Theoretical and empirical runtime analysis of evolutionary algorithms for the PARTITION problem

Bachelor Thesis of

Daniel Lipp

At the Department of Informatics and Mathematics
Chair of algorithms for intelligent systems

UNIVERSITÄT PASSAU

Advisor: Prof. Dr. Dirk Sudholt

Time Period: 14th May 2023 – 14th August 2023

Statement of Authorship

I hereby declare that this document has been composed by myself and describes my own work, unless otherwise acknowledged in the text.

Passau, August 8, 2023

Abstract

A short summary of what is going on here.

Deutsche Zusammenfassung

Kurze Inhaltsangabe auf deutsch.

Contents

1. Introduction

The question of $P = NP$ is still unanswered to this day and solving NP-hard problems for every instance still requires exponential time. To avoid the exponential running time on NP-hard optimisation problems one might use approximation algorithms. Those algorithms do not always return the best possible solution but only a solution with a guaranteed solution quality. For a minimisation problem a (1+0.5)-approximation algorithm will always return a solution that has at most 1.5 times the optimal value. An example of an NP-hard optimisation problem is PARTITION. An instance of this problem is a multiset of n positive numbers $\{w_1, \ldots, w_n\}$ which has to be divided in two subsets with sums that are as close as possible. So a solution of PARTITION is a subset of $I \subset \{1, \ldots, n\}$ which splits the multiset into two subsets. The quality of the solution then is $\max\{\sum_{i \in I} w_i, \sum_{i \notin I} w_i\}$. PARTITION is one of the easiest NP-hard problems and has even been dubbed the easiest NP-hard problem [Hay02]. There are multiple algorithms specifically designed for PARTITION. Some of them return approximations and others always return the best solution. The exact algorithms have a runtime exponential in the input size due to the NP-hardness. Problem specific approximation algorithms are mostly deterministic such as the greedy method, the KK-algorithm or the FPTAS for the subsetsum problem which can be used for partition as well. Another class of non-deterministic algorithms are the so-called Evolutionary Algorithms which mimic the behaviour of evolution. Those algorithms start with a random population of solutions which are then changed with random steps. If the solution is good enough it survives and replaces one of the worst individuals in the population. The EA continues to generate new offspring in an endless loop. In practice the algorithm is given stopping conditions such as reaching a number of iterations or a specific solution quality. This is the principle of an anytime algorithm which can be terminated at anytime and output a valid solution. The longer the waiting time the better the solution might get. The main usage of EA lies in problems without a problem specific algorithms as these mostly outperform the EAs. To better understand their behaviour analysing them on well researched problems might still be beneficial to learn more about this class of algorithms. This thesis researches the runtime of basic EAs such as (1+1) EA and variations of the RLS. The first part is a theoretical analysis with a main focus of lowering bounds that were previously shown. Additionally there are new results for other algorithm variants and also a lemma on different type of inputs. The remainder of this thesis is an empirical analysis of multiple base algorithms with different parameter setting on different kinds of inputs. Here not only the (1+1) EA with different mutation rate and variants of the RLS with different parameter values are researched but also a heavy tailed mutation operator. Apart from typical distributions such as the uniform, geometric and binomial distribution there are also results for problem specific instances such as an input where one values is as large as all other values combined. In the end the empirical results are condensed into a personal suggestion which algorithm should best be chosen to solve the problem, depending on the input but also in general.

2. Preliminaries

2.1. Notation

A short list of vocabulary used throughout the paper.

EA Evolutionary Algorithm

RSH Randomised Search Heuristic referring to all analysed Evolutionary algorithms

n The input length of the problem

w_i The i-th object of the input. If not mentioned otherwise the weights are sorted in non-increasing order so: $w_1 \geq w_2 \geq \ldots \geq w_{n-1} \geq w_n$

W The sum of all objects: $\mathrm{W} = \sum_{i=1}^{n} w_i$

bin When solving Partition a set of numbers is divided into two distinct subsets and in this paper both subsets are referred to as bins

b_F The fuller bin (the bin with more total weight)

b_E The emptier bin (the bin with less total weight)

opt The optimal solution for a given partition instance.

x A vector $x \in \{0,1\}^n$ describing a solution

$f(x)$ The fitness function for PARTITION. This means a solution x has a solution quality of $f(x) = \max\{\sum_{i=1}^{n} w_i \cdot x_i, \sum_{i=1}^{n} w_i \cdot (1 - x_i)\} = b_F$

2.2. Background

2.2.1. Known algorithms for partition

Multiple methods for generating a solution of PARTITION already exist. Solving the problem with a greedy approach in runtime $\mathcal{O}(n)$ results in an approximation ratio of 3/2 if the elements are not sorted or a ratio of 7/6 if the numbers are sorted [Gra66]. Greedy in this case means putting each element in the currently emptier set while looking at each value exactly once. Another approximation algorithm is the KK-algorithm or also called Largest Differencing Method. With expected time $\mathcal{O}(n \log n)$ it has the same runtime as greedy with sorting and also the same worst case approximation of 7/6. For inputs chosen uniform random from [0,1] the KK has an expected ratio of $1 + \frac{1}{n^{\Theta(\log n)}}$ in comparison to the greedy algorithm which only reaches an approximation ratio of $1 + \mathcal{O}(\frac{1}{n})$. Instead of putting each element in the currently emptier set the currently largest two values are combined to one value by either subtracting or adding them depending on which results in a better solution. Adding them corresponds to putting the elements in the same set whereas subtracting them means putting them in different sets. There is even a fully polynomial time approximation algorithm (FPTAS) for the subsetsum problem [KMPS03] which can be used for PARTITION by setting the required sum to $\lfloor W/2 \rfloor$. FPTAS return a solution of at most $(1+\epsilon)$ the optimum in a time that is polynomial both in n and in $\frac{1}{\epsilon}$. There are lots of other approximation-algorithms but also some algorithms that always return the best solution. The Pseudopolynomial time number partitioning algorithm which uses dynamic programming always returns an optimal solution but needs time and space $\mathcal{O}(n\frac{m}{2})$ where m is the largest number in the input [Kor09]. The runtime is only pseudopolynomial because to encode m in the input only $\log_2(m)$ bits are required which causes $m = 2^{\log_2(m)}$ to be exponential in the input size. The Complete Greedy Algorithm (CGA) [Kor98] traverses a binary tree depth first and searches the complete 2^n search space in a greedy way. It functions the same way as the simple greedy algorithm but instead of only looking at only the greedy option at each height it also evaluates the non-greedy option after evaluating the complete subtree of the greedy option. The algorithm continues the depth first search until it either finds a perfect partition or has traversed the whole tree. In the second case it will return the best value found on the way. While the space complexity in only $\mathcal{O}(n)$ the runtime is $\mathcal{O}(2^n)$. Another exact algorithm is the Complete Karmarkar-Karp (CKK) [Kor98]. This algorithm works similar to the complete greedy approach by traversing the binary tree of all solutions. Instead of greedily selecting the next edge here the algorithm behaves like the KK-algorithm described above. It performs better than the CGA for the same reasons as before but also has the same worst case running time as the GCA.

2.2.2. Evolutionary Algorithm

Evolutionary Algorithms mimic the process of evolution and normally behave mostly the same. A run typically looks like this:

1. Generate initial population at random
2. If stopping condition are met return the currently best solution
3. Generate offspring population (e.g. by mutation)
4. Evaluate fitness of the offspring
5. Select fittest individuals and update population
6. Go back to step 2.

For PARTITION a solution $x \in \{0,1\}^n$ separates all numbers into two different sets with $x_i = 0$ meaning w_i is in set 0 whereas $x_i = 1$ meaning w_i is in set 1. So every possible value of x describes a feasible solution but not necessarily a good one. To evaluate the quality of a solution the EA is given a fitness function. The fitness function in this case is $f(x) = \max\{\sum_{i=1}^{n} w_i \cdot x_i, \sum_{i=1}^{n} w_i \cdot (1 - x_i)\} = b_F$. The goal of the algorithm is to return a solution with minimal fitness. A mutation step in the PARTITION problem will change an algorithm-dependent number of bits from 1 to 0 or vice versa. In this case flipping a bit means putting the element in the other set. A simple implementation of an EA is the so called (1+1) EA (Algorithm 2.1). The first 1 in the brackets refers to size of the population and the second to the amount of mutants created in each iteration of the loop. So it always has only one solution and generates just one new solution in each step. The mutation of

Algorithm 2.1: (1+1) EA

choose x uniform from $\{0,1\}^n$
while *x not optimal* **do**
  $x' \leftarrow x$
  flip every bit of x' with probability $1/n$
  if $f(x') \leq f(x)$ **then**
    $x \leftarrow x'$

Algorithm 2.2: RLS

choose x uniform from $\{0,1\}^n$
while *x not optimal* **do**
  $x' \leftarrow x$
  flip one uniform random bit of x'
  if $f(x') \leq f(x)$ **then**
    $x \leftarrow x'$

the current individual is performed by flipping each bit independently with probability $1/n$. The amount of flipped bits is binomial distributed with an expected value of $n \cdot \frac{1}{n} = 1$. Another simple EA is the randomised local search or short RLS (Algorithm 2.2). Instead of flipping each bit with probability $1/n$ here one bit is flipped at uniform random. Apart from that the algorithms are the same.

2.2.3. Literature on the RLS and (1+1) EA for PARTITION

Carsten Witt proved that the RLS and the (1+1) EA find a $(4/3+\epsilon)$ approximation in expected time $\mathcal{O}(n)$ and a (4/3)-approximation in expected time $\mathcal{O}(n^2)$ [Wit05]. He then introduced an almost worst case input to prove the bound for the approximation ratio is at least almost tight. The input is defined as followed for any $0 < \epsilon < 1/3$ and even n:
The input contains two numbers of value $1/3 - \epsilon/4$ and $n-2$ elements of value $(1/3 + \epsilon/2)/(n-2)$. The total volume is normalised to 1. When the two large values are in the same bin, the RSHs are tricked into a local optimum, where only w_1 and w_2 are in the first bin and the remaining elements in the other bin. This results in an almost worst case. To leave this local optimum $\Omega(n)$ bits must be moved in a step separating the two large values. Such a step will never happen for the RLS and only in expected exponential time for the (1+1) EA. This worst case happens with probability $\Omega(1)$. He also proved both RSHs return a $(1+\epsilon)$-approximation for $\epsilon \geq 4n$ in expected time $\lceil en\ln(4/\epsilon)\rceil$ for the (1+1) EA and $\lceil en\ln(4/\epsilon)\rceil$ for the RLS with probability at least $2^{-(e\log e+e)\lceil 2/\epsilon\rceil\ln(4/\epsilon)-\lceil 2/\epsilon\rceil}$ for the (1+1) EA and at least $2^{-(\log e+1)\lceil 2/\epsilon\rceil\ln(4/\epsilon)-\lceil 2/\epsilon\rceil}$ for the RLS. Afterwards he proved both RHSs reach a solution where the difference between the two bins is at most 1 for uniform distributed inputs on [0,1] after expected time $\mathcal{O}(n^2)$ for the (1+1) EA and $\mathcal{O}(n\log n)$ for the RLS. The difference between the two bis is even bounded by $\mathcal{O}(\log n/n)$ after $\mathcal{O}(n^{c+4}\log n)$ steps with probability at least $1-\mathcal{O}(1-1/n^c)$. This leads to an expected difference of $\mathcal{O}(\log n/n)$ after $\mathcal{O}(n^{c+4}\log n)$ steps. He also analysed exponential distributed inputs with parameter 1. With probability $1-\mathcal{O}(1/n^c)$ the difference on those inputs is bounded by $\mathcal{O}(\log n)$ after $\mathcal{O}(n^2\log n)$ steps and even by $\mathcal{O}(\log n/n)$ after $\mathcal{O}(n^{c+4}\log^2 n)$ steps. Additionally he described a polynomial time randomised approximation scheme (PRAS) for the RLS and the (1+1) EA for values of $\epsilon = \Omega(\log\log n/\log n)$.
For MAKESPAN-SCHEDULING a list of processing times has to be distributed on a set

Algorithm 2.3: (1+1) EA WITH STATIC MUTATION RATE

```
choose x uniform from {0,1}^n
while x not optimal do
    x' ← x
    flip every bit of x' with probability c/n
    if f(x') ≤ f(x) then
        x ← x'
```

of machines while minimising the total time of the fullest machine. With 2 machines this problem is exactly the same as PARTITION. So in a sense MAKESPAN-SCHEDULING is a more general version of PARTITION. This lead to Christian Gunia generalising some results previously shown by C. Witt to MAKESPAN-SCHEDULING on k machines [Gun05]. Solutions for MAKESPAN-SCHEDULING are $x \in \{0, \ldots, k-1\}^n$ and therefore during a mutation x_i is set to a uniform random value from $\{0, \ldots, k-1\} \backslash \{x_i\}$ instead of $1 - x_i$. The adapted RSHs reach an approximation ratio of $(2k/k+1)$ in expected time $\mathcal{O}(Wn^{2k-2}/w_n)$. On an instance where every weight is the same the expected optimisation time is bounded by $\mathcal{O}(n \log n)$. He also adapted the almost worse case to the more general problem and proved the RLS does not find a solution better than $(2k/k+1) - \epsilon$ in finite time for any $\epsilon > 0$ with constant probability. The second statement for PARTITION on the uniform distributed inputs on [0,1] has an equivalent lemma for MAKESPAN-SCHEDULING on k machines as well.

Another way of dealing with NP-hard problems is identifying a parameter k which defines how hard the problem is to solve. One possible parametrisation of PARTITION is solving whether there is a solution of $f(x) \leq k$. A fixed-parameter tractable problem is a problem that can be solved in time polynomial in the size of the input and $g(k)$ where g is any arbitrary function. Partition falls into this category as it can be decided in time at most $\mathcal{O}(4^k)$[Fer05]. Andrew M. Sutton and Frank Neumann made a parametrised analysis of PARTITION[SN12]. They parametrised the problem in multiple ways. One of their parametrisation was: given an integer k, is there a solution of at most $W/2 + W/k$? They showed that a multistart of the (1+1) EA or RLS using runs of length $\lceil en \ln(2k) \rceil$ is a Monte Carlo-fpt algorithm for this parametrised version of PARTITION. They also analysed a parametrisation of the size of the critical path and also the discrepancy (the difference between the two bins).

2.3. Higher mutation rates and heavy tailed mutations

For OneMax and all other linear functions the mutation rate of $p_m = 1/n$ was proven to be optimal[Wit13]. This is not the case for every fitness function. Jump$_k$ has an optimal mutation rate of k/n and a small constant factor deviation from k/n results in an increase of the runtime exponential in $\Omega(k)$[DLMN17]. The same might hold for PARTITION, because the previously discussed literature only analyses mutation rates with a 1-bit flip in expectation.
One way of creating algorithms with higher mutation rates is adjusting the currently existing algorithms. For the (1+1) EA this can be done easily. By changing the mutation rate $1/n$ to c/n for any constant c the algorithm now flips $n \cdot c/n = c$ bits in expectation.

For the RLS it is not that simple, as the RLS chooses a random bit and flips it. Instead of flipping c bits in every step there should be the possibility to flip different amounts of bits in every step. The standard RLS chooses a random neighbour with Hamming distance one. So the variant of the RLS could simply choose neighbours that have a Hamming distance

Algorithm 2.4: RLS_k^B

```
choose x uniform from {0,1}^n
while x not optimal do
    x' ← uniform random neighbour of x with Hamming distance ≤ k
    if f(x') ≤ f(x) then
        x ← x'
```

Algorithm 2.5: RLS_k^S

```
choose x uniform from {0,1}^n
while x not optimal do
    y ← uniform random value ∈ {1,...,k}
    x' ← uniform random neighbour of x with Hamming Distance y
    if f(x') ≤ f(x) then
        x ← x'
```

larger than one. The selection should still be uniform random to keep the idea of the RLS intact. One possible way is to choose a random neighbour with Hamming distance $\leq k$. The amount of neighbours with Hamming distance y is given by $\binom{n}{y}$. For $k = 4$, this results in n neighbours with Hamming distance 1, $n(n-1)/2$ neighbours with Hamming distance 2, $n(n-1)(n-2)/6$ for 3 and $n(n-1)(n-2)(n-3)/24$ for 4. The probability to choose a random neighbour with Hamming distance $y \leq k$ for $k = \mathcal{O}(1)$ is given by

$$P(\mathrm{RLS}_k^B \text{ flips } y \text{ bits}) = \frac{\binom{n}{y}}{\sum_{i=1}^{k}\binom{n}{i}} = \frac{\Theta(n^y)}{\sum_{i=1}^{k}\Theta(n^i)} = \frac{\Theta(n^y)}{\Theta(n^k)} = \Theta(n^{y-k}) = \Theta(\frac{1}{n^{k-y}})$$

This variant of the RLS is likely to choose a neighbour with Hamming distance k as the number of neighbours with hamming distance k rises with k for $k \leq n/2$. The probability of flipping only one bit is $\mathcal{O}(\frac{1}{n^{k-1}})$. For some inputs flipping only one bit might be more optimal which is rather unlikely for this variant of the RLS. This algorithm will be called RLS_k^B from now on, because it chooses a random neighbour in the Hamming ball with radius k.
An alternative way of changing the RLS is to first choose $y \in \{1, \ldots, k\}$ uniform random and then choose a neighbour with Hamming distance y uniform random. Here the probability of flipping $y \leq k$ bits is given by $1/k$, so the algorithm is much more likely to choose to flip only one bit. This variant of the RLS will be referred to as RLS_k^S because it first choses the Hamming sphere and afterwards the neighbour within the selected Hamming sphere.

Both variants of the RLS change at most k bits in each step and therefore only a constant amount of bits. For the (1+1) EA the algorithm will also flip mostly $\mathcal{O}(c)$ bits which is also constant. So neither of the new variants is likely to change up to $\Theta(n)$ bits. Quinzan *et al.* therefore introduced another mutation operator in [FGQW18] called $pmut_\beta$. This operator chooses k from a powerlaw distribution D_n^β with exponent β and maximum value n and then k uniform random bits are flipped. This algorithm will mostly flip a small number of bits but occasionally up to n bits. Distributions like this are called heavy tailed mutations because their tail is not bounded exponentially.

