Universiteit Leiden
The Netherlands

Opleiding Informatica

Multi-valued decision variables and redundant coding

in evolutionary algorithms

Jaap Blok

Supervisors:
Prof.dr. T.H.W. Bäck & F. Ye MSc.

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

15/08/2019

Abstract

In this thesis we explore the relations between expected optimisation time (ERT) and multi-valued decision variables for several different Evolutionary Algorithms (EA) through empirical testing. This is achieved by running experiments on the IOHprofiler, a benchmark platform, under various parameter settings. A OneMax-like benchmark problem is considered, featuring a multi-valued search space where all variables are in the discrete domain $\{0, \ldots, r-1\}$. This problem is analogous to the classic OneMax problem, in the sense that it asks to optimise $F : \{0, \ldots, r-1\}^n \to \{0, \ldots, n(r-1)\}; \mathbf{x} \mapsto \sum_{i=1}^{n} x_i$. We find that the relationship between r and ERT is dependent on the algorithm in question, and in every experimental case leads to an increase in ERT. Lastly, we explore the effects on the ERT of redundant coding, where solution variables x_i can be in a larger integer domain than the problem variables. They are then mapped to the same domain as the problem variables through a modulo operation. For this experiment, a second benchmark problem is proposed, which asks to optimise $F : \{0, \ldots, r_s-1\}^n \to \{0, \ldots, n(r_p-1)\}; \mathbf{x} \mapsto \sum_{i=1}^{n} x_i \% r_p$, where r_s is the size of the solution variables' domain, and r_p is the size of the problem variables' domain. We find that redundancy has a minimal impact on the performance of the tested algorithms.

Contents

Chapter 1

Introduction

As noted by Doerr et al [1], in Natural Computing, the most popular solution candidate representation is the bit-string, where the solution is encoded as a string of length n of integer values in the set $\{0, 1\}$. This is a consequence of its versatility; many optimisation problems can be stated as a series of binary decisions, which are easily encoded in a vector of binary values.

When applying an Evolutionary Algorithm to an intrinsically unorthodox problem, however, two approaches can generally be taken. The first is to use a well known solution representation for the solutions, which can then be translated to fit the problem, or the problem reformulated to fit the solution representation. The second is to choose a solution representation that does fit the problem, but may not be readily understood in its characteristics. The first method carries with it the advantage that the functioning of the algorithm is relatively well understood, while a disadvantage is that the translation or reformulation might disturb the efficiency of the optimisation process. The second method avoids these potential pitfalls, but the performance of the algorithm used may be hard to predict beforehand due to lack of available research.

A set of problems that feature multi-valued decision variables exists, where the variables that make up a solution can take on more than two values, as is the case for the problems featuring binary decisions. For these problems the representation of solutions can consist of a string of integer values within a domain $\{0, \ldots, r-1\}$, where r is the size of the domain. The optimisation time for the $(1+1)$ RLS and $(1+1)$ EA that use these decision variables on a problem analogous to OneMax has been theoretically proven to be $\Theta(rn\log n)$ [1]. To both confirm these findings and expand slightly beyond the existing scope, results will be gathered through empirical IOHprofiler experiments on the same type of optimisation problem in this thesis.

In many cases, a problem which features multi-valued decision variables will not have an equal number of possible values for every decision variable. In these cases a solution representation $\mathbf{x}$ can be used with a domain size r_s for each solution variable x_i in $\mathbf{x}$ that is as large as, or larger than the domain $\{0, \ldots, r_p - 1\}$ of each problem variable. This representation can be mapped to fit the search space of the problem through a modulo operation. The effects of this encoding with a larger variable size (redundant coding) will be tentatively explored through IOHprofiler experiments in this thesis.

1.1 Applications

For optimisation problems where the problem can be stated as a series of multi-valued decisions, algorithms can be applied that represent solutions as integer vectors. An example of this type of problem would be Grammatical Evolution [8], where integer values represent choices in production rules. Increasing understanding of the effects of using integer values can serve as a guide for improving algorithm design and as baseline for further analysis of algorithms that search a multi-valued space. For the example of Grammatical Evolution specifically, redundant coding was used. This may have had an impact on the optimisation time, but as the paper by O'Neill et al [8] states, this is not yet clear at the time of writing. A better understanding of the effects of redundant coding may, again, help improve algorithm design, and support further analysis of the optimisation time of Grammatical Evolution algorithms.

1.2 Thesis Overview

In this thesis the relation between ERT and decision variable size will be investigated, as well as the relation between ERT, decision variable size and redundant coding. The academic context will be explained in chapter 2, the testing platform will be touched upon in chapter 3, the optimisation problem will be defined in chapter 4, the algorithms used in chapter 5. The experiments themselves, including the results, are outlined in chapter 6, and the conclusions drawn from them in chapter 7. Finally, suggestions for further research can be found in chapter 8. This work serves as a bachelor thesis under supervision of Prof.dr. T.H.W. Bäck and Furong Ye Msc.

Chapter 2

Related Work

The mutation operator used in this thesis has had its effect on running time of the $(1+1)$ EA theoretically proven for the value of $r = 3$ [3]. This algorithm changes solution variables to a random different value in a domain $\{0, \ldots, r-1\}$ with a probability of $1/n$, where n is the problem dimension. The authors conclude that for a domain $\{0, \ldots, 2\}$, the expected running time is $\mathcal{O}(n\log n)$. They also conclude that their approach can not be used to construct running time expectations for a higher value of r. The mutation operator has been considered for practical applications as well, e.g. in [6].

