$, an example of this is shown by Böttcher et al. (2010) where they demonstrated that the exact optimal value for the mutation probability in LeadingOnes is $p \approx 1.59/n$; another example is the work of Sudholt (2017) where he finds that for a (2+1) GA the optimal mutation probability is $p \approx 1.618/n$ on OneMax.

Furthermore Badkobeh et al. (2014) demonstrated that an implementation of adaptive schemes for the mutation rate is faster compared to the mutation rate $c = 1$ in the $(1 + \lambda)$ EA for the OneMax problem. In this study the mutation rate was fitness-dependent which means that in order to calculate the mutation rate a good knowledge of the problem at hand is needed.

Finally this problem was overcome by Doerr et al. (2017) where they used a self-adjusting mutation rate that only needs information of the success in the last offspring population; they rigorously proved that this algorithm achieved the same expected number of fitness evaluations as in Badkobeh et al. (2014) for the OneMax problem.

The (1 + λ) Evolutionary Algorithm with Self-Adjusting Mutation Rate

The basic idea behind this algorithm (Doerr et al., 2017) is to create $\lambda/2$ offspring with a mutation probability $p = c/(2n)$ and the other $\lambda/2$ offspring with $p = 2c/n$. The mutation rate c is initialized as $c^{init} \geq F$ (F being the update factor) and adjusted at every iteration with 50% probability with c equals to c/F or Fc depending on the mutation probability of the fittest ofspring, and 50% probability to a random value in $\{c/F, Fc\}$. If at any time the mutation rate goes outside of the boundaries $[F, 1/(2F)]$, the mutation rate is replaced by the boundary exceeded. A similar pseudocode as given by Doerr et al. (2017) is shown in Algorithm 2

In their study a value of 2 in F was used and they suggest that a smaller F could give better results but that it might not be worth to optimize it since the influence of the parameter is not very large.

Algorithm 2: The $(1 + \lambda)$ EA with offspring population size λ and two-rate standard bit mutation.

```
1  Initialization: Choose $x \in \{0,1\}^n$ uniformly at random and set $c \leftarrow c^{init}$
2  Optimization: for $t \in \{1, 2, \dots\}$ do
3      Mutation:
4      for each $i \in \{1, \dots, \lambda\}$ do
5          $c_t = c/2$ if $i \leq \lambda/2$ and $c_t = 2c$ otherwise;
6          Create $x'_i \in \{0,1\}^n$ by copying $x$ and, independently for each bit, flip this
           bit with probability $p = c_t/n$;
7      Selection:
8      if $\max\{f(x'_1), \dots, f(x'_\lambda)\} \geq f(x)$ then
9          Replace $x$ by some randomly chosen $x'_i$ with maximal $f$-value;
10         Replace $c^*$ with $c/F$ if $c_t = c/2$ of chosen $x'_i$ or $Fc$ otherwise;
11     Update:
12     Perform one of the following two actions with probability 1/2:
           • Replace $c$ with $c^*$;
           • Replace $c$ with either $c/F$ or $Fc$ , each with probability 1/2;
       Replace $c$ with min{max{F,$c$},$1/(2F)$};
```

2.3.2 Offspring population in $(1 + \lambda)$ EA

The offspring population has been less studied than the mutation rate, but still there have been a wide range of studies on the behaviour of this parameter especially in the $(1 + \lambda)$ EA. In Jansen et al. (2005) they made a broad study of the impact of λ; they found that for simple landscapes setting λ in the range of $1 \leq \lambda \leq \log n$ grant asymptotically equivalent sequential optimization times, and especifically for OneMax and LeadingOnes there is no benefit in increasing λ beyond $\lambda = 1$. In contrast with these landscapes the authors also showed that for more complex problems there is a benefit on using large values for λ; a tailored function was presented with such characteristics, the function is called SufSamp.

In the study above the focus was made on sequential processing time but for parallel processing time Gießen and Witt (2015) demonstrated that increasing λ generates a linear improvement in parallel optimization time on OneMax until a certain cut-off value of $\lambda = o(\ln(n)\ln(\ln(n))/\ln(\ln(\ln(n))))$. Given the knowledge that an improvement can be made by parallelization several models to exploit this have been studied in Lässig and Sudholt (2010b) and Lässig and Sudholt (2010a). Another important study is Lässig and Sudholt (2011) were the authors describe a parallel model that can be used with adaptive offspring population, they used two adaptive schemes, in the first scheme the population would be doubled or reduced to one depending on the success of the previous generation and the second scheme the offspring population would be halved instead of reduced to one in case of success. Other similar adaptive approaches that can be parallelized with the same model are used by Jansen et al. (2005) in $(1 + \lambda)$ EA and Doerr et al. (2015) in $(1 + (\lambda, \lambda))$ GA where the offspring population would adapt depending on the success of previous generation, but in these cases λ would be multiplied and divided by a different number. In the first work the authors multiplied by 2 and divide by λ/s with s being the number successful offspring, the reasoning given in their work was to set λ to roughly the reciprocal of the success probability, minimizing the total expected optimization time. For the $(1 + (\lambda, \lambda))$ GA they used the one-fifth success rule and proved to give greater improvements than other factors used to modify λ. Since the algorithm from Jansen et al. (2005) is used in this work, a more detailed description is given in Algorithm 3, based on the pseudocode they give.

There are studies where the population is regulated, not by explicit decision but rather by adding a natural behaviour such as aging, this operation eliminates an individual when it reaches a certain predefined age. A study in this and other similar aging models were compared in Eiben et al. (2010), even though they are not used in this work, they are worth looking to be implemented on future work.

Algorithm 3: The $(1 + \lambda)$ EA with self-adjusted offspring population size λ and mutation probability p.

```
Initialization: Choose $x \in \{0,1\}^n$ uniformly at random and set $\lambda := 1$
Optimization: for $t \in \{1, 2, \dots\}$ do
    Mutation:
    for each $i \in \{1, \dots, \lambda\}$ do
        Create $x'_i \in \{0,1\}^n$ by copying $x$ and, independently for each bit, flip this bit with probability $p = 1/n$;
    Counting Successes for the Adaptation of $\lambda$:
    $s := |\{x'_i \mid f(x'_i) \geq f(x) \wedge x'_i \neq x\}|$;
    Selection:
    if $\max\{f(x'_1), \dots, f(x'_\lambda)\} \geq f(x)$ then
        Replace $x$ by some randomly chosen $x'_i$ with maximal $f$-value;
    Update:
    if $s > 0$ then
        $\lambda := \lambda/s$;
    else
        $\lambda := 2\lambda$
```

2.4 Crossover and Genetic Algorithms (GA)

One of the main operators of EAs is the crossover (also called recombination). This operator is inspired by sexual reproduction in nature, and essentially it combines two parents into one offspring; there are several ways to implement crossover, and empirical research has been made in Picek and Golub (2010) showing that some of them perform better for certain problems while worse in others. From a range of seven crossover operators they concluded that uniform crossover performs better in almost all problems tested, following these results this operator is used in the experiments of this work. To give a better understanding of crossover operators a brief explanation of three of the most used is given below, for further information you can refer to Picek and Golub (2010).

Single-Point Crossover

The single-point crossover splits both parents at a randomly chosen point in the bit-string and then combines the first part of the first parent with the second part of the second parent. For example, the bit strings x_1 and x_2 would make an offspring x_3 as follows:

$$x_1 = 0110010110010, \;\; x_2 = 1110101100011$$

Selecting a randomly chosen point to split the strings

$$x_1 = 0110|010110010, \;\; x_2 = 1110|101100011$$

Taking the first part of x_1 and the second part of x_2 it ends up with

$$x_3 = 0110101100011$$

Two-Point Crossover

When applying crossover with two-point crossover the parents are split into three parts by two randomly chosen points and the offspring is generated with the first and third part of one parent and the second part of the other. Following the previous example two-point crossover would be:

$$x_1 = 0110010110010, \;\; x_2 = 1110101100011$$

Selecting two randomly chosen points to split the strings

$$x_1 = 0110|0101100|10, \;\; x_2 = 1110|1011000|11$$

Taking the first and third part from x_1 and the second part from x_2 it ends up with

$$x_3 = 0110101100010$$

Uniform Crossover

In uniform crossover a generalization of the single-point and two-point crossover is made where the offspring is formed selecting each bit independently at random from either parents with a probability p_c. Using the same parents as before uniform crossover would work like this:

$$x_1 = 0110010110010, \;\; x_2 = 1110101100011$$

Spliting the strings into bits

$$x_1 = 0|1|1|0|0|1|0|1|1|0|0|1|0, \;\; x_2 = 1|1|1|0|1|0|1|1|0|0|0|1|1$$

Taking for each bit independently at random from either x_1 or x_2 it ends up with

$$x_3 = 0110001100010$$

Crossover has been believed to improve the optimization time and this was empirically proved by Picek and Golub (2010), in the past decade there have been several studies proving that crossover was superior than EAs with only mutation but only on artificially constructed functions, up until the last year where Sudholt (2017) proved that crossover is at least twice as fast for building-block functions, attributing this to neutral mutations (mutations that doesn't affect directly the fitness) that can be stored in parents and combined later on. In general GAs benefit from parents with different building blocks that are mixed in the offspring, but a problem arises if the parents are the same since the crossover does not bring any benefits. This and other problems in GAs were ingeniously solved by Doerr et al. (2015) with the $(1 + (\lambda, \lambda))$ GA.

2.4.1 (1 + (λ, λ)) GA

The (1 + (λ, λ)) GA is an algorithm with only one parent but it still manages to implement crossover and mutation. It works by first mutating the parent λ times, then applying a uniform crossover λ times between the parent and the fittest mutated offspring, in the end it performs an eletist selection. In this algorithms the value of λ is self-adjusted with the one-fifth success rule which is mentioned in Section 2.3.2. The Algorithm 4 and Algorithm 5 shows pseudocode as given in Doerr et al. (2015) (changing the update factor for G since F is used for the update factor in mutation rate on this work) with and without the self-adjusted parameters.

Algorithm 4: The (1 + (λ, λ)) GA with offspring population size λ, mutation probability p, and crossover probability c.

```
1  Initialization: Sample $x \in 0, 1^n$ uniformly at random and query $f(x)$;
2  Optimization: for $t = 1, 2, \dots$ do
3      Mutation phase:
4      Sample $\ell$ from $\mathcal{B}(n, p)$;
5      for $i = 1, \dots, \lambda$ do
6          Sample $x^{(i)} \leftarrow \text{mut}_\ell(x)$ and query $f(x^{(i)})$;
7      Choose $x' \in \{x^{(1)}, \dots, x^{(\lambda)}\}$ with $f(x') = \max\{f(x^{(1)}), \dots, f(x^{(\lambda)})\}$ u.a.r.;
8      Crossover phase:
9      for $i = 1, \dots, \lambda$ do
10         Sample $y^{(i)} \leftarrow \text{cross}_c(x, x')$ and query $f(y^{(i)})$;
11     If exists, choose $y \in \{y^{(1)}, \dots, y^{(\lambda)}\} \backslash \{x\}$ with $f(y) = \max\{f(y^{(1)}), \dots, f(y^{(\lambda)})\}$
       u.a.r.;
12     otherwise, set $y := x$;
13     Selection step:
14     if $f(y) \geq f(x)$ then $x \leftarrow y$;
```

Studying this algorithm with adaptive population the authors of Doerr and Doerr (2015) demonstrated that it yields a better expected optimization time on OneMax than any other static parameter choice and the authors suggest that further improvement can be achieved with self-adjusting mechanisms for other parameters.

2.5 The OneMax function

The OneMax function was design as a simple benchmark function. Since it is one of the simplest functions and in particular to the (1 + 1) EA it is the easiest problem (Corus et al., 2017), it has been studied profoundly. Generally it is the first problem to be studied or tested in EAs, for this work the three most important studies are (Doerr et al., 2017), (Jansen et al., 2005) and (Doerr and Doerr, 2015), where the function was studied in the algorithms tested empirically here.

The objective of this problem is to get an encoded bit-string full of $'1'$. As mentioned before in order to evaluate the solution we need a fitness function, for this problem it is calculated by counting the number of correct values (i.e. $'1'$). A formal definition would be, OneMax : $\{0, 1\} \to \mathbb{N}$ is defined by

Algorithm 5: The self-adjusting (1 + (λ, λ)) GA with mutation probability p, crossover probability c, and update strength F.

```
Initialization: Sample $x \in 0, 1^n$ uniformly at random and query $f(x)$;
Initialize $\lambda \leftarrow 1$;
Optimization: for $t = 1, 2, \ldots$ do
    Mutation phase:
    Sample $\ell$ from $\mathcal{B}(n, p)$;
    for $i = 1, \ldots, \lambda$ do
        Sample $x^{(i)} \leftarrow \text{mut}_\ell(x)$ and query $f(x^{(i)})$;
    Choose $x' \in \{x^{(1)}, \ldots, x^{(\lambda)}\}$ with $f(x') = \max\{f(x^{(1)}), \ldots, f(x^{(\lambda)})\}$ u.a.r.;
    Crossover phase:
    for $i = 1, \ldots, \lambda$ do
        Sample $y^{(i)} \leftarrow \text{cross}_c(x, x')$ and query $f(y^{(i)})$;
    If exists, choose $y \in \{y^{(1)}, \ldots, y^{(\lambda)}\} \backslash \{x\}$ with $f(y) = \max\{f(y^{(1)}), \ldots, f(y^{(\lambda)})\}$ u.a.r.;
    otherwise, set $y := x$;
    Selection and update step:
    if $f(y) > f(x)$ then $x \leftarrow y$; $\lambda \leftarrow \max\{\lambda/G, 1\}$;
    if $f(y) = f(x)$ then $x \leftarrow y$; $\lambda \leftarrow \min\{\lambda G^{1/4}, n\}$;
    if $f(y) < f(x)$ then $\lambda \leftarrow \min\{\lambda G^{1/4}, n\}$;
```

$$\textsc{OneMax}(x) := \sum_{i=1}^{n} x_i \text{ for all } n \in \mathbb{N} \text{ and all } x \in \{0,1\}^n \tag{1}$$

2.6 The LeadingOnes function

In the case of the LeadingOnes function it has the same optimum as the OneMax function but the fitness function is different. The function LeadingOnes : $\{0, 1\} \rightarrow \mathbb{N}$ is defined by

$$\textsc{LeadingOnes}(x) := \sum_{i=1}^{n} \prod_{j=1}^{i} x_j \text{ for all } n \in \mathbb{N} \text{ and all } x \in \{0,1\}^n \tag{2}$$

As seen in the function above to calculate the fitness of an individual the number of consecutive "1" is counted starting from the left, e.g. "1101111111" would give a value of 2 while the fitness of "1111000100" is 4. As one can imagine this is a more complex problem for mutation based algorithms because an almost perfect solution as the first example has a worse fitness value than the second one which can drift the algorithm away from the result.

