Internal Report CS Bioinformatics Track 10-01 May 2010

Universiteit Leiden

Opleiding Informatica

Self-adaptive Mutation in Molecular Evolution

Alexander Aleman

MASTER'S THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Self-adaptive Mutation in Molecular Evolution

Alexander Aleman

ABSTRACT. This study is focused on an evolutionary algorithm for automated *de novo* drug design. It discusses how to improve the mutation operator of such an algorithm. By introducing a weighted mutation scheme it tries to improve the performance of the algorithm as well as facilitating for the use of self-adaptation. The two questions which this study tries to answer are: Does introduction of different weights for each type of mutation make a difference in the performance of the algorithm? Can self-adaptation help choosing good parameters for the algorithm and maybe improve the performance?

The proposed methodology is evaluated in a case-study on automated drug design where we aim to find molecular structures that could serve as inhibitors of Lipoxygenase, which is involved in the metabolism of fatty acids (and thus simply the fats we are eating).

Contents

CHAPTER 1

Introduction

Automated *de novo* discovery of molecules that can serve as drug candidates is a challenging field of science. Problems in molecular drug design are typically multi-objective (e.g. maximization of effects and minimization of undesirable side effects). Also, they usually consist of many soft and/or fuzzy constraints (predicting the feasibility and usability as drugs) which make it problematic to define a sharp boundary between feasible and infeasible solutions. Both interactive ([**12**]) and automated ([**10**], [**11**], [**17**], [**21**], [**9**]) approaches have been proposed.

A major bottleneck of automated molecular design is that it is difficult for experts to provide accurate objective- and constraint-functions which can be used for the automated search. On the one hand it is very difficult to determine the activity of candidate molecular structures on certain targeted cells which leads to the use of inaccurate approximation models for activity prediction. On the other hand, methods are missing to accurately determine the feasibility of candidate molecules as drugs (e.g. solubility in water/ blood). For both, the 'chemists eye' and real-world experiments are still necessary. Nonetheless it is important to keep improving the automated techniques in their ability to find a good solution with the given objective- and constraint-functions.

In this study an evolutionary algorithm that automates the design of molecules, as proposed by Kruisselbrink et al. ([**10**], [**11**]), will be considered. In particular, this study will focus on the effects of the mutation operator.

The mutation operator is a tool to explore the search space in this algorithm. The mutation operator does not make a difference between large and small mutations. A way to include this difference between large and small mutations would be to introduce different weights for each type of mutation. An interesting question that this study tries to answer is if introducing these different weights for each type of mutation makes a difference in the performance of the algorithm.

When using heuristic search methods such as an evolutionary algorithm, a major problem is to find good settings for the parameters of the algorithm. In particular, when the user is a non expert in algorithm design. Self-adaptation provides a way to learn good parameter settings automatically during the algorithm run. Self-adaptation has been successfully applied in some cases (e.g. [**20**]), but it is not clear beforehand that this will also have positive effects on an arbitrary evolutionary algorithm in a non-standard search space such as chemical space. Implementation of self-adaptation is relatively difficult in our case, because the mutation operator is graph based and there is no obvious notion of distance or neighbourhoods. In this study we will try to answer

the question if self-adaptation can help choosing good parameters for the algorithm and maybe improve the performance.

This report is structured as follows: In Chapter 2 the general framework of evolutionary algorithms and self-adaptation will be presented in a problem independent way. In Chapter 3 the problem of molecular design will be discussed, such as representation, mutation and evaluation. In Chapter 4 an approach will be proposed to incorporate self-adaptation in the scope of molecule evolution. In Chapter 5 the approach will be validated by means of an experimental study on a test-case in molecular design. Chapter 6 finalizes this report by summarizing the results and suggesting future work.

CHAPTER 2

Evolutionary Algorithms

Evolutionary algorithms are algorithms in the field of artificial intelligence, that have been used in the past to tackle a variety of optimization problems. They use an iterative process, learned from evolution in nature, in their search to keep optimizing an existing population. This population is being changed and improved until certain requirements, such as constraints or maximum time spent, are met. Such processes are inspired by biological mechanisms of evolution, such as mutation, recombination and survival of the fittest. Candidate solutions to the optimization problem play the role of individuals in a population, and the fitness function, which determines how well a solution (individual) performs, is an analogy of biological fitness in the sense that it determines the chance to survive and reproduce. In contrast to evolutionary biology, where fitness is measured by the reproduction rate of an individual, in evolutionary algorithms the fitness is determined by the quality of the individuals with respect to the objective function(s).

Evolution of the population takes place by iteratively applying the mentioned mechanisms of evolution. In this process, there are two main forces that form the basis of evolutionary systems: Recombination and mutation create the necessary diversity and thereby facilitate novelty, while survival and reproduction act as forces which increase quality. Many aspects of such an evolutionary process are stochastic, i.e. changed pieces of information due to recombination and mutation are randomly chosen. While selection operators can be either stochastic, individuals with a higher fitness value have a higher chance to survive and reproduce than individuals with a lower fitness, or deterministic, only the individuals with the highest fitness values will survive and reproduce.

In a typical evolutionary algorithm the objective is to find an object $x \in S$ that optimizes $f : S \to \mathbb{R}$, where S is the search space of all possible candidate solutions. A population of candidate solutions (individuals) is then slowly evolved to try and reach this optimal solution. Individuals consist of three parts: the object x that needs to be optimized, the so-called *decision parameters* or *object variables*, the *fitness* of the individual $f(x)$, reflecting the quality with respect to the objective function(s), and (optionally) the *endogenous* strategy parameters, such as mutation strength, which is of particular importance in this study in the context of *self-adaptation* (see Chapter 2.1). Parameters that describe population size and number of offspring are called *exogenous* strategy parameters, as they are kept constant and are not dependent on an individual.

In this research we will adopt the generational loop of two general evolutionary schemes: the

(μ, λ)-strategy (Algorithm 1) and the $(\mu + \lambda)$-strategy (Algorithm 2), both with a deterministic selection operator. Here μ defines the size of the parent population and λ defines the number of offspring that is generated via recombination and/or mutation at each iteration of the algorithm. The difference between the two algorithms can be found in the selection operator, in the (μ, λ)-strategy the parent population is not considered in the selection, so only the offspring have a chance to survive and become the new parent population. While in the $(\mu + \lambda)$-strategy the parents have as much chance to survive and reproduce again as the offspring. The advantage of the $(\mu + \lambda)$-strategy is that good individuals aren't replaced by worse ones. But the disadvantage is that when there is no individual within the range of one mutation with a higher fitness score the $(\mu + \lambda)$-strategy gets stuck in this region of the search space, while there could be regions in the search space with higher fitness values. So the $(\mu + \lambda)$-strategy is prone to getting stuck in a *local* optimum. The (μ, λ)-strategy is able to leave such local optima and search for a better local or even *global* optimum, at the risk of losing good solutions.

Algorithm 1 General (μ, λ)-evolution

1: $t = 0$
2: Initialize parent population P_t
3: Evaluate P_t
4: **while** not terminate **do**
5: Generate λ offspring $Q_t = \{q_1, \ldots, q_\lambda\}$ from P_t
6: Evaluate Q_t
7: $P_{t+1} =$ The μ best individuals in Q_t
8: $t = t + 1$
9: **end while**

Algorithm 2 General $(\mu + \lambda)$-evolution

1: $t = 0$
2: Initialize parent population P_t
3: Evaluate P_t
4: **while** not terminate **do**
5: Generate λ offspring $Q_t = \{q_1, \ldots, q_\lambda\}$ from P_t
6: Evaluate Q_t
7: $P_{t+1} =$ The μ best individuals in $Q_t \cup P_t$
8: $t = t + 1$
9: **end while**

2.1. Self-Adaptation

The basic idea of self-adaptation is to impose the evolutionary process not only to the decision parameters, but also to the endogenous strategy parameters, in particular the strength/impact of the mutations. The idea of self-adaptation originates from nature, where self-repair mechanisms exist, such as *repair enzymes* and *mutator genes*. In this way an individual can adapt the endogenous strategy parameters to the changing landscape, while the evolution progresses. Hence, there is no deterministic control of the mutation strategy for the user. With the use of self-adaptation, the link between the strategy and decision parameters can be exploited. Moreover, self-adaptation can help to improve the results of an evolutionary algorithm [**20**]. Self-adaptation is also a tool that can reduce the number of parameters that has to be set by the user of the algorithm.

CHAPTER 3

Molecule Evolution

Having described the basic technique used in this research, this chapter will describe how these are applied in the context of automated *de novo* drug design. Automated *de novo* drug design deals with the problem of finding new molecules that possess a number of wanted molecule properties which make them promising candidates to be used as drugs for a certain targeted disease. It consists of two parts: In the first place, the molecular structures should exhibit the desired behavior on the targeted cells. Secondly, the molecular structures should possess other pharmacological and biological properties which make them usable as drug (i.e. they should be drug-like [**13**]).

It is difficult to apply mathematical programming techniques in molecular design as the search space is discrete and has a irregular neighborhood structure that makes it infeasible to map it on a integer programming problem. Moreover, evaluation is partly done by software components of which the details are hidden to the user and thus also to the algorithm. On the other hand, evolutionary algorithms have often been used to heuristically solve problems of graph structure optimization with black box evaluation software. However, a major problem is that it is very difficult to compute the quality of candidate molecules. For both the activity on the targeted cells (objectives) as the determination of the drug-likeness (constraints) of candidate molecular structures it is only possible to estimate/approximate the performance.