3. Runtime Analysis of different EAs on PARTITION

This chapter focuses on the theoretical analysis of the runtime of Evolutionary Algorithms for PARTITION. The first section is focused on improving previously shown bounds for the (1+1) EA and the RLS. Afterwards there is also an analysis of Evolutionary Algorithms that flip more than one bit in expectation. The last section analyses inputs that follow a binomial distribution.

3.1. Improving bounds on the RLS and the (1+1) EA

As discussed in the Background section C. Witt already showed multiple results for the RLS and the standard (1+1) EA. One of the results is the approximation ratio of at most 4/3 after expected time at most $\mathcal{O}(n^2)$. In this section this bound will be lowered to $\mathcal{O}(n \log n)$ for both algorithms. The special input with $w_1 \geq W/2$ is also analysed for the (1+1) EA with mutation rate c/n. For the analysis of mutation rates c/n the work of C. Witt only has to be slightly adjusted. The first Lemma is a modification of a Lemma in [Wit05].

Lemma 3.1. *Let $w_1 \geq W/2$, then for any $\gamma > 1$ and $0 < \delta < 1$, the (1+1) EA with mutation rate c/n with constant $0 < c < \sqrt{n}$ (RLS_k^S) reaches an f-value at most $w_1 + \delta(W - w_1)$ in at most $\lceil \frac{e^c}{c \cdot (1-o(1))} n \ln(\gamma/\delta) \rceil$ ($\lceil kn \ln(\gamma/\delta) \rceil$) steps with probability at least $1 - \gamma^{-1}$. Moreover, the expected number of steps is at most $2 \lceil \frac{e^c}{c \cdot (1-o(1))} n \ln(2/\delta) \rceil$ ($2 \lceil kn \ln(2/\delta) \rceil$).*

Proof. This Lemma is very similar to C. Witt's Lemma 2 from section '2. Definitions and Proof Methods' in [Wit05]. The proof is mostly the same. He first defines a potential function $p(x) = f(x) - l$. While $p(x) > 0$ all steps moving only a small object to the emptier bin are accepted. The expected p-decrease is at least $p_0 \cdot q$ where q is a lower bound on the probability of the algorithm to flip one specific bit. This leads to a next p value of $(1 - q)p_0$. Since all steps of both algorithms are independent this argumentation remains valid even if the p value is only an expected value. With $q = 1/yn$ for a constant $y > 0$ the expected p value after $t = yn \ln(\gamma/\delta)$ steps is at most

$$p_t \leq p_0(1 - 1/yn)^t = p_0(1 - 1/yn)^{yn \ln(\gamma/\delta)} \leq p_0 \cdot e^{-\frac{1}{y} \cdot yn \ln(\gamma/\delta)} = p_0(\gamma/\delta)^{-1} = p_0(\delta/\gamma)$$

Applying Markov's inequality to the non-negative p value implies $p_t \leq p_0\delta$ with probability $1 - 1/\gamma$. Repeating independent phases of length $\lceil yn \ln(2/\delta) \rceil$ the expected number of

phases is at most 2. Up until here the proof is the same.
Instead of choosing $W/2$ as the general upper bound for p_0 as in the original lemma here the lower value $W - w_1 \leq W/2$ is chosen because it is more tight for the special case $w_1 \geq W/2$ with $l = w_1$. The probability of the RLS_k^S to flip one specific bit is $\frac{1}{k} \cdot \frac{1}{n}$ and for (1+1) EA with mutation rate c/n at least

$$\frac{c}{n}(1-\frac{c}{n})^{n-1} \geq \frac{c}{n}(1-\frac{c}{n})^{n} \geq \frac{c}{n}e^{-c}(1-\frac{c^2}{n}) = \frac{c}{e^c n}(1-o(1))$$

The inequality $(1+x/n)^n \geq e^x(1-x^2/n)$ requires $n \geq 1, |c| \leq n$ which both hold. Setting $y = \frac{e^c n}{c \cdot (1-o(1))}$ for the (1+1) EA and $y = k$ for the RLS_k^S concludes the result. □

The next Lemma also analyses inputs with $w_1 > W/2$. Its goal is to bound the probability of the (1+1) EA with mutation rate c/n and the $\text{RLS}_{k\geq 2}^S$ of flipping the first bit after a certain solution quality has been reached. Inputs with $w_1 > W/2$ are similar to linear functions which strongly suggest a runtime of $\mathcal{O}(n \log n)$ without flips of w_1. If such a step is too unlikely the algorithms might find an optimal solution before w_1 is flipped and the Hamming distance to the optimum might increase drastically.

Lemma 3.2. *For instances with $w_1 > W/2$ the probability of flipping w_1 when $b_E = c \cdot \frac{W-w_1}{2}$ for $1 < c < 2$ holds, is at most $\frac{2y(y-1)^2}{n(n-1)(c-1)}$ in a step where any algorithm flips $2 \leq y \leq n/2$ bits.*

Proof. For a successful flip of w_1 after $b_E \geq \frac{W-w_1}{2}$ holds a total volume of at least $z \geq 2 \cdot (b_E - \frac{W-w_1}{2})$ must be shifted from b_E to b_F. Otherwise the step is rejected because

$$b'_F = b_E + w_1 - z > b_E + w_1 - 2 \cdot (b_E - \frac{W-w_1}{2}) = b_E + w_1 - 2b_E + W - w_1 = W - b_E = b_F$$

which results in an increase of the fitness ($b_F = f(x), b'_F = f(x')$). Let I be the set of indices of all elements moved from b_E to b_F and $w_{\max} = \max\{w_i | i \in I\}$. $|I| \leq y-1$ holds because at least w_1 is moved from b_F to b_E. The sum of all elements is at most $w_{\max} \cdot |I| \leq (y-1)w_{\max}$. If $w_{\max} < 2 \cdot (b_E - \frac{W-w_1}{2})/(y-1)$ then $(y-1)w_{\max} < 2 \cdot (b_E - \frac{W-w_1}{2})$ and the step is rejected. Thus at least one of the objects moved from b_E to b_F must have a volume of at least $2 \cdot (b_E - \frac{W-w_1}{2})/(y-1)$. At most $d \leq \frac{b_E}{w_{\max}}$ of these objects can be in b_E if they made up the complete volume of b_E. Simplifying this inequality leads to at most

$$d \leq \frac{b_E}{w_{\max}} \leq \frac{W-w_1}{w_{\max}} \leq \frac{W-w_1}{2(c\frac{W-w_1}{2} - \frac{W-w_1}{2})/(y-1)} = \frac{(W-w_1)(y-1)}{(W-w_1)(c-1)} = \frac{(y-1)}{(c-1)}$$

objects having at least a volume of $w_{\max}$. For a successful flip w_1 and at least one of these d objects must switch bins and the probability for such a step flipping y bits is therefore at most

$$\mathbb{P}(y \text{ bits are flipped}) \cdot \mathbb{P}(\text{the correct } y \text{ bits are flipped} | y \text{ bits are flipped})$$
$$\leq 1 \cdot \frac{\binom{1}{1} \cdot \binom{\lceil d \rceil}{1} \cdot \binom{n-2}{y-2}}{\binom{n}{y}} = \lceil d \rceil \frac{\frac{(n-2)!}{(n-2-y+2)! \cdot (y-2)!}}{\frac{n!}{(n-y)! \cdot y!}} = \frac{\lceil d \rceil \cdot (n-2)! \cdot (n-y)! \cdot y!}{n! \cdot (n-y)! \cdot (y-2)!} = \frac{\lceil d \rceil y(y-1)}{n(n-1)}$$
$$\leq \frac{y(y-1)}{n(n-1)} \cdot (\frac{y-1}{c-1} + 1) = \frac{y(y-1)}{n(n-1)} \cdot (\frac{y-1+c-1}{c-1}) \leq \frac{2y(y-1)^2}{n(n-1)(c-1)}$$

□

Theorem 3.3 is the last analysis for inputs with $w_1 \geq \frac{W}{2}$. It uses the results of the two previous lemmas and gives an asymptotic bound on the runtime. Instead of giving a runtime for a 4/3-approximation this lemma gives the expected runtime for reaching one of the two optimal solutions. For a non-optimal solution moving one element from the fuller to the emptier bin will always result in an improvement. There are also no local optima which neither of the algorithms is unlikely to leave. So this input is rather easy to solve for the RLS and (1+1) EA and comparable to a linear function or even OneMax if $w_2 = \cdots = w_n$.

Theorem 3.3. *If $w_1 \geq \frac{W}{2}$ then the RLS and the (1+1) EA with mutation rate k/n with constant $0 < k < \sqrt{n}$ reach the optimal solution in expected time $\Theta(n \log n)$*

Proof. The optimal solution is putting w_1 in one bin and all other elements in the other bin. So the problem is almost identical to OneMax/ZeroMax. A single bit flip of the first bit can only happen, if the emptier bin has a weight of at most $\frac{W-w_1}{2}$. After this flip the weight of the emptier bin is at least $\frac{W-w_1}{2}$ and therefore another single bit flip of w_1 can only happen before another bit is flipped. The run of the RLS can be divided into two phases:

Phase 1: The RLS reaches a search point with $b_E > \frac{W-w_1}{2}$.

Phase 2: The RLS reaches an optimal solution $\Rightarrow$ w_1 is in one bin and all other elements are in the other bin.

The expected length of the first phase is at most $2n$ because the probability of flipping the first bit is $\frac{1}{n}$ and the expected time for such a step then is at most n. After such a step $b_E \geq \frac{W-w_1}{2}$ holds. If the solution is already optimal $b_E = W - w_1 > \frac{W-w_1}{2}$, otherwise there is at least one bit that can be flipped. This bit will be flipped in expected time at most n for the same reason as for w_1. The total length of first phase is at most $2n$. In the second phase the RLS can no longer flip w_1 as it does not result in an improvement ever again. Therefore the RLS behaves exactly as on OneMax/ZeroMax depending on the value of the first bit and reaches an optimal solution in $\Theta(n \log n)$ resulting in a total runtime of $\Theta(n \log n)$ (Theorem 3 in [Wit14]).
As long as w_1 does not flip the (1+1) EA has to minimize a linear function of $n-1$ bits which takes $(1+o(1))\frac{e^k}{k} n \ln n$ time (Corollary 4.2 in [Wit13]). The only steps that could hinder the algorithm from optimising the linear function in $\Theta(n \log n)$ would be a flip of the first bit. Such steps invert the optimal solution which could decrease the progress of minimising the linear function. If such a step has an expected time of $\omega(n \log n)$ the linear function is likely to be optimised in expectation before such a step happens.

The probability of the (1+1) EA to flip more than $k + \sqrt{6k \ln(n)} = k + k\sqrt{6 \ln(n)/k}$ is limited by Chernoff bounds:

$$\mathbb{P}((1{+}1) \text{ EA flips more than } k + k\sqrt{6\ln(n)/k} \text{ bits})$$
$$\leq \mathbb{P}(X \geq (1 + \sqrt{6\ln(n)/k}) \cdot k) \leq e^{-k \cdot \sqrt{6\ln(n)/k}^2/3} = e^{-k \cdot \frac{6 \ln n}{3k}} = n^{-2}$$

So the expected time for such a step is at least $n^2 = \omega(n \ln(n))$. Now let's look at steps that flip at most $k + \sqrt{6k \ln(n)}$ bits in a single step. Such a step only successfully flips w_1 if both w_1 is flipped and enough total volume is shifted from b_E to b_F. Due to Lemma 3.1 with $\delta = \frac{1}{n}$ (for $n > 1$) the solution is at most $w_1 + \delta(W - w_1)$ after expected time

$$2\lceil \frac{e^k}{k(1-o(1))} n \ln(2/\delta) \rceil = 2\lceil \frac{e^k}{k(1-o(1))} n \ln(2/\frac{1}{n}) \rceil = 2\lceil \frac{e^k}{k(1-o(1))} n \ln(2n) \rceil$$
$$= 2\lceil \frac{e^k}{k(1-o(1))} n(\ln(n) + \ln(2)) \rceil \leq \frac{2e^k}{k(1-o(1))} n(\ln(n) + 4)$$

The value of b_E is then at least $W-(w_1+\delta(W-w_1)) = (1-\delta)(W-w_1) = (1-\frac{1}{n})(W-w_1)$.

Lemma 3.2 states that the probability of a step flipping with w_1 together with $y-1$ other bits is at most $\frac{2y(y-1)^2}{n(n-1)(c-1)}$. Applying the bound $y \le k+\sqrt{6k\ln(n)}$ and the value $c = 2(1-\frac{1}{n})$ this simplifies to

$$\frac{2y(y-1)^2}{n(n-1)(c-1)} \le \frac{2(k+\sqrt{6k\ln(n)})^3}{n(n-1)(1-\frac{2}{n})} = \frac{2(k+\sqrt{6k\ln(n)})^3}{n(n-1)\frac{n-2}{n}} = \frac{2(k+\sqrt{6k\ln(n)})^3}{(n-1)(n-2)}$$

The probability of one of these steps to happen for any value of y is given by

$$\sum_{y=2}^{k+\sqrt{6k\ln(n)}} \mathbb{P}(y \text{ bits are flipped}) \cdot \mathbb{P}(\text{the correct } y \text{ bits are flipped} | y \text{ bits are flipped})$$
$$\le (k+\sqrt{6k\ln(n)}) \cdot \frac{2(k+\sqrt{6k\ln(n)})^3}{(n-2)(n-1)} = \frac{2(k+\sqrt{6k\ln(n)})^4}{(n-2)(n-1)}$$
$$= \frac{2(o(n^{1/8}))^4}{(n-2)(n-1)} = \frac{o(n^{0.5})}{\mathcal{O}(n^2)} = \mathcal{O}(n^{-1.5})$$

The expected time for any step successfully flipping w_1 is then $\Omega(n^{1.5}) = \omega(n \ln n)$. Let T be the time until the linear function is optimised and p the probability of successfully flipping w_1. Then the probability that w_1 flips after expected time $\frac{2e^k}{k(1-o(1))}n\ln(n)+4$ before the linear function is optimised is at most

$$p \cdot E(T) \le \frac{1}{\mathcal{O}(n^{1.5})} \cdot (1+o(1))\frac{e^k}{k} n \ln n = \frac{(1+o(1))\frac{e^k}{k}\ln n}{\mathcal{O}(n^{0.5})} = \frac{o(n^{0.1})}{\mathcal{O}(n^{0.5})} = o(\frac{1}{n^{0.4}}) = o(1)$$

The probability of w_1 not being flipped after expected time $\frac{2e^k}{k(1-o(1))}n\ln(n)+4$ is $1-o(\frac{1}{n^{0.4}}) = 1-o(1)$. If such a step happens the fitness does not decrease and the bound on the probability of flipping w_1 still holds. The algorithm will still find the solution in expected time at most $(1+o(1))\frac{e^k}{k}n\ln n$. Since even after a flip all condition are still true the expected time of optimising the linear function after expected time $\frac{2e^k}{k(1-o(1))}n\ln(n)+4$ is given by $\frac{1}{1-o(1)} \cdot (1+o(1))\frac{e^k}{k}n\ln n = \Theta(n\log n)$

The total runtime for the (1+1) EA is $\frac{2e^k}{k(1-o(1))}n\ln(n)+4+\frac{1+o(1)}{1-o(1)} \cdot \frac{e^k}{k}n\ln n = \Theta(n\log n)$.

□

The next few Lemmas and Corollaries help to prove the runtime of $\mathcal{O}(n\log n)$ for the (1+1) EA and the RLS on inputs with $w1 < W/2$ for reaching an approximation ratio of 4/3. They are mostly rather short with a proof of only a few lines. The only Lemma with a longer proof is Lemma 3.9. It shows the runtime bound for a more restricted type of input and is used to simplify the proof of Lemma 3.10

Lemma 3.4. *If $b_F \le \frac{2}{3} \cdot W$ the approximation ratio is at most $\frac{4}{3}$*

Proof. $\frac{b_F}{opt} \le \frac{(2/3)\cdot W}{opt} \le \frac{(2/3)\cdot W}{(1/2)\cdot W} = \frac{4}{3}$, since $opt \ge \frac{W}{2}$ □

Corollary 3.5. *If $w_1 \ge \frac{W}{3}$ and w_1 is in the emptier bin, then the approximation ratio is at most $\frac{4}{3}$*

Proof. w_1 is in the emptier bin, so $b_F \leq W - w_1 \leq W - \frac{W}{3} = \frac{2W}{3}$ and with Lemma 3.4 the assumption follows. □

Lemma 3.6. *Any object of weight v can be moved from b_F to b_E if and only if $b_F - b_E \geq v$*

Proof. $'' \Leftarrow''$:
$b_F - b_E \geq v \Leftrightarrow b_F \geq b_E + v$, so after moving an object with weight v from b_F to b_E, the new weight of b_E is at most the weight of b_F before moving the object, thus the RSH accepts the step.
$'' \Rightarrow''$:
$b_F - b_E < v \Leftrightarrow b_F < b_E + v$, so moving an object of weight v results in $b_F{}' = b_E + v > b_F$ which results in the step being rejected. □

Corollary 3.7. *The RLS is stuck in a local optimum if $b_F - b_E \leq w_n$ holds and $b_F > opt$.*

Proof. A single bit flip of weight v can only happen if $b_F - b_E \geq v$ (Corollary 3.7). If $b_F - b_E < w_n$ there is no weight which satisfies the condition and therefore no single bit flip is possible. If $b_F - b_E = w_n$ then only objects with weight w_n can be flipped, but those do not change the fitness ($b'_F = b_E + w_n = b_F - w_n + w_n = b_F$). Since the RLS can only move one bit at a time and only if it results in an improvement, the RLS is stuck. □

Corollary 3.8. *Every object $\leq \frac{W}{3}$ can be moved from b_F to b_E if $b_F \geq \frac{2W}{3}$*

Proof. $b_F \geq \frac{2W}{3} \Rightarrow b_E \leq W - \frac{2W}{3} \leq \frac{W}{3} \Rightarrow b_F - b_E \geq \frac{2W}{3} - \frac{W}{3} = \frac{W}{3}$ and with Lemma 3.6 the assumption follows. □

Lemma 3.9. *In expected time $\mathcal{O}(n \log n)$ the weight of the fuller bin can be decreased to $\leq \frac{2W}{3}$ by the RLS and the (1+1) EA if every object besides the biggest in the fuller bin is at most $\frac{W}{3}$ and $w_1 \leq \frac{W}{2}$.*