The work of Doerr et al [1] most closely coincides with this thesis. In their work, the authors note that very little theoretical work exists on the practices of using evolutionary algorithms on problems with multi-valued search spaces. Throughout their works they obtain several different ERT's of both RLS and $(1+1)$ EA algorithms with different mutation operators on OneMax-like problems, and find that a self adaptive mutation operator based on the success of previous generations offers the best results. The finding that a mutation operator which randomly selects a new value, as described in chapter 5, results in an ERT of $\Theta(rn\log n)$ is the one we will try to replicate in this thesis.

The work on grammatical evolution of, among others, O'Neill et al [8], has raised the question what the effects of their choice of solution representation are having on the running time of the algorithms. In this area, an evolutionary algorithm is used to evolve a derivation tree following grammar production rules in Backus-Naur-form. Every solution variable represents a decision of which production rule is applied to build the next part of the tree. Due to the amount of alternative choices from a grammar rule being unpredictable, every solution variable has to have a large enough size to fit the grammar rule with the highest amount of alternatives. To always have the solution variable apply to the choice at hand, the solution allows solution variables in a domain of integers that is at least as large as the grammar rule with the highest amount of alternatives. These values are then mapped ad hoc to the domain of the problem variable choices through a modulo operation, creating a manner of redundancy in the encoding. This thesis aims to shed light on the effect of this redundancy on the ERT as well.

Chapter 3

IOHprofiler

To perform the experiments the IOHprofiler (Iterative Optimisation Heuristics), which consists of the IOHexperimenter and IOHanalyzer [4], was used. These form a software platform that can be used as a tool for analysing the performance of a given set of algorithms on a given set of problems. With given implementation of algorithms and problems, the IOHexperimenter can generate formatted performance data that can then be interpreted by the IOHanalyzer. The IOHanalyzer generates and visualises statistics of the algorithms' performance for fixed-target running time and fixed-budget function values. The statistic we are most interested in in this thesis is the expected optimisation time. The IOHprofiler can record the optimisation times during a number of subsequent runs, and then give the average of those times for any fixed target value. In this case the optimal target value was recorded, creating the statistic of interest. Furthermore, because the IOHprofiler includes a number of usable optimisation problems including OneMax, modifications to the existing OneMax file served to create an implementation of our two benchmark problems, as will be discussed in chapter 4. For further information on the capabilities and design choices, see [5].

Chapter 4

Benchmark functions

For problems with a binary search space, the OneMax problem serves as a classic benchmark function. The problem that is proposed and experimented with for this thesis is analogous to the OneMax function, from here on called "OneMax-like". In [1], an identical function has been considered. Where for both problems the solution candidates $\mathbf{x}$ are evaluated by the sum of their constituent variables x_i, the OneMax function seeks to maximise the number of $1's$ (or $0's$, by symmetry), while the OneMax-like seeks to maximise the values of the variables to a maximal value of $r-1$. The problem is:

$$F : \{0, \ldots, r-1\}^n \to \{0, \ldots, n(r-1)\}; \mathbf{x} \mapsto \sum_{i=1}^{n} x_i$$

As such, solutions for a OneMax problem can be encoded as a binary string, while for the OneMax-like problem solutions can be encoded as a string of integers in a domain $\{0, \ldots, r-1\}$, where the size of the domain r can be determined beforehand and $r-1$ can be considered the maximal value of a single variable x_i. When a maximal value of $r = 2$ is chosen, the problem is the OneMax function. In every case where the binary case is tested, OneMax-like implementation is used to simulate OneMax. An important difference in these functions to note is the fact that the OneMax-like allows variables to mutate to an intermediate value, if not the optimal value, if it is considered an improvement on the parent fitness as per the evaluation function.

For gathering insight in the effects of redundant coding on the ERT, a second problem is considered which further modifies a OneMax-like problem. For this problem a distinction needs to be made between solution variable size r_s and problem variable size r_p. In order to handle solutions with variable values beyond an optimal value r_p, it maps every solution variable x_i through a modulo operation $\%r_p$ to a value equal to or lower than $r_p - 1$. To allow for the probability for every problem variable value to occur after the modulo mapping to be equal, we have introduced the additional rule that the solution variable domain r_s should always be a multitude of r_p, such that it always is $r_p m$. In an exact definition, it asks to optimise:

$$F : \{0, \ldots, r_s-1\}^n \to \{0, \ldots, n(r_p-1)\}; \mathbf{x} \mapsto \sum_{i=1}^{n} x_i \% r_p$$

Chapter 5

Algorithms

A selection of algorithms with different properties has been made to explore the effect of multi-valued variables and redundant coding. As this is largely achieved through modifying the mutation operator, as described below, crossover will not be used. The algorithms used can roughly be divided by mutation rate; we use a $(1+\lambda)$ RLS and a $(1+\lambda)$ EA which feature static mutation rates, and a $(1+\lambda)$ $\text{EA}_{2/r,2r}$ and $(1+\lambda)$ $\text{EA}_{log\text{-}n}$, which feature a self adaptive mutation rate. Further details on the functioning of these algorithms will be provided further on in this chapter.