2.7 The SufSamp function

This function was created by Jansen et al. (2005) as a problem where a EA with $\lambda > 1$ would have a faster expected optimization time than using $\lambda = 1$. The intuition behind

the function is that in a fitness landscape where there is a single narrow steep path to the global optimum and several branches guiding to a local optimum with a more gradual uphill path, a EA would need to have a sufficient large offspring population size in order to increase the probability to find the right path at every branch point B_n or jump out of a local optimum. In the function there are about $\sqrt{n}$ number of B_n that can lead to a local optimum.
The formal definition given in their work starts with a function $f : \{0,1\}^n \to \mathbb{N}$.

For $n \in \mathbb{N},\ k := \lfloor \sqrt{n} \rfloor$. The function $f : \{0,1\}^n \to \mathbb{N}$ for all $x \in \{0,1\}^n$ is:

$$f(x) := \begin{cases} (i+3)n + \textsc{OneMax}(x) & \text{if } (x = 0^{n-i}1^i \text{ with } 0 \le i \le n) \text{ or} \\ & (x = y0^{n-i-k}1^i \text{ with } i \in \{k, 2k, \dots, (k-2)k\}, y \in \{0,1\}^k) \\ 0 & \text{otherwise.} \end{cases} \tag{3}$$

With the previous definition of f a start point of 0^n is needed, in order to avoid this, they extended f into SufSamp

For $n \in \mathbb{N},\ m' := \lfloor n/2 \rfloor,\ m'' := \lceil n/2 \rceil$ .Then $\textsc{SufSamp} : \{0,1\}^n \to \mathbb{N}$ is defined as:

$$\textsc{SufSamp}(x) := \begin{cases} n - \textsc{OneMax}(x'') & \text{if } x' \ne 0^{m'} \wedge x'' \ne 0^{m''} \\ 2n - \textsc{OneMax}(x') & \text{if } x' \ne 0^{m'} \wedge x'' = 0^{m''} \\ f(x'') & \text{if } x' = 0^{m'} \end{cases} \tag{4}$$

for all $x = x_1x_2\cdots x_n \in \{0,1\}^n$ with $x' := x_1x_2\cdots x_{m'} \in \{0,1\}^{m'}$ and
$x'' := x_{m'+1}x_{m'+2}\cdots x_n \in \{0,1\}^{m''}$

2.8 The Makespan Scheduling Problem

The makespan scheduling problem is a combinatorial optimization problem, where the scheduling of n jobs in m machines takes place, with each job $1,\dots,n$ have processing times $p_1,\dots,p_n$ that could depend on the machine that it is processed on. Within this problem there are different sets of constraints that can be applied, in this work the description of Makespan Scheduling on 2 machines is used as given by Neumann and Witt (2010).

Makespan scheduling on 2 machines

There are n jobs with positive processing times $p_1, p_2, \dots, p_n$ and need to be scheduled on two identical machines in order to minimize the makespan. In order to encode the machines a bit-string $x \in \{0,1\}^n$ is used where the job i is scheduled in machine 1 if the $x_i = 0$ and on machine 2 otherwise. The fitness function would be described by:

$$f_{p_1,\dots,p_n}(x) := \max\left\{ \sum_{i=1}^{n} p_i x_i, \sum_{i=1}^{n} p_i(1-x_i) \right\} \tag{5}$$

For this problem a common practice is to initialize the weights/processing time randomly, in this work this practice arises the problem of knowing beforehand the solution in order to know when an algorithm has found it and can stop searching.

Makespan scheduling is an NP-hard problem, therefore an approximation is used here, instead of an exact solution. The algorithm used is called Longest Processing Time (LPT), it consists in sorting the jobs by processing time decreasingly, and using the sorted jobs to put every job in the currently emptier machine (Neumann and Witt, 2010). An example of this algorithm is given below for $x \in \{0,1\}^6$, with jobs $j_1, \dots, j_6$ and processing times $p_1, \dots, p_6$.

$$p_1 = 4,\ p_2 = 7,\ p_3 = 5,\ p_4 = 2,\ p_5 = 8,\ p_6 = 9$$

Sorting the jobs we get the ordered set:

$$J := \{j_6,\ j_5,\ j_2,\ j_3,\ j_1,\ j_4\}$$

Assigning each job to the currently emptier machine, the next set of jobs for each machine M_1 and M_2 its corresponding sum is achieved.

$$M_1 := \{j_6,\ j_3,\ j_1\} = 18,\ M_2 := \{j_5,\ j_2,\ j_4\} = 17$$

This solution encoded in $x \in \{0,1\}^6$ would be $x = 010110$ which is an equivalent solution to $x = 101001$ since both machines are identical.

3 Requirements and analysis

In this chapter an analysis of the main objective is made in order to describe functional and non-functional requirements. In the end of this chapter the way to test and evaluate both the algorithms and their implementation is given in detail.

3.1 Main objective

The main objective can be separated into two parts, the first part is the test of the algorithms showed in Doerr et al. (2017), Jansen et al. (2005) and Doerr and Doerr (2015) with toy functions, afterwards these algorithms are modified by changing and mixing them together, in order to finally compare them with the same toy functions.

3.1.1 Testing

The algorithms tested are the (1 + λ) EA with Self-Adjusting Mutation Rate (Doerr et al., 2017), the (1 + λ) EA with Self-Adjusting Offspring Population Size (Jansen et al., 2005) and the (1 + (λ,λ)) GA with its Self-Adjusting counterpart (Doerr and Doerr, 2015). They are compared against a (1 + λ) EA with optimal static parameters, if known, or a range of parameters otherwise. The algorithms are explained formally in detail in Section 2.

These algorithms contain some extra parameters that can be modified, during the testing these parameters are tuned empirically since in some of the research papers they only choose and tested one value. Two evaluation metrics are used for all of them, the sequential and parallel time, in order to give insights of the computational cost for every algorithm; if any run of the algorithms can not get to the optimum the Success Rate and Mean Best Fitness are mentioned. The toy functions used in the testing are OneMax, LeadingOnes, MakeSpan Scheduling and SufSamp.

3.1.2 Modifications

In this part of the project modifications in the algorithms are made, a brief explanation is given here and further explained in Section 4. For the (1 + λ) EA with Self-Adjusting Mutation Rate, the parameter F also affects the mutation probability of the created subpopulations, ending with $p = c/(Fn)$ and $p = Fc/n$ for both subpopulations, instead of $p = c/(2n)$ and $p = 2c/n$.

The changes in the (1 + λ) EA with Self-Adjusting Offspring Population Size are in the factors used to multiply and divide the current population size λ, and constructing an upper limit for the value of λ.

At last for the (1 + (λ,λ)) GA, two algorithms are made by changes in the mutation step, the first is mutating only one parent transforming the algorithm into $(1+(1,\lambda))$ GA and the second changes the method of selection of te second parent for crossover, instead of mutating λ times and selecting the second parent from the fittest individuals, only 0.37λ is mutated selecting the fittest of this offspring population then this individual is compared against the other 0.63λ offspring one by one; if there is a fitter individual it is chosen as a parent and the rest of the offspring is not compared, otherwise the previously chosen individual is used.

3.2 Requirements

Within the requirements of the project there are two types, functional and non-functional. The functional requirements are related to the main objective presented before, and the non-functional requirements are those that specify the criteria to judge the system as a whole.

3.2.1 Functional Requirements

In order to achieve the main objective of testing and modifying the algorithms a certain amount of information should be stored and available to the user when running the code. The description of the information needed is below.

- Average number of evaluations needed to solve the problem.
- Average number of generations needed to solve the problem.
- Average fitness value of the best individual at the end of every run.
- Average fitness value of the best individual in each generation across every run.
- Average value of λ in each generation across every run.
- Average value of c in each generation across every run.

Other than the information that is collected, some options should be available to the user (all of them are optional inputs for the user), these are shown below.

- Get a help command that describes all the possible inputs.
- Print results and other information during the runs.
- Seed for random numbers.
- Problem to solve.
- Size of the problem.
- Number of runs.
- Type of stop criteria.
- Maximum number of evaluations (if it is chosen as stop criteria).
- Maximum number of generations (if it is chosen as stop criteria).
- Algorithm used.
- Parameters specific for each algorithm.

3.2.2 Non-Functional Requirements

As mentioned before the non-functional requirements are based on the system, therefore they are related to the criteria used in design decisions. The requirements of this section are divided into four.

Robustness

To achieve robustness a program needs to be able to handle errors and unexpected input with ease. As such the code in this project needs to be able to detect errors, display relevant information and if possible continue running.

Usability

Usability refers to the ease of use of software to achieve a certain objective. As described in ISO 9241-11 (1998), usability is the "[e]xtent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use." In order to attain a good usability, the program needs to have a good help method that describes with detail each option and if an unexpected input is given describe the problem.

Replicability

Since this project is an empirical study it is crucial that the code is able to replicate the experiments even when using random numbers, therefore a library which accepts a seed in the random number generator is required.

Code recycling

To help future studies, the code should be explained in detail with comments and be designed with interchangable modules. The interchangable modules would help experimenting with new algorithms since you would not need to create everything from scratch but only create the new modules, using a similar idea to the plug and play.

3.3 Testing and evaluation

The testing and evaluation of the project will be done for two things, the algorithms and the implementation.

3.3.1 Algorithms

To test and evaluate the algorithms there exists several possible criteria, in this case we are interested in the parallel and sequential optimization time. Both of them are important but almost always there is a trade off between parallel and sequential optimization time, this is specially true in the $(1 + \lambda)$ EA on OneMax where the best sequential optimization time is achieved with $\lambda = 1$ but there is a linear improvement in parallel optimization time if λ is increased until a certain cut-off value. This relationship is important when using parallel computing since even you can run several evaluations in parallel you do not want to waste computer power in unnecessary evaluations, to capture this I decided to use a geometric mean to help compare the algorithms. For more complex problems the success rate and mean best fitness are used jointly. Since this testing is the main objective of the project it will only be shown in the results.

3.3.2 Implementation

The implementation will be tested repeating some experiments from other authors to ensure that the algorihms and problems are working as intended. To test robustness, test cases are created with several unexpected inputs, and the implementation should handle them correctly, depending on the severity it should raise an error and giving enough information to the user about it, or raise a warning and continue.

4 Design

In this project there are three design processes made, firstly the algorithms are selected and the modifications designed, afterwards the experiments are designed in order to achieve the main objective and ensure a valid result, lastly the program design is made to ensure all the functional and non-functional requirements are met.

4.1 Algorithm design

As mentioned in Section 3.1 previously proposed algorithms are tested and modified in this project. The description of the original algorithms is in the literature review in Algorithms 2, 3, 4, and 5 and the modifications proposed are explained in this section.

4.1.1 Modifications for the (1 + λ) EA with Self-Adjusting Mutation Rate

For this algorithm the change proposed is to include the parameter F into the mutation probability of the created subpopulations, affecting not only the new mutation rate but the actual mutation rate too, with this change the two subpopulations are created with $p = c/(Fn)$ and $p = Fc/n$. The authors of the algorithm did not give an explanation of why they decided to exclude the parameter from the created subpopulations; the hypothesis of this modifications is that using the parameter in the created subpopulations will adjust the mutation rate more precisely, and that otherwise the mutation rate is not adjusted optimally. The pseudocode of the original algorithm is in Algorithm 2 and the change is made in step 5 and 10 ending with the modified algorithm in Algorithm 6

4.1.2 Modifications for the (1 + λ) EA with Self-Adjusting Offspring Population Size

The changes for this algorithm are in the factors used to multiply and divide the current population size λ, all the changes are based on previously proposed self-adjusting schemes. The ways the factor is selected are:

- multiplying and dividing by a factor G. (Lässig and Sudholt, 2011)
- multiplying by a factor G and setting λ to 1. (Lässig and Sudholt, 2011)
- multiplying by a factor $G^{1/4}$ and dividing by G (Doerr et al., 2015)

From the original algorithm given in Algorithm 3 the steps 11 to 15 would change depending on the update scheme used.

4.1.3 Modifications for the Self-Adjusting (1 + (λ,λ)) GA

Since this is the most complete algorithm in which all the parameters are self-adjusted, the modification is not in the selection of parameters itself, but in the way the algorithm

Algorithm 6: The (1 + λ) EA with offspring population size λ and two-rate standard bit mutation. (Changed)

```
1  Initialization: Choose $x \in \{0,1\}^n$ uniformly at random and set $c \leftarrow c^{init}$
2  Optimization: for $t \in \{1, 2, \dots\}$ do
3      Mutation:
4      for each $i \in \{1, \dots, \lambda\}$ do
5          $c_t = c/F$ if $i \leq \lambda/2$ and $c_t = Fc$ otherwise;
6          Create $x'_i \in \{0,1\}^n$ by copying $x$ and, independently for each bit, flip this
           bit with probability $p = c_t/n$;
7      Selection:
8      if $\max\{f(x'_1), \dots, f(x'_\lambda)\} \geq f(x)$ then
9          Replace $x$ by some randomly chosen $x'_i$ with maximal $f$-value;
10         Replace $c^*$ with $c/F$ if $c_t = c/F$ of chosen $x'_i$ or $Fc$ otherwise;
11     Update:
12     Perform one of the following two actions with probability 1/2:
           • Replace $c$ with $c^*$;
           • Replace $c$ with either $c/F$ or $Fc$ , each with probability 1/2;
       Replace $c$ with min{max{F,$c$},1/(2$F$)};
```

decides the best parents from the mutated individuals and the fittest individual from the created offspring. There are four modifications made but only 2 are explained with pseudo-code. The first idea is based on the optimal stopping theory to select a lifetime partner, and it is used in nature by certain types of fish, and believed to be used by humans too (Fry, 2014). The method suggests to use the first 36.78% of the potential partners as a comparison, and then select the next partner that is better than the first 36.78%. Translated into the (1 + (λ,λ)) GA the procedure would be as follows; only mutate the parent into $\lfloor 0.37\lambda \rfloor$ offspring, select the fittest individual x', then start mutating more offspring and select for crossover the first one that is better than x', if no other better individual is found, use x' for crossover. The purpose is to reduce the amount of evaluations and tend to select the best offspring. The idea could be used in the crossover phase too, but here it is only used in the mutation to better appreciate its behaviour.

Using this idea the first two algorithms emerge from the original Self-Adjusting (1 + (λ,λ)) GA, using the same update rule in Algorithm 7 and a new update rule based on the optimal stopping theory in Algorithm 8

The changes in update rule for Algorithm 8 are in the last steps, where if the fittest mutation is found in the first $\lfloor 0.37\lambda \rfloor$ its value is decreased, if it is found after the first $\lfloor 0.37\lambda \rfloor$ the value stays the same, and is increased otherwise.

As a third modification the mutation process is only made once and the crossover is made λ times with it. The last modification is using the original algorithm, but select the fittest offspring from both the mutations and the crossover, since both of them are evaluated.