Proof. In expected time $\mathcal{O}(n \log n)$ the RSH can move every object $\leq \frac{W}{3}$ to the emptier bin as long as $b_F \geq \frac{2W}{3}$ due to Corollary 3.8 and Theorem 3.3. So in expected time $\mathcal{O}(n \log n)$ the solution can be shifted to w_1 being in one bin and all other objects in the other bin. The RSH will only stop moving the elements if the condition $b_F \geq \frac{2W}{3}$ is no longer satisfied (Corollary 3.8). If $w_1 \geq \frac{W}{3}$ and every object was moved to the bin without w_1, then $b_F = \max\{W - w_1, w_1\} = W - w_1 \leq \frac{2W}{3}$, because $w_1 \leq \frac{W}{2}$. So either the RSH moves all objects to the emptier bin or stops moving objects because $b_F < \frac{2W}{3}$ both resulting in $b_F \leq \frac{2W}{3}$. If w_1 is not in the fuller bin, then the result follows by Corollary 3.5.
Now assume $w_1 < \frac{W}{3}$. In this case the RLS will move one object per step to the emptier bin. Each object has weight $< \frac{W}{3}$ and therefore one step cannot decrease the weight of the fuller bin from $> \frac{2W}{3}$ to $\leq \frac{W}{3}$. If all objects except one where moved to one bin, the other bin would have a weight of at least $W - w_1 > \frac{2W}{3}$. Therefore the RLS will find a solution with $b_F < \frac{2W}{3}$ before moving all elements from the first to the second bin.
The proof for the (1+1) EA is mostly the same. The main difference is the (1+1) EA being able to flip more than one bit in a single step. Such a step could make the emptier bin the fuller bin or increase the number of bits that must be shifted to the emptier bin. But with the results of Theorem 3.3 the proof works exactly the same as for the RLS. The case $w_1 \geq \frac{W}{3}$ does not change only the bin containing w_1 might change. Apart from that there is no difference for the (1+1) EA. The case $w_1 < \frac{W}{3}$ is also rather similar. The (1+1) EA will move elements from the fuller bin to the emptier bin until $b_F < \frac{2W}{3}$ holds. The (1+1) EA can make the emptier bin the fuller bin by moving multiple objects in one step, but

this does not hinder it from reaching $b_F < \frac{2W}{3}$. After such a step it will continue moving elements until the condition holds. □

Lemma 3.10. *The RLS and the (1+1) EA reach an approximation ratio of at most $\frac{4}{3}$ in expected time $\mathcal{O}(n \log n)$ if $w_1 < W/2$*

Proof. If $w_1 + w_2 > \frac{2W}{3}$ after time $\mathcal{O}(n)$ w_1 and w_2 are separated and will remain separated afterwards (3. Average case analysis, Theorem 1 in [Wit05]). From then on the following holds. If w_1 is in the emptier bin, then the result follows directly by Corollary 3.5. Otherwise all elements in the fuller bin except w_1 have a weight of at most $\frac{1}{3}$ and therefore the result follows by Lemma 3.9 and Lemma 3.4. If $w_1 + w_2 \leq \frac{2W}{3}$ the result follows directly by Lemma 3.9 and Lemma 3.4. □

Corollary 3.11. *The RLS and the (1+1) EA reach an approximation ratio of at most $\frac{4}{3}$ for every input in expected time $\mathcal{O}(n \log n)$*

Proof. This follows directly from Theorem 3.3 and Lemma 3.10. □

3.2. Runtime Analysis of higher mutation rates

The first analysed input is again the input with $w_1 \geq W/2$. This input is very close to a weighted OneMax and therefore more or less the OneMax equivalent for Partition. The first analysed algorithm will be the RLS variants.

Corollary 3.12. *The expected time until the RLS_k^S and RLS_k^B flips $w_1 \geq W/2$ after they reached a solution with $b_E = c \cdot \frac{W - w_1}{2}$ with $1 < c < 2$ is at least $\frac{n(n-1)(c-1)}{2k^2(k-1)^2}$. For constant values of c the expected time is $= \Omega(n^2)$.*

Proof. Using Lemma 3.2, the upper bound of $y \leq k$ and the fact that both algorithms flip at most k bits leads to the first part

$$(k \cdot \frac{2y(y-1)^2}{n(n-1)(c-1)})^{-1} = \frac{n(n-1)(c-1)}{2ky(y-1)^2} \leq \frac{n(n-1)(c-1)}{2k^2(k-1)^2}$$

k is constant and if c is constant too, this leads to expected time $= \Omega(n^2)$ for a flip of the first bit if $b_E = c \cdot \frac{W - w_1}{2}$ with $1 < c < 2$ which concludes the second statement. □

Lemma 3.13. *The RLS_k^S reaches the optimal solution on an input with $w_1 \geq W/2$ in expected time $\mathcal{O}(n \log n)$*

Proof. This proof is similar to proof for the (1+1) EA in Theorem 3.3 and is divided in the same two parts. The first part is proving the RLS_k^S does not flip the first bit in expected time $\mathcal{O}(n \log n)$ after some time T has passed. After time T the algorithm then minimises the linear function in expected time $\mathcal{O}(n \log n)$ before the first bit is flipped and the progress of the linear function might be reset. After expected time $(2\lceil kn \ln(2/0.4) \rceil) = 2\lceil kn \ln(5) \rceil \leq 2\lceil 1.61 \cdot kn \rceil \leq 4kn$ the RLS_k^S reaches a solution of $f(x) \leq w_1 + 0.4(W - w_1)$ (Lemma 3.1). This means $b_E \geq 0.6(W - w_1) = 1.2 \cdot \frac{W - w_1}{2}$. With Lemma 3.12 the expected time of at least $\frac{n(n-1)(c-1)}{k^2(k-1)^2} \frac{n(n-1)}{5k^2(k-1)^2} = \Omega(n^2)$ for a successful flip of w_1 follows. This concludes the first part
The RLS_k^S can be seen as an unbiased unary black box algorithm with a sequence $\mathcal{D} =$

$(p_1, \ldots, p_n)$ where $p_i = 1/k$ for $1 \leq i \leq k$ and $p_i = 0$ otherwise. The mean of this sequence is $\mathcal{X} = \sum_{i=1}^{n} \mathbb{P}(X = i)i = \sum_{i=1}^{k} \frac{i}{k} = \frac{1}{k} \cdot \sum_{i=1}^{k} i = \frac{k+1}{2}$. For any constant value k both the probability $p_1 = \frac{1}{k} = \Theta(1)$ and mean $\mathcal{X} = \frac{k+1}{2} = O(1)$ meet the conditions of Theorem 1 in [DJL23] and therefore the RLS_k^S optimises the linear function in expected time $(1+o(1))\frac{1}{p_1} n \ln n = (1+o(1))kn \ln n$. Applying the same arguments as in Theorem 3.3 this results in a probability of not flipping w_1 after expected time $4kn$ of at least

$$1 - \frac{(1+o(1))kn \ln n \cdot 5k^2(k-1)^2}{n(n-1)} = 1 - \frac{(1+o(1))5k^5 \ln n}{n-1} = 1 - \frac{o(k^5 n^{0.5})}{n-1} = 1 - \frac{1}{o(\sqrt{n})}$$

The expected time of minimising the linear function then is

$$\frac{1}{1 - \frac{1}{o(\sqrt{n})}} \cdot (1+o(1))kn \ln n = \frac{1}{1 - o(1)} \cdot (1+o(1))kn \ln n = \Theta(n \log n)$$

The total expected runtime therefore is at most $4kn + \frac{1+o(1)}{1-o(1)} \cdot kn \ln n = \Theta(n \log n)$.

□

3.3. Binomial distributed input

This section discusses binomial inputs. At first they seem uninteresting as all values are rather close to the expected value of the distribution. So it seems like the Evolutionary Algorithm only has to find a search point where both bins have an equal amount of values. After investigating this input experimentally this is not true to that extend. Solutions with an equal amount of bits with value 0 and value 1 are indeed close to the optimum but just close. There is mostly still a difference of at least one to the optimum. This might cause algorithm like the RLS to get stuck despite the input looking easy at first glance. If elements can be swapped this input should become solvable again because there are many small values slightly bigger or smaller than the expected value. If the algorithm switches those the small difference to the optimum can be closed. The first lemma tries to prove that these input are indeed easy to solve in a sense that this input is very likely to have a perfect partition.

Lemma 3.14. *A binomial distributed input ~$B(m,p)$ has a perfect partition ($b_F - b_E = 0$ for even W and $b_F - b_E = 1$ for uneven W) with high probability if the input size n is large enough.*

Proof. Sketch:

- The initial distribution is likely rather close to the optimum
- The difference between the bins is probably not more than 10 expected values
- the large values

Consider a random separation of all values into two sets with equal size if n is even or one set with one value more than the other if n is odd. The sum X of one set is a sum of $\frac{n}{2} \cdot m$ independent Bernoulli trials with probability p. With Chernoff Bounds the following inequality follows:

$$\mathbb{P}(X \geq (\frac{n}{2} + \sqrt{\frac{n}{2}}) \cdot m \cdot p) = \mathbb{P}(X \geq (1 + 2\sqrt{\frac{2}{n}}) \cdot \frac{nmp}{2}) \leq e^{-\frac{mnp}{2} \cdot 2\sqrt{\frac{2}{n}}^2/3} = e^{-\frac{2mp}{3}}$$

For $mp \geq 1.5$ the probability is less than $\frac{1}{e}$. Otherwise the input is rather trivial, since the numbers will be concentrated around $mp \leq 1.5$ and most values will be below 10.
After moving $\mathcal{O}(\sqrt{\frac{n}{2}}/2)$ objects to the emptier set, the difference between the two sets is at most half the expected value mp of a single value. ... □

Lemma 3.15. *With high probability the RLS does not find a perfect partition for an input with distribution ~$B(m,p)$ if n is large enough, $mp \geq 50$ and $m - mp \geq 50$.*

Proof. The RLS will always move one object per step from fuller to the emptier bin. As long as $b_F - b_E > w_i$ holds moving any object of weight at most w_i from b_F to b_E results in a decrease of the fitness ($b'_F = b_E + w_i < b_F - w_i + w_i = b_F$). If the RLS does not decrease the difference $b_F - b_E$ to at most w_n in any step the solution is never optimal and the RLS therefore must be stuck. So now assume the RLS eventually reaches a search point with $b_F - b_E \leq w_n$. Consider the step which decreases the difference from more than w_n to at most w_n. If this step does not decrease the difference to 0 for even n or 1 for odd n the RLS is stuck due to Corollary 3.7. For a step to decrease $b_F - b_E$ from more than w_n to 0 the RLS must move an object of weight y which is given by $b'_F - b'_E = (b_F - y) - (b_E + y) = 0 \Leftrightarrow b_F - b_E - 2y = 0 \Leftrightarrow y = (b_F - b_E)/2$. Such an element can only exist for even n because only for even n either both bins have an even sum or neither of them. Odd inputs will always have exactly one bin with an even sum and one with an odd sum. For odd n a difference of 1 suffices for a perfect partition and therefore the value of y must be either $\lceil (b_F - b_E)/2 \rceil$ or $\lfloor (b_F - b_E)/2 \rfloor$. For any binomial distribution the value which occurs the most in expectation is the expected value if $mp \in \mathbb{N}$ or $\lceil mp \rceil$ and $\lfloor mp \rfloor$ otherwise. Even if the behaviour of the algorithm forces values smaller or bigger than mp in the emptier bin, the number of bigger/smaller values in the fuller bin will still be less in expectation than the amount of mp in both bins. Let's assume that objects with exactly this volume must be selected and that there are two options because $mp \notin \mathbb{N}$. This will give an upper bound on the probability of flipping the right object in the crucial step. The probability of a number to be $\lfloor mp \rfloor$ is $p^{\lfloor mp \rfloor}(1-p)^{m - \lfloor mp \rfloor} \leq 0.5^{50} = \frac{1}{1024^5} < 10^{-15}$ because both $mp \geq 50$ and $m - mp \geq 50$ and $p \leq 0.5$ or $1 - p \leq 0.5$. For $\lceil mp \rceil$ the same bound applies with the same arguments. The probability of either of these numbers to be drawn from the binomial distribution is at most $2 \cdot 0.5^{50} = 0.5^{49} \leq 10^{-15}$. This leads to $n \cdot 10^{-15}$ values with the right value in expectation. All bits are flipped uniformly at random and therefore the probability of flipping a good bit is at most $\frac{n \cdot 10^{-15}}{|b_F|} \leq \frac{n \cdot 10^{-15}}{n} = 10^{-15}$.

This means the RLS is very unlikely to find a perfect partition. Given the solution has a perfect partition the RLS will get stuck in a local optimum. ☐

The restriction of $mp \geq 50$ and $m - mp \geq 50$ was not chosen to make the proof work but is also necessary. Without those restrictions the distribution might have many small values and especially elements close to 1. When there are many small values these can be used to fill the small gaps which makes it easy for the RLS to find a perfect partition. On the other hand if p is almost one, then almost every element will be same for smaller values of m. If every element is the same then the RLS must only find a search point with equal amounts of 0s and 1s.

4. Experimental Results

In the following chapter the different variants of the RLS and the (1+1) EA are now analysed empirically for the best algorithm depending on the input. Additionally for most lemmas from the previous chapters there are also tests if they actually hold in practice.

4.1. Code

The complete java code used for all empirical studies is available on GitHub.

4.1.1. The Algorithms

All different variants of the RLS function more or the less the same. They start with an initial random value and then optimise this one value in the loop. The loop can be summarised like this:

1. generate a number k of bits to be flipped (algorithm specific)
2. flip k random bits
3. evaluate fitness of the mutated individual
4. replace old value with new value if new value is better
5. repeat if not optimal

The (1+1) EA variants behave differently at first glance as they flip each bit independently with probability c/n. This can be seen as n independent Bernoulli trials with probability c/n. The amount of bits that are flipped is therefore binomial distributed and the algorithm can be implemented exactly as the versions of the RLS. The same holds for the *pmut* operator which generates a number k from a powerlaw distribution and then flips k bits. This leads to only one implementation of a partition solving algorithm which is not only given the input array of numbers but also a generator for the amount of bits to be flipped in each step. The random values for the amount of bits to be flipped are generated according to this table:

Algorithm	Returned value
RLS	1
RLS_k^B	$y \in \{1, \dots, k\}$ with probability $\frac{\binom{n}{y}}{\sum_{i=1}^{k} \binom{n}{i}}$
RLS_k^S	uniform random value $y \in \{1, \dots, k\}$
(1+1) EA	binomial distributed value from $\sim B(n, c/n)$
pmut	powerlaw distributed value from $\sim D_{n/2}^{\beta}$

Algorithm 4.1: GenericPartitionSolver

choose x uniform random from $\{0,1\}^n$
while x *not optimal* **do**
  $x' \leftarrow x$
  $k \leftarrow$ kGenerator.generate()
  flip k uniform random bits of x'
  if $f(x') \leq f(x)$ **then**
    $x \leftarrow x'$

4.1.2. Random number generation

Java only provides a random number generator for uniform distributed values for any integer interval or random double values $\in [0,1)$. The same holds for the MersenneTwister with an implementation used from this page. All experiment were executed with both uniform random number generators. The results were rather similar, so only the results for the Fast Mersenne Twister are shown. For this project uniform random numbers do not suffice as for an efficient way of implementing the (1+1) EA or simply for generating a binomial distributed input another random number generator is needed. One of the needed distributions is a binomial distribution. The simplest way to generate a number $\sim B(m,p)$ would be to run a loop m times and add 1 to the generated number if a uniform random value $\in [0,1)$ is less than p. This works perfectly fine and generates numbers according to the distribution. With low values for p this approach is rather inefficient and especially for values of $p = 1/m$. The expected value in this case is 1 but generating a random number takes time $\mathcal{O}(m)$. Another more efficient way was implemented by StackOverflow user pjs on stackoverflow inspired by Devroyes method introduced in [Dev06]. This method has an expected running time of $\mathcal{O}(mp)$ which is equal to the expected value of the distribution. For the case of $p = 1/m$ this runs in expected constant time in comparison to $\mathcal{O}(m)$ for the naive way. This number generation was also used for the implementation of the (1+1) EA instead of running a loop in every step. Algorithm 4.2 uses the second waiting time method which uses the fact that X$\sim B(m,p)$ if X is the smallest integer so that $\sum_{i=1}^{X+1} \frac{E_i}{n-i+1} < -\log(1-p)$ for E_i iid exponential random variables (Lemma 4.5 section X.4 [Dev06]).

Algorithm 4.2: Binomial random number generator

$q \leftarrow \ln(1.0 - p)$
$x \leftarrow 0$
$sum \leftarrow 0$
while *true* **do**
  $sum \leftarrow sum + \ln(\text{random}())/(n-x)$
  `// random() generates a random value` $\in [0,1)$
  if $sum < q$ **then**
    return x
  $x \leftarrow x + 1$

The next generator needed is for geometric distributed values. This generator is only necessary for the generation of geometric distributed inputs but not for the algorithms. The easiest way to generate geometric distributed values is the naive way: generating a uniform random value p' until $p' < p$ holds. The expected running time of this algorithm is equal to the expected value of the distribution $1/p$. So this method is comparably effective to the approach used for binomial random number generation.