5.1 Mutation operator

In order for the algorithms we consider to search a discrete, multi-valued space, the standard mutation operator that allows a search through binary space has been modified. Where it originally flips one of the binary solution variables x_i in a candidate solution $\mathbf{x}$, it now changes a solution variable x_i that is to be mutated to a random uniformly selected different value within $[0, \ldots, r-1]$, where r is the size of the domain of integers. Practically, this means repeatedly selecting a random value within $[0, \ldots, r-1]$, until one is selected that is different from the original value of x_i. For the algorithm to find the optimal solution to a OneMax-like problem, both the size of the solution variable domain and problem variable domain need to be equal to r.

When a domain size of $r = 2$ is selected, the algorithms we use effectively simulate their binary variant, albeit computationally less efficient. This implementation has been used to gather all data, including for the performance on the binary OneMax benchmark function. The process of selecting the variable(s) x_i to be mutated has not been modified, and depends on the algorithm in question.

Algorithm 1: Pseudocode for the mutation operator

```
1 temp := x_i;
2 while x_i = temp do
3 |   x_i ~ U(0, r − 1);
```

5.2 Algorithms used

The algorithms that were used in the experiments are described in this section.

(1) Random Local Search (RLS). This algorithm uses a $(1+\lambda)$ strategy to explore its neighbouring solutions by mutating a randomly selected single value x_i. This can be flipping one bit in the binary variant, or changing one integer value in the multi-valued variant.

Algorithm 2: $(1+\lambda)$ Random Local Search algorithm

Initialization: Sample $\mathbf{x} \in \{0, \ldots, r-1\}^n$ uniformly at random and evaluate $f(\mathbf{x})$;
Optimisation: for $t = 1, 2, 3, \ldots$ **do**
  for $i = 1, \ldots, \lambda$ **do**
    create offspring $\mathbf{y}^{(i)} \leftarrow \mathbf{x}$;
    $j \sim \cup[0, n-1]$;
    mutate $(\mathbf{y}^{(i)}[j])$;
    evaluate $f(\mathbf{y}^{(i)})$;
  $\mathbf{x}^* \leftarrow \arg\max\{f(\mathbf{y}^{(1)}), \ldots, f(\mathbf{y}^{(\lambda)})\}$;
  if $f(\mathbf{x}^*) \geq f(\mathbf{x})$ **then**
    $\mathbf{x} \leftarrow \mathbf{x}^*$;

(2) $(1+\lambda)$ EA. An evolutionary algorithm with fixed mutation rate defined by a mutation probability of $1/n$ for every solution variable x_i, distributed uniformly.

Algorithm 3: $(1+\lambda)$ EA algorithm

Initialization: Sample $\mathbf{x} \in \{0, \ldots, r-1\}^n$ uniformly at random and evaluate $f(\mathbf{x})$;
Optimisation: for $t = 1, 2, 3, \ldots$ **do**
  for $i = 1, \ldots, \lambda$ **do**
    create offspring $\mathbf{y}^{(i)} \leftarrow \mathbf{x}$;
    for $j = 1, \ldots, n$ **do**
      mutate $(\mathbf{y}^{(i)}[j])$ with probability $1/n$;
    evaluate $f(\mathbf{y}^{(i)})$;
  $\mathbf{x}^* \leftarrow \arg\max\{f(\mathbf{y}^{(1)}), \ldots, f(\mathbf{y}^{(\lambda)})\}$;
  if $f(\mathbf{x}^*) \geq f(\mathbf{x})$ **then**
    $\mathbf{x} \leftarrow \mathbf{x}^*$;

(3) $(1+\lambda)$ $\mathrm{EA}_{2/r,2r}$. As proposed here [2], this algorithm introduces a variable r which controls the mutation rate. The algorithm creates half of the offspring with half the current mutation rate, and half with double the mutation rate. This value is doubled or halved depending on the selected offspring, or, with a probability of 0.5 doubled or halved at random. The value for r always stays within preset bounds.

Algorithm 4: $(1+\lambda)$ $\mathrm{EA}_{2/r,2r}$

Initialization: Sample $\mathbf{x} \in \{0,\dots,r-1\}^n$ uniformly at random and evaluate $f(\mathbf{x})$;
Initialise $r = 2$;
Optimisation: for $t = 1, 2, 3, \dots$ **do**
    for $i = 1, \dots, \lambda$ **do**
        if $i < \lambda/2$ **then**
            sample $\ell^{(i)} \sim_{>0} (n, r/(2n))$;
        else
            sample $\ell^{(i)} \sim_{>0} (n, 2r/n)$;
        create offspring $\mathbf{y}^{(i)} \leftarrow \mathbf{x}$;
        for $j = 1, \dots, n$ **do**
            mutate $(\mathbf{y}^{(i)}[j])$ with probability $\ell^{(i)}$;
        evaluate $f(\mathbf{y}^{(i)})$;
    $\mathbf{x}^* \leftarrow \arg\max\{f(\mathbf{y}^{(1)}), \dots, f(\mathbf{y}^{(\lambda)})\}$;
    if $f(\mathbf{x}^*) \geq f(\mathbf{x})$ **then**
        $\mathbf{x} \leftarrow \mathbf{x}^*$;
    if $\mathbf{x}^*$ *has been created with mutation rate* $r/2$ **then** $s \leftarrow 3/4$ **else** $s \leftarrow 1/4$;
    Sample $q \in [0,1]$ u.a.r.;
    if $q \leq s$ **then** $r \leftarrow \max\{r/2, 2\}$ **else** $r \leftarrow \min\{2r, n/4\}$;