The basis for this study will be the algorithm (Algorithm 3) as used in the automated version of the Molecule Evoluator ([**10**], [**11**]). The population of μ molecules is initialized (see Chapter 3.2) and then the main loop starts. Each generation offspring is created by randomly selecting a parent λ times and applying mutations (see Chapter 3.2) to these parents. The offspring population is then evaluated using the fitness function (see Chapter 3.3, 3.3.1 and 3.3.2). Depending on the chosen scheme, the μ best ranked molecules, according to their fitness score, are selected from the offspring population Q_t (in case of (μ, λ)-selection) or the offspring and parent population $Q_t \cup P_t$ (in case of $(\mu+\lambda)$-selection). These selected molecules then form the new parent population P_t.

3.1. Representation

The choice of representation of individuals in an Evolutionary Algorithm is of great importance, since it defines the neighborhood landscape. Moreover, the complexity of the fitness function as well as the mutation operators depend on the chosen representation. In this study the graph-like

Algorithm 3 $(\mu^+\lambda)$ molecule evolution

```
t = 0
Initialize parent population P_t of size μ
Evaluate P_t
while not terminate do
    for i = 1 to λ do
        Randomly select a molecule p from P_t
        Randomly select n mutation(s) and apply this to p
        Add this new molecule to the offspring population Q_t
    end for
    Evaluate Q_t
    P_{t+1} = The μ best individuals in Q_t(∪P_t)
    t = t + 1
end while
```

representation of the Molecule Evoluator is adopted. Due to the structure of molecules (see Figure 3.1.1), molecules consist of *atoms* which are 'connected' by *bonds*, graphs seem like good candidates to represent them.

A graph G is an ordered pair $G = (V, E)$, where V is the collection of *vertices* and E the

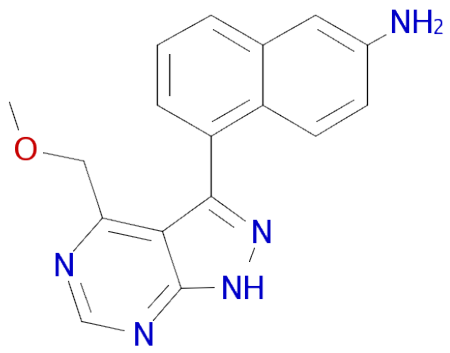


FIGURE 3.1.1. Graph representation of a molecule

collection of *edges* (the connection between two vertices). An example of such a graph is shown

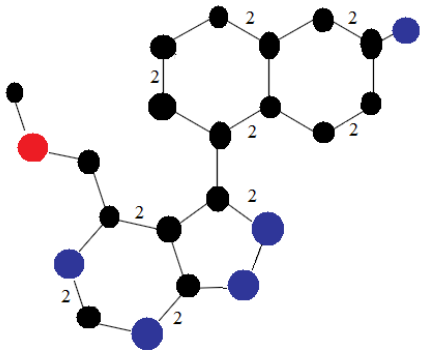


FIGURE 3.1.2. A conversion of Figure 3.1.1 to an edge-weighted labeled graph

in Figure 3.1.2. To apply this concept to molecules each vertex can represent an atom, while each edge could represent a bond. Since molecules are a special kind of graph with certain constraints an expansion of the graph concept is needed. First of all some atoms have multiple bonds between two atoms, therefore the edges are given different weights depending on the number of bonds, a so-called *edge-weighted graph*. Secondly atoms in molecules have many different types (carbon, hydrogen, etc.), therefore we need a label for every vertex, a so-called *labeled* graph. Combining these two types we get the *edge-weighted labeled graph* as representation of molecules. Important to note is that only when certain constraints are satisfied an edge-weighted labeled graph represents a molecule.

3.2. Mutation/Recombination

The design of evolutionary algorithms for the evolution of molecular structures requires the design of sets of mutation operators which are complete (i.e. with which it should be possible to reach any part of the search space) and with which neighborhoods of molecular structures can be described. The mutation operators that are used in this research are adopted from [**12**] and [**10**] which are shown in Table 1, although with some slight modifications: For the underlying representation of the molecular structures a graph-based representation is used instead of the TreeSmiles-notation used in [**10**] and [**11**]. For the *AddGroup* mutation different fragments were used. The set of fragments was constructed using some large molecular databases (DrugBank [**24**], HMDB [**25**], and ZINC [**8**]) from which the most frequent ring-systems and linkers were extracted. The top 150 organic ring-systems and linkers were used as fragments for the *AddGroup* mutation operator. These fragments are also used to initialize the first population (i.e. molecular

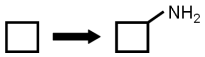
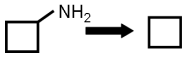
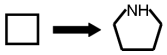
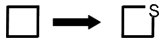
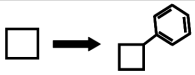
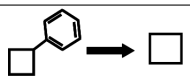
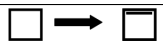
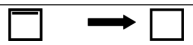
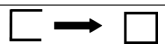
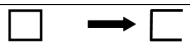
Mutation	Example
Add atom	
Remove atom	
Insert atom	
Uninsert atom	
Mutate atom	
Add group	
Remove group	
Increase bond order	
Decrease bond order	
Make ring	
Break ring	

TABLE 1. Mutation operators of the Molecule Evoluator

structures are generated from scratch by combining and mutating fragments from the linker- and ring-system-set e.g. Figure 3.1.1). The currently implemented mutation operator, as seen in Algorithm 3, works by applying a fixed number of mutations n (e.g. 1 or 2). These mutations are chosen randomly, with uniformly distributed probability, from this list of possible mutations, as shown in Table 1.

For the sake of completeness we will also introduce the recombination operator. The implementation of the recombination operator is inspired by the subtree crossover as used in genetic programming. However, as molecules are represented as graphs and not trees, the recombination operator is restricted to be only applied to subtrees which are connected at exactly one point to the base molecule. The recombination operator will not be used in this study to focus on mutation.

3.3. Fitness Evaluation

In this study, the problem of optimizing a molecule towards desired drug properties is transformed into a single-objective optimization problem with constraints. The biological activity serves

as an objective function, while the constraints provide a way to distinguish molecules in their ability to be used as drugs.

Support vector machine models (SVMs) are used for the prediction of the biological activity based on molecular similarity [**1**] to a set of steroidal as well as non-steroidal reference compounds. A support vector machine is a type of classifier that works in a transformed feature space defined by nonlinear transformations of the original variables [**23**]. They map these features as vectors in a high-dimensional feature space. In this space a 'best' separating hyperplane (the *maximal margin* hyperplane) is constructed (e.g. Figure 3.3.1).

In the case of molecule design this hyperplane is used to predict the biological activity of the

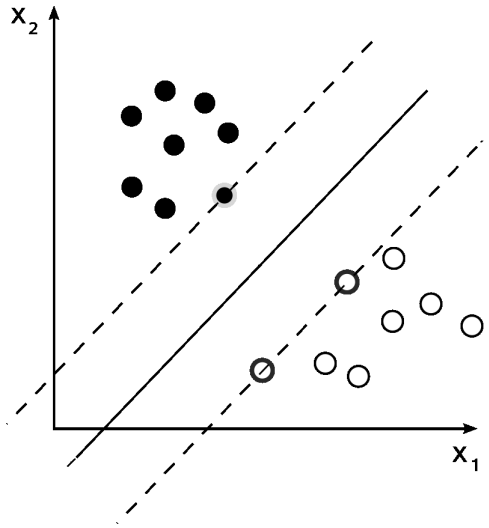


FIGURE 3.3.1. Maximal margin hyperplane for a trained SVM separating samples from two classes in the x_1-x_2-feature space

molecule, by reflecting the desirable properties regarding activity. As SVMs cannot work with the molecule structures directly, features are extracted from the molecules which are used for the training and prediction. For each test-case three support vector machines are used which predict the activity of the molecules based on different types of features (ECFP6 fingerprints, MDL keys and AlogP counts [**7**]). Similar to [**11**], the activity scores of the three are (if needed) mapped to a value in the interval [0,1] and aggregated into one function. This is the first part of the objective function which needs to be maximized:

$$f_{objectives} = f_{ECFP6} \cdot f_{MDL} \cdot f_{AlogP}. \tag{3.3.1}$$

3.3.1. Constraints. For the constraints, Lipinski's rule-of-five [**15**] is used to predict the drug-likeness of the molecules. This rule consists of simple boundary values for five easy calculable molecular properties which can be used to estimate the drug-likeness of molecules. Besides this, a maximal bound is set for the estimated minimal energy conformation which estimates the stability of the molecular structures. As the boundary values of both Lipinski's rule-of-five and the minimal energy conformation are vague, these *fuzzy constraints* are modeled by *desirability functions* (DFs) [**11**].