4.2 Experiment design

The importance of designing an experiment is to be able to ensure a valid and replicable result. In evolutionary algorithms there are two methods to achieve the validity and replicability of the experiments. The first one is to adjust the size of the problem "n",

Algorithm 7: The self-adjusting (1 + (λ, λ)) GA with mutation probability p, crossover probability c, update strength F, and optimal stopping theory for parent selection.

```
1  Initialization: Sample $x \in 0, 1^n$ uniformly at random and query $f(x)$;
2  Initialize $\lambda \leftarrow 3$;
3  Optimization: for $t = 1, 2, \ldots$ do
4      Mutation phase:
5      Sample $\ell$ from $\mathcal{B}(n, p)$;
6      for $i = 1, \ldots, \lfloor 0.37\lambda \rfloor$ do
7          Sample $x^{(i)} \leftarrow \text{mut}_\ell(x)$ and query $f(x^{(i)})$;
8      Choose $x' \in \{x^{(1)}, \ldots, x^{(\lfloor 0.37\lambda \rfloor)}\}$ with $f(x') = \max\{f(x^{(1)}), \ldots, f(x^{(\lfloor 0.37\lambda \rfloor)})\}$
       u.a.r.;
9      for $i = \lceil 0.37\lambda \rceil, \ldots, \lambda$ do
10         Sample $x^{(i)} \leftarrow \text{mut}_\ell(x)$ and query $f(x^{(i)})$;
11         if $f(x^{(i)}) > f(x')$ then
12             $x' \leftarrow x^{(i)}$;
13             break
14     Crossover phase:
15     for $i = 1, \ldots, \lambda$ do
16         Sample $y^{(i)} \leftarrow \text{cross}_c(x, x')$ and query $f(y^{(i)})$;
17     If exists, choose $y \in \{x', y^{(1)}, \ldots, y^{(\lambda)}\} \backslash \{x\}$ with
       $f(y) = \max\{f(x'), f(y^{(1)}), \ldots, f(y^{(\lambda)})\}$ u.a.r.;
18     otherwise, set $y := x$;
19     Selection and update step:
20     if $f(y) > f(x)$ then $x \leftarrow y$; $\lambda \leftarrow \max\{\lambda/G, 3\}$;
21     if $f(y) = f(x)$ then $x \leftarrow y$; $\lambda \leftarrow \min\{\lambda G^{1/4}, n\}$;
22     if $f(y) < f(x)$ then $\lambda \leftarrow \min\{\lambda G^{1/4}, n\}$;
```

this value is increased to reduce the possibility of solving the problem by chance, but in return, the algorithm takes more time to solve it. Another way is to increase the amount of runs and averaging the results, this decreases the weight of outlier runs where the algorithm solves the problem faster or slower than expected because of random chance.

In this work the selection of these two values is made by following the selection made by the authors of the algorithms studied were the values of n range between 18 and 5000, and the number of runs are around 50 for small tests and 10000 for rigorous tests. The values of $n = 100$ and runs = 500 are used to test the algorithms.

4.3 Code design

The code design has as objective fulfill all the functional requirements, while taking into account the non-functional requirements, and reduce possible bugs. In the design process first a programming paradigm is selected with the requirements in mind. Subsequently a programming language is selected taking into account the previous step. Afterwards several recommended practices for the programming language selected are explained and used during the coding of the project. Finally the architecture of the code is detailed.

Algorithm 8: The self-adjusting (1 + (λ, λ)) GA with mutation probability p, crossover probability c, update strength F, and optimal stopping theory for parent selection.

```
1  Initialization: Sample $x \in 0, 1^n$ uniformly at random and query $f(x)$;
2  Initialize $\lambda \leftarrow 3$;
3  Optimization: for $t = 1, 2, \ldots$ do
4      Initialize $h \leftarrow 0$;
5      Mutation phase:
6      Sample $\ell$ from $\mathcal{B}(n, p)$;
7      for $i = 1, \ldots, \lfloor 0.37\lambda \rfloor$ do
8          Sample $x^{(i)} \leftarrow \mathrm{mut}_\ell(x)$ and query $f(x^{(i)})$;
9      Choose $x' \in \{x^{(1)}, \ldots, x^{(\lfloor 0.37\lambda \rfloor)}\}$ with $f(x') = \max\{f(x^{(1)}), \ldots, f(x^{(\lfloor 0.37\lambda \rfloor)})\}$
       u.a.r.;
10     for $i = \lceil 0.37\lambda \rceil, \ldots, \lambda$ do
11         Sample $x^{(i)} \leftarrow \mathrm{mut}_\ell(x)$ and query $f(x^{(i)})$;
12         if $f(x^{(i)}) > f(x')$ then
13             $x' \leftarrow x^{(i)}$;
14             $h \leftarrow 1$;
15             break
16     Crossover phase:
17     for $i = 1, \ldots, \lambda$ do
18         Sample $y^{(i)} \leftarrow \mathrm{cross}_c(x, x')$ and query $f(y^{(i)})$;
19     If exists, choose $y \in \{x', y^{(1)}, \ldots, y^{(\lambda)}\} \backslash \{x\}$ with
       $f(y) = \max\{f(x'), f(y^{(1)}), \ldots, f(y^{(\lambda)})\}$ u.a.r.;
20     otherwise, set $y := x$;
21     Selection and update step:
22     if $f(y) > f(x)$ and $h = 0$ then $x \leftarrow y$; $\lambda \leftarrow \max\{\lambda/G, 3\}$;
23     if $f(y) > f(x)$ and $h = 1$ then $x \leftarrow y$;
24     if $f(y) = f(x)$ then $x \leftarrow y$; $\lambda \leftarrow \min\{\lambda G^{1/4}, n\}$;
25     if $f(y) < f(x)$ then $\lambda \leftarrow \min\{\lambda G^{1/4}, n\}$;
```

4.3.1 Programming paradigm

In order to ensure the non-functional requirements the programming paradigm needs to be able to help the code be understandable and extensible without affecting the robustness and usability of the program. To achieve this a combination of two programming paradigms is used, modular programming and class-based programming.

The modular programming will help separate the functionality into interchangable parts, called *modules*, letting the program choose between different problems and algorithms with ease. It also helps the reusability because you could create a new module to extend the functionality into other algorithms or problems, without modifying the previous code.

The class-based programming is used to give the modules standard functions that are needed, enhancing further the reusability. Since all problems and all algorithms share the same functions such as initialization or evaluation of fitness a class-based programming would make the modules able to be programmed independently and interchangeably with ease.

4.3.2 Programming language

Since two programming paradigms are used, a flexible language that can implement both of them simultaneously is needed. Python and Java are both good candidates and do not seem to have a clear disadvantage from each other. In order to decide between them, my familiarity is taken into account since the code also needs to be readable, simple and efficient, therefore the programming language that is used is Python.

4.3.3 Best practices

In this section a breif description of the programming practices used is given, they are taken from a guide given by the Python Software Foundation (Rossum et al., 2001).

- **Indentation**: Use indentation for continuation lines aligning wrapped elements vertically.
- **Maximum Line Length**: Use a maximum of 79 characters per line.
- **Blank Lines**: Use two blank lines to separate functions and class definitions, one blank line for method definitions inside a class and inside functions, sparingly, to divide logical sections.
- **Imports**: Import at the beginning of the file, use separate lines for each import, and order them by standard library imports, third party imports and local application/library specific imports.
- **Space**: Use single spaces around assignements, comparisons, booleans and operators (with some exceptions).
- **Comments**: Comments should be complete sentences. The first word should be capitalized. Start comments with # and a single space.
- **Class Names**: Use *CapitalizedWords*.
- **Package and Module Names**: Modules should have short, all-lowercase names.
- **Function and Variable Names**: Written in all lowercase, with words separated by underscores. Name functions with verbs and variables with nouns.
- **Constant names**: Written in all capital letters with underscores separating words.

4.3.4 Code structure

The code is structured in modules that work individually to make a certain action and are imported by a "master" program that follows the flow diagram in Figure 1. In this section a detailed description of each module including the master program is given.

Master program

This part of the code has the objective of taking the inputs given by the user in order to create instances of the corresponding modules that contain the algorithm and problem chosen, make the empirical experiments with them, and reporting the outputs described in the Section 3.2.1.

Input module

This module reads the inputs from the command line and process them. The inputs accepted by the program are described in Table 1.

Cmd	Accepted values	Default	Description
-h			Displays a description of the inputs.
-t	{OneMax, LeadingOnes, SufSamp, MakespanScheduling}	OneMax	Select the problem to solve.
-n	[10, 5000]	100	Size of the problem "n".
-r	[1, 10000]	100	Number of runs.
-s	{solved, ngenerations, nevaluations}	solved	Stop criteria.
-g	[10, 100000]	1000	Maximum number of generations.
-e	[10, 1000000]	10000	Maximum number of evaluations.
-a	{OnePlusLambda, OnePlusLambdaSAMut, OnePlusLambdaSAOff, OnePlusLambdaCommaLambda, OnePlusLambdaCommaLambdaSA, OnePlusLambdaSAMutImp, OnePlusLambdaSAOffImp1, OnePlusLambdaSAOffImp2, OnePlusLambdaSAOffImp3, OnePlusLambdaSAMutOff, OnePlusLambdaCommaLambdaSAImp1, OnePlusLambdaCommaLambdaSAImp2, OnePlusLambdaCommaLambdaSAImp3, help}	OnePlusLambda	Algorithm used.
-p	[0, 1]	$1/n$	Mutation probability "p".
-l	[1, $50n \log_2 n$]	1	Offspring size population "λ".
-c	[0, 1]	$1/\lambda$	Crossover probability.
-F	[1.0, 5.0]	2.0	Mutation update factor.
-G	[1.0, 5.0]	2.0	Offspring size population update factor.
--seed	$\mathbb{Z}$	None	Seed for random number generation.
-v	None	None	If present print information per run

Table 1: Input description

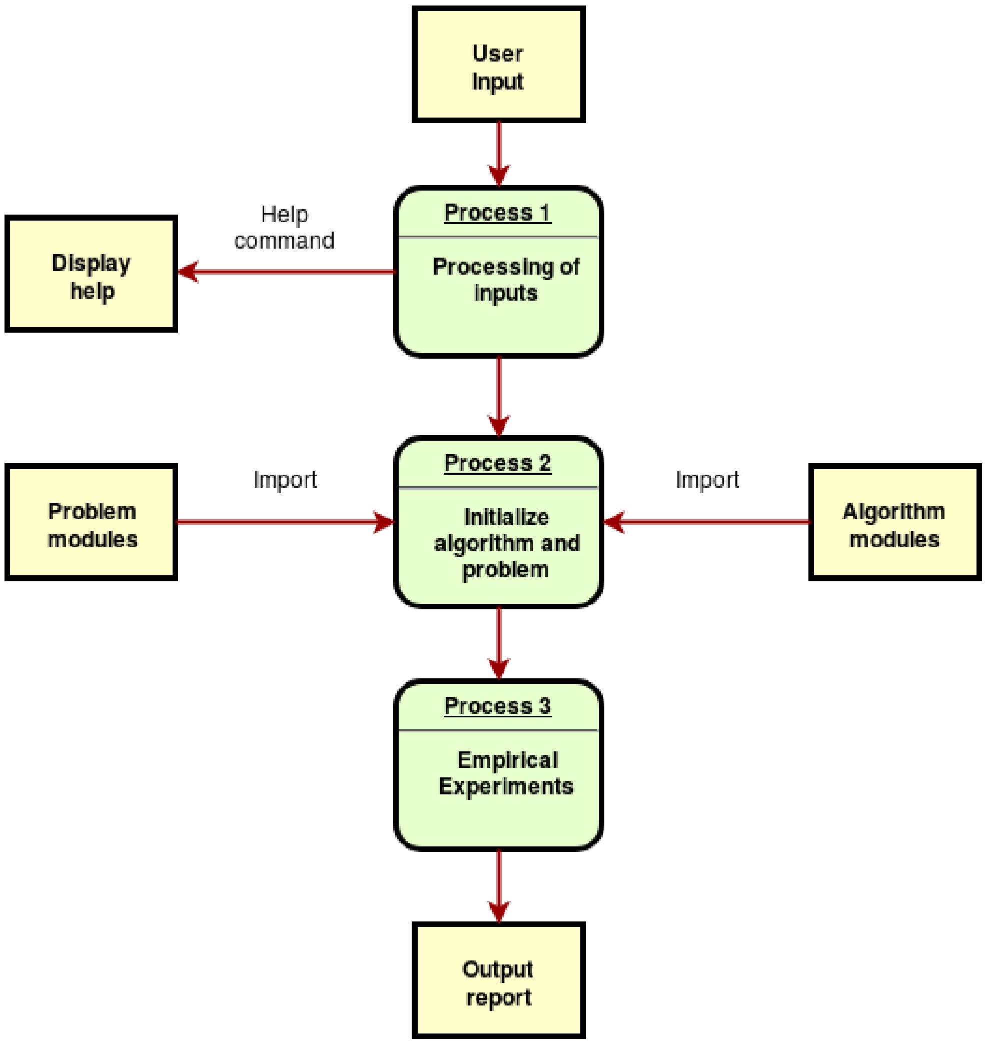

Figure 1: Flow diagram of master program

Problem modules

The problem modules are composed by a class that have several standardized methods (internal functions) and attributes (internal varables). The methods are:

- *__init__*: Initialization of the problem with the size of the problem "n" as input.
- *evaluate*: Evaluation of the fitness function with a bitstring of size "n" as input, giving as output the fitness value and a boolean to describe if the value is the optimal.

And the attributes are:

- *max_fitness*: Maximum fitness for the initialized problem.

Algorithm modules

As the previous modules, the algorithm modules are composed by a class with standardized methods and attributes. The methods are:

- *__init__*: Initialization of the algorithm with the flowing inputs in order: problem object (an instance of the problem modules), size of the problem "n", mutation probability, offspring population size, crossover probability, mutation update factor

and offspring update factor. If any of those inputs is not needed put a representative variable name instead (eg. not_used).
- *__next__*: Run a generation updating the actual generation and number of evaluations made, and saving in a list the actual offspring size and actual mutation probability.

And the attributes are:
- *solved*: A boolean variable describing if the optimal value has been achieved.
- *evaluations*: Number of actual evaluations.
- *generations*: Number of actual generations.
- *parent*: A tuple of two elements, the output of the evaluate method in the problem object and the bitstring.
- *offspring_size*: Actual offspring size.
- *fit_gen*: List with the maximum fitness in each generation.
- *lambda_gen*: List with the offspring size in each generation.
- *mut_gen*: List with the mutation rate in each generation.

5 Implementation and Testing

This chapter illustrates the implementation of the project and the testing made to the implementation of algorithms, problems and error handling.

5.1 Implementation

As per design the code is separated in modules, this section will revise the implementation of each type of them; in the case of the problem and algorithm modules only one will be shown here, since it would be infeasible to show all of them, due to the size of the code.

5.1.1 Input module

In this module the inputs are read from the command line options and preprocessed using the Python standard library "argparse", if any input is out of bounds the module raises an error and prints the help information. All the inputs are saved as attributes of the class CommandLine, in order to be accessed later by the master program.