Algorithm 4.3: GEOMETRIC RANDOM NUMBER GENERATOR

```
sum ← 0
// random() generates a random value ∈ [0,1)
while random() ≥ q do
    sum ← sum + 1
return sum
```

The last generator needed is for powerlaw distributed values. This generator is in contrast to the geometric number generator needed for both the algorithm with the $pmut_\beta$ mutation operator and for generating inputs. This implementation is also from stackoverflow. The user gnovice provided the following formula on this page on stackoverflow:

$$x = [(b^{n+1} - a^{n+1}) * y + a^{n+1}]^{1/(n+1)}$$

a is the lower bound, b the upper bound, n the parameter of the distribution and y the number generated uniform random $\in [0, 1)$. The idea behind the formula and the formula itself is explained in a mathworld page. For a powerlaw distribution $P(x) = Cx^n$ for $x \in [a, b]$ normalisation gives

$$\int_a^b P(x)dx = C\frac{[x^{n+1}]_a^b}{n+1} = 1 \Leftrightarrow C = \frac{n+1}{b^{n+1} - a^{n+1}}$$

Let Y be a uniformly random distributed variate on [0,1]. Then

$$D(x) = \int_a^x P(x')dx' = C\int_a^x x'^n dx' = \frac{C}{n+1}(x^{n+1} - a^{n+1}) = \frac{(x^{n+1} - a^{n+1})}{b^{n+1} - a^{n+1}} \equiv y$$

and the variate is given by

$$X = [(b^{n+1} - a^{n+1}) * y + a^{n+1}]^{1/(n+1)}$$

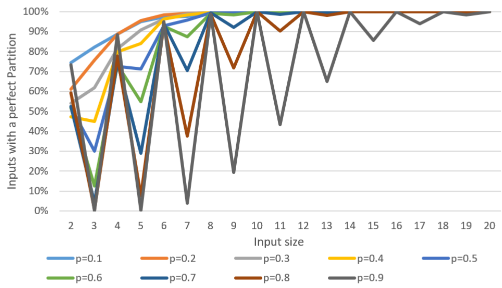
The values inserted in this formula must be negative. In the original paper for the $pmut_\beta$ operator and in the definition normally a powerlaw distribution is $P(x) = Cx^{-n}$ and therefore any positive value for n in this case was negated. Apart from this negation the generator was not changed.

4.2. Do binomial inputs have perfect partitions?

Lemma 3.14 is only valid for larger n. In practice the bound is much smaller depending on the expected value of a single number. Another factor deciding how likely an input is to have a perfect partition is whether n is even or odd. To determine the influence of all factors multiple experiments were conducted. The goal of the first experiment was to determine the influence of the array size to the input having a perfect partition and the fact if n is even or odd. So for every possible combination of $p \in \{0.1, 0.2, \ldots, 0.8, 0.9\}, m \in \{10, 100, 1000, 10^4, 10^5\}$ and $n \in \{2, 3, 4, \ldots, 19, 20\}$ 1000 randomly generated inputs of size n were tested for a perfect partition. Due to the small values for n it was possible to brute force the results in a short amount of time. The results are visualised in figure 4.1 to figure 4.9.

On the x-axis is the size of the input and on the y-axis the percentage of inputs that had a perfect partition. The different graphs in each figure resemble the different values of p

Figure 4.1.: Percentage of Binomial inputs with perfect partitions for $m = 10$



used for generating the inputs. The graph for $p = 0.1$ resembles the percentage of inputs that had a perfect partition with values generated from the distribution $\sim B(m, 0.1)$ with m being dependent on the figure. For figure 4.1 m has the value 10.
Figure 4.1 is a bit overloaded with information and the zigzag makes it hard to gain any benefit from the graphs. That's why for $m \in \{10, 100, 1000\}$ there is one figure for the even input sizes of n and one for the odd. Figures which show only results for either even or odd values of n have dotted graphs, because the values in between the points do not exist. The dotted lines are only in the figure for a better visualisation of the trend and not meant for interpretation apart from the marked values. For $n \geq 10,000$ all values for the odd input sizes are 0 %, so there is no point in showing the data in a separate figure.

Figure 4.2.: Percentage of Binomial inputs with perfect partitions for $m = 10$ for even n

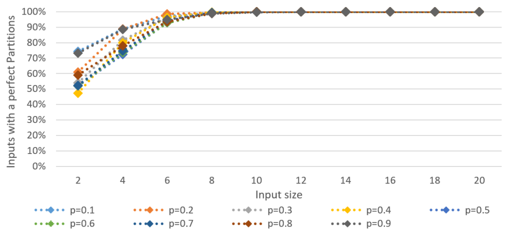


Figure 4.3.: Percentage of Binomial inputs with perfect partitions for $m = 10$ for odd n

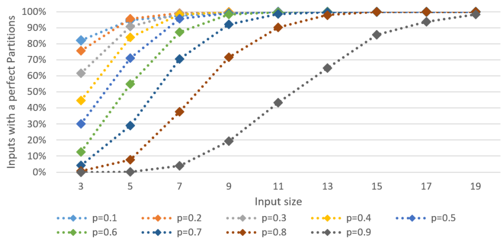


Figure 4.4.: Percentage of Binomial inputs with perfect partitions for $m = 100$ for even n

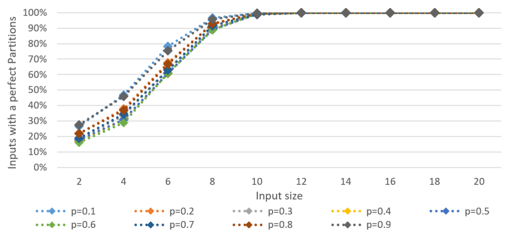
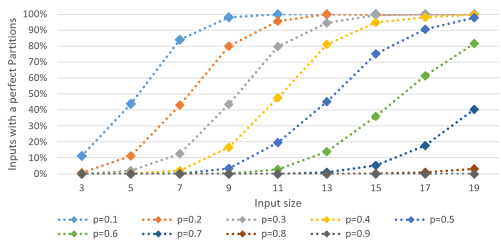


Figure 4.5.: Percentage of Binomial inputs with perfect partitions for $m = 100$ for odd n



It is easy to see that for small inputs sizes it is relevant if n is even or odd for higher expected values as all curves in figure 4.9 oscillate between 0% and 100% for $n \geq 14$. For odd inputs the probability of a perfect partition decreases much more drastically with m as for even inputs because the expected value of a single number increases with m. If all values are much higher the small differences between the values can no longer even out the fact of one set having more elements than the other. The oscillation therefore increases with increasing m. For $n = 20$ all 1000 inputs had a perfect partition for every combination

Figure 4.6.: Percentage of Binomial inputs with perfect partitions for $m = 1000$ for even n

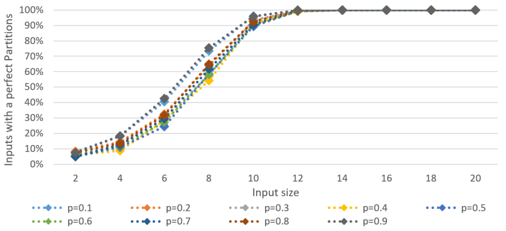


Figure 4.7.: Percentage of Binomial inputs with perfect partitions for $m = 1000$ for odd n

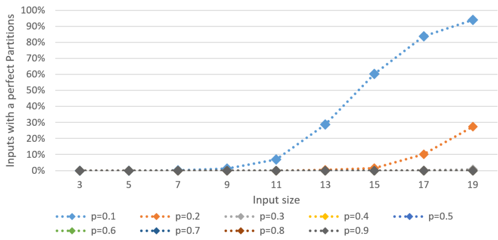


Figure 4.8.: Percentage of Binomial inputs with perfect partitions for $m = 10,000$

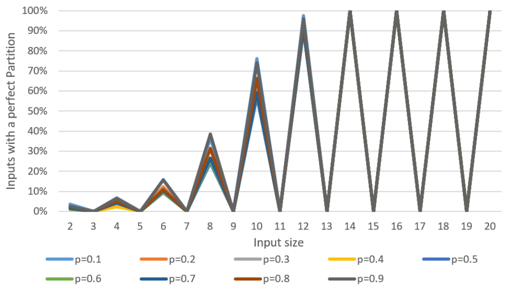
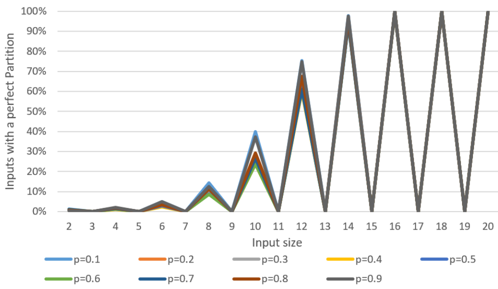


Figure 4.9.: Percentage of Binomial inputs with perfect partitions for $m = 100,000$



of p and m but for $n = 19$ only combinations where $mp \leq 300$ holds had at least one input with a perfect partition. For expected values of up to 10^5 it seems to be almost granted that an input of length 20 has a perfect partition if it is binomial distributed. Even for only 12 binomial generated values more than 50% of the inputs had a perfect partition (see figure 4.9). Another visible effect is the decreasing percentage with rising p. This may be a direct result of the value chosen for p but can also be an indirect result as the value for p changes the expected value for a constant m. The expected value may have an influence on the number of perfect partitions because it influences the highest value of the input. For uniform distributed inputs Borgs *et. al.* showed that the coefficient of number of bits needed to encode the max value/n has a huge impact on the number of perfect partitions [BCP01]. For a coefficient < 1 the probability of a perfect partition tends to 1 and for a coefficient > 1 it tends to 0. This was only proven for the uniform distributed input, but it might also hold for a binomial distributed input. This leads to the second experiment.

In the second experiment the inputs were generated a bit differently. Here the goal was to keep the expected value fixed for any combination of p and n and set the value of m to e/p for all $e \in \{10, 20, 30, 40, 50, 100, 200, 500, 1000, 2000, 5000, 10000, 50000\}$ so that $E(X) = mp = e/p \cdot p = e$. With this setup the influence of the expected value is almost isolated from the other parameters. The probability is still linked to p as p also influences the variance $mp(1 - p)$.

Figure 4.10 to figure 4.13 again show the percentage of perfect inputs with different settings of m, p, n. The x-axis is the expected value mp of a single number of the input. The different graphs show the percentage for different input sizes. It seems as if the value of p has a much smaller influence than the expected value. For a fixed expected value and a fixed input size a higher value for p seems to only slightly increase the percentage of inputs with a perfect partition. The expected value influences the percentage significantly more. For $p = 0.1, n = 14$ the value decreases from 100% at $E(X) = 10$ to below 20% at

$E(X) = 50000$ (figure 4.10). For $p = 0.9$ the percentage only drops below 50% but still decreases by a factor of 2 (figure 4.13).

Figure 4.10.: Percentage of Binomial inputs with perfect partitions for p = 0.1

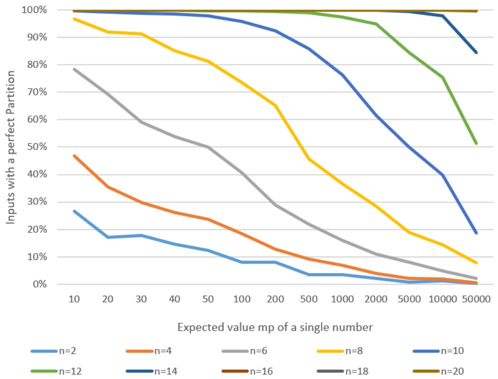


Figure 4.11.: Percentage of Binomial inputs with perfect partitions for p = 0.2

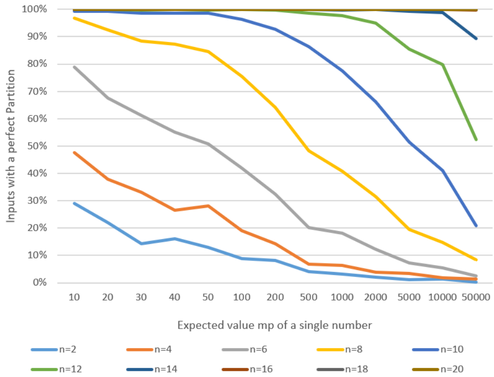


Figure 4.12.: Percentage of Binomial inputs with perfect partitions for p = 0.5

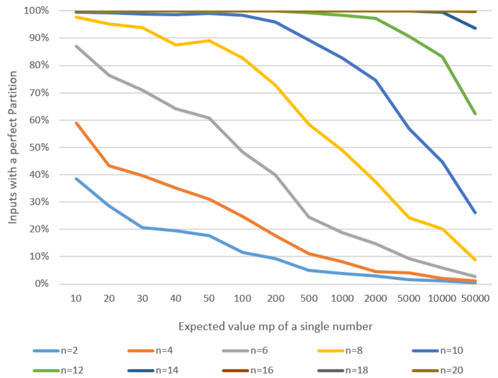
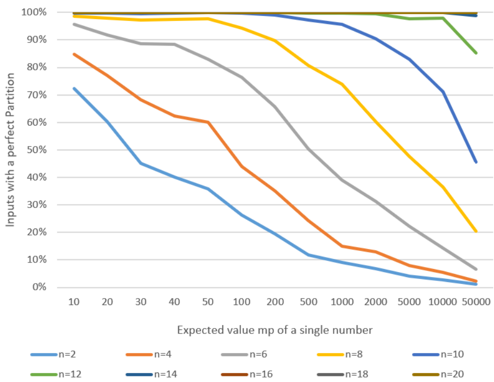


Figure 4.13.: Percentage of Binomial inputs with perfect partitions for p = 0.9



The last experiment showed that for $n = 20$, 1000/1000 inputs had a perfect partition. This raised the question of how the amount of perfect partition changes with changing values for m, p, n. Figure 4.14 to figure 4.17 show the amount of perfect partitions a binomial distribution $\sim B(m, p)$ has. For these figures 10,000 random binomial inputs with the given values for m and p were generated. Each input was then tested for the number of perfect partitions it has. The used method was again brute force to ensure correctness which was only possible due to the small input sizes. After all runs the average values were combined in the given figures. The value of p is dependent on the picture and each value of $m \in \{10, 100, 1000, 10000\}$ has its own graph within the figure. The x-axis is the size of the input and the y-axis the number of perfect partitions the input has. Notice that all graphs have a y-axis with a logarithmic scale. Since the graphs are all linear the actual values rise exponentially. The number of perfect partitions is mostly multiplied by a factor between 3 and 4 when the input size increases by 2.

The higher the value of m the closer the curves of p and $1 - p$ get. For $m = 1000$ and $m = 10000$ the values are almost the same for every input size. For $p = 0.1, m = 10$ an input with expected values of 1 seems to much more likely to have a perfect partition than an input with expected value $10 \cdot 0.9 = 9$. With growing m this has less impact.

Figure 4.14.: Amount of perfect partitions for $p = 0.1$

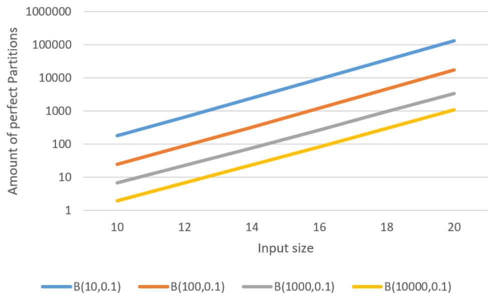


Figure 4.15.: Amount of perfect partitions for $p = 0.9$

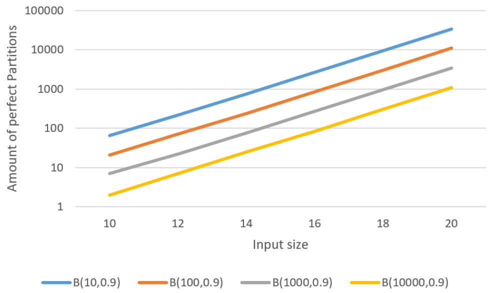


Figure 4.16.: Amount of perfect partitions for $p = 0.2$

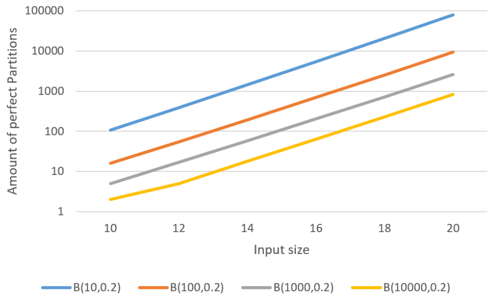
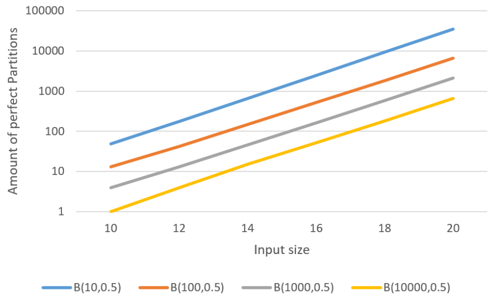


Figure 4.17.: Amount of perfect partitions for $p = 0.5$



So the binomial input should be easy to solve due to the exponential number of perfect partitions. It might be harder for the smaller values of n as there are only a few perfect partitions. Due to the small number of total possibilities it should still be easy to solve for the small values of n as long as the RSH is not stuck in a local optimum. The number of iterations might be high in terms of the big-O notation but should still be small in the absolute value.

4.3. Binomial distributed values

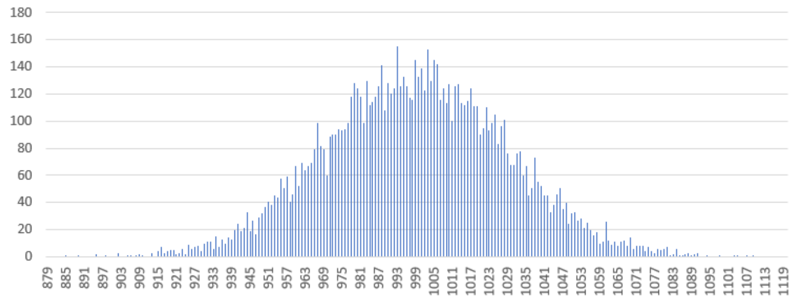
In the following subsections the performance of the different algorithms is tested for different kinds of inputs. The exact distributions of the input are explained separately in each subsection. The procedure for each comparison is always the same. A random input is generated according to the distribution and then solved by every algorithm. All algorithms had the same two stopping conditions. The first was reaching a perfect partition and the second was taking more than $10 \cdot n \ln(n)$ steps. For the lower values of n the step limit of 100,000 was used instead. For $n = 20$ giving the algorithm only 600 steps is rather small. In some cases the smaller inputs are even more difficult to solve. Most modern computer should be able to handle 100,000 iterations in a short amount of time anyway. So the minimum step limit of 100,000 seemed reasonable. If either of these conditions was met, the algorithm returned its current best solution. This step is repeated 10,000 times. The results are presented in a table containing multiple statistics for each algorithm over all 10,000 runs. The data is explained in the table below.

column name	meaning
algo type	type of algorithm (RLS, RLS_k^B, RLS_k^S, (1+1) EA or pmut)
algo param	parameter of the algorithm or '-' if it is the standard variant
avg mut/change	average #bits flipped for iterations leading to an improvement
avg mut/step	average #bits flipped for any iteration
total avg count	average #iterations for all runs
avg eval count	average #iterations of runs returning an optimal solution
max eval count	maximum #iterations of runs returning an optimal solution
min eval count	minimum #iterations of runs returning an optimal solution
fail ratio	ratio of unsuccessful runs to all runs
avg fail dif	average value of $b_F - f(opt)$ for non-optimal solutions

Sometimes every algorithm managed to find an optimum in each run. To avoid redundancy and shorten the thesis the rows 'total avg count' and 'avg fail dif' are not shown in the table in those cases.
Firstly the different variants of the RLS are compared with values of $k \in \{2, 3, 4\}$, then the performance of the (1+1) EA with static mutation rate c/n with $c \in \{1, 2, 3, 5, 10, 50, 100\}$ and lastly the performance of the $pmut_\beta$ mutation operator with the parameter $\beta \in \{1.25, 1.5, \ldots, 2.75, 3.0, 3.25\}$. Afterwards there is a comparison for multiple input sizes of the best algorithms because the best algorithm is often dependent on the size of the input. Normally there are three tables for each input. The first states how often the algorithms did not find an optimal solution for the different input sizes ('fails' in top left cell). The second gives their average performance for the successful runs ('avg' in top left cell) and the last the performance for all runs ('total avg' in top left cell). The last two tables differ in the unsuccessful runs. Often the algorithm is stuck in a local optima it won't leave in reasonable time or even never for variants of the RLS. In these cases the step limit is the deciding factor on how big the penality for this run is. So neither of the two average values alone is enough to give a complete insight on the performance. Sometimes a variant of the RLS is much faster than the other algorithms for a specific input but is also the only algorithm to get stuck in a local optimum. This creates the possibility to start the RLS variant with a low step limit and switch to the (1+1) EA if the RLS variant does not return an optimal solution. Giving both tables for the different average values might help with this decision.