Desirability functions were first introduced by Harrington [**6**] to model a number of quality criteria. In order to allow for a better comparison between the differently scaled quality criteria with respect to their desired levels Harrington proposed to map the quality criteria to the open unit interval $(0, 1)$, where a value close to zero stands for 'poor quality', whereas a value close to 1 stands for 'high quality'. Derringer [**4**] then extended this work by incorporating the value 0, when a constraint is strictly violated and thus the desirability should be minimal, and the value 1, when a value completely satisfies a constraint. Given a set of objective variables $X = x_1, \ldots, x_n$ and a set of quality criteria $Y = y_1, \ldots, y_m$, where $y_i = f_i(X)$, Derringer desirability functions have a signature:

$$d_i(y_i) : \mathbb{R} \to [0, 1], i = 1, \ldots, m$$

We will use these desirability functions to model the fuzzy constraints, as done by Kruisselbrink et al. [**11**]. To do this we define the relation $A \lesssim B$, which means A is essentially smaller than B. There are three types of fuzzy constraints that we will describe using this relation:

$$\begin{aligned} LB_j < A_j \lesssim g_j(x) \\ g_j(x) \lesssim B_j < UB_j \\ LB_j < A_j \lesssim g_j(x) \lesssim B_j < UB_j \end{aligned} \tag{3.3.2}$$

Here, LB_j and UB_j denote the absolute lower and higher cutoff bounds beyond which the constraints are absolutely not satisfied, the regions (LB_j, A_j) and (B_j, UB_j) denote the gray areas. At the first type of constraints the region $[A_j, \infty)$, at the second type the region $(-\infty, B_j]$ and at the third type the region $[A_j, B_j]$ denote the area where the constraints are absolutely satisfied. The first two constraints are so-called *one-sided constraints* and the third one is a so-called *two-sided constraint*.

For each constraint j a desirability function $\hat{g}_j$ is constructed which maps the original value of the molecule property g_j to the interval $[0, 1]$. The first type of (3.3.2) can be modeled by means

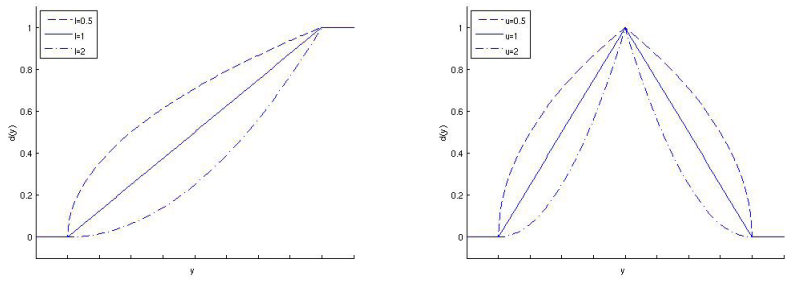
of a desirability function in the following way:

$$\hat{g}_j(x) = \begin{cases} 0 & , g_j(x) \leq LB_j \\ \left(\dfrac{g_j(x) - LB_j}{A_j - LB_j} \right)^{l_j} & , LB_j < g_j(x) < A_j \\ 1 & , g_j(x) \geq A_j \end{cases} \tag{3.3.3}$$

Here, l_j describes the shape of the function in the gray area. In the case of $l_j = 1$ the shape will be linear, in the case of $l_j < 1$ the function is relatively mild (easier to get close to 1) for values in the gray area, while in the case of $l_j > 1$ the function is more strict. An example of this function can be seen in Figure 3.3.2(a). The second type of (3.3.2) can be modeled similar to (3.3.3):

$$\hat{g}_j(x) = \begin{cases} 1 & , g_j(x) \leq B_j \\ \left(\dfrac{g_j(x) - UB_j}{B_j - UB_j} \right)^{u_j} & , B_j < g_j(x) < UB_j \\ 0 & , g_j(x) \geq UB_j \end{cases} \tag{3.3.4}$$

Here, u_j describes the shape of the function in the gray area in the same way as l_j. This function would look like a mirrored (in the y-axis) version of Figure 3.3.2(a). The third type of (3.3.2) is



(a) One sided desirability function (b) Two sided desirability function

FIGURE 3.3.2. Derringer desirability functions

the modeled by combining (3.3.3) and (3.3.4):

$$\hat{g}_j(x) = \begin{cases} 0 & , g_j(x) \leq LB_j \\ \left(\frac{g_j(x) - LB_j}{A_j - LB_j}\right)^{l_j} & , LB_j < g_j(x) < A_j \\ 1 & , A_j \leq g_j(x) \leq B_j \\ \left(\frac{g_j(x) - UB_j}{B_j - UB_j}\right)^{u_j} & , B_j < g_j(x) < UB_j \\ 0 & , g_j(x) \geq UB_j \end{cases} \tag{3.3.5}$$

A simplified example of this can be seen in Figure 3.3.2(b), where $A_j = B_j$ and $l_j = u_j$.

3.3.2. Aggregation. The objective function (3.3.1) and the constraints can now be combined by means of aggregation. All m desirability functions are aggregated into one (our second) part of the objective function similar to (3.3.1) which also needs to be maximized:

$$f_{constraints} = \prod_{i=1}^{m} \hat{g}_i \tag{3.3.6}$$

Now (3.3.1) and (3.3.6) can be aggregated into one function:

$$f_{combined} = f_{objectives} \cdot f_{constraints} = f_{ECFP6} \cdot f_{MDL} \cdot f_{AlogP} \cdot \prod_{i=1}^{m} \hat{g}_i. \tag{3.3.7}$$

This type of aggregation has a particular interesting quality: When one of the constraints is strictly violated the value of $f_{constraints}$ will be 0 and thus the value of $f_{combined}$ will be 0. While in the case that one of the constraints is in the gray area $f_{combined}$ will only be penalized, but not totally rejected. The goal of the algorithm will from now on be to maximize $f_{combined}$.

CHAPTER 4

Scalable Mutation in Chemical Space

In this study we wish to apply self-adaptation to our algorithm (see Chapter 2.1). Self-adaptation could be helpful in our case, because it is not quite clear beforehand what the mutation strength of the evolution should be. On the one hand it is important to get a high *progress rate*, the rate at which the algorithm approaches the optimum. On the other hand it is important to keep the harmful effects of mutation within reasonable bounds. Self-adaptation can be applied to try and tackle these issues.

The problem when trying to apply this to the chemical space arises when trying to think of a reasonable way to scale the mutation. Some of the mutations have a rather big structural effect on the molecule, while other mutations only change it slightly. The mutation operators are graph based and therefor difficult to scale by some real valued factor. If we would simply change the number of mutations that are applied to a molecule we could have the case that one mutation is giving more changes to the molecule then two mutations if in the second case two relatively minor changes are made, while in the first case one massive change was made. To tackle this issue we introduce the concepts of mutation impact and mutation budget, as derived from [**5**].

4.1. Mutation Impact

In applied genetics it is common to speak of mutations as either harmful or beneficial. A harmful mutation is a mutation that decreases the fitness of the organism. A beneficial mutation is a mutation that increases fitness of the organism, or which promotes traits that are desirable. Analysis of the effect on the fitness score that mutations have, could give us insight into the impact a specific mutation has in the evolution. This effect we will call mutation impact. Since it is impossible to predict the mutation impact of a specific mutation with a given, support vector machine based fitness score, we will try to estimate this by means of random walks through the molecule space as shown in Algorithm 4.

A random walk in the molecule space is a path of molecules that consists of taking successive random steps (mutations). Given a starting point in the molecule space, a randomly created molecule, we can randomly, with evenly distributed probability, choose one of the available mutations and apply this to the current molecule. If we keep applying random mutations to the last created molecule and evaluate the fitness of the molecule at each step, we can estimate the effect of a given mutation by calculating the difference between the fitness score before and after applying

Algorithm 4 Random walk in molecule space

```
t = 0
Initialize molecule path P = ∅
Initialize starting molecule rw_0
P = P ∪ {rw_0}
while t < T_max do
    rw_{t+1} = Mutate(rw_t)
    P = P ∪ {rw_{t+1}}
    t = t + 1
end while
```

this mutation. If we do multiple random walks, we can get an estimate for the mutation impact by averaging these effects for each mutation respectively.

For a given fitness function on the molecule space M,

$$f : M \to \mathbb{R}$$

the mutation impact MI_j, for a given mutation j, can now be estimated with

$$MI_j = \frac{1}{\#A_j} \sum_{i \in A_j} |f(rw_i) - f(rw_{i-1})|, \tag{4.1.1}$$

where

$$rw_i = \text{the obtained molecule after } i \text{ steps in the random walk}$$
$$A_j = \{i \in \mathbb{N} | rw_i \text{ was formed after applying mutation } j \text{ on } rw_{i-1}\}.$$

In cases where there is not enough time to learn the mutation impact by performing random walks they could also be learned during the evolution loop by using a cumulative approach. Given a fixed starting mutation impact for mutation j, the mutation impact could be updated each generation by applying:

$$MI_j^{new} = c \cdot MI_j^{old} + (1 - c) \cdot MI_j^{current}, \tag{4.1.2}$$

with $c \in (0, 1)$. MI_j^{old} represents the mutation impact from the previous generation, $MI_j^{current}$ represents the average impact mutation j had in this generation and MI_j^{new} will be the new value for the mutation impact of mutation j. In case mutation j wasn't applied this generation the impact should not be updated. Also β should be much larger then δ to prevent too much fluctuation. A disadvantage of this method is that it is necessary to store information about the applied mutation and fitness of the parent, also in case multiple mutations were performed it is difficult to decide which mutation had what impact. Furthermore finding a good starting value for the mutation impacts can be hard without an in-depth knowledge of the fitness landscape. Due

to these disadvantages in this study the approach of random walks is chosen for determining the mutation impacts.