(4) $(1+\lambda)$ $\text{EA}_{log\text{-}n}$. As the SA3 strategy proposed here [7], this algorithm features a log-normal self adaptation rule of the mutation rate p_m following the formula, where $\gamma = 0.22$:

$$p'_m = \frac{1.0}{1.0 + \frac{1.0 - p_m}{p_m} * exp(\gamma * \mathcal{N}(0,1))}$$

Algorithm 5: $(1+\lambda)$ $\text{EA}_{log\text{-}n}$

Initialization: Sample $\mathbf{x} \in \{0, \ldots, r-1\}^n$ uniformly at random and evaluate $f(\mathbf{x})$;
Initialise mutation_rate := 0.2;
Optimisation: for $t = 1, 2, 3, \ldots$ **do**
    for $i = 1, \ldots, \lambda$ **do**
        create offspring $\mathbf{y}^{(i)} \leftarrow \mathbf{x}$;
        temp_mutation_rate := $1.0/(1.0 + (((1.0 - mutation\_rate)/mutation\_rate) * exp(0.22 * \mathcal{N}(0,1))))$;
        for $j = 1, \ldots, n$ **do**
            mutate $(\mathbf{y}^{(i)}[j])$ with probability = temp_mutation_rate;
        evaluate $f(\mathbf{y}^{(i)})$;
    $\mathbf{x}^* \leftarrow \arg\max\{f(\mathbf{y}^{(1)}), \ldots, f(\mathbf{y}^{(\lambda)})\}$;
    if $f(\mathbf{x}^*) \geq f(\mathbf{x})$ **then**
        $\mathbf{x} \leftarrow \mathbf{x}^*$;
        update mutation rate to temp_mutation_rate of $\mathbf{x}^*$;

Chapter 6

Experiments

6.1 Parameter settings

The experiments have been performed under the following parameter settings.

(1) A baseline run with traditional binary solutions is included, followed by runs with categorical variables in a domain $[0, \ldots, r-1]$. This baseline result can be found under the result for $r = 2$. Several different values for r have been tested.
(2) The experiments have been performed with a dimension of $n = 1000$.
(3) Every run has been given a budget of $500n$, in every case more than enough to reach the optimal solution.
(4) Every final result is the average of the results of 100 runs.
(5) In the case of the self adaptive $(1+\lambda)\text{EA}_{2/r,2r}$ and $(1+\lambda)$ $\text{EA}_{log\text{-}n}$, offspring counts of $\lambda = 10$, $\lambda = 50$ and $\lambda = 100$ were used to keep the functioning of the self adaptive rule intact. For the $(1+\lambda)$ RLS and $(1+\lambda)$EA offspring counts of $\lambda = 1$, $\lambda = 50$ and $\lambda = 100$ were used.
(6) For the experiments on redundant coding, an additional parameter of a modulo divisor was introduced. This has been set to values in the domain $[2, \ldots, 6]$, along with the size of the problem variables r_s. Parameter values for solution variable domain upper boundary are in this case always set to a value $10^p r_p$. The experiments with $r_s = r_p$ function effectively as if there is no modulo operation, and are used as a baseline. When these values are set to 2, the result is equal to the ERT of a binary algorithm optimising OneMax.

The exact combinations of parameters are laid out in the tables with results in section 6.3.

6.2 Expectation

As per [1], for the RLS and $(1+1)$ EA algorithms, an ERT of $\Theta(rn\log n)$ is to be expected. For variations on these algorithms, a positive relation between domain size r is to be expected, due to the problem allowing

intermediate values, whose amount only increases along with r.

When considering the similarity of the OneMax problem and our problem, an expectation of the optimisation time for a given value for r could be constructed using the expected optimisation time of the binary version $E(T)$ on the OneMax problem of the same dimensions. One way to look at the process of optimisation, is to view a one step improvement of the current best solution as one bit x_i reaching its maximal value. This occurs when this bit flips from 0 to 1. Similarly, when using categorical variables, an integer x_i will settle into its maximal value when its value mutates to $r-1$. Given that the probability of this mutation occurring is $1/(r-1)$ (a value cannot mutate to remain the same), the expected amount of mutations to a single integer for it to attain its optimal value is $r-1$. This is $(r-1)/1 = (r-1)$ times more than for the binary version. From this the expectation is raised that the relation between domain size r and binary optimisation time $E(T)$ is $(r-1)E(T)$.

In the case of using a modulo operator the same logic can be applied, with the additional consideration that a solution value can mutate to no effect on the fitness due to the modulo operation, a concept known for some time in biology [9]. While we expect the probability of a mutation to the optimal value occurring is $1/(r_p-1)$ without a modulo operation, with the modulo operation we expect it to move towards $1/r_p$, as the redundancy of the encoding increases with r_s. As a result, we expect the ratio between the non-redundant and redundant versions to approach $\frac{1/(r_p-1)}{1/r_p} = \frac{r_p}{r_p-1}$ as solution variable domain r_s is increased.