5.1.2 Master program

First it calls an instance of the input module and saves all the command line options into it with the next line.

```
configuration = CommandLine()
```

Then it calls for instances of the problem and algorithm selected, this is made with the function `eval()` that lets Python run code from strings. Since this function can run arbitrary code written in the inputs, a whitelisting in the inputs module is used to avoid malicious or erroneous use of the code. The creation of the string for every problem uses the inputs "problem" and "problem_size", afterwards the algorithm uses the created instance problem and other inputs. As stated in the design all algorithm modules should have the same inputs since the code always gives the same amount of inputs to avoid using if cases, making the code compact and easily recyclable.

```
problem_string = (configuration.problem + '(' +
                  str(configuration.problem_size) + ')')
problem = eval(problem_string)
algorithm_string = (configuration.algorithm + '(' + 'problem,' +
                    str(configuration.problem_size) + ',' +
                    str(configuration.mut_prob) + ',' +
                    str(configuration.offspring_size) + ',' +
                    str(configuration.cross_prob) + ',' +
                    str(configuration.mut_update_factor) + ',' +
                    str(configuration.offspring_update_factor) + ')')
```

After creating the algorithm string, several runs of the algorithm are made with the stop criteria chosen. For the first criteria "solved" a problem arises, some problems have local minima and the algorithms can get stuck there forever. To overcome this, two other stop conditions are added, if 100,000 evaluations are made without improvement in a local optimum (1,000,000 for SufSamp) or the overall amount of evaluations surpasses $100n^4$ the code stops. If any of the alternative stop conditions are used, that run is not taken into account for optimization time averages but it is counted for the mean best fitness. When all the runs have been finished the results are saved and shown (written and plotted) by the output module.

5.1.3 Problem modules

From design all the problem modules need to have at least the standard methods (__init__, evaluate) and the attribute max_fitness. To illustrate the structure of all the modules, the code for the simplest problem is shown, where in the method __init__ initializes the maximum fitness and the method evaluate, evaluates a bit string returning the actual fitness value and if the optimal fitness is reached.

```
class OneMax:
    def __init__(self, problem_size):
        self.max_fitness = problem_size

    def evaluate(self, bit_string):
        fitness_value = bit_string.count('1')
        if fitness_value == self.max_fitness:
            optimal_fitness = True
        else:
            optimal_fitness = False
        return fitness_value, optimal_fitness
```

5.1.4 Algorithm modules

The algorithm modules are more complex and are composed of two standard modules (__init__, __next__) and several attributes, in the initialization of each algorithm a random bit string is created using the following lines

```
import numpy as np

bit_array = np.random.choice(['0', '1'], self.problem_size)
self.bit_string = ''.join(bit_array)
```

After the initialization the __next__ method represents a complete generation containing all evolutionary operations necesary. As an example the __next__ method of the (1+λ) EA is shown.

```
def __next__(self):
    offspring = []
    self.parent = [self.problem.evaluate(self.bit_string),
                   self.bit_string]
    self.mut_prob_gen.append(self.mutation_probability)
    self.lambda_gen.append(self.offspring_size)
    for i in range(self.offspring_size):
        mutated_string = self.mutate()
        offspring.append((self.problem.evaluate(mutated_string),
                          mutated_string))
    self.select(offspring)
    self.generations += 1
    self.evaluations += self.offspring_size
```

First the actual parent is stored with it's evaluation, then it is mutated λ times, in the end selects the best offspring. It is worth noticing that the evolution operators are methods itself. The code of the mutation in (1+λ) EA is explained to illustrate, but for every algorithm is different.

```
def mutate(self):
    p = [1 - self.mutation_probability, self.mutation_probability]
    mut_array = np.random.choice(['0', '1'], self.problem_size, p = p)
    mutated_list = [str(xor(int(b), int(m))) for b, m in
                    zip(self.bit_string, mut_array)]
    return ''.join(mutated_list)
```

In the mutate method a mutation array with the same size of the bit string is created and populated with 0's and 1's following the mutation probability, with the use of the xor function between the bitstring and mutate array the mutated string is created.

5.1.5 Output module

This module saves all the results into a pickled file, which is a serialized binary protocol used to save objects in Python. If the verbose option is used by the user the scalar results are printed and plots of the average fitness, lambda and mutation rate through every generation are given.

5.2 Testing

In this section two tests were carried out in the code, the first one is a robustness test with fifteen test cases that have unexpected inputs. For each case an expected behaviour was given and compared against the real behaviour of the code giving a grade from 1 to 5 with the following meanings.

- 1 - The code does not detect the erroneous input and fails during run.
- 2 - The code does not detect the erroneous input and have a succesful run.
- 3 - The code detects the erroneous input but does not display the error.
- 4 - The code detects the erroneous input and display the error but not clear.

- 5 - The code detects the erroneous input and display the error with relevant information.

The results of this test cases were compiled in Table 2.

Test No.	Input	Expected behaviour	Real behaviour	Result
1	-t JobShop	Error message with allowed values.	Error message printing the usage of the program, the invalid choice ”JobShop” and all the allowed problems.	5
2	-n 5001	Error message with allowed values.	Error message printing the usage of the program, the invalid choice ”5001” and the allowed range of values.	5
3	-r 1.4	Error message with allowed values.	Error message printing the usage of the program, the invalid choice ”1.4” and the caption ”invalid int value”, but it does not give the allowed values.	4
4	-s solve	Error message with allowed values.	Error message printing the usage of the program, the invalid choice ”solve” and all the allowed stop criteria.	5
5	-s solved -g 50000	Warning for input not used and continue with the runs, explaining how to take into account the value.	Warning message printing the unexpected argument, give the changes needed to take into account the value and continue.	5
6	-s ngenerations -e 5	Warning for input not used and continue with the runs, explaining how to take into account the value.	Warning message printing the unexpected argument, give the changes needed to take into account the value and continue.	5
7	-a help	Show the meaning of the acronyms used for the argument -a.	Show the meaning of the acronyms used for the argument -a and stop the program.	5
8	-p 1.0	Error message with allowed values.	Error message printing the usage of the program, the invalid choice ”1.0” but it is not clear on the allowed values since the range says 0.0 - 1.0	4

9	-l 0	Error message with allowed values.	Error message printing the usage of the program, the invalid choice ”0” and all the allowed values.	5
10	-c 0.5	Warning for input not used and continue with the runs, explaining how to take into account the value.	Warning message printing the unexpected argument, give the argument ”-a” to change but not the correct options.	4
11	-F 1.2	Warning for input not used and continue with the runs, explaining how to take into account the value.	Warning message printing the unexpected argument, give the argument ”-a” to change but not the correct options.	4
12	-G 3.5	Warning for input not used and continue with the runs, explaining how to take into account the value.	Warning message printing the unexpected argument, give the argument ”-a” to change but not the correct options.	4
13	–seed 4.1	Error message with allowed values.	Error message printing the usage of the program, the invalid choice ”4.1” and the caption ”invalid int value”, but does not give the allowed values.	4
14	-p 0.3 -a OnePlusLambda CommaLambdaSA	Warning for input not used and continue with the runs, explaining how to take into account the value.	Warning message printing the unexpected argument, give the argument ”-a” to change but not the correct options.	4
15	-n 50 -l 100 -a OnePlusLambda CommaLambdaSA	Warning for input changed and continue with the runs.	Warning message printing the change made and continue with the runs.	5

Table 2: Implementation test cases

From this test it can be derived the conclusion that the code is robust enough to handle several types of unexpected inputs with ease and give relevant information when needed.

In the second part of the tests, previous experiments by other authors were repeated and compared to ensure the algorithms and problems were correctly programmed. The results are shown in Table 3.

Modules tested	Test description	Expected result	Real result
OneMax $(1 + \lambda)$ EA	Average number of evaluations to solve the OneMax problem with size $n = 1000$ using the $(1 + 1)$ EA	17001 (Doerr et al., 2015)	16819.84 (100 runs)
LeadingOnes	Expected optimization times of $(1 + 1)$ EA for $n = 550$ with $p = 1.5936/n$ and $p = 1/n$	$0.77201n^2 \approx 233533$ $0.85914n^2 \approx 259890$ (Böttcher et al., 2010)	233467.78 262733.53 (100 runs)
SufSamp	Number of successful runs with $n = 42$ of the $(1 + \lambda)$ EA with $\lambda = 1, 2, 7, 100, 420,$ 2100.	1 - 14%, 2 - 18%, 7 - 18%, 100 - 62%, 420 - 64%, 2100 - 68% (Jansen et al., 2005)	1 - 4%, 2 - 0%, 7 - 2%, 100 - 42%, 420 - 54%, 2100 - 48% (50 runs)
$(1 + \lambda)$ EA with self-adjusting mutation rate (Algorithm 2)	Average number of evaluations to solve the OneMax problem with $\lambda = 600$ better than $(1 + \lambda)$ EA	$\approx$ 6% better with $n = 5000$ (Doerr et al., 2017)	12.4% better with $n = 500$ (100 runs)
$(1 + \lambda)$ EA with self-adjusting offspring population size (Algorithm 3)	Average number of evaluations to solve the LeadingOnes problem with size $n = 90$ slightly higher than $(1 + 1)$ EA	$\approx$ 7400 (Jansen et al., 2005)	7400.152 (1000 runs)
$(1 + (\lambda, \lambda))$ GA (Algorithm 4)	Average number of evaluations to solve the OneMax problem with size $n = 500$, $\lambda = 8$, $p = 0.016$ and $c = 0.125$	5411 (Doerr et al., 2015)	5424.61 (100 runs)
Self-adjusting $(1 + (\lambda, \lambda))$ GA (Algorithm 5)	Average number of evaluations to solve the OneMax problem with size $n = 500$, $G = 1.5$ $p = \lambda/n$ and $c = 1/\lambda$	5143 (Doerr et al., 2015)	5152.95 (100 runs)

Table 3: Replicability tests

We can see that some results vary, but they could be explained by the number of runs used in the testings. For the Makespan Scheduling problem to the best of my knowledge there are no empirical results for any of the algorithms used in this work in the literature, therefore the only testing was made by checking manually the evaluations and approximated maximum fitness for $n = 10$.

6 Results and discussion

In this section the results of the experiments are shown with a discussion on the benefits and disadvantages of each algorithm in different problem settings. First the original algorithms are tested with different configurations and problems, afterwards the modificated algorithms are compared with the original and against each other with the same problems. In the end a discussion of interesting findings and further work is made.

6.1 Empirical results on original algorithms

From the original algorithms several parameters were chosen, firstly as a reference the parameters selected by their respective authors are used, secondly a selection of parameters following the insights given by other research works is made.

In the following figures all the different results of the algorithms with different parameters are shown, to see the complete tables with parameters and results refer to the Appendices. The experiments performed here used 500 independent runs for each algorithm. The problem size n is set to 100. All the runs that find the optimum were used to compute the average optimization times and all runs are used to calculate success rate and mean best fitness.

6.1.1 OneMax

The first problem tested is the OneMax showed in Figure 2. An important point to notice is that the best algorithms should be in the lower part for sequential computing, and one could argue that the ones in the lower left corner are the best for parallel computing since they reduce the amount of generations without drastically increasing the number evaluations, this relationship is calculated using the geometric mean and represented with the size of the circles (smaller circles are better).

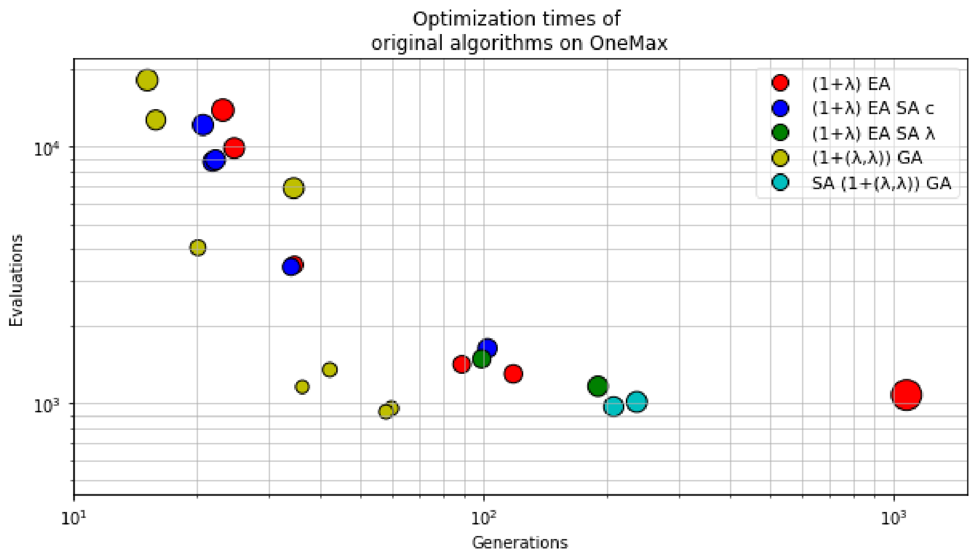

Figure 2: Optimization times from original algorithms on OneMax

From the experiments we can see the linear improvements in parallel optimization time the (1 + λ) EA gets from increasing the λ until the cut-off value ($\lambda = 16$), but interestingly we see that it could be viable to increase λ past the cut-off value, since the decrease of evaluations is proportionaly larger than the increase in number of generations at first. This behaviour can be seen in Table 4 where the geometric mean decreases from $\lambda = 16$ to $\lambda = 100$ and then increases again with $\lambda = 400$.

Algorithm	Parameters	Evaluations	Generations	Geometric mean
(1 + λ) EA	$\lambda = 1$, $p = 0.01$	1076.83	1076.83	1076.83
(1 + λ) EA	$\lambda = 11$, $p = 0.01$	118.29	1301.23	392.33
(1 + λ) EA	$\lambda = 16$, $p = 0.01$	88.64	1418.27	354.564
(1 + λ) EA	$\lambda = 100$, $p = 0.01$	34.61	3461.6	346.16
(1 + λ) EA	$\lambda = 400$, $p = 0.01$	24.69	9876.0	493.8

Table 4: (1 + λ) EA on OneMax

This same relation between λ and the geometric mean is seen in (1 + λ) EA with self-adjusting mutation rate and (1 + (λ, λ)) GA, leaving the user of the algorithms with the responsability to tune it in order to optimize the use of parallel computing power. With the (1 + λ) EA with self-adjusting offspring size does not need to tune this parameter and the geometric mean is not increased dramatically compared to the other algorithms.

Focusing in the sequential optimization time in Figure 2 we can see that the (1 + (λ, λ)) GA outperforms every other algorithm tested. But still it has the problem of selecting the appropriate parameters to achieve the best performance, since it also has the worst performance, this is solved by its self-adjusting counterpart and still achieves the second best sequential optimization time.

6.1.2 LeadingOnes

In Figure 3 the results of the LeadingOnes problem are shown. We can infer that this problem does not benefit greatly from crossover, making the (1 + (λ, λ)) GA and the self-adjusting (1 + (λ, λ)) GA perform badly.

As with the OneMax problem there is a linear improvement in parallel optimization time when increasing λ until a cut-off point, and the geometric mean decreases further past the cut-off $\lambda = n$. In this problem the algorithm with the smallest geometric mean and the smallest number of evaluations is the (1 + λ) EA, given that you choose the appropriate λ. The (1 + λ) EA with self-adjusting offspring size achieves almost the same number of evaluations, reducing the number of generations and taking away the manual tuning of the offspring size.

In this problem the (1 + λ) EA with self-adjusting mutation rate performs worse than the classic (1 + λ) EA in every case, from the results we can infer that if λ is increased the self-adjusting mutation rate would outperform the (1 + λ) EA with the same choice of λ in sequential and parallel optimization, but the geometric mean would increase, making it a waste of computational resources.