4.2. Mutation Budget

Now that we have introduced the concept of mutation impact, we can apply this to introduce scalable mutation to the molecule evoluator. To incorporate this with the molecule evolution approach of [**10**] we introduce the idea of a mutation budget. The mutation budget can be compared to an amount of money that is available to be spent. In the case of the molecule evoluator this is the amount of "money" that can be spent, by the algorithm, on mutations. For each molecule there is a separate budget, each mutation has a different price (mutation impact) and the total spendings on a molecule should not exceed the budget. The only mutations that are considered are the mutations that are, according to their respective mutation impacts, within the budget. With equal probability one of the considered mutations will be applied. Subsequently the cost of this mutation, the mutation impact, will be subtracted from the remaining budget and this process continues until there are no more mutations that have a lower cost than the remaining budget. The total budget can in this way be used to vary the mutation strength, by either increasing or decreasing this budget.

To do this we define an individual of the population of molecules as an ordered pair $I = (X, \sigma)$, where X is a molecule (decision parameter) and σ its mutation budget (strategy parameter). To apply self-adaptation we must not only use mutation on the decision parameter, in our case the molecule X, but also on the strategy parameter σ. Moreover, the strategy parameter should be adapted before the molecule to be able to learn, by selection of individuals based on objective function, from this change in the strategy parameter. The mutation step size a two point operator, as described in [**18**], with a learning rate $\alpha = 1.1$ has been chosen:

$$\sigma_{adapted} = \alpha^{\xi} \cdot \sigma, \tag{4.2.1}$$

where α is the mutation strength of the strategy parameter and ξ a uniformly distributed random variable on $\{-1, 1\}$. This means that σ is either multiplied or divided by a factor α, depending on the random variate of the random variable ξ. According to Beyer [**2**] the value for α can stay fixed for the starting phase (first 1000 generations) of the algorithm, although in contrast to Beyer the value is not set equal to 1.3. This is due to the fact that the mutation impacts (see Figure 5.2.1 and Chapter 5.2) are relatively close together and therefore a large α would ignore many intermediate values of the budget.

One of the main advantages of this adaptation scheme is that it is unbiased. Therefore a tendency to increase or decrease the budget is exclusively caused by the algorithm itself and not by the adaptation scheme. To make sure that the mutation doesn't stop altogether σ has a lower bound of σ_{min}, which should be a big enough budget to allow for at least one smaller mutation. A rough outline of this scheme is shown in Algorithm 5, where MI_j is the impact of a mutation j as

seen in equation (4.1.1). Depending on whether we want to use a $(\mu + \lambda)$- or a (μ, λ)-strategy, the parent population P_t is considered in the selection.

Algorithm 5 $(\mu^{+}_{,}\lambda)$-evolution with mutation budget

$t = 0$
Initialize parent population P_0
while $t < T_{max}$ **do**
    for $i = 1$ to λ **do**
        Select a random parent individual $p = (X_p, \sigma_p) \in P_t$
        $\xi =$ random $\in \{-1, 1\}$
        $\sigma_i = \max(\alpha^{\xi} \cdot \sigma_p, \sigma_{min})$
        budget $B = \sigma_i$
        $X_i = X_p$
        while $\{k | MI_k \leq B\} \neq \emptyset$ **do**
            Randomly select a mutation $j \in \{k | MI_k \leq B\}$
            $X_i = X_i \circ j$
            $B = B - MI_j$
        end while
        Add individual $q_i = (X_i, \sigma_i)$ to the offspring population Q_t
    end for
    Evaluate Q_t and P_t
    $P_{t+1} =$ The μ best individuals in $Q_t (\cup P_t)$
    $t = t + 1$
end while

CHAPTER 5

Experiments/Results

To validate the proposed method we use the following test-case: The target will be to find inhibitors of the so-called LOX, or Lipoxygenase, which is involved in the metabolism of fatty acids (and thus simply the fats we are eating). It catalyzes the following reaction:

$$\text{fatty acid } + O_2 = \text{fatty acid hydro-peroxide}$$

Lipoxygenases are found in plants, animals and fungi. Products of lipoxygenases are involved in diverse cell functions [**16**].

The goal of the algorithm is to maximize $f_{combined}$ (see equation (3.3.7)), which will maximize the predicted biological activity, while making sure the candidate molecules satisfy certain fuzzy constraints. For the boundaries of the fuzzy constraints (as described in Chapter 3.3.1), the settings of Table 1 are used.

Descriptor	LB	A	B	UB
Num H-acceptors	0	1	6	10
Num H-donors	0	1	3	5
Molecular solubility	-6	-4	∞	∞
Molecular weight	150	250	450	600
ALogP	0	1	4	5
Minimized energy	0	0	80	150

TABLE 1. The settings of the boundary values for the properties that are used as fuzzy constraints in the automated molecular search.

5.1. Experimental Setup

The first step is to estimate the mutation impacts, as described in Chapter 4.1, 500 random walks (Algorithm 4) with $T_{max} = 100$ steps were conducted. For each mutation this gives approximately 5000 (the mutations in the random walk were chosen with equal probability) estimations.

After the mutation impacts are estimated we can incorporate this in the mutation budget algorithm (Algorithm 5). To test the self-adaptation mechanism it is compared with the same algorithm but with a fixed budget. These fixed budgets vary, depending on the mutation impacts, from the budget needed for one smaller mutation up until three big mutations. Both the $(\mu + \lambda)$- and the (μ, λ)-strategies are considered and compared. Each test case is run three times with $T_{max} = 200$. The size of the parent population will be $\mu = 15$ and the number of offspring will be $\lambda = 100$.

5.2. Results

The first step in testing the mutation impact/budget concept was to estimate the mutation impact of the different mutations. This was done by conducting the random walks as described in Chapter 4.1. This yielded, for the mean mutation impacts, the results as shown in Figure 5.2.1.

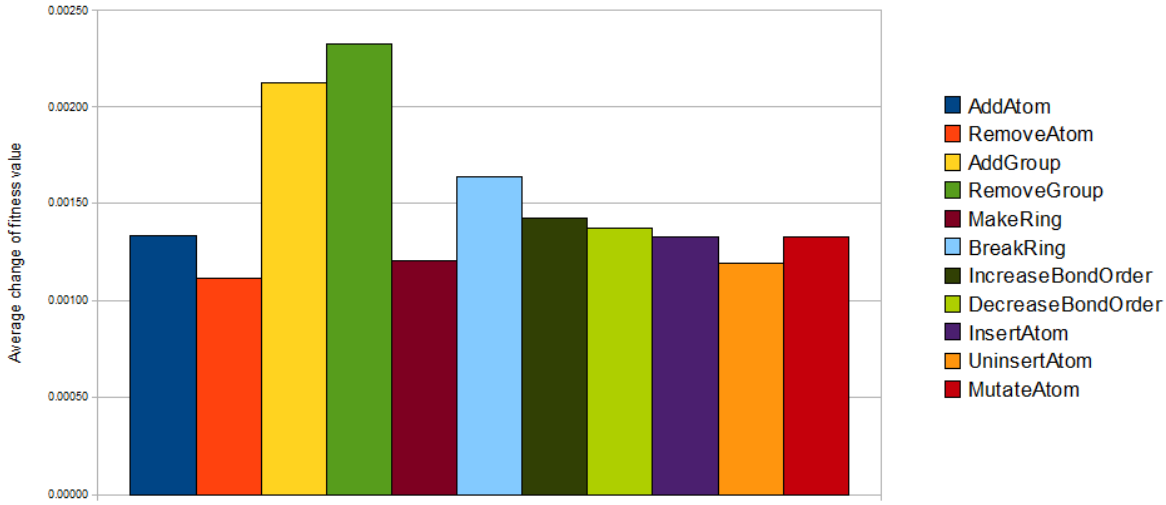


FIGURE 5.2.1. Estimated means of the mutation impacts

Since the mean says very little about the distributions of the mutation impacts themselves, further analysis is conducted. To analyze the distributions of the mutation impacts we introduce the concept of the empirical cumulative distribution function [**19**] (empirical CDF):

$$\hat{F}_n(x) = \frac{\text{number of elements in the sample } \leq x}{n} = \frac{1}{n}\sum_{i=1}^{n} I(X_i \leq x), \tag{5.2.1}$$

where n is the total number of estimations (approximately 5000), $X_1, \ldots, X_n$ i.i.d. real random variables and $I(A)$ is the indicator of an event A:

$$I(A) = \begin{cases} 1 & \text{if } A \text{ is true} \\ 0 & \text{if } A \text{ is false} \end{cases} \tag{5.2.2}$$

The empirical CDF is known to converge to the true CDF by the strong law of large numbers [**19**], therefore the empirical CDF is a good tool to analyze the distributions. For each mutation the empirical CDF is shown in Figure 5.2.2. In this figure we can see that the gradient of all the empir-