The first analysed inputs are inputs following a binomial distribution ~$B(m, p)$ as those inputs have been researched in the previous subsection. The results showed that the expected value of a single number is the main driver for the amount of perfect partitions the input has. The results also suggested the inputs tend to have more perfect partitions if the expected value is lower. The more perfect partitions an input has relative to the number of all possible partitions, the more likely the different RSHs are to find one of those. Therefore researching inputs with higher expected values seems more interesting but generating higher values takes more time with a random number generator that needs $\mathcal{O}(mp)$ time. To keep the time for generating one set of numbers reasonable the values chosen for all tests are $m = 10000, p = 0.1, n = 10000$ with the expected value for a single number being $mp = 1000$. Figure 4.18 shows a random binomial distributed input of length $n = 10000$. For this input type almost every time all elements were sharply concentrated around the expected value with all values being at 1000 ± 200 (for figure 4.18 even closer at 1000 ± 121). So after reaching a difference between the two bins of below $(1000 - 200)/2 = 400$ the algorithm can no longer achieve an improvement by flipping a single bit (Corollary 3.7)

Figure 4.18.: Distribution of a random binomial input



4.3.1. RLS Comparison

algo type	RLS_b^B	RLS_b^B	RLS_s^S	RLS_s^S	RLS_s^S	RLS_b^B	RLS
algo param	b=2	b=4	s=2	s=4	s=3	b=3	-
avg mut/change	2.000	4.000	1.602	2.563	2.000	2.735	1.000
avg mut/step	2.000	4.000	1.500	2.500	2.000	3.000	1.000
total avg count	295	409	454	538	636	488,329	919,832
avg eval count	295	409	454	538	636	380,018	103
max eval count	1,900	3,717	3,562	4,300	6,057	920,419	199
min eval count	4	2	5	4	3	9	9
fail ratio	0.000	0.000	0.000	0.000	0.000	0.200	0.999
avg fail dif	-	-	-	-	-	1	246

The RLS_2^B seems to perform the best as it mostly switches two elements which works great for binomial distributed inputs. The same algorithm with $b = 4$ performs a bit worse but still good as switching 4 elements can be beneficial as well. The variant of RLS_3^B on the other hand does not reach the optimal solution in 20% of the inputs with an average difference of 1. It also needs 1000 times more iterations to find an optimum on average compared to the best algorithm RLS_2^B. The RLS_s^S variants behave mostly the same with $s = 2$ being the best, followed by $s = 4$ and $s = 3$. In this case the variant of $s = 3$ is by far not as bad as for the RLS_3^B because the probability of flipping 2 bits is $1/3$ as compared to $\mathcal{O}(n^{-1})$ for the RLS_3^B. The RLS_k^S seems all to be a good option for binomial inputs with values of $k \in \{2, 3, 4\}$. The standard RLS on the other hand performs by far the worst as it only moves one element per step. It only managed to reach the optimal solution 13 times for 10,000 different inputs. The average number of iterations for those inputs was only 103 so the RLS likely had a good initialisation with a few lucky steps leading directly to the optimum. For all other cases the average difference between the bins was 246 which is close to the median of the values from 0 to $(1000 - 100)/2 = 450$. This is likely due to the RLS being unable to improve the solution once the current solution has a difference below half of the lowest value (Corollary 3.7).

4.3.2. (1+1) EA Comparison

For the (1+1) EA the best static mutation rate seems to be $3/n$. The probability of flipping 2 or 4 bits as n goes to infinity for mutation rate $1/n$ approaches $13/24e \approx 0.199$, for $2/n$ approaches $8/3e^2 \approx 0.361$, for $3/n$ approaches $63/8e^3 \approx 0.392$, for $4/n$ approaches $56/3e^4 \approx 0.342$ and for $5/n$ approaches $77/2e^5 \approx 0.259$. So the highest probability has $c = 3$, followed by $c = 4$ and $c = 2$ then $c = 5$ and lastly $c = 1$. For higher values of c the probability decreases further as the expected number of flipped bits is c for mutation rate c/n.

algo type	EA-SM	EA-SM	EA-SM	EA-SM	EA-SM	EA-SM	EA-SM	EA-SM
algo param	$3/n$	$4/n$	$2/n$	$5/n$	-	$10/n$	$50/n$	$100/n$
avg mut/change	3.103	3.953	2.339	4.861	1.695	9.727	49.587	99.529
avg mut/step	3.000	4.000	2.000	4.999	1.000	10.000	50.001	100.001
avg eval count	594	642	645	731	1,080	1,370	7,052	13,624
max eval count	6,084	5,368	6,151	8,083	8,767	18,297	113,206	155,424
min eval count	3	3	0	3	7	7	3	5
fail ratio	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

The static mutation rate $3/n$ seems to perform the best with both $4/n$ and $2/n$ being a close second place. The next best values are $5/n$ and the standard $1/n$. From then on the number of iterations rises monotonically with rising mutation rate. The higher mutation rates perform significantly worse but the still find a solution within the limit as opposed to the standard RLS.

4.3.3. pmut Comparison

algo type	pmut	pmut	pmut	pmut	pmut	pmut	pmut	pmut	pmut
algo param	2.00	2.25	2.50	1.75	2.75	3.00	3.25	1.50	1.25
avg mut/change	6.667	3.890	2.815	15.093	2.289	2.009	1.833	38.481	95.474
avg mut/step	8.413	4.356	2.900	22.396	2.270	1.935	1.729	70.698	224.871
avg eval count	610	612	629	641	645	677	717	721	974
max eval count	6,462	5,408	7,478	5,681	6,321	8,262	6,415	8,077	11,014
min eval count	3	0	1	3	7	8	3	7	3
fail ratio	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

For the $pmut_\beta$ mutation operator the choice of β seems to be much more insignificant than for the RLS or (1+1) EA. Here all values perform comparably good with only the value of $\beta = 1.25$ having a clear performance difference compared to next best value. All values of β reach an optimal solution in every case. The worst variant of the $pmut_\beta$ operator still performs much better than the worst value for the (1+1) EA and even better than the worst RLS variant. There is no clear winner but because $\beta = 2.0$ had the best performance in this experiment, it was used for the comparison of the best variants.

4.3.4. Comparison of the best variants

Looking at the previous tables for this setting of $m = 10000, p = 0.1, n = 10000$ the RLS_2^B performs better than the (1+1) EA and $pmut_\beta$ mutation for all values of c/n and β by a factor of at least 2. This is likely from the fact that this version of the RLS flips almost only two bits which seems to be close to optimal for this kind of input. There are many values close to the expected value which can be switched to make small adjustments to the fitness value. To further investigate which algorithm performs best on all binomial distributed inputs now a comparison with different input lengths follows. The parameters of the distribution were not changed.
The following table shows the number of runs in which the algorithms did not find an optimal solution within 50,000 steps. The time limit was set 50,000 because the algorithms normally reach the optimal solution within a few thousand steps. If the solutions is not found after 50,000 steps, the algorithm is most likely stuck in a local optimum which could only be left by flipping more bits than possible for the algorithm.

fails in 1000 runs	20	50	100	500	1000	5000	10000
RLS_2^S	231	22	3	0	0	0	0
RLS_4^B	0	1	2	0	0	0	0
RLS_2^S	243	4	0	0	0	0	0
RLS_4^S	1	0	0	0	0	0	0
(1+1) EA ($3/n$)	0	0	0	0	0	0	0
pmut(2.5)	0	0	0	0	0	0	0

The (1+1) EA and $pmut_\beta$ always reach an optimal solution but the RLS variants do not. The RLS variants that can only flip two bits per step perform significantly worse for small inputs. They are probably more likely to get stuck in a local optimum where a step flipping 4 bits or more would be necessary. So the RLS_2^B does perform better for larger inputs but is much more likely to get stuck in a local optima. The next table contains the average number of iterations the algorithm needed to find an optimal solution for all runs where the algorithms managed to find an optimal solution. Here it still looks like the RLS_2^B finds the solution with the lowest amount of steps, because the cases where the algorithm is stuck in a local optima are not contained in this table.

avg	20	50	100	500	1000	5000	10000
RLS_2^S	164	257	337	256	257	293	310
RLS_4^B	349	355	661	412	412	436	425
RLS_2^S	267	435	408	428	442	452	496
RLS_4^S	601	491	492	491	534	562	560
(1+1) EA ($3/n$)	716	570	569	580	614	625	650
pmut(2.5)	1596	576	600	637	657	700	685

The next table contains the overall average amount of iterations for every run. So runs where no optimal result was found add 50,000 to the sum of all iterations.

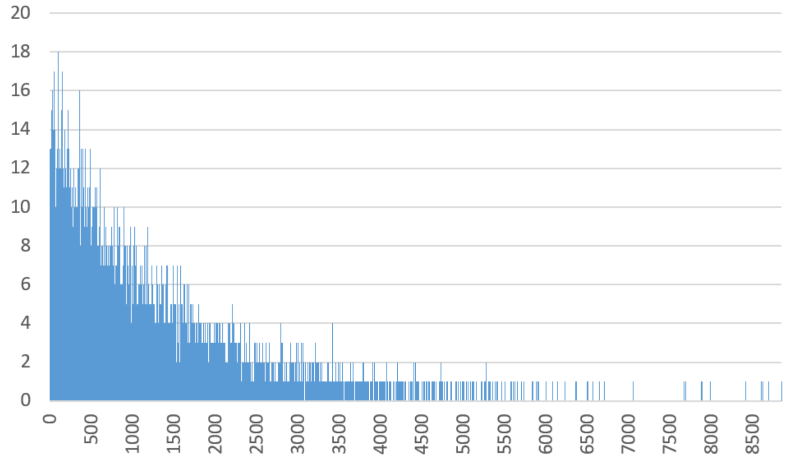
total avg	20	50	100	500	1000	5000	10000
RLS_2^S	11676	1351	486	256	257	293	310
RLS_4^B	349	405	760	412	412	436	425
RLS_2^S	12352	633	408	428	442	452	496
RLS_4^S	650	491	492	491	534	562	560
(1+1) EA ($3/n$)	716	570	569	580	614	625	650
pmut(2.5)	1596	576	600	637	657	700	685

Here the RLS_2^B is only the best algorithm for values of $n \geq 500$. Below this bound choosing the (1+1) EA with static mutation rate $3/n$ is a safer choice as the (1+1) EA reaches an optimal solution for every input (in this experiment).

4.4. Geometric distributed values

For the geometric distribution the chosen default value is $p = 0.001$. This results in an expected value of 1000 which is the same as for the binomial distribution in the last subsection. This should make the results more comparable. The maximum value is theoretically not limited but for the implementation in Java the maximum value was set to the maximum value of a long value $= 2^{63} - 1 = 9,223,372,036,854,775,807$. Without this maximum the value might overflow and instead be negative with high absolute value. Figure 4.19 shows a random geometric distributed input. The span of all values is way higher than for the binomial distribution, although they have same expected value. Here the values are not in the interval $[800, 1200]$ but rather between 0 and 9000. The theoretical limitation of the values being at most $2^{63} - 1$ seems to not have an influence on the results. The geometric distribution does not only have low values close or equal to 1 but also has mostly values that are very small. This should lead to 1-bit flips being effective as the small values can remove the small differences. Because there are so many small values moving only one bit might be better than switching two elements.

Figure 4.19.: Distribution of a random geometric input



4.4.1. RLS Comparison

algo type	RLS_s^S	RLS_s^S	RLS_s^S	RLS_b^B	RLS	RLS_b^B	RLS_b^B
algo param	s=2	s=3	s=4	b=2	-	b=3	b=4
avg mut/change	1.483	1.958	2.428	2.000	1.000	3.000	4.000
avg mut/step	1.500	2.000	2.500	2.000	1.000	3.000	4.000
total avg count	2,218	2,597	2,924	3,130	3,816	4,062	4,853
avg eval count	2,218	2,597	2,924	3,130	1,886	4,062	4,853
max eval count	34,066	38,333	48,275	43,116	68,298	46,410	51,427
min eval count	0	0	1	7	4	4	0
fail ratio	0.000	0.000	0.000	0.000	0.002	0.000	0.000
avg fail dif	-	-	-	-	1	-	-

For these inputs the variants of the RLS perform differently to the binomial input. The only similarity is the RLS being the only algorithm that did not find an optimal solution for every input. If the RLS did find an optimal solution in those 21 cases it instead might be the best RLS variant. The other algorithms are ranked by their probability of flipping only one bit. This means at first the three RLS_s^S variants from 2 to 3 to 4 and then the same for the RLS_b^B variants. So it does seem like moving mostly one element at once is better for the geometric input in comparison to two elements for the binomial distribution. In the 21 cases where the RLS did not find an optimal solution it was most likely stuck in a local optimum where no small value was left.

4.4.2. (1+1) EA Comparison

algo type	EA-SM	EA-SM	EA-SM	EA-SM	EA-SM	EA-SM	EA-SM	EA-SM
algo param	$2/n$	-	$3/n$	$4/n$	$5/n$	$10/n$	$50/n$	$100/n$
avg mut/change	2.246	1.551	3.048	3.936	4.861	9.822	49.750	99.707
avg mut/step	2.000	1.000	3.000	4.000	5.000	10.000	50.000	100.001
avg eval count	3,097	3,505	3,518	4,009	4,807	7,758	18,457	25,993
max eval count	39,490	60,533	39,048	47,881	56,204	91,305	173,851	354,479
min eval count	10	0	6	5	3	5	9	3
fail ratio	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

The results for the (1+1) EA are similar to the results of the RLS. From mutation rate $2/n$ on the runtime increases with rising mutation rate. The only part that does not fit into the theory of 1 bit flips being superior is the mutation rate $2/n$ performing better than the standard $1/n$. All variants reach an optimal solution within the given limit for the number of iterations.

4.4.3. pmut Comparison

algo type	pmut	pmut	pmut	pmut	pmut	pmut	pmut	pmut	pmut
algo param	3.25	3.00	2.75	2.50	2.25	2.00	1.75	1.50	1.25
avg mut/change	1.689	1.862	2.162	2.681	3.834	6.885	15.848	41.832	104.749
avg mut/step	1.729	1.934	2.270	2.905	4.367	8.500	22.219	70.666	224.556
avg eval count	2,207	2,268	2,376	2,480	2,586	2,813	3,196	3,898	5,399
max eval count	40,626	45,193	39,295	41,558	40,339	46,110	43,223	61,121	60,767
min eval count	0	5	9	5	7	2	3	1	10
fail ratio	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

The results for the $pmut_\beta$ operator are even more clear than for the (1+1) EA. With decreasing values for β the amount of flipped bits per step increases. The performance decreases as well with decreasing values for β which fits into the theory of one bit flips being better for geometric distributed inputs. The number of repetitions of the algorithm might simply be too small to make the small difference in the performance between the two values visible. The difference in the performance for the $pmut_\beta$ operator is not as drastic as for the (1+1) EA. Only $\beta = 1.5$ and $\beta = 1.25$ perform significantly worse the next best value.

4.4.4. Comparison of the best variants

The setup for the evaluation of lower values for n is mostly the same except for having a fixed time limit of 100,000 instead of using 50,000 as the limit. The first try was executed with 50,000 but there the algorithms performed too bad for $n = 20$. Therefore in the second attempt the step limit was increased to 100,000. The first table lists the number of runs where the different algorithms did not find the optimal solution within the time limit.

fails in 1000 runs	20	50	100	500	1000	5000	10000
RLS	983	951	885	619	425	42	0
RLS_2^S	881	578	171	0	0	0	0
(1+1) EA ($1/n$)	464	132	48	1	1	0	0
(1+1) EA ($2/n$)	149	11	1	0	0	0	0
pmut(3.25)	284	67	27	0	0	0	0
pmut(3.5)	312	91	38	0	1	0	0

For small inputs the geometric distributed input seems to have inputs without a perfect partition because there were many iterations where neither of the algorithms found an optimal solution within the time limit. It is still likely to have a perfect partition even for the small values in comparison to other distributions which follow afterwards. Many algorithms especially the variants of the RLS seem to be likely to get stuck in a local optimum. The (1+1) EA finds an optimum in most of the runs, so the geometric distributed inputs also seem to be likely to have a perfect partition for small values. They definitely are harder to solve for smaller input sizes than the binomial inputs, but they still have a perfect partition most times. The next table visualises the average number of iterations the algorithms needed for finding an optimal solution if the algorithm managed to do so.

avg	20	50	100	500	1000	5000	10000
RLS	35	78	140	566	904	2119	2188
RLS_2^S	357	2024	5369	4687	3945	2752	2583
(1+1) EA ($1/n$)	21827	20775	14583	9459	7702	4358	3924
(1+1) EA ($2/n$)	18211	11613	7529	5266	4359	3858	3293
pmut(3.25)	22530	16421	11312	5909	5375	2843	2246
pmut(3.5)	24202	17414	11731	6503	5216	2773	2388

The variants of the (1+1) EA and of the *pmut* algorithm seem to take about 20,000 iterations for $n = 20$ if they manage to find the optimal solution. They also perform better and better the bigger the input gets. This is probability caused by the many additional small values that can be used for smaller adjustments to the fitness. Also a really high value does not have as much of an effect, because there are possibly other larger values which cancel each other out, if they are in different bins. The standard (1+1) EA does not only find a perfect partition less often, it also needs more iterations on average if it does. So the (1+1) EA with $p_m = 2/n$ performs indeed better for every input size. The last table again lists the total average number of steps.