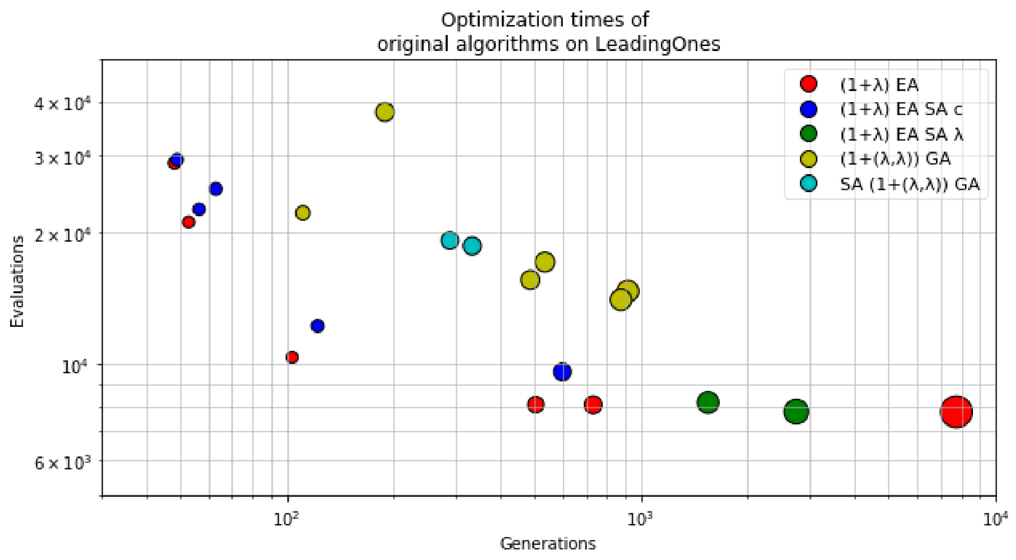

Figure 3: Optimization times from original algorithms on LeadingOnes

6.1.3 Makespan Scheduling

The third problem tested is the Makespan Scheduling problem, in this problem the initialization of the bit strings was changed to $x \in \{0\}^n$ because when initializing randomly there was a big probability to find a good approximation to the optimal solution and increasing the offspring size increased the probability of randomly generating the optimal solution in the first generation, making the results less interesting for the purpose of this work. Even though they are not taken into account in this section, the results with the random initialization are given in the Appendices.

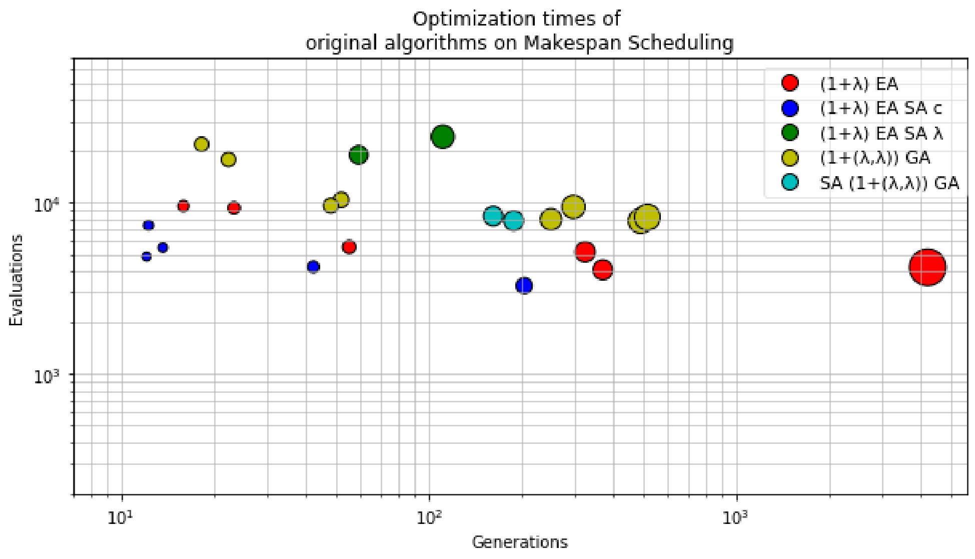

Figure 4: Optimization times from original algorithms on Makespan Scheduling

From Figure 4 we can observe that the selection of offspring population size has a small effect on the number of evaluations up until $\lambda = 100$, but it decreases the number of generations needed drastically. To ilustrate this, the results of the number of generations and evaluations needed by the $(1+\lambda)$ EA are given in Figure 5. Here we see that the number of generations needed to get to the optimal solution decrease rapidly when increasing λ, but the number of evaluations keeps steady.

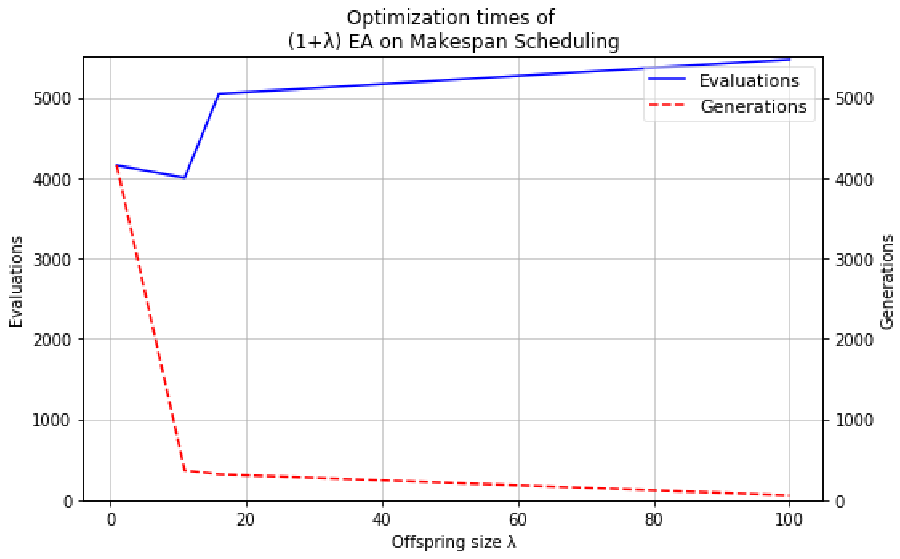

Figure 5: Optimization times from $(1+\lambda)$ EA on Makespan Scheduling

In this problem the best algorithm was the $(1+\lambda)$ EA with self-adjusting mutation rate, followed by the classic $(1+\lambda)$ EA, the most likely reason for this is that the first algorithm adjusts the mutation rate to the problem and in the second algorithm the standard mutation rate of $1/n$ was used, which most likely is not the best for this problem in particular.

In contrast the $(1+\lambda)$ EA with self-adjusting offspring population size used the most number of evaluations, but it was also the only algorithm to get to the optimum 100% of the runs while the other algorithms had a success rate between 96.6% and 99.6%.

6.1.4 SufSamp

The last problem is similar to OneMax and LeadingOnes but with several local optima, therefore the algorithms have a similar behaviour when changing the offspring population size as seen in Figure 6. Another thing to notice is that some algorithms are not showing and it is because those algorithms were not able to get to the optima in any run. Given the similarity to other problems and the low success rate of all the algorithms, in Figure 7 the algorithms are compared with their success rate and mean best fitness.

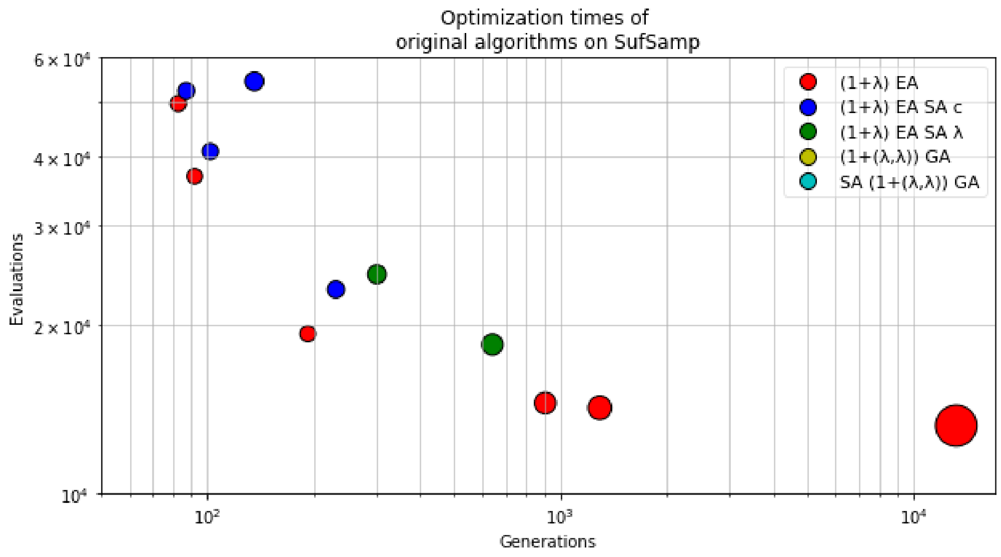

Figure 6: Optimization times from original algorithms on SufSamp

In Figure 7 we can see that the $(1 + \lambda)$ EA is the best algorithm for this problem and there are several reasons. Firstly the probability to find the branch to the global optima seem to decreas if you have a different mutation rate than 1/n and secondly if you do not have a large enough offspring population size in a branch point the probability of finding the correct branch decreases. Given these two characteristics we can see why the other algorithms with self-adjusting parameters could not get to the global optimum as many times as the classical $(1 + \lambda)$ EA.

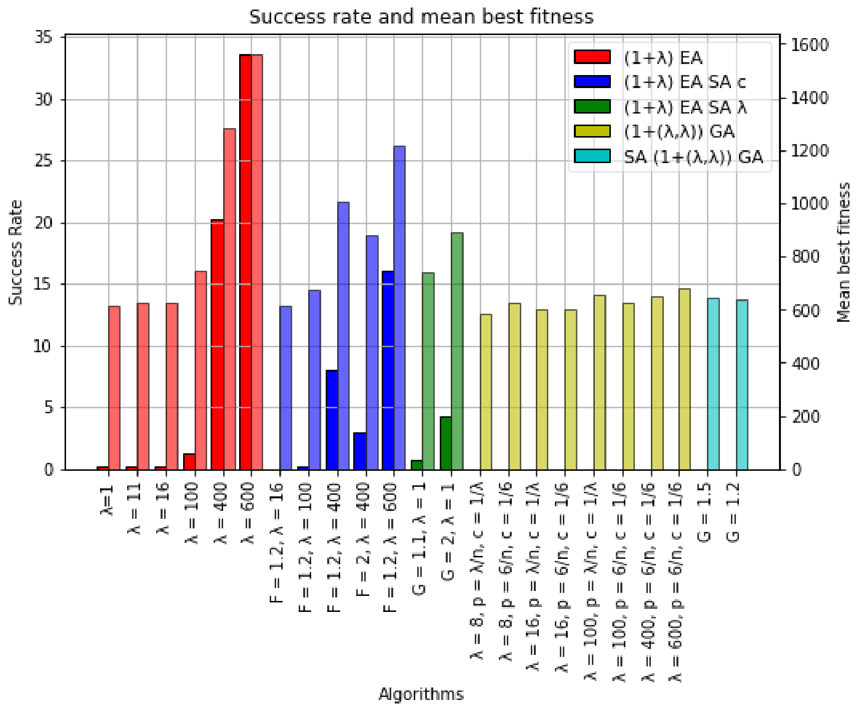

Figure 7: Success rate and mean best fitness from original algorithms on SufSamp

6.2 Empirical results on modified algorithms

To compare the modified algorithms, the same parameters chosen in the original algorithms were used. In the figures of this section the original algorithm results are shown in translucent color.

6.2.1 OneMax

As shown in Figure 8 in general the modifications done to the algorithms did not change the results drastically for the OneMax problem, the only algorithm that obtained a considerable improvement is the self adjusted $(1 + (\lambda, \lambda))$ GA, that have an improvement of 60% in the number of generations and an improvement of 20% in the number of evaluations with the second modification, making it the best algorithm in sequential optimization time on OneMax. An important point to make is that this algorithm cannot be used completely in parallel computing due to the way each generation is calculated.

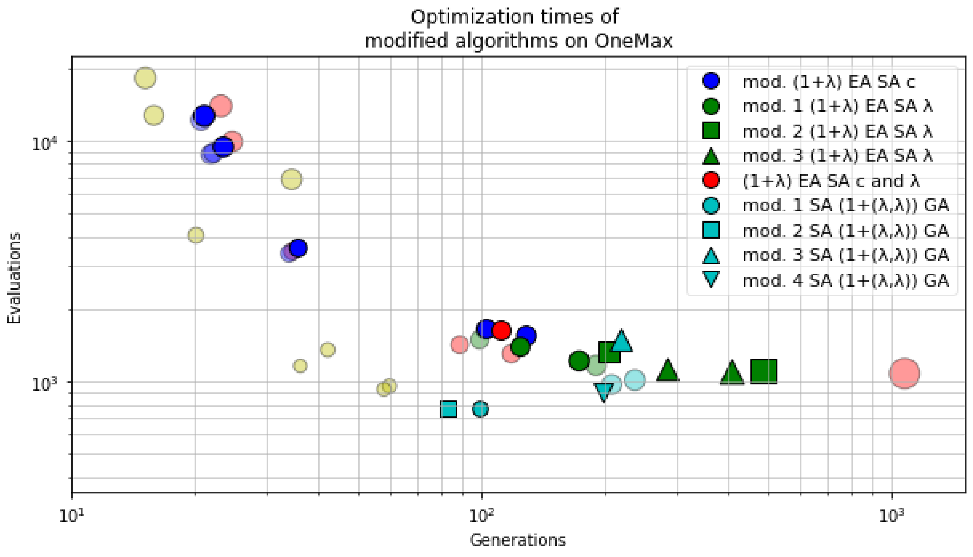

Figure 8: Optimization times from modified algorithms on OneMax

6.2.2 LeadingOnes

For the LeadingOnes problem the same behaviour as the previous problem emerged, the modified algorithms have comparable performance as their original counterparts, but overall the best algorithm is still the $(1 + 1)$ EA. The only noticeable improvement is the third modification of the self adjusted $(1 + (\lambda, \lambda))$ GA where it improved 40% in the number of evaluations. It is important to notice that the second modification which had the best performance for OneMax had also an improvement but only of 7% in the number of evaluations but given the performance of the original algorithm neither of the modifications are near the best algorithms.