6.3 Results

r	$(1+1)$ RLS	$(1+50)$ RLS	$(1+100)$ RLS	$(1+1)$ EA	$(1+50)$ EA	$(1+100)$ EA
2	6753.03 1.00	28595.01 1.00	53210.67 1.00	10786.65 1.00	22510.1 1.00	34091.46 1.00
3	14140.05 1.05	42543.21 0.74	74566.74 0.70	22814.61 1.06	41740.39 0.93	59979.47 0.88
4	22115.09 1.09	53086.5 0.62	87520.97 0.55	34770.72 1.07	58836.09 0.87	83556.97 0.82
5	30103.99 1.11	63196.22 0.55	100134.63 0.47	46449.63 1.08	74975.37 0.83	103401.32 0.76
6	36940.88 1.09	73467.56 0.51	111349.41 0.42	58737.22 1.09	90637.77 0.81	124923.02 0.73
7	42904.12 1.06	82810.29 0.48	123602.03 0.39	72211.85 1.12	108214.52 0.80	141580.39 0.69
8	51598.56 1.09	93767.63 0.47	134618.25 0.36	84655.51 1.12	120813.66 0.77	157743.28 0.66
9	58258.21 1.08	103462.45 0.45	146542.49 0.34	95329.05 1.10	135104.05 0.75	173542.05 0.64
10	67333.64 1.11	112974.84 0.44	156067 0.33	111794.75 1.15	151370.68 0.75	189575.49 0.62
16	113865.67 1.12	167915.83 0.39	- -	187644.41 1.16	236196.43 0.70	- -
32	230121.28 1.10	301420.89 0.346	- -	394694.46 1.18	462356.52 0.66	- -
64	478307.22 1.12	569139.47 0.32	- -	792077.86 1.17	886563.55 0.63	- -

Table 6.1: **The average ERT w.r.t. r and ratio for r for the $(1+\lambda)$ RLS and $(1+\lambda)$ EA.**

In tables 6.1, 6.2 and 6.3 the average ERT can be seen as the upper value in each square. The value below represents the weight for r under the assumption that r is a multiplier of the baseline ERT result for the binary case. This calculation follows the formula $weight = \frac{ERT}{(r-1)baseERT}$.

Looking at table 6.1, for the $(1+1)$ RLS, the ERT increases with an increase of r in every case. For the algorithms with higher offspring count, the ERT seems to increase by a lower factor than for algorithms with lower offspring count. The $(1+1)$ RLS performs better than the $(1+1)$ EA for every r that we have data for. For the cases with higher offspring count the RLS performs worse than the EA for a value of r up to $r = 3$ for $\lambda = 50$, and $r = 4$ for $\lambda = 100$, after which the RLS performs better in terms of ERT.

As can be seen in table 6.1 as well, the factor of r in the ERT of the $(1+1)$ RLS seems to increase for the first few experiments, then fluctuates when increasing r. Its weight is in every case higher than the baseline weight of 1 for the binary case. A possible explanation may be that the last part of optimisation, where progress costs the most time, is disproportionately affected by the decreased chance to mutate a variable to its optimal value, resulting in a large variance between results. A positive relation between r and ERT was expected, as explained in chapter 6, but the non-linearity was not. The factor of r in the ERT in our test does not dispute the theoretical result in [1], however; the results do not suggest quadratic or exponential behaviour.

For the RLS variants with $\lambda = 50$ and $\lambda = 100$, an inverse pattern can be seen, where the the factor of r on the

ERT declines at a diminishing rate with the increase of r. A possible explanation starts with the fact that a larger variable domain size r allows for more possible mutations. An algorithm with higher offspring count may be less affected by this, by increasing the probability of a large improvement when compared to a low offspring count. These results do not seem in line with the ERT as expected in chapter 6, which describes a linear relation. In a different comparison, an increase in offspring population size λ generally seems to have a negative effect on the ERT. Looking at figure 6.1, the $(1+1)$ RLS makes progress the fastest during the start of the optimisation process and slows down near the end, while the $(1+50)$ RLS and $(1+100)$ RLS perform in a more stable manner.

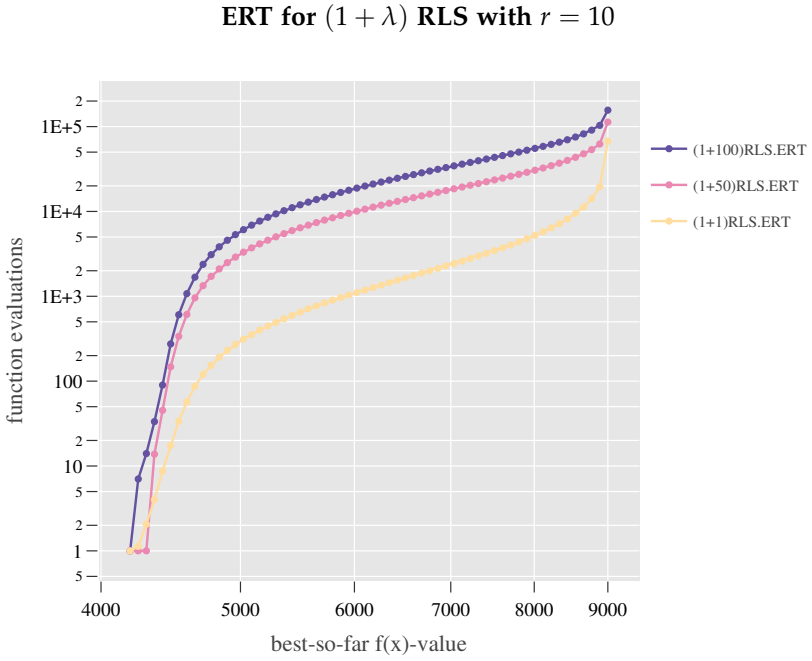

Figure 6.1: **The expected running time under various settings for λ and $n = 1000$.**