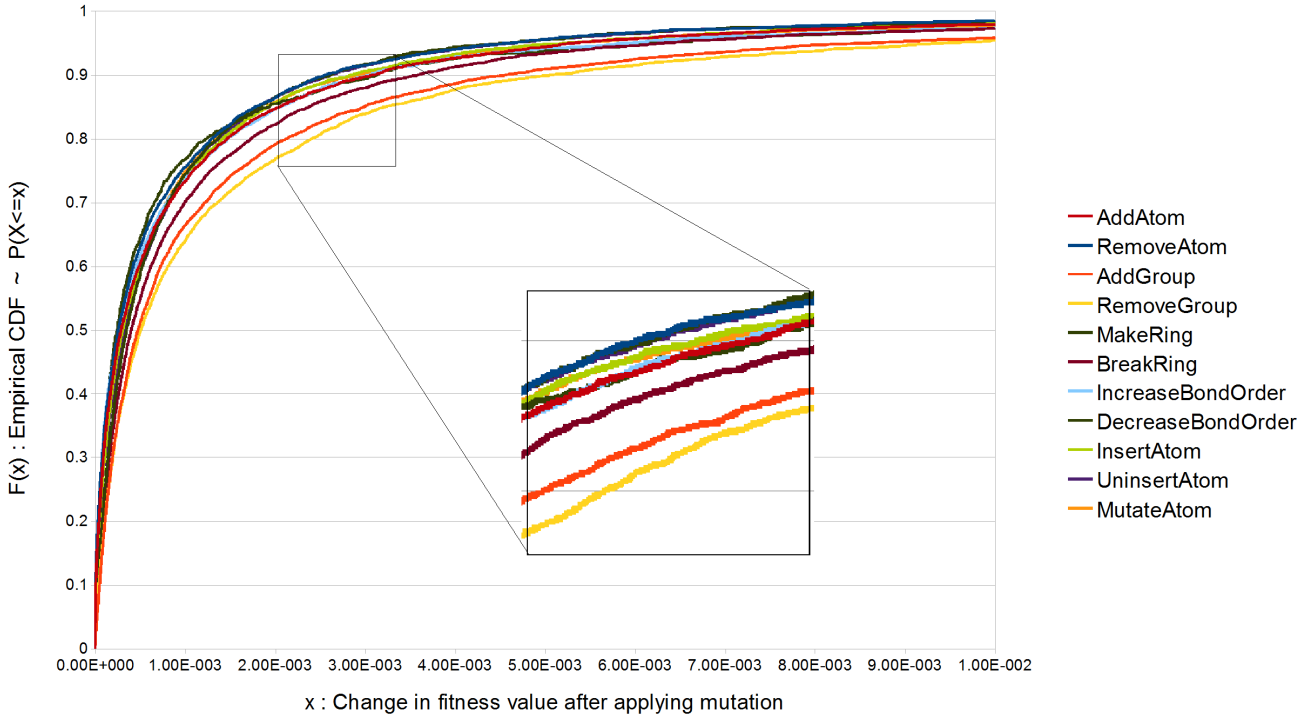


FIGURE 5.2.2. Estimated cumulative distributions for each mutation

ical CDFs is relatively high in the beginning and low at the end. A high gradient in a region of the graph implies that there is a high probability to be in that region. Likewise a low gradient implies a low probability. Many mutations share almost the same mutation impact, but a few stand out. The empirical CDFs of the *AddGroup* and *RemoveGroup* mutations have relatively low gradients at the start and higher towards the midsection and end of the diagram compared to the other mutations, which means that these two mutations have a higher probability to have bigger impact and lower probability to have small impact than the other mutations. The *BreakRing* mutation is

somewhere in between *AddGroup* and *RemoveGroup* and the rest of the mutations. This gives rise to three groups of mutations, the high impact mutations (*AddGroup*, *RemoveGroup*), the medium impact mutations (*BreakRing*) and the low impact mutations (the remaining mutations). To get a better impression of what the distributions of these groups an estimation of the probability density function (pdf) for a representative of each group is shown in Figure 5.2.3. Notice that with

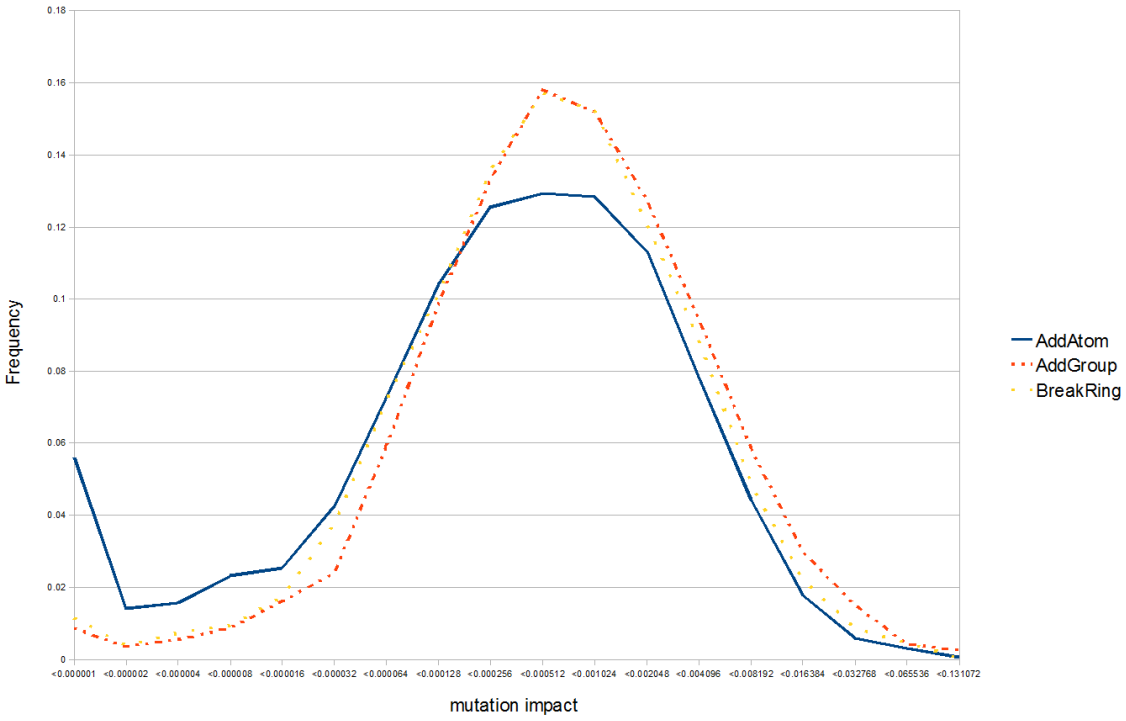


FIGURE 5.2.3. Estimated probability density function for a representative of each group of mutations

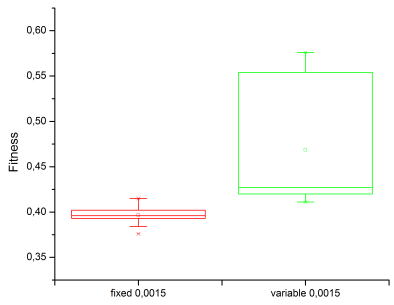
the logarithmic scale on the x-axis they resemble a normal distribution, so the mutation impacts appear to be lognormally distributed. Higher impact mutations have a similar distibution, but are shifted more to the right then lower impact mutations. Considering the shape of the distributions is similar for all mutations, the mean impact gives a good representation of the different impacts the mutations have.

To test whether the concept of mutation impact/budget can be effective even without incorporating self-adaptation, the original mutation (see Chapter 3.2) scheme is transformed to the mutation impact/budget system. Every mutation will get the same impact equal to the average of the impacts as shown in Figure 5.2.1. This average is approximately equal to ~ 0.0015. The mutation

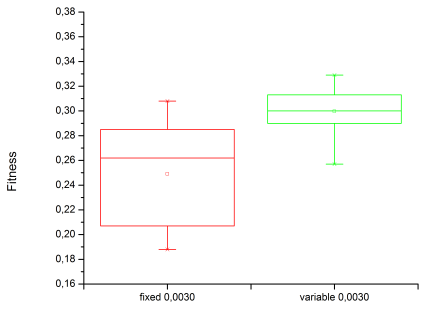
budget is set to 0.0015 and 0.0030, which corresponds to 1 respectively 2 mutations. This is then compared to a setup with same budgets, but with the mutation dependent impacts (see Chapter 4.1) from Figure 5.2.1.