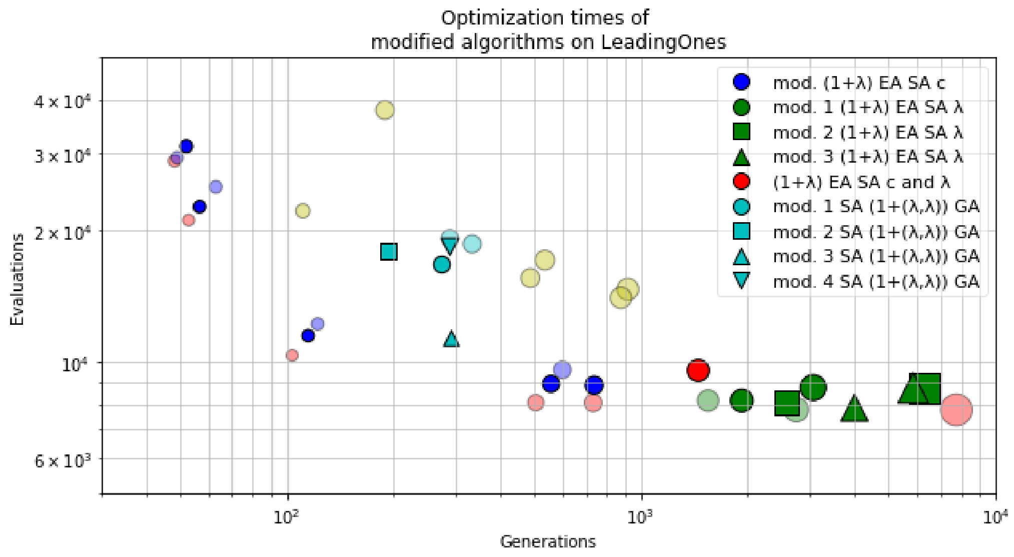

Figure 9: Optimization times from modified algorithms on LeadingOnes

6.2.3 Makespan Scheduling

The modified algorithms on the makespan scheduling problem have in general a better success rate than the original algorithms, and all of the modified versions of $(1 + \lambda)$ EA with self-adjusting offspring population size used less evaluations than the original version. Overall the best algorithm is the original $(1 + \lambda)$ EA with self-adjusting mutation rate with an average of 3277.46 evaluations to get to the optimum using an offspring population size of 16.

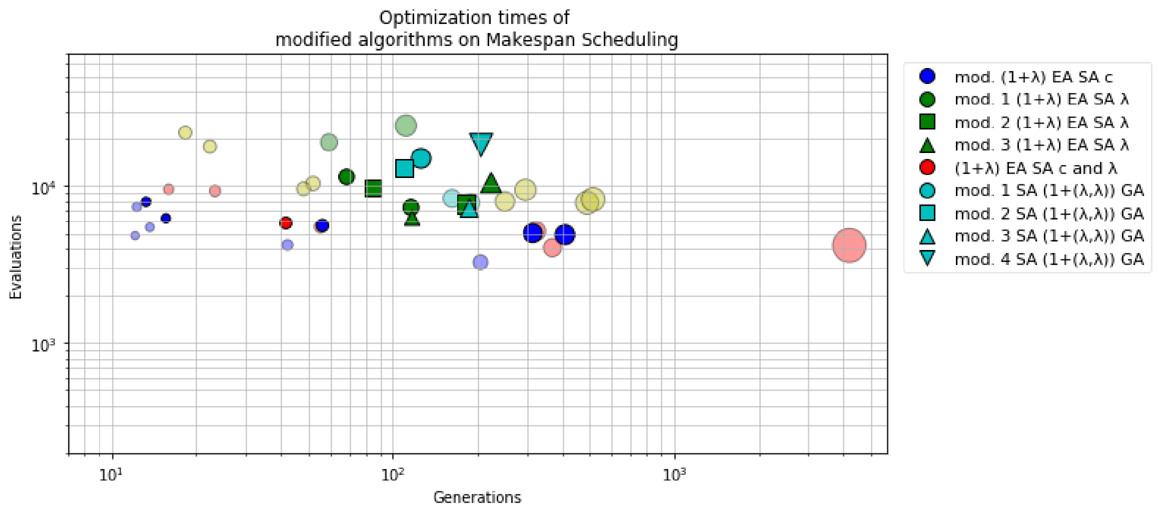

Figure 10: Optimization times from modified algorithms on Makespan Scheduling

6.2.4 SufSamp

At last the modified algorithms on the SufSamp problem also had almost the same performances in both optimization times and success rate. The algorithms does not improve in the success rate because the problems mentioned before are not addressed by the modifications, because of this the best algorithm in sequential and success rate is still the $(1 + \lambda)$ EA.

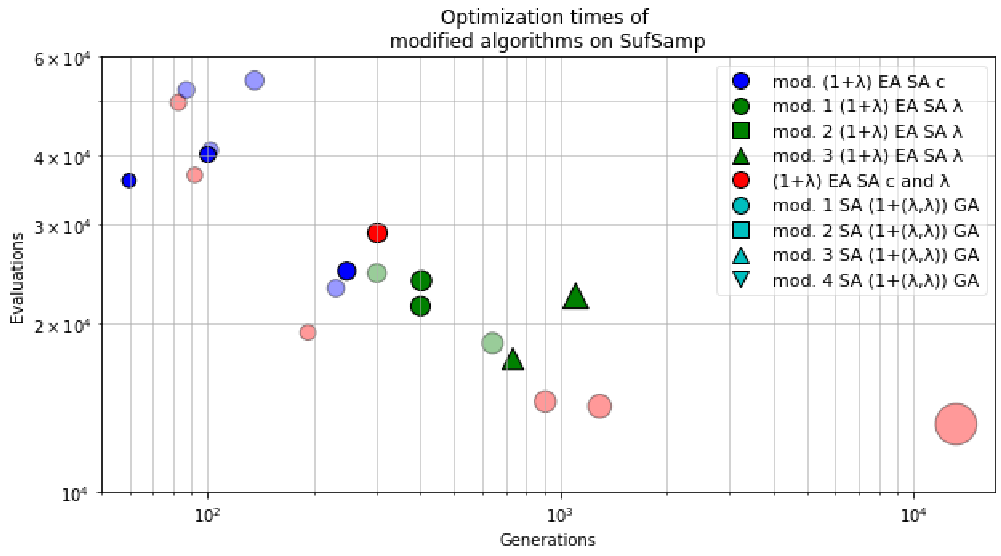

Figure 11: Optimization times from modified algorithms on SufSamp

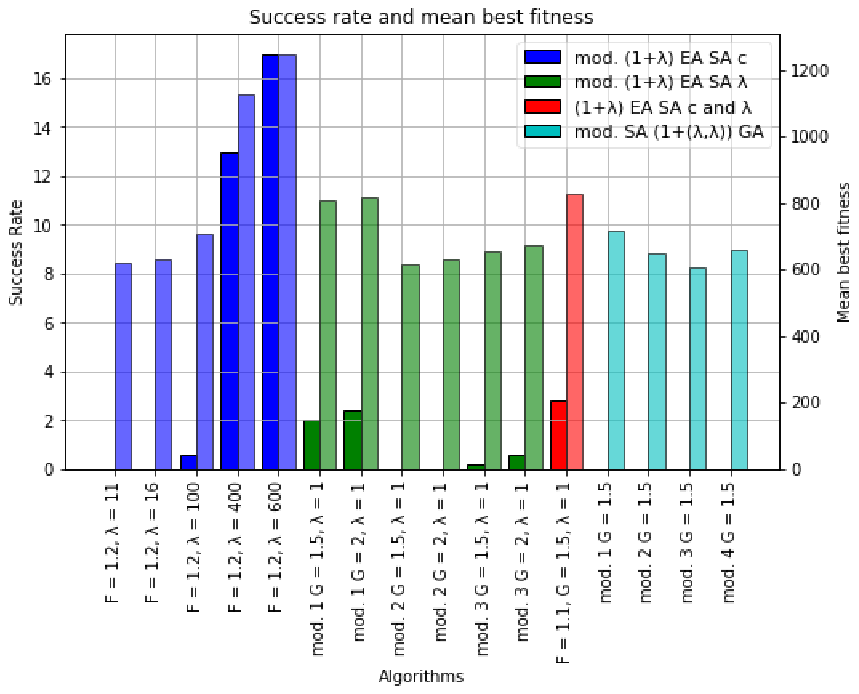

Figure 12: Success rate and mean best fitness from modified algorithms on SufSamp

With these final results all the objectives of testing evolutionary algorithms and modifying them to search for improvements were achieved. From the modifications I have found one new interesting way to do the selection of parents that could lead to an interesting field of research. As follow up to this work an empirical study of this mechanism in different algorithms will be done in order to improve the understanding of how it works and to encourage a riguruous analytical study of this mechanism.

7 Conclusions

In summary this dissertation provides an empirical study of several self-adjusting algorithms with modifications that could improve them depending on the problem settings. This study also could help to define the best parameters and/or the best agorithms depending on what kind of optimization is needed.

It also proposes a new metric to evaluate the performance of Evolutionary Algorithms that uses parallel computing, calculating the geometric mean of the evaluations and generations, which joins both metrics into one to ensure that the improvement in the number of generations is taken into account but without neglecting the number of evaluations. From this metric derives an important insight, that it could be benefitial to increase the offspring population size past the cut-off value where there is no linear improvement in parallel optimization time, since it still decreases proportionally quickier than the increase of secuential optimization time.

At last from the modifications made to the algorithms, I found a new selection scheme to reduce the amount of evaluations per generation without increasing the number of generations, and proposed an update rule using the same selection scheme. I hope that this work inspires theoretical analyses of this selection scheme.

References

Badkobeh, G., Lehre, P. K., Sudholt, D., 2014. Unbiased black-box complexity of parallel search. In: Lecture Notes in Computer Science. Vol. 8672 of PPSN 2014. Springer.

Bartz-Beielstein, T., Branke, J., Mehnen, J., Mersmann, O., May 2014. Evolutionary algorithms. In: Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery. Vol. 4. pp. 178–195.

Beyer, H.-G., Brucherseifer, E., Jakob, W., Pohlheim, H., Sendhoff, B., Binh To, T., 2002. Evolutionary algorithms - terms and definitions.
URL https://homepages.fhv.at/hgb/ea-glossary/node38.html

Böttcher, S., Doerr, B., Neumann, F., 2010. Optimal fixed and adaptive mutation rates for the LeadingOnes problem. In: Proceedings of Parallel Problem Solving from Nature. Vol. 6238 of PPSN 2010. Springer, pp. 1–10.

Corus, D., He, J., Jansen, T., Oliveto, P. S., Sudholt, D., Zarges, C., 2017. On easiest functions for mutation operators in bio-inspired optimisation. In: Algorithmica. Vol. 78. Springer, pp. 714–740.

Doerr, B., Doerr, C., July 2015. Optimal parameter choices through self-adjustment: Applying the 1/5-th rule in discrete settings. In: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation. pp. 1335–1342.

Doerr, B., Doerr, C., Ebel, F., 2015. From black-box complexity to designing new genetic algorithms. In: Theoretical Computer Science. Vol. 567. pp. 87–104.
URL https://doi.org/10.1016/j.tcs.2014.11.028

Doerr, B., Gießen, C., Witt, C., Yang, J., June 2017. The (1+λ) evolutionary algorithm with self-adjusting mutation rate. In: Proceedings of the Genetic and Evolutionary Computation Conference. GECCO '17. ACM, pp. 1351–1358.
URL https://doi.org/10.1145/3071178.3071279

Dyer, D. W., 2008. Evolutionary computation in java: A practical guide to the watchmaker framework.
URL https://watchmaker.uncommons.org/manual/ch01s02.html

Eiben, A. E., Marchiori, E., Valk, V. A., 2010. Evolutionary algorithms with on-the-fly population size adjustment. In: Proceedings of the 8th Parallel Problem Solving From Nature. PPSN VIII. Springer, pp. 41–50.

Fry, H., April 2014. The mathematics of love [video file].
URL https://www.ted.com/talks/hannah_fry_the_mathematics_of_love

Gießen, C., Witt, C., June 2015. The interplay of population size and mutation probability in the $(1 + \lambda)$ EA on OneMax. Algorithmica 78 (2), 587–609.
URL https://doi.org/10.1007/s00453-016-0214-z

ISO 9241-11, 1998. Ergonomic requirements for office work with visual display terminals (VDTs) - Part 11: Guidance on usability.

Jansen, T., De Jong, K. A., Wegener, I., 2005. On the choice of the offspring population size in evolutionary algorithms. Evolutionary Computation 13 (4), 413–440.

Lässig, J., Sudholt, D., 2010a. The benefit of migration in parallel evolutionary algorithms. In: Proceedings of the Genetic and Evolutionary Computation Conference. GECCO 2010. pp. 1105–1112.

Lässig, J., Sudholt, D., 2010b. General scheme for analyzing running times of parallel evolutionary algorithms. In: Parallel Problem Solving from Nature. PPSN XI. Springer, pp. 234–243.

Lässig, J., Sudholt, D., 2011. Adaptive population models for offspring populations and parallel evolutionary algorithms. In: Proceedings of the 11th Workshop Proceedings on Foundations of Genetic Algorithms. FOGA '11. ACM, pp. 181–192.
URL http://doi.acm.org.sheffield.idm.oclc.org/10.1145/1967654.1967671

Neumann, F., Witt, C., 2010. Bioinspired Computation in Combinatorial Optimization. Natural Computing Series. Springer.

Picek, S., Golub, M., September 2010. Comparison of a crossover operator in binary-coded genetic algorithms. WSEAS Transactions on Computers 9 (9), 1064–1073.

Rossum, G. v., Warsaw, B., Coghlan, N., 2001. Style guide for python code.
URL https://www.python.org/dev/peps/pep-0008/

Sudholt, D., 2017. How crossover speeds up building-block assembly in genetic algorithms. In: Evolutionary Computation. Vol. 25. pp. 237–274.
URL https://doi.org/10.1162/EVCO_a_00171

Witt, C., 2013. Tight bounds on the optimization time of a randomized search heuristic on linear functions. Combinatorics, Probability and Computing 22 (2), 294–318.