For the $(1+\lambda)$ EA algorithm, results are similar to the RLS for $\lambda = 1$; when considering r as a factor of the ERT, it increases for the first few experiments, then starts to fluctuate. As can be seen in table 6.1, for $\lambda = 50$ and $\lambda = 100$, the factor of r decreases at a diminishing rate when increasing r, though not as much as for the RLS. This suggests, again, that the increased offspring count dampens the effects of more variance in mutations. Continuing this line of thinking, this may explain the slower decrease of the weight for r, as the EA algorithm already features a greater variance in mutations than the RLS, and profits less relatively from the wider range of possible mutations caused by increasing r. Lastly, while the $(1+50)$ and $(1+100)$ EA's perform better with lower r, with increasing the value for r they are overtaken in terms of performance in ERT by the $(1+50)$ and $(1+100)$ RLS respectively, possibly due to a more favourable balance of offspring count and mutation variance.

r	$(1+10)$ $\mathrm{EA}_{2/r,2r}$	$(1+50)$ $\mathrm{EA}_{2/r,2r}$	$(1+100)$ $\mathrm{EA}_{2/r,2r}$
2	23879.69 1.00	32044.26 1.00	40644 1.00
3	51103.76 1.07	63207.53 0.99	80353.36 0.99
4	80070.76 1.12	97662.39 1.02	118315.89 0.97
5	110149.44 1.15	128910.72 1.01	153954.71 0.95
6	139044.01 1.16	162626.74 1.02	186797.62 0.92
7	165141.67 1.15	190185.38 0.99	227811.83 0.93
8	198914.86 1.19	220680.11 0.98	258001.62 0.91
9	230028.07 1.20	252038.94 0.98	291534.93 0.90
10	256784.56 1.19	291672.61 1.01	314842.1 0.86
16	440777.03 1.23	470068.04 0.98	-
32	922135.85 1.25	963601.39 0.97	-
64	1927901.54 1.28	1994607.28 0.99	-

Table 6.2: **The average ERT w.r.t. r and ratio for r for the $(1+\lambda)$ $\mathbf{EA}_{2/r,2r}$.**

r	$(1+10)$ $EA_{log\text{-}n}$	$(1+50)$ $EA_{log\text{-}n}$	$(1+100)$ $EA_{log\text{-}n}$
2	16317.99 1.00	23126.32 1.00	32902.97 1.00
3	30067.03 0.92	42816.59 0.93	59703.07 0.91
4	44778.87 0.91	58120.04 0.84	81026 0.82
5	57853.85 0.89	81979.17 0.89	100673.25 0.76
6	70831.23 0.87	94274.85 0.82	121003.57 0.74
7	88142.57 0.90	117895.76 0.85	138522.61 0.70
8	99276.54 0.87	130268.55 0.80	154648.06 0.67
9	108210.06 0.83	140732.12 0.76	174333.65 0.66
10	131951.77 0.90	159973.14 0.77	192937.41 0.65
16	200593.93 0.82	251249.13 0.72	-
32	418882.77 0.83	470256.18 0.66	-
64	871927.52 0.85	905293.75 0.62	-

Table 6.3: **The average ERT w.r.t. r and ratio for r for the $(1+\lambda)$ $EA_{log\text{-}n}$.**

The algorithms with a self adaptive mutation rate, the $(1+\lambda)$ $EA_{2/r,2r}$ and $(1+\lambda)$ $EA_{log\text{-}n}$, of which results can be seen in tables 6.2 and 6.3, show behaviour that is reminiscent of the ones touched upon above. While the $(1+\lambda)$ $EA_{log\text{-}n}$ performs similar to the $(1+\lambda)$ EA, the $(1+\lambda)$ $EA_{2/r,2r}$ performs the worst of all algorithms tested, included on the baseline binary program. As can be seen in figure 6.2, the $(1+1)$ $EA_{log\text{-}n}$ can improve quicker during the first stage of optimisation, but the versions with higher offspring count perform more stable, and catch up near the end of the process.

Looking at tables 6.1, 6.2 and 6.3, for every experiment, a relation between r and the ERT seems to exist that is not entirely linear. In general, the experiments with higher offspring count show results that follow a clear trend instead of fluctuating. This may be an indication of progression trough mutation becoming dependent on a small probability near the end of the optimisation process, which could result in relatively dissimilar optimisation times between runs. A higher number of restarts per experiment might provide more accurate average results.