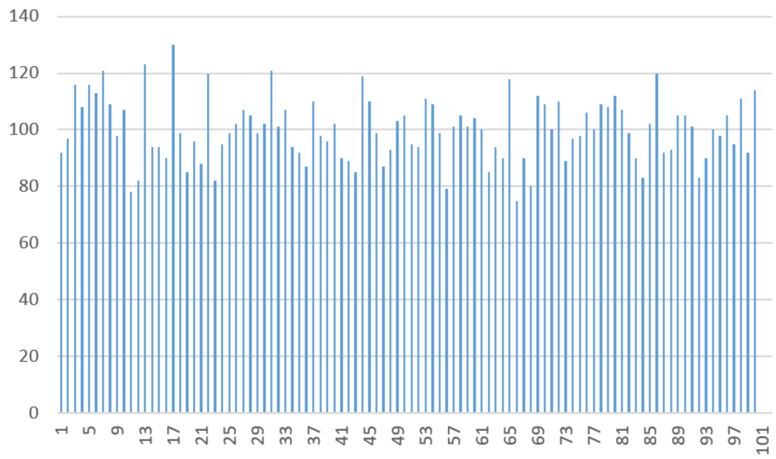
total avg	20	50	100	500	1000	5000	10000
RLS	98300	95103	88516	62115	43019	6230	2188
RLS_2^S	88142	58654	21551	4687	3945	2752	2583
(1+1) EA ($1/n$)	58099	31232	18683	9550	7795	4358	3924
(1+1) EA ($2/n$)	30397	12585	7622	5266	4359	3858	3293
pmut(3.25)	44531	22021	13707	5909	5375	2843	2246
pmut(3.5)	47851	24929	15086	6503	5311	2773	2388

The RLS is only an option if the input is large enough ($n \geq 10,000$). For smaller input sizes especially for $n \leq 100$ choosing the (1+1) EA with mutation rate $2/n$ seems like the best choice. For larger values this (1+1) EA does not find an optimal solution the fastest but is still fast enough to be a viable option. Another rather save option is $pmut_{3.25}$. This algorithm performs worse for $n \leq 100$ but is still good in comparison to the other algorithms. For $n \geq 1000$ $pmut_{3.25}$ starts to outperform the best version of the (1+1) EA and almost all other researched algorithms.

4.5. Uniform distributed inputs

For the uniform distribution the default values were 1 for the lower bound and 50000 for the upper bound (exclusive). The range was limited to 50000 to reduce the time the algorithms needs to find an optimal solution. The higher the values are with too few values the more likely the input is to not have a perfect partition[BCP01]. This will cause the algorithms to always reach the limit for the number of iterations which drastically increases the time needed for the experiment. The length of the input was 50000.

Figure 4.20.: Distribution of a random uniform input (10000 values between 1 and 100)



4.5.1. RLS Comparison

algo type	RLS_b^B	RLS_s^S	RLS_s^S	RLS_s^S	RLS_b^B	RLS_b^B	RLS
algo param	b=2	s=3	s=4	s=2	b=3	b=4	-
avg mut/change	2.000	1.991	2.481	1.501	3.000	4.000	1.000
avg mut/step	2.000	2.000	2.500	1.500	3.000	4.000	1.000
total avg count	73,797	93,552	95,746	96,698	101,538	112,431	2,134,880
avg eval count	73,797	93,552	95,746	96,698	101,538	112,431	45,419
max eval count	727,262	1,216,841	1,769,247	1,954,892	1,211,069	1,532,872	404,899
min eval count	23	30	40	36	15	27	38
fail ratio	0.000	0.000	0.000	0.000	0.000	0.000	0.390
avg fail dif	-	-	-	-	-	-	1

The picture for the RLS variants on this type of input is not clear. There in no obvious tendency for neither of the variants. The only obvious thing is the RLS being the worst of the RLS variants again. Every variant reaches the optimal solution in every case except for the RLS which only manages for 61 % of the inputs. All other variants reach an optimal solution for every input. The RLS_2^B seems to be the best variant for these kinds of inputs. The next best variants are the three RLS_s^S variants which differ only slightly between each other.

4.5.2. (1+1) EA Comparison

algo type	EA-SM	EA-SM	EA-SM	EA-SM	EA-SM	EA-SM	EA-SM	EA-SM
algo param	$4/n$	$3/n$	$2/n$	$5/n$	$10/n$	-	$50/n$	$100/n$
avg mut/change	4.012	3.096	2.292	4.945	9.887	1.580	49.788	99.729
avg mut/step	4.000	3.000	2.000	5.000	10.000	1.000	50.000	100.000
total avg count	107,309	111,459	115,595	115,597	166,608	188,301	369,943	551,189
avg eval count	107,309	111,459	115,595	115,597	166,608	188,301	369,943	548,386
max eval count	943,738	1,166,443	1,099,884	1,368,459	1,579,593	2,516,372	4,093,515	4,552,116
min eval count	182	173	22	228	107	245	365	157
fail ratio	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.001
avg fail dif	-	-	-	-	-	-	-	1

The (1+1) EA seems to perform better with a lower mutation rate. $p_m = 4/n$ reaches an optimal solution the fastest. To both sides the speed of convergence decreases with decreasing/increasing mutation rate. For the uniform distributed input all variants of the (1+1) EA reach an optimal solution within the step limit except for $100/n$ which does not find an optimal solution in about 0.1 % of the inputs.

4.5.3. pmut Comparison

algo type	pmut	pmut	pmut	pmut	pmut	pmut	pmut	pmut	pmut
algo param	2.00	2.25	1.75	2.75	2.50	3.00	3.25	1.50	1.25
avg mut/change	7.943	4.039	24.963	2.220	3.079	1.905	1.708	99.853	314.277
avg mut/step	10.095	4.562	34.759	2.274	2.931	1.934	1.729	158.122	719.683
avg eval count	106,010	109,947	110,591	110,601	113,455	116,213	117,593	137,223	167,556
max eval count	1,468,480	1,490,905	1,104,274	1,588,457	1,571,705	1,675,031	1,269,567	1,473,465	1,814,702
min eval count	67	137	223	257	109	128	316	50	379
fail ratio	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

The optimal value for β seems to be somewhere around 2.0. The neighbouring values in both directions need more time the greater the difference to 2.0 A major increase of the runtime happens at $\beta = 1.5$ and even worse at $\beta = 1.25$, yet even the worst parameter manages to find an optimal solution in every run.

4.5.4. Comparison of the best variants

For the uniform distributed input the best variant of the RLS once again seems to perform the best. But by looking at the smaller values this does not hold in general.

fails in 1000 runs	20	50	100	500	1000	5000	10000	50000
RLS_2^S	996	980	892	360	289	14	0	0
RLS_3^S	974	768	645	461	426	54	3	0
RLS_4^S	902	621	541	460	431	53	4	0
(1+1) EA ($2/n$)	885	690	641	510	483	85	5	0
(1+1) EA ($3/n$)	772	596	549	425	449	53	2	0
(1+1) EA ($4/n$)	697	509	453	439	396	44	3	0
pmut (2.5)	840	738	696	553	520	86	10	0

The RLS variants are the most likely to get stuck in a local optimum for $n \leq 100$. The (1+1) EA variants also often do not find an optimal solution, but this happens less frequently. The more values the input has the more likely it is for any of the algorithms to find a perfect partition. Between $n = 100$ and $n = 500$ the performance of the RLS_2^B drastically increases and for $n \geq 500$ this variant of the RLS stays the best variant for the remaining input sizes. Uniform distributed inputs seem to be much less likely to have a perfect partition for the small input sizes which can be explained by the coefficient of Borgs *et. al.* [BCP01].

avg	20	50	100	500	1000	5000	10000	50000
RLS_2^S	364	1540	6759	36358	37324	77210	83500	82738
RLS_3^S	5204	33492	39795	40336	38838	104864	118393	106196
RLS_4^S	17681	39895	38795	39959	38857	98693	108780	107857
(1+1) EA ($2/n$)	36004	41914	40131	42050	38937	108903	133264	122042
(1+1) EA ($3/n$)	32791	38954	38422	40617	41017	103831	112166	111402
(1+1) EA ($4/n$)	39665	39926	40709	39479	41889	99899	127578	110099
pmut (2.5)	39232	36918	37162	39866	40588	118671	144665	128531

The amount of steps needed to find an optimal solution seems to be nearly constant for every algorithm as the number of steps does not strictly increase with n but sometimes even decreases for $n \leq 1000$. This is caused by the number of steps the algorithm was given. For $n \leq 1000$ the time limit was 100,000 and for the bigger values it was $10n \ln(n)$. Interestingly enough the average runtime decreases from $n = 10000$ to $n = 50000$ for most algorithms despite the bigger input size.

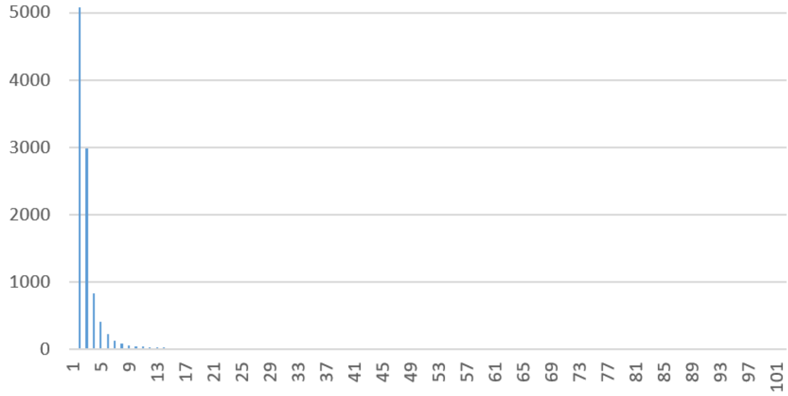
total avg	20	50	100	500	1000	5000	10000	50000
RLS_2^S	99601	98030	89930	59269	55438	82091	83500	82738
RLS_3^S	97535	84570	78627	67841	64893	122198	120801	106196
RLS_4^S	91932	77220	71907	67578	65209	116033	112029	107857
(1+1) EA ($2/n$)	92640	81993	78507	71604	68430	135844	137203	122042
(1+1) EA ($3/n$)	84676	75337	72228	65854	67500	120899	113784	111402
(1+1) EA ($4/n$)	81718	70503	67568	66047	64901	114241	129959	110099
pmut (2.5)	90277	83472	80897	73120	71482	145089	152429	128531

My general advice would be choosing the RLS_2^B for $n \geq 500$ and the (1+1) EA with $p_m = 4/n$ otherwise.

4.6. powerlaw distributed inputs

This distribution has mostly small values, but occasionally it also generates bigger values. The lower the parameter the higher the values get and also the amount of big values increases. For a parameter of $\beta = 2.75$ and a maximum value of 10,000 the distribution

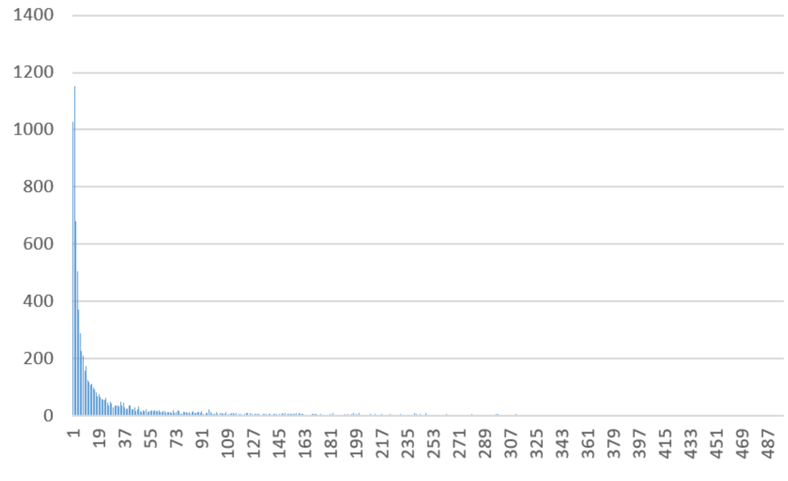
Figure 4.21.: Distribution of a random powerlaw input with $\beta = 2.75$



looks like in Figure 4.21. All values are rather small and less than 100, also half of the values are one. So this input seems rather easy for $\beta = 2.75$.

For a value of $\beta = 1.25$ the distribution looks a bit different. There are less small values close to one and instead also big values even over 1000. Figure 4.22 is cropped to get a more clear view for the smaller values. The higher values mostly occurred 0 to 2 times. The highest value 9948 occurred only once. Researching inputs like this should be more interesting which is why $\beta = 1.25$ was chosen for the experiment. To give a better view on this type of input there is also a table for $\beta = 2.75$ at the evaluation of the (1+1) EA. The results for the other algorithms were mostly the same, but these are not shown here for better readability.

Figure 4.22.: Distribution of a random powerlaw input with $\beta = 1.25$



4.6.1. RLS Comparison

algo type	RLS_s^S	RLS_b^B	RLS_s^S	RLS_b^B	RLS_b^B	RLS_s^S	RLS
algo param	s=4	b=3	s=3	b=2	b=4	s=2	-
avg mut/change	2.458	3.000	1.975	2.000	4.000	1.488	1.000
avg mut/step	2.501	3.000	2.000	2.000	4.000	1.500	1.000
avg eval count	302	330	342	359	384	417	577
max eval count	1,214	1,616	1,414	1,380	2,574	2,077	2,400
min eval count	4	3	7	10	5	15	16
fail ratio	0.000	0.000	0.000	0.000	0.000	0.000	0.000

The input is even easier to solve for the RLS variants than the binomial distributed inputs. There is no clear tendency and all algorithms have a rather equal runtime. All algorithm manage to find an optimal solution in every run.

4.6.2. (1+1) EA Comparison

The first table shows the results for parameter $\beta = 2.75$

algo type	EA-SM	EA-SM	EA-SM	EA-SM	EA-SM	EA-SM	EA-SM	EA
algo param	$50/n$	$100/n$	$10/n$	$5/n$	$4/n$	$3/n$	$2/n$	-
avg mut/change	49.922	99.873	10.009	5.053	4.105	3.156	2.280	1.534
avg mut/step	49.989	100.017	9.999	4.999	4.005	3.003	2.000	0.999
avg eval count	84	103	111	157	184	208	273	461
max eval count	1,281	1,488	1,946	3,030	3,043	3,283	4,744	7,036
min eval count	0	1	3	1	0	0	2	0
fail ratio	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

For $\beta = 2.75$ the results are much different from $\beta = 1.25$. Until $p_m \leq 50/n$ the speed of convergence increases but at $p_m = 100/n$ the speed decreases again. The optimal value seems to be somewhere around $p_m = 50/n$. The (1+1) variants are generally faster than all RLS variants when comparing the maximum number of iterations. For mutation rates $3/n \leq p_m \leq 100/n$ the (1+1) EA is also faster on average. The next table shows the results for a powerlaw distribution with $\beta = 1.25$.

algo type	EA-SM	EA-SM	EA-SM	EA-SM	EA-SM	EA-SM	EA-SM	EA-SM
algo param	$4/n$	$3/n$	$5/n$	$2/n$	-	$10/n$	$50/n$	$100/n$
avg mut/change	3.955	3.074	4.860	2.267	1.562	9.618	49.506	99.633
avg mut/step	4.003	2.999	4.999	1.999	1.000	10.001	50.000	100.000
avg eval count	284	295	311	366	640	1,190	47,165	87,514
max eval count	1,224	1,214	1,734	1,510	2,723	14,032	419,043	1,123,496
min eval count	15	8	15	3	0	15	13	1
fail ratio	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

With this setting the optimal value is shifted to somewhere around $p_m = 4/n$. The higher mutation rates perform drastically slower with $p_m = 100/n$ being 500 times slower than the optimal value. The speed of convergence is sometimes even to slow find an optimal solution in time $10 \cdot n \ln(n)$. So the parameter of the distribution does change the optimal parameter for the Evolutionary Algorithm solving the input. The same was true for the RLS and also for *pmut*.

4.6.3. pmut Comparison

algo type	pmut	pmut	pmut	pmut	pmut	pmut	pmut	pmut	pmut
algo param	1.50	1.75	1.25	2.00	2.25	2.50	2.75	3.00	3.25
avg mut/change	48.278	17.438	124.596	7.062	3.895	2.708	2.173	1.876	1.692
avg mut/step	99.756	26.795	370.116	9.098	4.440	2.921	2.275	1.932	1.728
avg eval count	181	193	219	225	265	301	337	366	388
max eval count	691	750	1,118	762	1,088	1,173	1,488	1,534	1,755
min eval count	8	11	5	12	13	11	11	8	13
fail ratio	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

The optimal value here seems to be somewhere around $\beta = 1.5$ instead of $\beta = 1.25$ for inputs from $\sim D^{2.75}_{20.000}$. So the optimal value is only lightly smaller in comparison to the (1+1) EA where the optimal value almost change from one side of the spectrum to the other.