The result is shown in Figure 5.2.4(a) and 5.2.5(a). Especially in the case of the (μ, λ)-strategy

(a) Average fitness of the top $\mu = 15$ individuals over 3 runs of the algorithm

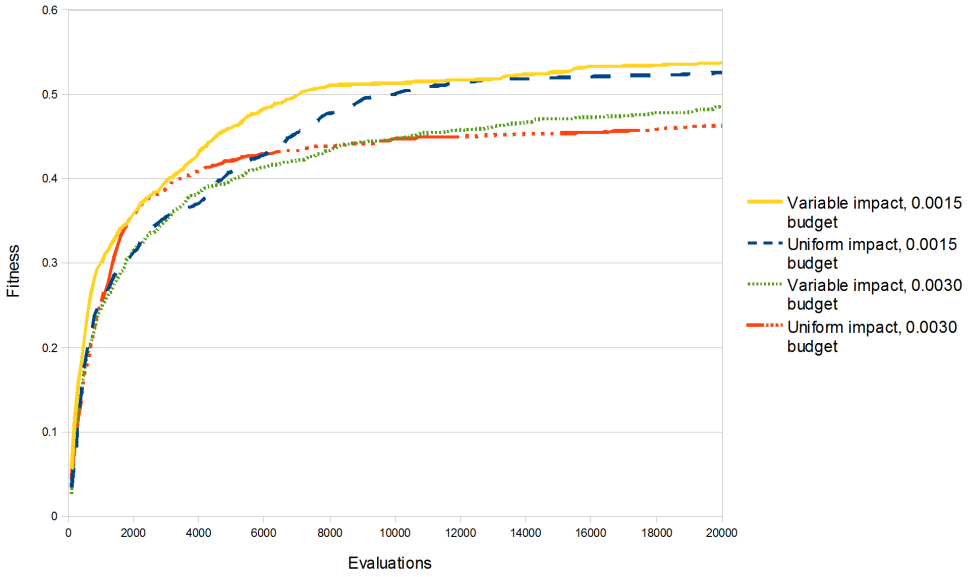


(b) Boxplot of the last 10 iterations for variable and uniform impact of 0.0015

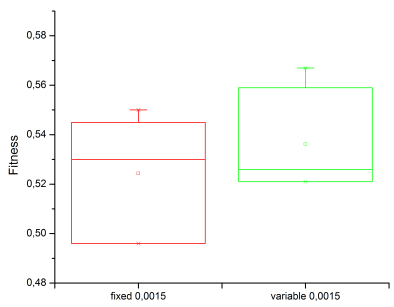


(c) Boxplot of the last 10 iterations for variable and uniform impact of 0.0030

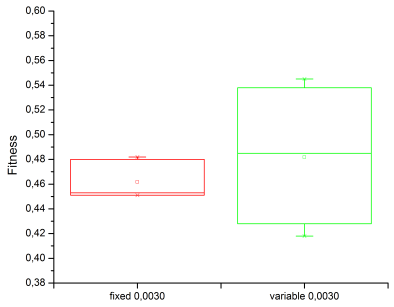
FIGURE 5.2.4. Comparison of fitness performance of uniform mutation impacts versus mutation type dependent (variable) impacts in the (μ, λ)-strategy



(a) Average fitness of the top $\mu = 15$ individuals over 3 runs of the algorithm



(b) Boxplot of the last 10 iterations for variable and uniform impact of 0.0015



(c) Boxplot of the last 10 iterations for variable and uniform impact of 0.0030

FIGURE 5.2.5. Comparison of fitness performance of uniform mutation impacts versus mutation type dependent (variable) impacts in the $(\mu + \lambda)$-strategy

a significant performance gain for the newly proposed method is apparent, for both budgets. In the case of the $(\mu + \lambda)$-strategy the new method only slightly outperforms the old strategy. The clear advantage of the new method in the (μ, λ)-strategy is likely caused by assigning a high impact to the *RemoveGroup* and *AddGroup* mutations. These mutations will be applied less in the new

method which appears to have a positive effect. This is likely caused by the need of the (μ, λ)-strategy to maintain decent solutions, without destroying them by applying a big mutation.

We can now use these mean impacts to create a set of fixed mutation budgets for the validation of the self-adaptation mechanism. These will range from one smaller mutation, around 0.0015, up until three bigger mutations, around 0.0060. These results also give rise to a value for σ_{min} (Chapter 4.2) of 0.0015. The values for σ will be initialized as a random value for each individual between 0.0015 and 0.0060. Plots of the average fitness values during the course of the evolution loop are shown in Figure 5.2.6 and 5.2.7. From these plots it can be seen that the self-adaptive approach is performing very well and reaches top spot for both the (μ, λ)- and $(\mu + \lambda)$-strategy, although the difference with the best fixed budget strategy is small is both cases.

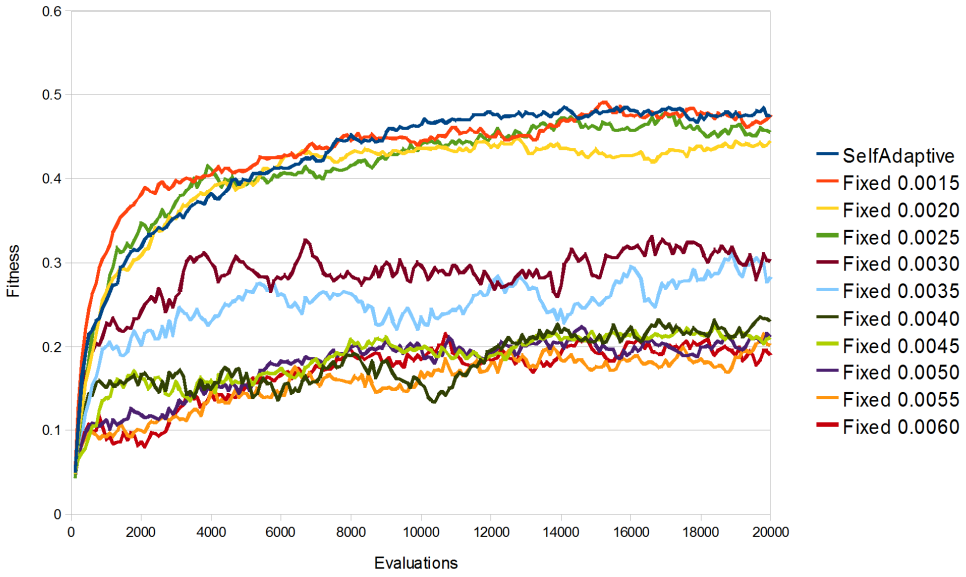


FIGURE 5.2.6. A plot of average fitness values during the course of evolution using the (μ, λ)-strategy

In order to better compare all of these fixed mutation budgets and the self-adaptive approach, we must try to find a way to quantify the performance of these approaches. Li et al. [**14**] proposed to quantify this by means of comparing the progress rate at different stages of the evolution. The progress rate at a point x was defined by Beyer [**2**] as the expectation of the distance covered towards the optimum in one mutation step starting from position x. However due to the discrete and rugged landscape of the fitness function this measure was lacking stability. Therefore we introduce

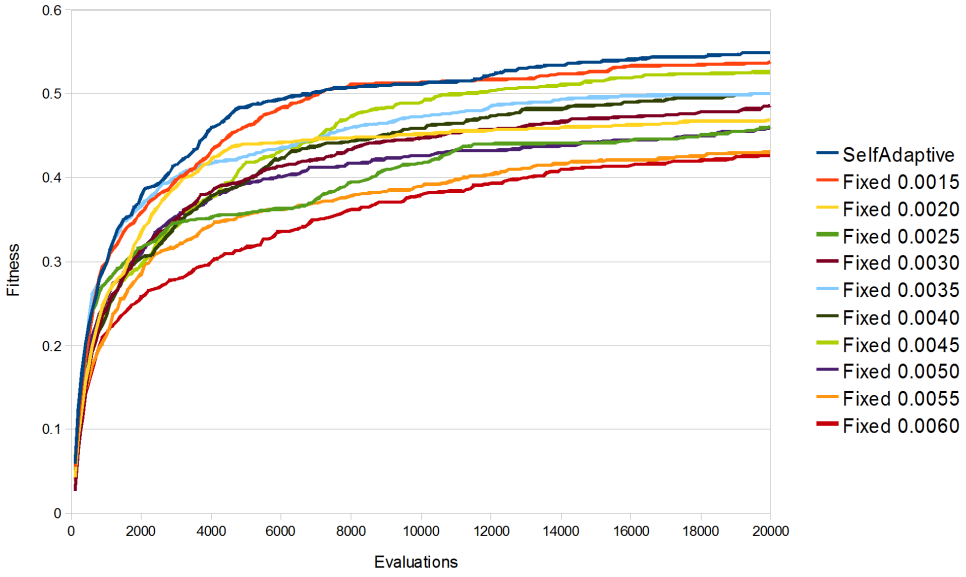


FIGURE 5.2.7. A plot of average fitness values during the course of evolution using the $(\mu + \lambda)$-strategy

a variation on this progress rate:

$$\phi(X) = \frac{1}{\psi(X)}, \tag{5.2.3}$$

where X is a partition of fitness function values and $\psi(X)$ the iterations spent by the algorithm in the range of X. The fitness function values at each iteration as used for X are determined by averaging the μ best individuals of each iteration and averaging this number for all the three different runs of the algorithm.