ERT for $(1+\lambda)$ EA$_{log\text{-}n}$ with $r = 10$

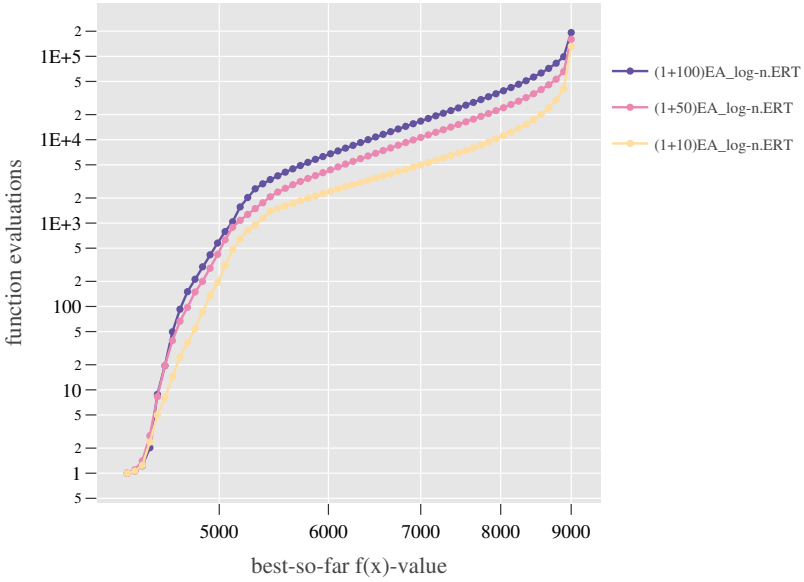

Figure 6.2: **The expected running time under various settings for λ and $n = 1000$.**

Modulo 2, r_s	$(1+1)$ RLS	$(1+1)$ EA	$(1+10)$ $EA_{2/r,2r}$	$(1+10)$ $EA_{log\text{-}n}$
2	6753.03	10786.65	23879.694	16317.99
20	12855.73	13574.67	21232.61	15080.2
200	13522.93	13662.77	20322.65	16003.22
2000	13701.57	13310.93	20894.84	16563.31
Modulo 3, r_s				
3	14140.05	22814.61	51103.76	30067.03
30	21309.49	24723.42	47050.35	29504.85
300	21906.82	24970.4	46671.53	30138.73
3000	21438.88	25131.71	46841.64	29637.65
Modulo 4, r_s				
4	22115.09	34770.72	80070.76	44778.87
40	28639.75	34566.91	74175.74	43185.49
400	28874.78	36713.43	75067.28	42888.29
4000	28084.37	37360.51	74548.72	43686.52
Modulo 5, r_s				
5	30103.99	46449.63	110149.44	57853.85
50	35011.78	48486.9	99552.48	54734.78
500	35400.04	48808.97	102499.24	56442.51
5000	35742.29	49510.57	102772.96	56164.17
Modulo 6, r_s				
6	36940.88	58737.22	139044.01	70831.23
60	43875.53	60472.88	129783.79	70524.86
600	42743.4	60192.08	132686.94	68212.07
6000	43846.99	58936.68	134852.7	68012.97

Table 6.4: **The average ERT w.r.t.** r_s **and modulo operation for the** $(1+1)$ **RLS,** $(1+1)$ **EA,** $(1+10)$ $\mathbf{EA}_{2/r,2r}$ **and** $(1+10)$ $\mathbf{EA}_{log\text{-}n}$**.**

When looking at table 6.4, our expectation of a ratio between the multi-valued baselines and the redundant versions approaching $\frac{r_p}{r_p-1}$ only seems to hold for the $(1+1)$ RLS algorithm, as can be seen in figure 6.3. One exception is the experiment for modulo 5, where the effect of the redundancy is slightly smaller than expected. The effect is much less pronounced for the $(1+1)$ EA, which may be caused by the redundancy allowing for mutations to multiple variables to increase fitness while not changing variables already optimal. For the self adaptive algorithms, the performance seems to improve slightly when increasing redundancy. These algorithms feature a larger offspring size λ, which may increase the quality of offspring selection. The graph 6.4 shows a faster improvement for the redundant coding instances in the early stages, but losing the advantage towards the end.