Appendices

Algorithm	Parameters	Generations	Evaluations	Geometric mean
$(1 + \lambda)$ EA	$\lambda = 1$, $p = 0.01$	1076.83	1076.83	1076.83
$(1 + \lambda)$ EA	$\lambda = 11$, $p = 0.01$	118.29	1301.23	392.33
$(1 + \lambda)$ EA	$\lambda = 16$, $p = 0.01$	88.64	1418.27	354.564
$(1 + \lambda)$ EA	$\lambda = 100$, $p = 0.01$	34.61	3461.6	346.16
$(1 + \lambda)$ EA	$\lambda = 400$, $p = 0.01$	24.69	9876.0	493.8
$(1 + \lambda)$ EA	$\lambda = 600$, $p = 0.01$	23.15	13891.2	567.08
$(1 + \lambda)$ EA with self-adjusting mutation rate	$\lambda = 16$, $p = 0.01$, $F = 1.2$	102.39	1638.24	409.56
$(1 + \lambda)$ EA with self-adjusting mutation rate	$\lambda = 100$, $p = 0.01$, $F = 1.2$	33.97	3396.8	339.69
$(1 + \lambda)$ EA with self-adjusting mutation rate	$\lambda = 400$, $p = 0.01$, $F = 1.2$	21.87	8747.2	437.38
$(1 + \lambda)$ EA with self-adjusting mutation rate	$\lambda = 400$, $p = 0.01$, $F = 2$	22.22	8888.0	444.4
$(1 + \lambda)$ EA with self-adjusting mutation rate	$\lambda = 600$, $p = 0.01$, $F = 1.2$	20.72	12163.2	502.02
$(1 + \lambda)$ EA with self-adjusting offspring size	$\lambda = 1$, $p = 0.01$, $G = 1.1$	190.35	1163.83	470.67
$(1 + \lambda)$ EA with self-adjusting offspring size	$\lambda = 1$, $p = 0.01$, $G = 2$	99.08	1488.14	383.99
$(1 + (\lambda, \lambda))$ GA	$\lambda = 8$, $p = 0.08$, $c = 0.125$	59.69	955.07	238.76
$(1 + (\lambda, \lambda))$ GA	$\lambda = 8$, $p = 0.06$, $c = 0.16667$	57.89	926.27	231.56
$(1 + (\lambda, \lambda))$ GA	$\lambda = 16$, $p = 0.16$, $c = 0.0625$	42.21	1350.78	238.78
$(1 + (\lambda, \lambda))$ GA	$\lambda = 16$, $p = 0.06$, $c = 0.16667$	36.172	1157.50	204.62

(1 + (λ, λ)) GA	$\lambda = 100$, $p = 0.999$, $c = 0.01$	34.46	6892.0	487.34
(1 + (λ, λ)) GA	$\lambda = 100$, $p = 0.06$, $c = 0.16667$	20.14	4038.4	285.19
(1 + (λ, λ)) GA	$\lambda = 400$, $p = 0.06$, $c = 0.16667$	15.89	12710.4	449.41
(1 + (λ, λ)) GA	$\lambda = 600$, $p = 0.06$, $c = 0.16667$	15.15	18175.2	524.74
self-adjusting (1 + (λ, λ)) GA	$\lambda = 1$, $G = 1.5$	207.89	969.64	448.97
self-adjusting (1 + (λ, λ)) GA	$\lambda = 1$, $G = 1.2$	236.72	1011.58	489.35

Table 5: Original algorithms on OneMax

Algorithm	Parameters	Generations	Evaluations	Geometric mean
(1 + λ) EA	$\lambda = 1$, $p = 0.0159$	7761.58	7761.58	7761.58
(1 + λ) EA	$\lambda = 11$, $p = 0.0159$	732.23	8054.53	2428.53
(1 + λ) EA	$\lambda = 16$, $p = 0.0159$	504.03	8064.51	2016.12
(1 + λ) EA	$\lambda = 100$, $p = 0.0159$	103.514	10351.4	1035.14
(1 + λ) EA	$\lambda = 400$, $p = 0.0159$	52.85	21141.6	1057.04
(1 + λ) EA	$\lambda = 600$, $p = 0.0159$	48.19	28914.0	1180.41
(1 + λ) EA with self-adjusting mutation rate	$\lambda = 16$, $p = 0.0159$, $F = 1.2$	599.43	9590.88	2397.72
(1 + λ) EA with self-adjusting mutation rate	$\lambda = 100$, $p = 0.0159$, $F = 1.2$	122.15	12215.2	1221.51
(1 + λ) EA with self-adjusting mutation rate	$\lambda = 400$, $p = 0.0159$, $F = 1.2$	56.54	22615.2	1130.78
(1 + λ) EA with self-adjusting mutation rate	$\lambda = 400$, $p = 0.0159$, $F = 2$	63.0	25200.8	1260.02
(1 + λ) EA with self-adjusting mutation rate	$\lambda = 600$, $p = 0.0159$, $F = 1.2$	49.0	29401.2	1200.27

$(1+\lambda)$ EA with self-adjusting offspring size	$\lambda = 1$, $p = 0.0159$, $G = 1.1$	2741.40	7769.60	4615.15
$(1+\lambda)$ EA with self-adjusting offspring size	$\lambda = 1$, $p = 0.0159$, $G = 2$	1545.0	8153.21	3549.18
$(1 + (\lambda, \lambda))$ GA	$\lambda = 8$, $p = 0.08$, $c = 0.125$	917.48	14679.62	3669.91
$(1 + (\lambda, \lambda))$ GA	$\lambda = 8$, $p = 0.06$, $c = 0.16667$	877.0	14031.97	3508.0
$(1 + (\lambda, \lambda))$ GA	$\lambda = 16$, $p = 0.16$, $c = 0.0625$	534.94	17118.14	3026.08
$(1 + (\lambda, \lambda))$ GA	$\lambda = 16$, $p = 0.06$, $c = 0.16667$	486.72	15575.04	2753.30
$(1 + (\lambda, \lambda))$ GA	$\lambda = 100$, $p = 0.999$, $c = 0.01$	110.98	22196.0	1569.49
$(1 + (\lambda, \lambda))$ GA	$\lambda = 100$, $p = 0.06$, $c = 0.16667$	189.102	37820.4	2674.31
$(1 + (\lambda, \lambda))$ GA	$\lambda = 400$, $p = 0.06$, $c = 0.16667$	112.79	90235.2	3190.24
$(1 + (\lambda, \lambda))$ GA	$\lambda = 600$, $p = 0.06$, $c = 0.16667$	98.87	118596.0	3424.26
self-adjusting $(1 + (\lambda, \lambda))$ GA	$\lambda = 1$, $G = 1.5$	288.53	19198.44	2353.58
self-adjusting $(1 + (\lambda, \lambda))$ GA	$\lambda = 1$, $G = 1.2$	333.52	18641.16	2493.43

Table 6: Original algorithms on LeadingOnes

Algorithm	Parameters	Generations	Evaluations	Geometric mean
$(1+\lambda)$ EA	$\lambda = 1$, $p = 0.01$	3391.68	3391.68	3391.68
$(1+\lambda)$ EA	$\lambda = 11$, $p = 0.01$	37.15	408.67	123.22
$(1+\lambda)$ EA	$\lambda = 16$, $p = 0.01$	31.91	510.53	127.64
$(1+\lambda)$ EA	$\lambda = 100$, $p = 0.01$	6.89	689.2	68.91
$(1+\lambda)$ EA	$\lambda = 400$, $p = 0.01$	1.86	744	37.2

$(1 + \lambda)$ EA	$\lambda = 600$, $p = 0.01$	3.42	2052.0	83.77
$(1 + \lambda)$ EA with self-adjusting mutation rate	$\lambda = 16$, $p = 0.01$, $F = 1.2$	10.68	170.82	42.71
$(1 + \lambda)$ EA with self-adjusting mutation rate	$\lambda = 100$, $p = 0.01$, $F = 1.2$	80.49	8049.4	804.92
$(1 + \lambda)$ EA with self-adjusting mutation rate	$\lambda = 400$, $p = 0.01$, $F = 1.2$	2.59	1034.4	51.76
$(1 + \lambda)$ EA with self-adjusting mutation rate	$\lambda = 400$, $p = 0.01$, $F = 2$	6.77	2708.8	135.42
$(1 + \lambda)$ EA with self-adjusting mutation rate	$\lambda = 600$, $p = 0.01$, $F = 1.2$	10.69	6416.4	261.90
$(1 + \lambda)$ EA with self-adjusting offspring size	$\lambda = 1$, $p = 0.01$, $G = 1.1$	65.91	4638.60	552.93
$(1 + \lambda)$ EA with self-adjusting offspring size	$\lambda = 1$, $p = 0.01$, $G = 2$	17.14	1036.92	133.31
$(1 + (\lambda, \lambda))$ GA	$\lambda = 8$, $p = 0.08$, $c = 0.125$	995.06	15920.96	3980.24
$(1 + (\lambda, \lambda))$ GA	$\lambda = 8$, $p = 0.06$, $c = 0.16667$	479.16	7666.53	1916.64
$(1 + (\lambda, \lambda))$ GA	$\lambda = 16$, $p = 0.16$, $c = 0.0625$	61.35	1963.2	347.05
$(1 + (\lambda, \lambda))$ GA	$\lambda = 16$, $p = 0.06$, $c = 0.16667$	36.24	1159.62	205.0
$(1 + (\lambda, \lambda))$ GA	$\lambda = 100$, $p = 0.999$, $c = 0.01$	7.65	1530.0	108.19
$(1 + (\lambda, \lambda))$ GA	$\lambda = 100$, $p = 0.06$, $c = 0.16667$	9.18	1835.2	129.80
$(1 + (\lambda, \lambda))$ GA	$\lambda = 400$, $p = 0.06$, $c = 0.16667$	6.58	5265.6	186.14
$(1 + (\lambda, \lambda))$ GA	$\lambda = 600$, $p = 0.06$, $c = 0.16667$	6.58	7898.4	227.97

self-adjusting (1 + (λ, λ)) GA	$\lambda = 1$, $G = 1.5$	61.02	1571.46	309.66
self-adjusting (1 + (λ, λ)) GA	$\lambda = 1$, $G = 1.2$	134.06	7437.62	998.54

Table 7: Original algorithms on Makespan Scheduling

Algorithm	Parameters	Generations	Evaluations	Geometric mean (gm) Success rate (sr)

(1 + λ) EA	$\lambda = 1$,
	$p = 0.01$
		4197.07	4197.07	gm = 4197.07,
				sr = 99%

(1 + λ) EA	$\lambda = 11$,
	$p = 0.01$
		368.92	4058.13	gm = 1223.57,
				sr = 98.6%

(1 + λ) EA	$\lambda = 16$,
	$p = 0.01$
		322.94	5167.11	gm = 1291.77,
				sr = 97.6%

(1 + λ) EA	$\lambda = 100$,
	$p = 0.01$
		55.23	5523.03	gm = 552.30,
				sr = 99%

(1 + λ) EA	$\lambda = 400$,
	$p = 0.01$
		23.32	9330.04	gm = 466.45,
				sr = 97.2%

(1 + λ) EA	$\lambda = 600$,
	$p = 0.01$
		15.95	9570.67	gm = 390.71,
				sr = 98.2%

(1 + λ) EA with
self-adjusting
mutation rate
	$\lambda = 16$,
	$p = 0.01$,
	$F = 1.2$
		204.84	3277.46	gm = 819.36,
				sr = 99.6%

(1 + λ) EA with
self-adjusting
mutation rate
	$\lambda = 100$,
	$p = 0.01$,
	$F = 1.2$
		42.24	4223.67	gm = 422.38,
				sr = 98%

(1 + λ) EA with
self-adjusting
mutation rate
	$\lambda = 400$,
	$p = 0.01$,
	$F = 1.2$
		13.70	5478.23	gm = 273.96,
				sr = 99.2%

(1 + λ) EA with
self-adjusting
mutation rate
	$\lambda = 400$,
	$p = 0.01$,
	$F = 2$
		12.12	4848.19	gm = 242.40,
				sr = 99.6%

(1 + λ) EA with
self-adjusting
mutation rate
	$\lambda = 600$,
	$p = 0.01$,
	$F = 1.2$
		12.30	7378.67	gm = 301.26,
				sr = 99.4%

(1 + λ) EA with
self-adjusting
offspring size
	$\lambda = 1$,
	$p = 0.01$,
	$G = 1.1$
		111.36	24320.56	gm = 1645.70,
				sr = 100%

(1 + λ) EA with
self-adjusting
offspring size
	$\lambda = 1$,
	$p = 0.01$,
	$G = 2$
		59.27	19065.25	gm = 1063.01,
				sr = 100%

(1 + (λ, λ)) GA	$\lambda = 8$,
	$p = 0.08$,
	$c = 0.125$
		490.58	7849.34	gm = 1962.33,
				sr = 99%

(1 + (λ, λ)) GA	$\lambda = 8$,
	$p = 0.06$,
	$c = 0.16667$
		515.21	8243.29	gm = 2060.83,
				sr = 98.2%

(1 + (λ, λ)) GA	$\lambda = 16$, $p = 0.16$, $c = 0.0625$	296.10	9475.17	gm = 1674.99, sr = 99%
(1 + (λ, λ)) GA	$\lambda = 16$, $p = 0.06$, $c = 0.16667$	250.20	8006.35	gm = 1415.34, sr = 97.8%
(1 + (λ, λ)) GA	$\lambda = 100$, $p = 0.999$, $c = 0.01$	52.02	10403.73	gm = 735.66, sr = 96.6%
(1 + (λ, λ)) GA	$\lambda = 100$, $p = 0.06$, $c = 0.16667$	48.12	9623.27	gm = 680.49, sr = 98%
(1 + (λ, λ)) GA	$\lambda = 400$, $p = 0.06$, $c = 0.16667$	22.37	17899.60	gm = 632.78, sr = 98.8%
(1 + (λ, λ)) GA	$\lambda = 600$, $p = 0.06$, $c = 0.16667$	18.29	21949.8	gm = 633.61, sr = 98.8%
self-adjusting (1 + (λ, λ)) GA	$\lambda = 1$, $G = 1.5$	162.33	8361.34	gm = 1165.03, sr = 97.6%
self-adjusting (1 + (λ, λ)) GA	$\lambda = 1$, $G = 1.2$	189.17	7865.16	gm = 1219.78, sr = 98.4%

Table 8: Original algorithms on Makespan Scheduling with special initialization

Algorithm	Parameters	Evaluations (e) Generations (g) Geometric mean (gm)	Success rate (sr) Average fitness (af)
(1 + λ) EA	$\lambda = 1$, $p = 0.01$	$e = 13195$ $g = 13195$ $gm = 13195$	$sr = 0.2\%$ $af = 612.64$
(1 + λ) EA	$\lambda = 11$, $p = 0.01$	$e = 14201$ $g = 1291$ $gm = 4281.76$	$sr = 0.2\%$ $af = 626.21$
(1 + λ) EA	$\lambda = 16$, $p = 0.01$	$e = 14480$ $g = 905$ $gm = 3620$	$sr = 0.2\%$ $af = 624.06$
(1 + λ) EA	$\lambda = 100$, $p = 0.01$	$e = 19250$ $g = 192.5$ $gm = 1925$	$sr = 1.2\%$ $af = 745.88$
(1 + λ) EA	$\lambda = 400$, $p = 0.01$	$e = 36780.20$ $g = 91.95$ $gm = 1839.0$	$sr = 20.2\%$ $af = 1284.75$
(1 + λ) EA	$\lambda = 600$, $p = 0.01$	$e = 49603.57$ $g = 82.67$ $gm = 2025.02$	$sr = 33.6\%$ $af = 1561.25$