4.6.4. Comparison of the best variants

The RLS_4^S performs equally good as the (1+1) EA variant with $p_m = 4/n$, but both are slower than $pmut_{1.5}$. Choosing any algorithm should solve the input quite fast if the parameter of the algorithm is somewhat close to the optimum.

fails in 1000 runs	20	50	100	500	1000	5000	10000	50000
RLS_3^S	822	400	87	0	0	0	0	0
RLS_4^S	806	389	80	0	0	0	0	0
(1+1) EA ($3/n$)	803	364	51	0	0	0	0	0
(1+1) EA ($4/n$)	803	359	49	0	0	0	0	0
pmut (1.5)	802	353	41	0	0	0	0	0
pmut (1.75)	804	354	41	0	0	0	0	0

The RLS is once again the algorithm that is the most likely to be stuck in a local optimum. Compared to the other algorithms it is not as drastic as for the binomial input for example. Only for $n < 500$ the algorithms do not find a global optimum in every run. The setting of the parameter and the choice of the algorithm almost doesn't affect the amount of runs without an optimal result. This type of input is probably easy to solve if it has a perfect partition. The two stopping conditions where a step limit and finding a perfect partition or a partition with difference of one between the two bin for uneven n. So in 80% of the runs the algorithms might have found an optimal solution, but the stopping conditions did not trigger as the solutions where not close to a perfect partition.

avg	20	50	100	500	1000	5000	10000	50000
RLS_3^S	523	2517	1237	139	158	226	279	509
RLS_4^S	1291	1976	1935	141	151	203	246	426
(1+1) EA ($3/n$)	554	1849	1218	152	168	210	248	401
(1+1) EA ($4/n$)	382	1107	633	175	190	223	252	359
pmut (1.5)	667	487	164	147	148	163	178	206
pmut (1.75)	465	665	229	135	135	160	179	244

Looking at the time the algorithms needed on average the runs that hit the step limit could have possibly been no failed runs. The easiest are inputs with size $n = 500$. For smaller values of n the algorithms sometimes fail and even in a good run they need more iterations to find an optimal solution. Due to the increasing size of the input the algorithms need more time for the bigger values.

total avg	20	50	100	500	1000	5000	10000	50000
RLS_3^S	82293	41510	9829	139	158	226	279	509
RLS_4^S	80850	40107	9780	141	151	203	246	426
(1+1) EA ($3/n$)	80409	37576	6256	152	168	210	248	401
(1+1) EA ($4/n$)	80375	36609	5502	175	190	223	252	359
pmut (1.5)	80332	35615	4258	147	148	163	178	206
pmut (1.75)	80491	35829	4320	135	135	160	179	244

$pmut_{1.75}$ and $pmut_{1.5}$ are not only the best variant for the bigger values of n but also for smaller inputs as well. It is the least likely to be stuck in a local optimum, and it is also the fastest if it reaches a global optimum.

4.7. Eqiuvalent of linear functions for PARTITION

The input of this section is more or less equivalent to linear functions. All values except the first follow any distribution whereas the first value is the sum of all other values. The optimal solution is therefore the 100. . . 00 or the 011. . . 11 string. So the input is almost identical to a linear function with positive weights that is maximised/minimised depending

on the value of the first bit.
For linear functions the mutation rate of 1/n was proven to be optimal for the (1+1) EA [Wit13]. This should also hold for this input. The RLS variants should also perform worse than the standard RLS. The higher the value for β the better the $pmut_\beta$ mutation should perform. Flips of the first bits could decrease the runtime, depending on how often they happen. By doing some testing with various algorithm variants of the RLS and the (1+1) EA it looked like the last bit was only flipped at most once for every input. There was only one case where it was flipped twice, but it was never flipped more than twice per run. The average number of flips was also mostly closer to zero than to one.
The experiments where conducted with the variant where the smaller values are all one. So for every run of each algorithm the input was $[n-1, 1, 1, \ldots, 1, 1]$. An input like this takes less time for every algorithm, but the results are mostly the same.

4.7.1. RLS Comparison

algo type	RLS	RLS_s^S	RLS_s^S	RLS_s^S	RLS_b^B	RLS_b^B	RLS_b^B
algo param	-	s=2	s=3	s=4	b=3	b=2	b=4
avg mut/change	1.000	1.181	1.688	1.865	3.000	1.998	3.998
avg mut/step	1.000	1.500	2.000	2.500	3.000	2.000	4.000
total avg count	91,068	168,429	235,016	309,492	921,030	921,030	921,030
avg eval count	91,068	168,429	235,016	309,492	-	-	-
max eval count	175,757	357,695	545,716	793,900	-	-	-
min eval count	62,400	110,193	153,263	197,812	-	-	-
fail ratio	0.000	0.000	0.000	0.000	1.000	1.000	1.000
avg fail dif	-	-	-	-	36	53	263

As expected the standard RLS reaches an optimal solution the fastest. It also reaches an optimal value for every instance. The RLS_s^S variants need more iterations to find an optimal solution. By looking at the average values more closely it seems like the average number of steps for the RLS_s^S is roughly $25,000 + 70,000s \pm 5,000$. The standard RLS is equivalent to RLS_k^S or RLS_k^B with $k = 1$. So the value of $k = 1$ seems to be optimal for the RLS variants too. The RLS_b^B variants on the other hand do not reach any of the two optimal solutions in any run. This is most likely caused by their very low possibility of flipping only one bit in a single step. They would eventually reach the optimal solution as well, but this would take much longer than for the RLS. The probability of flipping only one bit in a step is $\mathcal{O}(n^{1-k})$ which results in a single bit flip every $\mathcal{O}(n^{k-1})$ steps in expectation. Because the fitness can only improve for OneMax making steps flipping more bits does not harm the fitness. The bound for OneMax is $\mathcal{O}(n \log n)$ and with the previous result the expected number of steps is bounded by $\mathcal{O}(n \cdot \mathcal{O}(n^{k-1}) \cdot \log(n \cdot \mathcal{O}(n^{k-1}))) = \mathcal{O}(n^{k-1+1} \cdot (k-1+1) \cdot \log(n)) = \mathcal{O}(kn^k \cdot \log(n))$ This problem is not equivalent to OneMax, as a flip of the bit with the highest value inverts the fitness function to ZeroMax but the result might still hold as the bound for the standard RLS for this input is the same as for the RLS on OneMax.

4.7.2. (1+1) EA Comparison

algo type	EA-SM	EA-SM	EA-SM	EA-SM	EA-SM	EA-SM
algo param	-	$2/n$	$3/n$	$4/n$	$5/n$	$10/n$
avg mut/change	1.272	1.751	2.334	2.967	3.638	7.874
avg mut/step	1.000	2.000	3.000	4.000	5.000	10.000
total avg count	231,317	292,626	494,410	867,801	921,030	921,030
avg eval count	231,317	292,626	493,983	815,543	-	-
max eval count	522,203	609,129	913,330	920,992	-	-
min eval count	153,161	189,643	287,830	576,429	-	-
fail ratio	0.000	0.000	0.001	0.495	1.000	1.000
avg fail dif	-	-	1	1	17	569

For this input the same as for OneMax holds. The static mutation rate $p_m = 1/n$ is the optimal value and the performance of the (1+1) EA decreases with rising mutation rate. Only for $p_m \leq 2/n$ the (1+1) EA managed to find one of the two optimal solutions in $10 \cdot n \ln(n)$ steps every time. With mutation rate $p_m = 4/n$ the (1+1) EA only managed to find the optimal solution in about 50 % of the inputs. The remaining mutation rates did not manage to find an optimal solution in any of the runs. Another interesting fact is the average number of bits flipped in a successful step. For the other inputs the overall average number of bits flipped in any step was mostly the same as for the average value of the successful steps. Here this is not the case. All mutation rates flipped fewer bits in the successful steps than in the average step. The only exception is the standard mutation rate which is caused by the steps where the algorithm would flip no bit. Those steps decrease the number of the average case but not of the successful case as those steps were skipped.

4.7.3. pmut Comparison

For *pmut* only 1798/10000 repetitions were executed as there was a clear tendency which of the algorithms performs better.

algo type	pmut	pmut	pmut	pmut	pmut	pmut	pmut	pmut	pmut
algo param	3.25	3.00	2.75	2.50	2.25	2.00	1.75	1.50	1.25
avg mut/change	1.286	1.359	1.459	1.598	1.814	2.152	2.722	3.719	5.197
avg mut/step	1.729	1.935	2.270	2.905	4.359	8.479	22.233	70.692	224.570
total avg count	142,937	153,214	167,454	181,340	206,641	243,262	302,445	422,885	698,772
avg eval count	142,937	153,214	167,454	181,340	206,641	243,262	302,445	422,885	686,374
max eval count	262,484	294,880	315,836	359,104	376,693	431,924	615,506	899,296	920,942
min eval count	97,595	107,234	113,518	126,834	142,210	162,092	205,614	286,108	472,286
fail ratio	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.053
avg fail dif	-	-	-	-	-	-	-	-	1

The results for the *pmut* operator are pretty similar to the results for the (1+1) EA and the RLS. The parameter $\beta = 3.25$ which flips the least bits on average finds the solution the fastest. Decreasing value for β lead to more time needed for finding one of the two optimums. All variants find an optimum in every run except for $\beta = 1.25$ which has a much higher value for the number of flipped bits per steps. The average number of bits flipped in a successful mutation is much lower than for the other inputs especially for the lower values for β. For the binomial and geometric input the successful average was around 100 for $\beta = 1.25$ but for the OneMax equivalent it was only at 5.

4.7.4. Comparison of the best variants

For this comparison neither of the algorithms failed to find one of the two optimal solutions. The following table lists the amount of iterations the algorithms needed to find an optimal solution.

avg	20	50	100	500	1000	5000	10000	50000
RLS	56	187	446	3050	6725	42056	90485	537292
(1+1) EA ($1/n$)	131	438	1059	7435	16963	106410	230310	1379632
pmut (3.25)	82	277	667	4666	10432	66450	142919	846722
pmut (3.5)	77	265	642	4393	9930	62337	137048	801570

The results for this experiment are as expected. The RLS performs better than the (1+1) EA because it does only single bit flips. The $pmut_{3.25}$ performs better than the standard (1+1) EA although flipping more bits on average. This is most likely cause by the few steps where *pmut* flips many bits which increase the average. But *pmut* most likely chooses to flip only one bit more often as the (1+1) EA. In a previous chapter the $\mathcal{O}(n \log n)$ bound was proven for the (1+1) EA and the RLS (Theorem 3.3). This seems to hold in practice at least for the easiest version of this input where the small values are one (see figure 4.23). The scale of this figure is $n \ln n$ which means that a value of 2 on the y-axis means the algorithm needs $2n \ln n$ steps on average to reach the optimal solution. The standard (1+1) EA performs a bit worse than the other three algorithms and approaches $en \ln n$ instead of staying close to $n \ln n$.

Figure 4.23.: Runtime for the OneMax equivalent with a $n \ln(n)$ scale

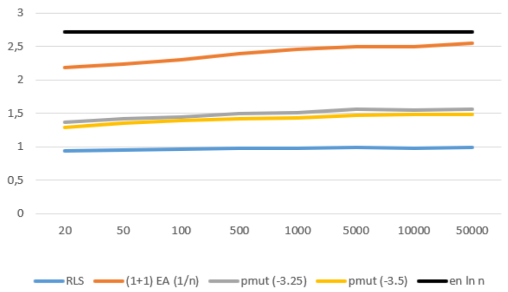
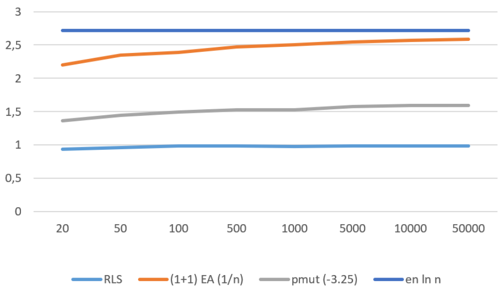


Figure 4.24.: Runtime for the OneMax equivalent with uniform distribution on a $n \ln n$ scale



Another variant of this input are uniform distributed inputs for example. The small values in this case are chosen from $\sim U(1, 49999)$ and the first value again is the sum of all other values. This input is harder because switching multiple small values for a big value increases the fitness but also increases the Hamming distance to the optimum. Looking at figure 4.24 the equivalent to uniform distributed linear functions looks not much harder compared to the variant where all values are one. The graphs are almost identical. It was clear for the RLS because the RLS can't switch elements which leads to the exact same behaviour. But even the other algorithms that are able to switch seem to not harm the hamming distance drastically by switching elements.

Figure 4.25 shows the average number of steps needed by the RLS_k^S for the equivalent to linear functions with uniform distribution. Lemma 3.13 proved the RLS_k^S needs time $4k + \frac{1+o(1)}{1-o(1)} \cdot kn \ln n$ but the actual expected running time seems to be lower. All RLS_k^S variants in the figure approach the value of $kn \ln n$ from below and for at least $n \leq 50,000$ do not need more time than $kn \ln n$. The RLS_k^B variants on the other hand have a significantly worse performance. Their average runtime is shown in figure 4.26 with a logarithmic y-axis. A value of 2 on the x-axis here means that the algorithm needed n^2 steps on average to find an optimal solution. All RLS_k^B variants needs at least n^2 steps on average, but all variants seem to approach n^k from below. The higher the value of k the less clear the asymptotic runtime appears. Testing higher values for $k = 4$ should reveal a better view on the asymptotic runtime but if the asymptotic runtime is actually close to n^4 then it would take $1000^4 = 10^{12}$ steps for $n = 1000$. With a runtime that high it was out of this thesis time budget to test multiple runs of the RLS_4^B with higher input sizes.

Figure 4.25.: Runtime of the RLS_k^S variants for the OneMax equivalent with uniform distribution on a $n \ln n$ scale

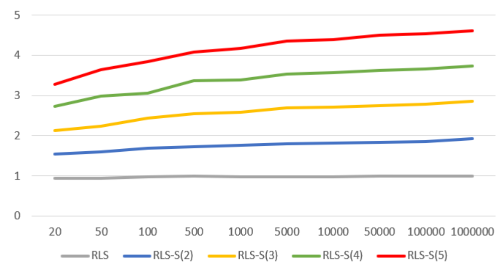
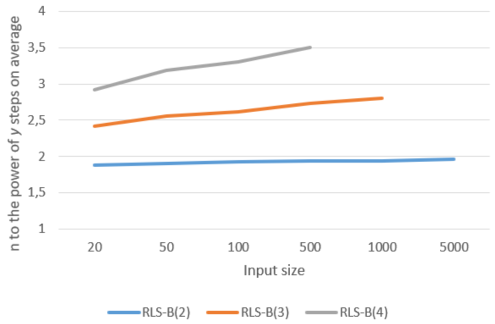


Figure 4.26.: Runtime of the RLS_k^B variants for the OneMax equivalent with uniform distribution on a n^y scale



4.8. Carsten Witt's worst case input

This input is the worst case input from C. Witt in [Wit05] as discussed in the background section. As all experimentally researched inputs in this paper contained only integer values this input is adjusted a bit. To prevent the small values to be below zero they are instead normalised to 1. The two big values are scaled by the same factor of $((1/3 + \epsilon/2)/(n-2))^{-1}$. The higher the value for ϵ the more likely the input is to get stuck in the local optima. With increasing ϵ the local optima becomes less bad. For the small values of ϵ there were only a few cases where some algorithms did not find an optimal solution. To make this effect more visible the value of ϵ was set to $\epsilon = 0.3$.
For $n = 10,000$ this evaluates to $w_1 = w_2 = 5344$ and $W = 9998 \cdot 1 + 2 \cdot 5344 = 20686$. The input then looks like this: $[5344, 5344, 1, 1, \ldots, 1, 1]$. The fitness of the local optimum is $f(x) = 2 \cdot 5433 = 10688$. To leave the local optimum the algorithm therefore has to flip at least $5433 + 9998 - 10688 = 4654$ bits as well in the same step. The best fitness is $f(x) = 5344 + 9998/2 = 10343$, which leads to a difference of $f(localOptimum) - f(opt) = 345$ and a approximation ratio of $f(localOptimum)/f(opt) = 10688/10343 = 1.033$. This is not really close to the worst case of 4/3 any more but with this setting at least many algorithms are stuck in the local optimum at least once for the 10000 runs.

4.8.1. RLS Comparison

algo type	RLS_b^B	RLS_s^S	RLS_b^B	RLS_b^B	RLS_s^S	RLS_s^S	RLS
algo param	b=4	s=4	b=3	b=2	s=3	s=2	-
avg mut/change	3.998	2.367	3.000	1.998	1.964	1.306	1.000
avg mut/step	4.000	2.500	3.000	2.000	2.000	1.500	1.000
total avg count	4,481	4,912	6,353	11,698	15,214	32,204	102,913
avg eval count	4,297	1,510	1,295	6,945	1,794	2,157	2,211
max eval count	77,441	11,617	10,125	80,083	12,221	13,074	12,149
min eval count	0	0	0	0	0	0	0
fail ratio	0.000	0.004	0.006	0.005	0.015	0.033	0.110
avg fail dif	595	345	380	398	345	345	345

The RLS is by far most likely to get stuck in the local optimum. The general tendency is the more bits the algorithm can flip the more unlikely the local optimum becomes. The only case where this does not hold is the RLS_2^B being better than the RLS_3^S, although the RLS_3^S can also flip 3 bits. All RLS_s^B variants had runs where they neither found the global nor one of the two local optima. The algorithms were most likely tricked into the direction of the local optimum and did not manage to leave it. But they were also not fast enough to reach the local optimum because of their low probability to flip only one bit.

4.8.2. (1+1) EA Comparison

algo type	EA-SM	EA-SM	EA-SM	EA-SM	EA-SM	EA-SM	EA-SM	EA
algo param	$100/n$	$50/n$	$10/n$	$5/n$	$4/n$	$3/n$	$2/n$	-
avg mut/change	99.924	49.989	10.038	5.084	4.106	3.143	2.218	1.441
avg mut/step	100.003	50.001	10.000	4.999	4.000	3.000	2.000	1.000
total avg count	69	104	407	801	1,166	2,766	10,495	34,147
avg eval count	69	104	407	801	982	1,294	1,855	3,217
max eval count	697	1,375	5,490	9,451	11,099	11,587	13,921	19,383
min eval count	0	0	0	0	0	0	0	0
fail ratio	0.000	0.000	0.000	0.000	0.000	0.002	0.009	0.034
avg fail dif	-	-	-	-	345	345	345	345

For the (1+1) EA the result is the inversion of the results for the OneMax equivalent. The higher the mutation rate the faster the algorithm reaches a global optimum. This holds at least up to $p_m \leq 100/n$. With mutation rate $p_m \leq 4/n$ the algorithm reaches the worst case at least once in 10000 runs. If the algorithm did not manage to find an optimal solution the fitness was always the same. So there was no run where any algorithm neither found a global nor the local optimum.