In Table 2 and Table 3 the progress rates are shown for respectively the $(\mu + \lambda)$- and the (μ, λ)-strategy. In Table 4 the progress rates are shown for the self-adaptive approach. A '0' in these tables means that the algorithm entered this partition, but did not reach the next partition within the given time-frame and therefore the progress rate is set to 0. In case a partition isn't reached at all by the algorithm within the given time-frame, the value 'NA' is added in the table, in practice this value is lower then the previous ones and is thus likely to be near 0. One of the first things that stand out in these tables is the fact that the progress rates of the (μ, λ)-strategy are lower than those of the $(\mu + \lambda)$-strategy. This is at least expected for the first few ranges, but with the higher risk of the $(\mu + \lambda)$-strategy to get stuck in local optima, one perhaps could expect the (μ, λ)-strategy to perform better at later stages. This is not the case in this experimental setup. Another thing of interest is the existence of multiple peaks in the case of the $(\mu + \lambda)$-strategy. When looking at the progress rates of the self-adaptive approach it seems to be performing good

	0.0015	0.0020	0.0025	0.0030	0.0035	0.0040	0.0045	0.0050	0.0055	0.0060
$X \leq 0.2$	0.25	0.17	**0.33**	0.17	0.25	0.17	0.17	0.20	0.13	0.13
$0.2 < X \leq 0.3$	**0.17**	0.10	0.08	0.09	**0.17**	0.03	0.07	0.08	0.07	0.03
$0.3 < X \leq 0.4$	0.05	**0.07**	0.01	0.03	0.05	0	0.04	0.02	0.01	0.01
$0.4 < X \leq 0.5$	**0.03**	0	0	0	0.01	NA	0.02	0	0	0

TABLE 2. Progress rates $\phi(X)$ for ten different fixed mutation budgets and four partitions using the $(\mu + \lambda)$-strategy

	0.0015	0.0020	0.0025	0.0030	0.0035	0.0040	0.0045	0.0050	0.0055	0.0060
$X \leq 0.2$	**0.25**	0.17	0.20	0.17	0.07	0.01	0.01	0.01	0.01	0.01
$0.2 < X \leq 0.3$	**0.14**	0.09	**0.14**	0.01	0	0	0	0	0	0
$0.3 < X \leq 0.4$	0.02	**0.03**	**0.03**	0	NA	NA	NA	NA	NA	NA
$0.4 < X \leq 0.5$	0	0	0	NA	NA	NA	NA	NA	NA	NA

TABLE 3. Progress rates $\phi(X)$ for ten different fixed mutation budgets and four partitions using the (μ, λ)-strategy

fitness partition \ strategy	$\mu + \lambda$	μ, λ
$X \leq 0.2$	0.33*	0.25*
$0.2 < X \leq 0.3$	0.14	0.09
$0.3 < X \leq 0.4$	0.06*	0.03*
$0.4 < X \leq 0.5$	0.02*	0*

TABLE 4. Progress rates $\phi(X)$ for the self-adaptive approach, * denotes values which differ no more then 0.01 from the optimal value of σ from Table 2 and 3

for both strategies.

In Figure 5.2.8 and 5.2.9 the progress rates for the fixed budgets are graphically shown for all four regions of X. It is apparent in the case of the (μ, λ)-strategy (Figure 5.2.9) that the lower fixed mutation budgets progress much faster than the higher ones. This implies that the advantage of low budget mutations, low budget mutations are less destructive compared to high budget mutations with respect to the parent population, is a highly needed quality in the (μ, λ)-strategy that outweighs the advantage of high budget mutations. High budget mutations explore the space

of molecules faster. In the case of the $(\mu + \lambda)$-strategy (Figure 5.2.8) there is not a clear advantage for one type of budget, there seem to be more the one peak in this diagram and thus more then one budget that performs well.

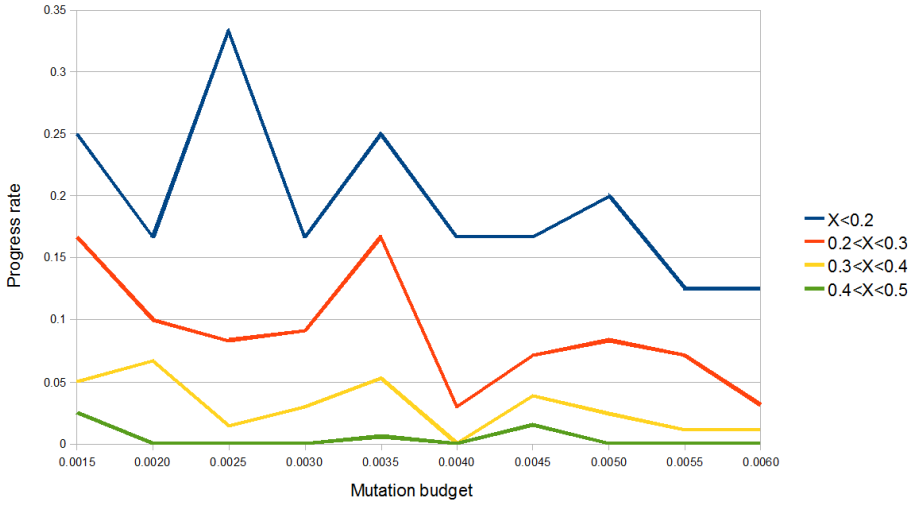


FIGURE 5.2.8. Progress rate for the $(\mu + \lambda)$-strategy

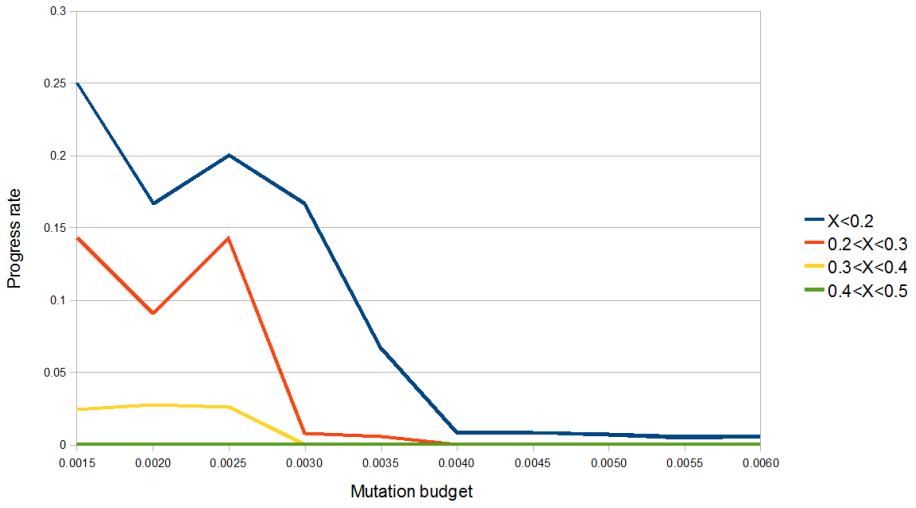
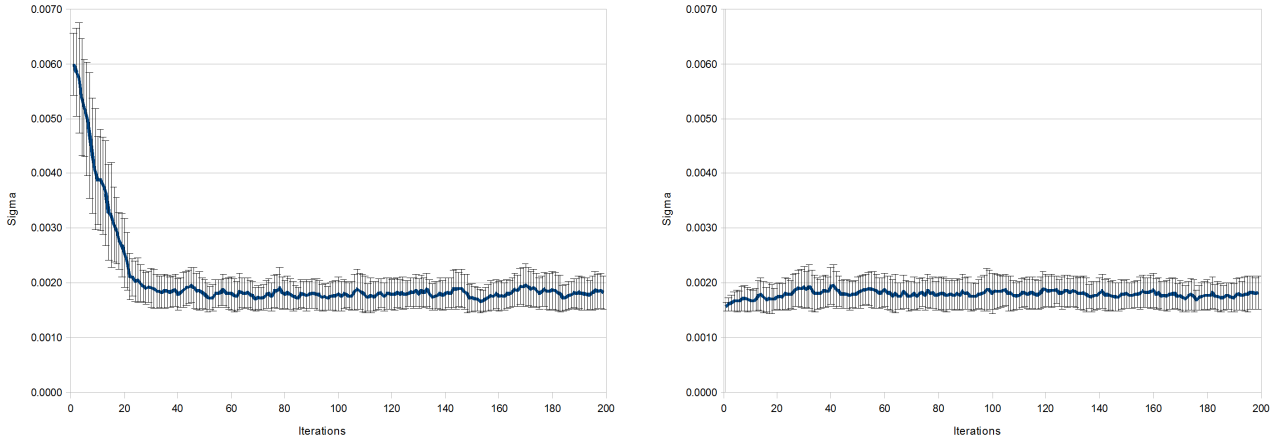


FIGURE 5.2.9. Progress rate for the (μ, λ)-strategy

In order to analyze how the self-adaptive algorithm adapted its mutation budget σ over time, two cases for each strategy were considered. In contrast to our previous random initialization of σ in the self-adaptation approach two fixed settings are considered, a high setting (0.0060) and a low setting (0.0015). The evolution of σ can be seen in Figure 5.2.10 for the (μ, λ)-strategy and Figure 5.2.11 for the $(\mu + \lambda)$-strategy.



(a) (b)

FIGURE 5.2.10. Average of σ during the course of the evolution, given a starting point of 0.0060 (Figure 5.2.10(a)) and 0.0015 (Figure 5.2.10(b)) using the self-adaptive (μ, λ)-strategy

From these figures we can see that in the case of the (μ, λ)-strategy the algorithm chooses a conservative approach (for both the high and low initialization of σ) and is highly similar to a strategy with a fixed low budget. The low budget strategies are the best performing strategies with respect to their fitness and progress rate (see Figure 5.2.6, 5.2.9 and Table 3). So the self-adaptive approach is able to find the best performing value of σ regardless of initialization for the (μ, λ)-strategy. In practice this means that the *AddGroup* and *RemoveGroup* mutations were excluded by the algorithm. In the case of the $(\mu + \lambda)$-strategy the algorithm evolves σ to two different values (multistability), depending on the initialization of σ. When σ is initialized at 0.0015 the algorithm doesn't evolve σ much and is highly similar to a strategy with a fixed low budget just as in the case of the (μ, λ)-strategy. As visible in Figure 5.2.7 and 5.2.8 and Table 2 again this is one of the best performing fixed budget strategies and thus the algorithm seems to be able to reach a good value for σ in this case. While in the case of a higher initialization of σ the evolution seems to stabilize at a value between 0.0030 and 0.0040, which again seems to be a region of high performance (notice the peak at 0.0035 in Figure 5.2.8). From this we can see