ERT for $(1+1)$ RLS using modulo 2

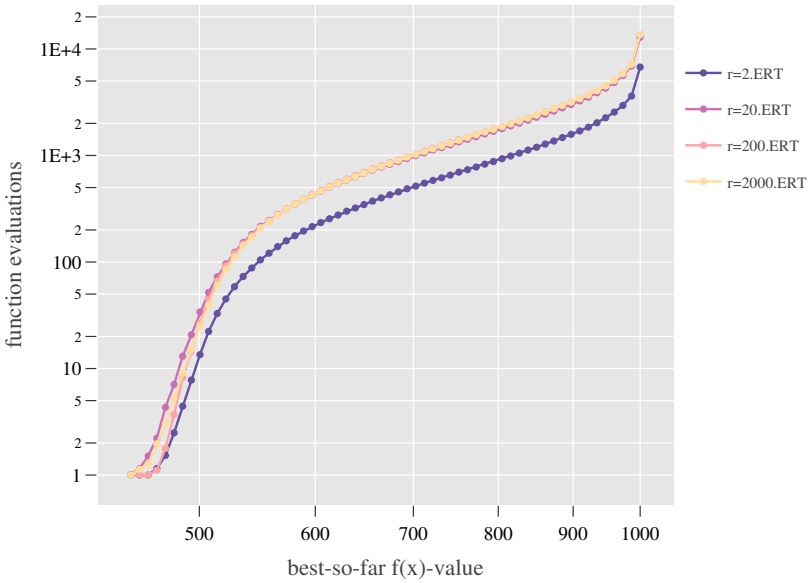

Figure 6.3: **The ERT under various settings for r_s and $n = 1000$.**

ERT for $(1+10)$ EA$_{log\text{-}n}$ using modulo 2

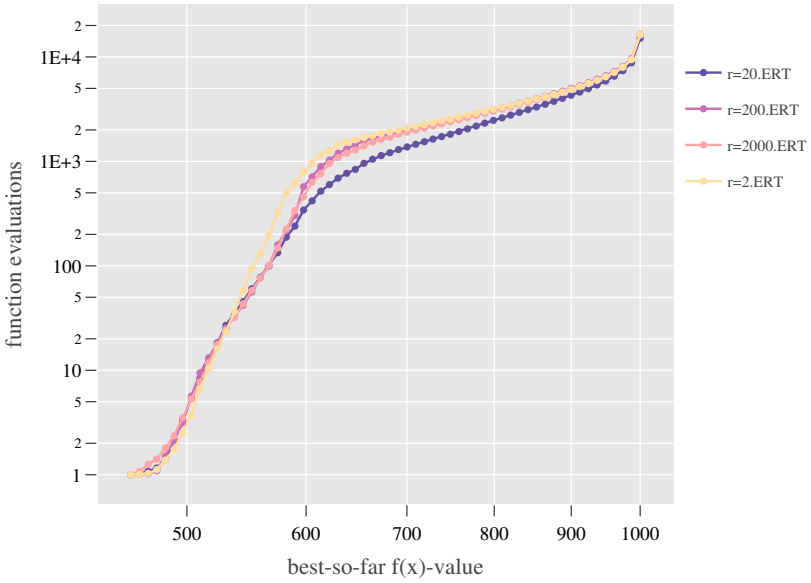

Figure 6.4: **The ERT under various settings for r_s and $n = 1000$.**

Chapter 7

Conclusions

7.1 Multi-valued decision variable experiment

From the data gathered it is very clear that increasing the size r of the variable domain has a detrimental effect on the ERT of the algorithms that were tested. The exact measure of the influence of r differs depending on algorithm and parameter basis, where especially a higher offspring count dampens the effect of an increase of r.

7.2 Redundant coding experiment

Our experiments with redundant coding proved that for the $(1+1)$ RLS algorithm an ERT of $\frac{r_p}{r_p-1}$ times the binary ERT can be expected for most cases. An anomalous result for modulo 5 seems to disprove this as a general rule however. The self adaptive algorithms, which used a larger offspring size λ than the algorithms featuring static mutation rates, seemed to benefit in a slight manner from redundancy. In general, the effect increasing problem variable size r_p has a marginal effect on the effect of redundancy on the ERT.

Chapter 8

Future research

As our preliminary results for the ERT w.r.t. multi-valued variables show, the interplay between variable domain size r and offspring size λ and the mutation method offers interesting results. One possible area of further research would be to further investigate the relation between offspring count λ and ERT when using various sizes for domain r. A second subject would be to repeat the experiments for the $(1+1)$ RLS and $(1+1)$ EA with more independent restarts to see if better average values can be found that show a trend.

For redundant coding, our expectation for ERT only applied to the RLS algorithm. While the results for the other algorithms imply that the effect is negligible, an understanding of why this difference exists may be worthwhile to create. For the specific application in grammatical evolution, a factor that is still confounding is the problem variables being of varying sizes. In order to apply any expectation for a specific variable size, a method must be constructed to find an expected variable size.

Bibliography

[1] Doerr, B., Doerr, C., and Kötzing, T. Static and self-adjusting mutation strengths for multi-valued decision variables. *Algorithmica 80* (07 2017).

[2] Doerr, B., Gieen, C., Witt, C., and Yang, J. The (1+λ) evolutionary algorithm with self-adjusting mutation rate. pp. 1351–1358.

[3] Doerr, B., Johannsen, D., and Schmidt, M. Runtime analysis of the (1+1) evolutionary algorithm on strings over finite alphabets. In *Proceedings of the 11th Workshop Proceedings on Foundations of Genetic Algorithms* (New York, NY, USA, 2011), FOGA '11, ACM, pp. 119–126.

[4] Doerr, C., Wang, H., Ye, F., van Rijn, S., and Bäck, T. IOHprofiler: A Benchmarking and Profiling Tool for Iterative Optimization Heuristics. *arXiv e-prints:1810.05281* (Oct. 2018).

[5] Doerr, C., Ye, F., Horesh, N., Wang, H., Shir, O. M., and Bäck, T. Benchmarking discrete optimization heuristics with iohprofiler. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion* (New York, NY, USA, 2019), GECCO '19, ACM, pp. 1798–1806.

[6] Gunia, C. On the analysis of the approximation capability of simple evolutionary algorithms for scheduling problems. In *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation* (New York, NY, USA, 2005), GECCO '05, ACM, pp. 571–578.

[7] Kruisselbrink, J., Li, R., Reehuis, E., Eggermont, J., and Bäck, T. On the log-normal self-adaptation of the mutation rate in binary search spaces. pp. 893–900.

[8] O'Neill, M., and Ryan, C. Grammatical evolution. *Trans. Evol. Comp 5*, 4 (Aug. 2001), 349–358.

[9] Yokoyama, S. The neutral theory of molecular evolution. by m. kimura. new york: Cambridge university press. 1983 xv + 367 pp., figures, tables, references, indices. $74.50 (cloth). *American Journal of Physical Anthropology 65*, 4 (1984), 401–402.