$(1+\lambda)$ EA with self-adjusting mutation rate	$\lambda = 16$, $p = 0.01$, $F = 1.2$	N/A	$sr = 0\%$ $af = 611.84$
$(1+\lambda)$ EA with self-adjusting mutation rate	$\lambda = 100$, $p = 0.01$, $F = 1.2$	$e = 23100$ $g = 231$ $gm = 2310$	$sr = 0.2\%$ $af = 675.47$
$(1+\lambda)$ EA with self-adjusting mutation rate	$\lambda = 400$, $p = 0.01$, $F = 1.2$	$e = 40730$ $g = 101.83$ $gm = 2036.55$	$sr = 8\%$ $af = 1004.59$
$(1+\lambda)$ EA with self-adjusting mutation rate	$\lambda = 400$, $p = 0.01$, $F = 2$	$e = 54320$ $g = 135.8$ $gm = 2716$	$sr = 3\%$ $af = 880.19$
$(1+\lambda)$ EA with self-adjusting mutation rate	$\lambda = 600$, $p = 0.01$, $F = 1.2$	$e = 52252.5$ $g = 87.09$ $gm = 2133.23$	$sr = 16\%$ $af = 1216.49$
$(1+\lambda)$ EA with self-adjusting offspring size	$\lambda = 1$, $p = 0.01$, $G = 1.1$	$e = 18419$ $g = 641.25$ $gm = 3436.74$	$sr = 0.8\%$ $af = 740.0$
$(1+\lambda)$ EA with self-adjusting offspring size	$\lambda = 1$, $p = 0.01$, $G = 2$	$e = 24580.19$ $g = 301.76$ $gm = 2723.48$	$sr = 4.2\%$ $af = 890.72$
$(1+(\lambda,\lambda))$ GA*	$\lambda = 8$, $p = 0.08$, $c = 0.125$	N/A	$sr = 0\%$ $af = 585.42$
$(1+(\lambda,\lambda))$ GA*	$\lambda = 8$, $p = 0.06$, $c = 0.16667$	N/A	$sr = 0\%$ $af = 628.26$
$(1+(\lambda,\lambda))$ GA*	$\lambda = 16$, $p = 0.16$, $c = 0.0625$	N/A	$sr = 0\%$ $af = 599.7$
$(1+(\lambda,\lambda))$ GA*	$\lambda = 16$, $p = 0.06$, $c = 0.16667$	N/A	$sr = 0\%$ $af = 599.7$
$(1+(\lambda,\lambda))$ GA*	$\lambda = 100$, $p = 0.999$, $c = 0.01$	N/A	$sr = 0\%$ $af = 656.82$
$(1+(\lambda,\lambda))$ GA*	$\lambda = 100$, $p = 0.06$, $c = 0.16667$	N/A	$sr = 0\%$ $af = 628.26$
$(1+(\lambda,\lambda))$ GA*	$\lambda = 400$, $p = 0.06$, $c = 0.16667$	N/A	$sr = 0\%$ $af = 649.68$
$(1+(\lambda,\lambda))$ GA*	$\lambda = 600$, $p = 0.06$, $c = 0.16667$	N/A	$sr = 0\%$ $af = 678.24$
self-adjusting $(1+(\lambda,\lambda))$ GA*	$\lambda = 1$, $G = 1.5$	N/A	$sr = 0\%$ $af = 642.54$

self-adjusting $(1 + (\lambda, \lambda))$ GA*	$\lambda = 1$, $G = 1.2$	N/A	$sr = 0\%$ $af = 638.97$

Table 9: Original algorithms on SufSamp

Algorithm	Parameters	Generations	Evaluations	Geometric mean
$(1 + \lambda)$ EA with self-adjusting mutation rate modified	$\lambda = 11$, $p = 0.01$, $F = 1.2$	128.69	1544.28	445.80
$(1 + \lambda)$ EA with self-adjusting mutation rate modified	$\lambda = 16$, $p = 0.01$, $F = 1.2$	103.05	1648.77	412.20
$(1 + \lambda)$ EA with self-adjusting mutation rate modified	$\lambda = 100$, $p = 0.01$, $F = 1.2$	35.76	3576.0	357.6
$(1 + \lambda)$ EA with self-adjusting mutation rate modified	$\lambda = 400$, $p = 0.01$, $F = 1.2$	23.48	9392.0	469.6
$(1 + \lambda)$ EA with self-adjusting mutation rate modified	$\lambda = 600$, $p = 0.01$, $F = 1.2$	21.09	12652.8	516.57
$(1 + \lambda)$ EA with self-adjusting offspring size modified 1	$\lambda = 1$, $p = 0.01$, $G = 1.1$	173.12	1214.73	458.58
$(1 + \lambda)$ EA with self-adjusting offspring size modified 1	$\lambda = 1$, $p = 0.01$, $G = 2$	124.52	1388.07	415.74
$(1 + \lambda)$ EA with self-adjusting offspring size modified 2	$\lambda = 1$, $p = 0.01$, $G = 1.1$	488.04	1105.74	734.61
$(1 + \lambda)$ EA with self-adjusting offspring size modified 2	$\lambda = 1$, $p = 0.01$, $G = 2$	205.03	1330.0	522.19
$(1 + \lambda)$ EA with self-adjusting offspring size modified 3	$\lambda = 1$, $p = 0.01$, $G = 1.1$	409.05	1092.64	668.54

$(1 + \lambda)$ EA with self-adjusting offspring size modified 3	$\lambda = 1$, $p = 0.01$, $G = 2$	285.07	1117.24	564.35
$(1 + \lambda)$ EA with self-adjusting offspring size and mutation rate	$\lambda = 1$, $p = 0.01$, $F = 1.1$ $G = 1.5$	111.43	1624.0	425.40
self-adjusting $(1 + (\lambda, \lambda))$ GA modified 1	$\lambda = 3$, $G = 1.5$	98.64	769.43	275.49
self-adjusting $(1 + (\lambda, \lambda))$ GA modified 2	$\lambda = 3$, $G = 1.5$	82.52	767.24	251.62
self-adjusting $(1 + (\lambda, \lambda))$ GA modified 3	$\lambda = 1$, $G = 1.5$	218.29	1502.08	572.62
self-adjusting $(1 + (\lambda, \lambda))$ GA modified 4	$\lambda = 1$, $G = 1.5$	197.15	902.2	421.75

Table 10: Modified algorithms on OneMax

Algorithm	Parameters	Generations	Evaluations	Geometric mean
$(1 + \lambda)$ EA with self-adjusting mutation rate modified	$\lambda = 11$, $p = 0.01$, $F = 1.2$	736.85	8842.22	2552.53
$(1 + \lambda)$ EA with self-adjusting mutation rate modified	$\lambda = 16$, $p = 0.01$, $F = 1.2$	556.59	8905.44	2226.36
$(1 + \lambda)$ EA with self-adjusting mutation rate modified	$\lambda = 100$, $p = 0.01$, $F = 1.2$	114.85	11485.0	1148.5
$(1 + \lambda)$ EA with self-adjusting mutation rate modified	$\lambda = 400$, $p = 0.01$, $F = 1.2$	56.77	22707.2	1135.38
$(1 + \lambda)$ EA with self-adjusting mutation rate modified	$\lambda = 600$, $p = 0.01$, $F = 1.2$	52.03	31216.8	1274.45
$(1 + \lambda)$ EA with self-adjusting offspring size modified 1	$\lambda = 1$, $p = 0.01$, $G = 1.1$	3059.70	8731.35	5168.69

$(1 + \lambda)$ EA with self-adjusting offspring size modified 1	$\lambda = 1$, $p = 0.01$, $G = 2$	1924.73	8152.97	3961.35
$(1 + \lambda)$ EA with self-adjusting offspring size modified 2	$\lambda = 1$, $p = 0.01$, $G = 1.1$	6281.13	8689.776	7387.94
$(1 + \lambda)$ EA with self-adjusting offspring size modified 2	$\lambda = 1$, $p = 0.01$, $G = 2$	2570.69	8050.86	4549.31
$(1 + \lambda)$ EA with self-adjusting offspring size modified 3	$\lambda = 1$, $p = 0.01$, $G = 1.1$	5864.69	8718.63	7150.67
$(1 + \lambda)$ EA with self-adjusting offspring size modified 3	$\lambda = 1$, $p = 0.01$, $G = 2$	4001.33	7854.02	5605.94
$(1 + \lambda)$ EA with self-adjusting offspring size and mutation rate	$\lambda = 1$, $p = 0.01$, $F = 1.1$ $G = 1.5$	1448.74	9606.39	3730.57
self-adjusting $(1 + (\lambda, \lambda))$ GA modified 1	$\lambda = 3$, $G = 1.5$	273.84	16762.83	2142.51
self-adjusting $(1 + (\lambda, \lambda))$ GA modified 2	$\lambda = 3$, $G = 1.5$	193.59	17932.28	1863.20
self-adjusting $(1 + (\lambda, \lambda))$ GA modified 3	$\lambda = 1$, $G = 1.5$	290.84	11380.51	1819.32
self-adjusting $(1 + (\lambda, \lambda))$ GA modified 4	$\lambda = 1$, $G = 1.5$	287.35	18477.98	2304.27

Table 11: Modified algorithms on LeadingOnes

Algorithm	Parameters	Generations	Evaluations	Geometric mean (gm) Success rate (sr)
$(1 + \lambda)$ EA with self-adjusting mutation rate modified	$\lambda = 11$, $p = 0.01$, $F = 1.2$	409.29	4911.48	gm = 1417.82, sr = 100%

(1 + λ) EA with self-adjusting mutation rate modified	$\lambda = 16$, $p = 0.01$, $F = 1.2$	314.25	5028.0	gm = 1257.0, sr = 100%
(1 + λ) EA with self-adjusting mutation rate modified	$\lambda = 100$, $p = 0.01$, $F = 1.2$	56.19	5619.44	gm = 561.92, sr = 99.8%
(1 + λ) EA with self-adjusting mutation rate modified	$\lambda = 400$, $p = 0.01$, $F = 1.2$	15.60	6240.8	gm = 312.02, sr = 100%
(1 + λ) EA with self-adjusting mutation rate modified	$\lambda = 600$, $p = 0.01$, $F = 1.2$	13.26	7958.72	gm = 324.86, sr = 99.8%
(1 + λ) EA with self-adjusting offspring size modified 1	$\lambda = 1$, $p = 0.01$, $G = 1.1$	116.07	7317.79	gm = 921.62, sr = 99.8%
(1 + λ) EA with self-adjusting offspring size modified 1	$\lambda = 1$, $p = 0.01$, $G = 2$	68.47	11486.96	gm = 886.86, sr = 99.8%
(1 + λ) EA with self-adjusting offspring size modified 2	$\lambda = 1$, $p = 0.01$, $G = 1.1$	181.54	7731.61	gm = 1184.73, sr = 99.8%
(1 + λ) EA with self-adjusting offspring size modified 2	$\lambda = 1$, $p = 0.01$, $G = 2$	84.80	9759.38	gm = 909.72, sr = 99.2%
(1 + λ) EA with self-adjusting offspring size modified 3	$\lambda = 1$, $p = 0.01$, $G = 1.1$	223.61	10539.67	gm = 1535.18, sr = 99.6%
(1 + λ) EA with self-adjusting offspring size modified 3	$\lambda = 1$, $p = 0.01$, $G = 2$	117.14	6309.49	gm = 859.71, sr = 99.4%
(1 + λ) EA with self-adjusting offspring size and mutation rate	$\lambda = 1$, $p = 0.01$, $F = 1.1$ $G = 1.5$	41.44	5956.75	gm = 496.84, sr = 100%
self-adjusting (1 + (λ, λ)) GA modified 1	$\lambda = 3$, $G = 1.5$	124.62	15342.54	gm = 1382.75, sr = 99.2%

self-adjusting
$(1 + (\lambda, \lambda))$ GA
modified 2
	$\lambda = 3$,
	$G = 1.5$
		109.04	13027.35	gm = 1191.85,
				sr = 99.6%

self-adjusting
$(1 + (\lambda, \lambda))$ GA
modified 3
	$\lambda = 1$,
	$G = 1.5$
		186.89	7362.7	gm = 1173.04,
				sr = 99.6%

self-adjusting
$(1 + (\lambda, \lambda))$ GA
modified 4
	$\lambda = 1$,
	$G = 1.5$
		204.94	18333.75	gm = 1938.38,
				sr = 100%

Table 12: Modified algorithms on Makespan Scheduling with special initialization

Algorithm	Parameters	Evaluations (e) Generations (g) Geometric mean (gm)	Success rate (sr) Average fitness (af)
$(1 + \lambda)$ EA with self-adjusting mutation rate modified	$\lambda = 11$, $p = 0.01$, $F = 1.2$	N/A	$sr = 0\%$ $af = 621.12$
$(1 + \lambda)$ EA with self-adjusting mutation rate modified	$\lambda = 16$, $p = 0.01$, $F = 1.2$	N/A	$sr = 0\%$ $af = 630.40$
$(1 + \lambda)$ EA with self-adjusting mutation rate modified	$\lambda = 100$, $p = 0.01$, $F = 1.2$	$e = 248.0$ $g = 24800.0$ $gm = 2480.0$	$sr = 0.6\%$ $af = 707.06$
$(1 + \lambda)$ EA with self-adjusting mutation rate modified	$\lambda = 400$, $p = 0.01$, $F = 1.2$	$e = 40049.23$ $g = 100.12$ $gm = 2002.43$	$sr = 13\%$ $af = 1128.16$
$(1 + \lambda)$ EA with self-adjusting mutation rate modified	$\lambda = 600$, $p = 0.01$, $F = 1.2$	$e = 35957.65$ $g = 59.93$ $gm = 1467.97$	$sr = 17\%$ $af = 1247.17$
$(1 + \lambda)$ EA with self-adjusting offspring size modified 1	$\lambda = 1$, $p = 0.01$, $G = 1.5$	$e = 21447.8$ $g = 401.4$ $gm = 2934.13$	$sr = 2\%$ $af = 807.64$
$(1 + \lambda)$ EA with self-adjusting offspring size modified 1	$\lambda = 1$, $p = 0.01$, $G = 2$	$e = 23821.83$ $g = 403.67$ $gm = 3100.99$	$sr = 2.4\%$ $af = 817.81$
$(1 + \lambda)$ EA with self-adjusting offspring size modified 2	$\lambda = 1$, $p = 0.01$, $G = 1.5$	N/A	$sr = 0\%$ $af = 614.69$

$(1 + \lambda)$ EA with self-adjusting offspring size modified 2	$\lambda = 1$, $p = 0.01$, $G = 2$	N/A	$sr = 0\%$ $af = 628.97$
$(1 + \lambda)$ EA with self-adjusting offspring size modified 3	$\lambda = 1$, $p = 0.01$, $G = 1.5$	$e = 22389.0$ $g = 1104.0$ $gm = 4971.67$	$sr = 0.2\%$ $af = 655.48$
$(1 + \lambda)$ EA with self-adjusting offspring size modified 3	$\lambda = 1$, $p = 0.01$, $G = 2$	$e = 17268.33$ $g = 732.67$ $gm = 3556.96$	$sr = 0.6\%$ $af = 670.65$
$(1 + \lambda)$ EA with self-adjusting offspring size and mutation rate	$\lambda = 1$, $p = 0.01$, $F = 1.1$ $G = 1.5$	$e = 29006.14$ $g = 301.29$ $gm = 2956.22$	$sr = 2.8\%$ $af = 829.41$
self-adjusting $(1 + (\lambda, \lambda))$ GA modified 1	$\lambda = 3$, $G = 1.5$	N/A	$sr = 0\%$ $af = 713.94$
self-adjusting $(1 + (\lambda, \lambda))$ GA modified 2	$\lambda = 3$, $G = 1.5$	N/A	$sr = 0\%$ $af = 649.68$
self-adjusting $(1 + (\lambda, \lambda))$ GA modified 3	$\lambda = 1$, $G = 1.5$	N/A	$sr = 0\%$ $af = 606.84$
self-adjusting $(1 + (\lambda, \lambda))$ GA modified 4	$\lambda = 1$, $G = 1.5$	N/A	$sr = 0\%$ $af = 656.82$

Table 13: Modified algorithms on SufSamp