4.8.3. pmut Comparison

algo type	pmut	pmut	pmut	pmut	pmut	pmut	pmut	pmut	pmut
algo param	1.25	1.50	1.75	2.00	2.25	2.50	2.75	3.00	3.25
avg mut/change	197.409	70.534	23.050	8.724	4.351	2.777	2.111	1.770	1.563
avg mut/step	224.442	70.480	22.299	8.470	4.368	2.906	2.271	1.934	1.729
total avg count	42	87	216	503	1,094	4,063	10,961	18,727	27,644
avg eval count	42	87	216	503	910	1,488	1,676	1,814	1,909
max eval count	303	867	2,843	6,230	10,487	916,298	501,346	411,742	12,386
min eval count	0	0	0	0	0	0	0	0	0
fail ratio	0.000	0.000	0.000	0.000	0.000	0.003	0.010	0.018	0.028
avg fail dif	-	-	-	-	345	345	345	345	345

For *pmut* the result is the exact same as for the (1+1) EA. The lower β the better the performance as more bits are flipped in each step. For the OneMax input $pmut_{1.25}$ flipped 224 bits on average per step, but the average of the successful steps was only 5. Here the average of all steps is again 224, but the average of the successful steps is at 194. The heavy tail here really increases the performance as most of the high values are accepted. The algorithm is tricked into the local optima only for $\beta \leq 2.25$. If the algorithm is on the path to the local optimum it is always fast enough to reach it within the time limit.

4.8.4. Comparison of the best variants

The $pmut_{1.25}$ and the (1+1) EA with $p_m = 100/n$ perform the best and always find an optimal solution within 700 iterations and even under 100 on the average case. The RLS_4^B performs significantly worse. In the experiment with different input sizes the mutation rate of $p_m = 100/n$ is ≥ 1 for $n \leq 100$. If the algorithm flips every bit then it won't change its solution. In these cases the mutation rate was then set to $p_m = 1/2$.

fails in 1000 runs	20	50	100	500	1000	5000	10000	50000
RLS_4^B	4	0	3	0	0	0	0	0
(1+1) EA ($100/n$)	0	0	0	0	0	0	0	0
pmut (1.25)	0	0	0	0	0	0	0	0

Only the RLS variant had runs where it did not reach a global optimum. This happened in less than 0.5 % of the inputs for $n = 20$ and $n = 100$. For the other input sizes it also managed to reach a global optimum for all inputs.

avg	20	50	100	500	1000	5000	10000	50000
RLS_4^B	12	26	53	221	442	2166	4194	20690
(1+1) EA ($100/n$)	10	16	24	32	34	47	69	222
pmut (1.25)	7	10	11	19	23	35	43	63

For the lower input sizes the RLS is slower than the remaining algorithm even it manages to find a global optimum.

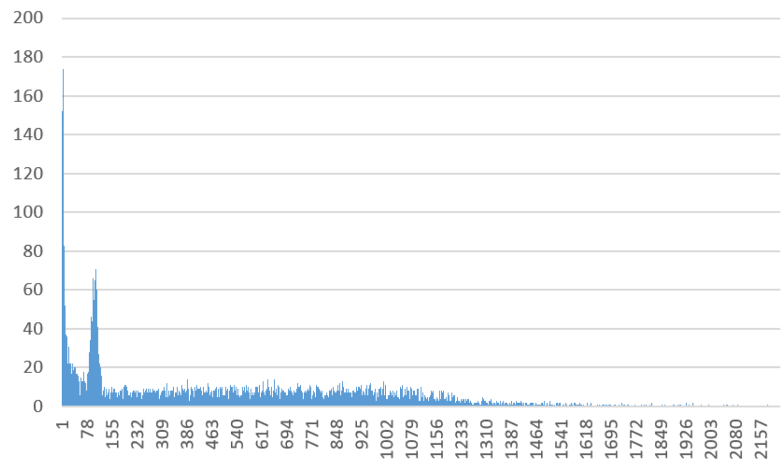
total avg	20	50	100	500	1000	5000	10000	50000
RLS_4^B	412	26	353	221	442	2166	4194	20690
(1+1) EA ($100/n$)	10	16	24	32	34	47	69	222
pmut (1.25)	7	10	11	19	23	35	43	63

The $pmut_{1.25}$ is the best variant closely followed by the (1+1) EA. The RLS version is by far slower than the other to variants for the bigger input sizes. Even for the smaller inputs it is still slower.

4.9. Multiple distributions combined

This subsection covers inputs that do not follow a specific distribution. So instead of sampling from one distribution here instead the value is chosen from a set of distributions. The used distributions were $\sim U(1, 49999)$, $\sim B(10000, 0.1)$, $\sim Geo(0.001)$, $\sim D^{1.25}_{50000}$. Each distribution is chosen with probability 1/8 and with remaining probability 1/2 one value is drawn from every distribution and added together. An input of this distribution is shown in figure 4.27. There are a lot of values close to 0 drawn from the powerlaw (and geometric) distribution. The amount of values then decreases to around 50. From then on the amount of generated numbers rises again until 100, the expected value of the binomial distribution. After the spike caused by the binomial distribution the values look more uniform distributed until 1200 where they fall of again. The higher the numbers get the less often they occur.

Figure 4.27.: Distribution of a mixed and overlapped input with $\sim U(1, 999)$, $\sim B(1000, 0.1)$, $\sim Geo(0.01)$, $\sim D^{1.25}_{1000}$



4.9.1. RLS Comparison

algo type	RLS	RLS_s^S	RLS_s^S	RLS_s^S	RLS_b^B	RLS_b^B	RLS_b^B
algo param	-	s=2	s=3	s=4	b=2	b=3	b=4
avg mut/change	1.000	1.431	1.831	2.214	2.000	3.000	3.999
avg mut/step	1.000	1.500	2.000	2.500	2.000	3.000	4.000
avg eval count	422	589	793	1,004	2,764	16,927	50,810
max eval count	2,293	2,934	4,879	5,722	24,569	207,460	588,295
min eval count	29	28	30	14	10	14	36
fail ratio	0.000	0.000	0.000	0.000	0.000	0.000	0.000

There is a clear preference of algorithms with higher probability to flip only one bit per step similar to the geometric distribution. For geometric distributions all variants had a rather equal runtime but here the RLS_4^B needs more than 100 time the iterations on average than the fastest RLS variant. This type of input punishes the worse mutation operators more than geometric inputs but not as extreme as the equivalent to linear functions. Here still every variant reaches an optimal solution in every case as opposed to the linear function equivalent. For the geometric input only the RLS failed to reach an optimal value in every run but here it does not. This leads to the RLS being the best RLS variants for this type of input in contrast to the geometric inputs.

4.9.2. (1+1) EA Comparison

algo type	EA-SM	EA-SM	EA-SM	EA-SM	EA-SM	EA-SM
algo param	-	$2/n$	$3/n$	$4/n$	$5/n$	$10/n$
avg mut/change	1.481	2.061	2.727	3.458	4.268	9.414
avg mut/step	1.000	2.000	3.001	3.999	5.000	10.000
total avg count	872	965	1,474	2,552	4,785	85,000
avg eval count	872	965	1,474	2,552	4,785	84,916
max eval count	4,201	5,151	8,431	18,891	37,928	856,656
min eval count	43	18	53	7	3	28
fail ratio	0.000	0.000	0.000	0.000	0.000	0.000
avg fail dif	-	-	-	-	-	1

The results here are pretty similar to the results of the RLS. The lower the mutation rates the faster the algorithm reaches an optimal solution. For $p_m \leq 10/n$ every mutation rate managed to find an optimal solution in $10n \ln n$ steps in all 10000 runs.
TODO: delete this or rerun The higher mutation rates had unsuccessful runs and $100/n$ even failed to reach the optimal solution in about 15 % of the inputs.

4.9.3. pmut Comparison

algo type	pmut	pmut	pmut	pmut	pmut	pmut	pmut	pmut	pmut
algo param	3.25	3.00	2.75	2.50	2.25	2.00	1.75	1.50	1.25
avg mut/change	1.583	1.737	2.002	2.423	3.303	5.830	12.519	30.910	73.182
avg mut/step	1.729	1.934	2.274	2.895	4.360	8.452	22.278	70.532	224.421
avg eval count	540	569	594	641	712	808	967	1,285	2,081
max eval count	3,110	2,891	3,504	3,896	5,152	4,274	5,610	6,190	14,984
min eval count	22	9	36	25	28	27	27	13	33
fail ratio	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

Here the results are similar again, but there are a few differences. $\beta = 2.5$ seems to perform better despite not being the highest value for β. Here using a worse mutation rates are has less impact on the runtime in contrast to the geometric inputs where the penalty is higher. Apart from that these inputs seem similar as every value of β also reaches an optimal value in every run.

4.9.4. Comparison of the best variants

The ranking of the algorithm is the same as for the other inputs with a similar preference of low mutation rates. The RLS variant has the best performance closely follow by $pmut_{3.25}$ and lastly be the standard (1+1) EA. For smaller values of n the results are similar too.

fails in 1000 runs	20	50	100	500	1000	5000	10000	50000
RLS	997	945	736	51	2	0	0	0
RLS_2^S	972	625	211	0	0	0	0	0
(1+1) EA ($1/n$)	868	436	146	0	0	0	0	0
(1+1) EA ($2/n$)	757	277	81	0	0	0	0	0
pmut (3.0)	781	364	120	0	0	0	0	0
pmut (3.25)	787	417	128	1	0	0	0	0

The RLS variants perform the worst for $n \leq 100$. For this input it is also rather hard to find a perfect partition for $n \leq 50$. For the lower values there were probability multiple inputs generated with mostly small values and an uneven number of large values. Even if the amount of large numbers is even there might still be no perfect partition, hence no algorithm is able to reach one and the run is treated as a failed run.

avg	20	50	100	500	1000	5000	10000	50000
RLS	60	169	261	425	390	404	436	570
RLS_2^S	412	1889	3552	791	626	591	607	695
(1+1) EA ($1/n$)	19040	15992	9758	1463	883	874	914	1105
(1+1) EA ($2/n$)	20169	16391	8562	1536	1088	1012	1018	1076
pmut (3.0)	26596	17183	8399	1060	606	578	578	691
pmut (3.25)	25871	16012	7669	1229	593	547	553	673

The input gets easier to solve up until $n = 5000$ and from then on gets harder again with increasing input size. The increase for larger input sizes is much smaller than the decrease for the small values.

total avg	20	50	100	500	1000	5000	10000	50000
RLS	99700	94509	73669	5503	589	404	436	570
RLS_2^S	97211	63208	23902	791	626	591	607	695
(1+1) EA ($1/n$)	89313	52619	22933	1463	883	874	914	1105
(1+1) EA ($2/n$)	80601	39550	15969	1536	1088	1012	1018	1076
pmut (3.0)	83924	47328	19391	1060	606	578	578	691
pmut (3.25)	84210	51035	19488	1328	593	547	553	673

This type of inputs are best solved by the (1+1) EA with $p_m = 2/n$ for $n \leq 100$ and also relatively good for $n = 500$. After $n \geq 1000$ the standard RLS becomes the best option and it seems like it stays that way for the remaining input sizes.

4.10. Conclusion of empirical results

There is no clear best algorithm for every input for PARTITION and not even a best parameter for every algorithm. For inputs that are comparable to a linear function/OneMax for all base algorithms the parameters with the lowest mutation rate have the best runtime. Other instances like the worst case input of C. Witt on the other hand require much higher mutation rates for the optimal performance. Inputs generated from a powerlaw distribution showed that the optimal parameter for every algorithm is not even fixed with a specific distribution. For inputs drawn from $\sim D^{2.75}_{50000}$ the higher mutation rates reached an optimum faster than the lower mutation rate for every algorithm variant. If the input was drawn from $\sim D^{1.25}_{50000}$ then the fastest mutation rates for the (1+1) EA on $\sim D^{2.75}_{50000}$ distributed inputs then instead become the slowest. So almost no general advice is possible, but a

Table 4.1.: Best algorithms variants for all researched inputs

	RLS variants			(1+1) EA variants			*pmut* variants		
	1st	2nd	3rd	1st	2nd	3rd	1st	2nd	3rd
binomial	RLS_2^B	RLS_4^B	RLS_2^S	$3/n$	$4/n$	$2/n$	2.25	2.0	1.75
geometric	RLS_2^S	RLS_3^S	RLS_4^S	$2/n$	$1/n$	$3/n$	3.25	3.0	2.5
polwerlaw	RLS_4^S	RLS_3^S	RLS_3^B	$4/n$	$3/n$	$2/n$	1.5	1.75	2.0
uniform	RLS_2^B	RLS_3^S	RLS_4^S	$3/n$	$2/n$	$4/n$	2.5	2.0	2.25
linear function	RLS	RLS_2^S	RLS_3^S	$1/n$	$2/n$	$3/n$	3.5	3.25	3.0
worst case	RLS_4^B	RLS_3^B	RLS_2^B	$100/n$	$50/n$	$10/n$	1.25	1.5	1.75
combined	RLS	RLS_2^S	RLS_3^S	$1/n$	$2/n$	$3/n$	3.25	3.0	2.75

Table 4.2.: Suggestions for the fastest algorithm on each input depending on the input size

<table>
<tr><th></th><th>$n \leq 100$</th><th>$100 < n < 500$</th><th>$500 \leq n < 10,000$</th><th>$10,000 \leq n$</th></tr>
<tr><td>binomial</td><td colspan="2">(1+1) EA $_{3/n}$</td><td colspan="2">RLS_2^B</td></tr>
<tr><td>geometric</td><td colspan="2">(1+1) EA $_{2/n}$</td><td colspan="2">$pmut_{3.25}$</td></tr>
<tr><td>uniform</td><td colspan="2">(1+1) EA $_{4/n}$</td><td colspan="2">RLS_2^B</td></tr>
<tr><td>powerlaw</td><td colspan="4">$pmut_{1.5}$ or $pmut_{1.75}$</td></tr>
<tr><td>linear function</td><td colspan="4">RLS</td></tr>
<tr><td>worst case</td><td colspan="4">$pmut_{1.25}$</td></tr>
<tr><td>combined</td><td>(1+1) EA $_{2/n}$</td><td colspan="2">$pmut_{1.25}$</td><td>RLS</td></tr>
</table>

few points still hold for every input type. The first one is the RLS being most likely to be stuck in a local optimum especially for the smaller input size. Even if a variant of the RLS is the fastet for the bigger input sizes it is most likely to be stuck in a local optimum for $n \leq 100$ for most input types. So if the input size $n \leq 100$ choosing the (1+1) EA or *pmut* mutation operator is a better choice. Another noticeable relation is that inputs that require higher mutation rates are generally easy to solve and are also solved very fast by the lower mutation rates. A lower number of iterations also does not imply a shorter runtime in every case. If the mutation rate $1/n$ needs only a few iterations more than $100/n$ it will still be much faster since one iteration is much shorter. The lower mutation rates are therefore generally a better choice as they will need less time in most cases and are still rather fast if they are not the fastest. Only if the algorithm is trapped due to its low mutation rate a higher mutation rate makes a huge difference.

Now to round this paper up there are two tables that summaries the previous results. For each input and each algorithm the best three variants are listed in Table 4.1 ordered by their average runtime. This implies a general tendency of better algorithms but is not necessarily a complete insight as the best parameter changes depending on n. Table 4.2 list my personal preference based on the previous results depending on the distribution and size of the input.

Bibliography

[BCP01] Christian Borgs, Jennifer Chayes, and Boris Pittel. Phase transition and finite-size scaling for the integer partitioning problem. *Random Structures & Algorithms*, 19(3-4):247–288, 2001.

[Dev06] Luc Devroye. Nonuniform random variate generation. *Handbooks in operations research and management science*, 13:83–121, 2006.

[DJL23] Carola Doerr, Duri Andrea Janett, and Johannes Lengler. Tight runtime bounds for static unary unbiased evolutionary algorithms on linear functions. *arXiv preprint arXiv:2302.12338*, 2023.

[DLMN17] Benjamin Doerr, Huu Phuoc Le, Régis Makhmara, and Ta Duy Nguyen. Fast genetic algorithms. In *Proceedings of the genetic and evolutionary computation conference*, pages 777–784, 2017.

[Fer05] Henning Fernau. *Parameterized algorithmics: A graph-theoretic approach*. PhD thesis, Citeseer, 2005.

[FGQW18] Tobias Friedrich, Andreas Göbel, Francesco Quinzan, and Markus Wagner. Evolutionary algorithms and submodular functions: Benefits of heavy-tailed mutations. *arXiv preprint arXiv:1805.10902*, 2018.

[Gra66] Ronald L Graham. Bounds for certain multiprocessing anomalies. *Bell system technical journal*, 45(9):1563–1581, 1966.

[Gun05] Christian Gunia. 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*, pages 571–578, 2005.

[Hay02] Brian Hayes. Computing science: The easiest hard problem. *American Scientist*, 90(2):113–117, 2002.

[KMPS03] Hans Kellerer, Renata Mansini, Ulrich Pferschy, and Maria Grazia Speranza. An efficient fully polynomial approximation scheme for the subset-sum problem. *Journal of Computer and System Sciences*, 66(2):349–370, 2003.

[Kor98] Richard E Korf. A complete anytime algorithm for number partitioning. *Artificial Intelligence*, 106(2):181–203, 1998.

[Kor09] Richard Earl Korf. Multi-way number partitioning. In *Twenty-First International Joint Conference on Artificial Intelligence*, 2009.

[SN12] Andrew M Sutton and Frank Neumann. A parameterized runtime analysis of simple evolutionary algorithms for makespan scheduling. In *International Conference on Parallel Problem Solving from Nature*, pages 52–61. Springer, 2012.

[Wit05] Carsten Witt. Worst-case and average-case approximations by simple randomized search heuristics. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 44–56. Springer, 2005.

[Wit13] Carsten Witt. Tight bounds on the optimization time of a randomized search heuristic on linear functions. *Combinatorics, Probability and Computing*, 22(2):294–318, 2013.

[Wit14] Carsten Witt. Fitness levels with tail bounds for the analysis of randomized search heuristics. *Information Processing Letters*, 114(1-2):38–41, 2014.

Appendix

A. Appendix Section 1

ein Bild

Figure A.1.: A figure