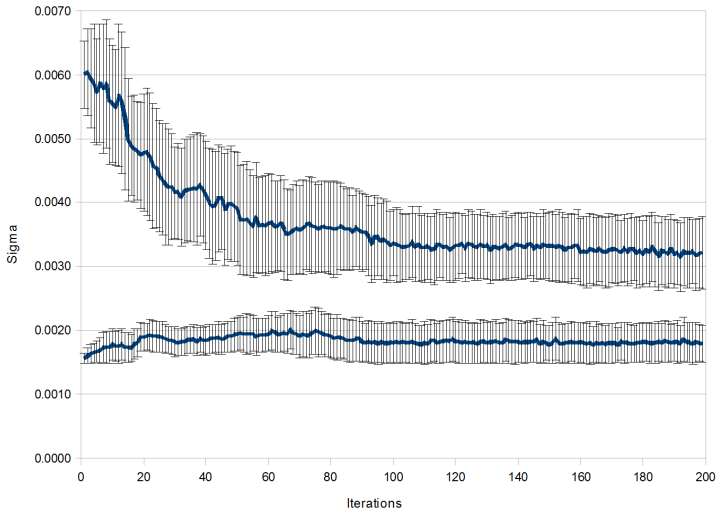


FIGURE 5.2.11. Average of σ during the course of the evolution, given a starting point of 0.0015 and 0.0060 using the self-adaptive $(\mu + \lambda)$-strategy

that the self-adaptive approach is able to find the values of σ which optimize the progress rate, although they do depend on the initialization here. A possible downside of this dependence can be that when even more local optima exists in the progress rate landscape the algorithm could pick one of these instead of the global maximum. This problem can be partly avoided by random initialization.

CHAPTER 6

Conclusions and Outlook

This report has presented a method to incorporate weighted mutation and self-adaptation to graph-based mutation operators in the context of automated *de novo* design of drugs. The concepts of mutation impact and mutation budget have been introduced to incorporate this weighted mutation scheme, while at the same time providing the algorithm with a parameter that can be used for self-adaptation. The use of this new mutation scheme improves the performance of the algorithm, especially in the case of (μ, λ)-selection. Thereby making (μ, λ)-selection a good alternative to $(\mu + \lambda)$-selection. Overall small mutation budgets performed good, even at the start of the algorithm.

The performance of the self-adaptation approach was able compete with the best handpicked choices for the mutation parameter and it was able to effectively optimize the progress rate. Therefore self-adaptation can be successfully applied to help choosing good parameters for the algorithm. However, the self-adaptation did not help to significantly increase the performance of the algorithm further.

A lot can still be improved to the current technique described here in the context of automated *de novo* design of drugs. For the concept of self-adaptation future work could focus on a suitable way of incorporating the process of learning the mutation impact in the algorithm itself by cumulative stepsize adaptation. This would be preferable especially when the behavior of the fitness function changes during the course of evolution. Moreover, the effect of the initial stepsize for the self-adaptation could be studied in more detail. Especially in the case of $(\mu + \lambda)$-selection different budgets were found by self-adaptation, depending on the initialization of the budget.

Other possible work could be focused on reaching solutions with desirable levels of quality (in terms of fitness and constraints), while spending the remaining time not on improving the quality even more but providing alternative solutions (an extension of [**9**]). It could also be interesting to apply these methods to more accurate, but computationally demanding, fitness models.

Bibliography

[1] A. Bender and R. C. Glen. Molecular similarity: a key technique in molecular informatics. *Organic and Biomolecular Chemistry*, 2:3204–3218, 2004.

[2] H. G. Beyer. *The Theory of Evolution Strategies*. Natural Computing Series. Springer, Berlin, 1st edition, 2001.

[3] H. G. Beyer, H. P. Schwefel, and I. Wegener. How to analyse evolutionary algorithms. *Theoretical Computer Science*, 1:101–130, 2002.

[4] G. Derringer and R. Suich. Simultaneous optimization of several response variables. *Journal of Quality Technology*, 12:214–219, 1980.

[5] M. Emmerich, M. Grötzner, and M. Schütz. Design of graph-based evolutionary algorithms: A case study for chemical process networks. *Evolutionary Computation*, 9(3):329–354, 2001.

[6] E. C. Harrington. The desirability function. *Industrial Quality Control*, 21:494–498, 1965.

[7] Accelrys Software Inc. Pipeline pilot, January 2010. `http://accelrys.com/products/scitegic/`.

[8] J. J. Irwin and B. K. Soichet. Zinc-a free database of commercially available compounds for virtual screening. *Journal of Chemical Information and Modeling*, 45(1):177–82, 2005.

[9] J. W. Kruisselbrink, A. Aleman, Michael T.M. Emmerich, A. P. IJzerman, A. Bender, T. Bäck, and E. van der Horst. Enhancing search space diversity in multi-objective evolutionary drug molecule design using niching. In *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 217–224, New York, NY, USA, 2009. ACM.

[10] J. W. Kruisselbrink, T. Bäck, A. P. Ijzerman, and E. Horst. Evolutionary algorithms for automated drug design towards target molecule properties. In *GECCO '08: Proceedings of the Conference on Genetic and Evolutionary Computation*, pages 1555–1562, New York, NY, USA, 2008. ACM.

[11] J. W. Kruisselbrink, M. T. Emmerich, T. Bäck, A. Bender, A. P. Ijzerman, and E. Horst. Combining aggregation with pareto optimization: A case study in evolutionary molecular design. In *EMO '09: Proceedings of the 5th International Conference on Evolutionary Multi-Criterion Optimization*, Lecture Notes in Computer Science, pages 453–467, Berlin, Heidelberg, 2009. Springer-Verlag.

[12] E. W. Lameijer, J. N. Kok, T. Bäck, and A. P. IJzerman. The molecule evoluator: An interactive evolutionary algorithm for the design of drug-like molecules. *Journal of Chemical Information and Modeling*, 46(2):545–552, 2006.

[13] Q. Li, A. Bender, J. Pei, and L. Lai. A large descriptor set and a probabilistic kernel-based classifier significantly improve druglikeness classification. *Journal of Chemical Information and Modeling*, 47(5):1776–86, 2007.

[14] R. Li. *Mixed-Integer Evolution Strategies for Parameter Optimization and Their Applications to Medical Image Analysis*. PhD thesis, Leiden University, 2010.

[15] C. Lipinski, F. Lombardo, B. Dominy, and P. Feeney. Experimental and computational approaches to estimate solubility and permeability in drug discovery and developments settings. *Advanced Drug Delivery Reviews*, 46(1-3):3–26, 2001.

[16] P. Needleman, J. Turk, B. A. Jakschik, Morrison A. R., and Lefkowith J. B. Arachidonic acid metabolism. *Annual Review of Biochemistry*, 55:69–102, 1986.

[17] C. A. Nicolaou and C. S. Pattichis. Multi-objective de novo drug design using evolutionary graphs. *Chemistry Central Journal 2008*, 2(1):7, 2007.

[18] I. Rechenberg. *Evolutionsstrategie '94*. Fromman-Holzboog, Stuttgart, 1994.

[19] J. A. Rice. *Mathematical Statistics and Data Analysis*. Duxbury Press, Belmont, California, 2nd edition, 1995.

[20] H. P. Schwefel. Collective phenomena in evolutionary systems. In P. Checkland and I. Kiss, editors, *Problems of Constancy and Change - The Complementarity of Systems Approaches to Complexity, Proc. 31st Annual Meeting*, pages 1025–1033, Budapest, 2008. Int'l Soc. for General System Research.

[21] B. Sharma, I. C. Parmee, M. Whittaker, and A. Sedwell. Drug discovery: exploring the utility of cluster oriented genetic algorithms in virtual library design. *Congress on Evolutionary Computation*, pages 668–675, 2005.

[22] O. M. Shir. *Niching in Derandomized Evolution Strategies and its Applications in Quantum Control; A Journey from Organic Diversity to Conceptual Quantum Designs*. PhD thesis, Leiden University, 2008.

[23] A. R. Webb. *Statistical Pattern Recognition*. John Wiley & Sons, Ltd., Chichester, England, 2nd edition, 2002.

[24] D. S. Wishart, C. Knox, A. C. Guo, D. Cheng, S. Shrivastava, D. Tzur, B. Gautam, and M. Hassanali. Drugbank: a knowledgebase for drugs, drug actions and drug targets. *Nucleic Acids Research*, 36:D901–6, 2008.

[25] D. S. Wishart, D. Tzur, C. Knox, R. Eisner, A. C. Guo, N. Young, D. Cheng, K. Jewell, D. Arndt, S. Sawhney, C. Fung, L. Nikolai, M. Lewis, M. A. Coutouly, I. Forsythe, P. Tang, S. Shrivastava, K. Jeroncic, P. Stothard, G. Amegbey, D. Block, D. D. Hau, J. Wagner, J. Miniaci, M. Clements, M. Gebremedhin, N. Guo, Y. Zhang, G. E. Duggan, G. D. Macinnis, A. M. Weljie, R. Dowlatabadi, F. Bamforth, D. Clive, R. Greiner, L. Li, T. Marrie, B. D. Sykes, H. J. Vogel, and L. Querengesser. Hmdb: the human metabolome database. *Nucleic Acids Research*, 35:D521–6, 2007.

APPENDIX A

Other data

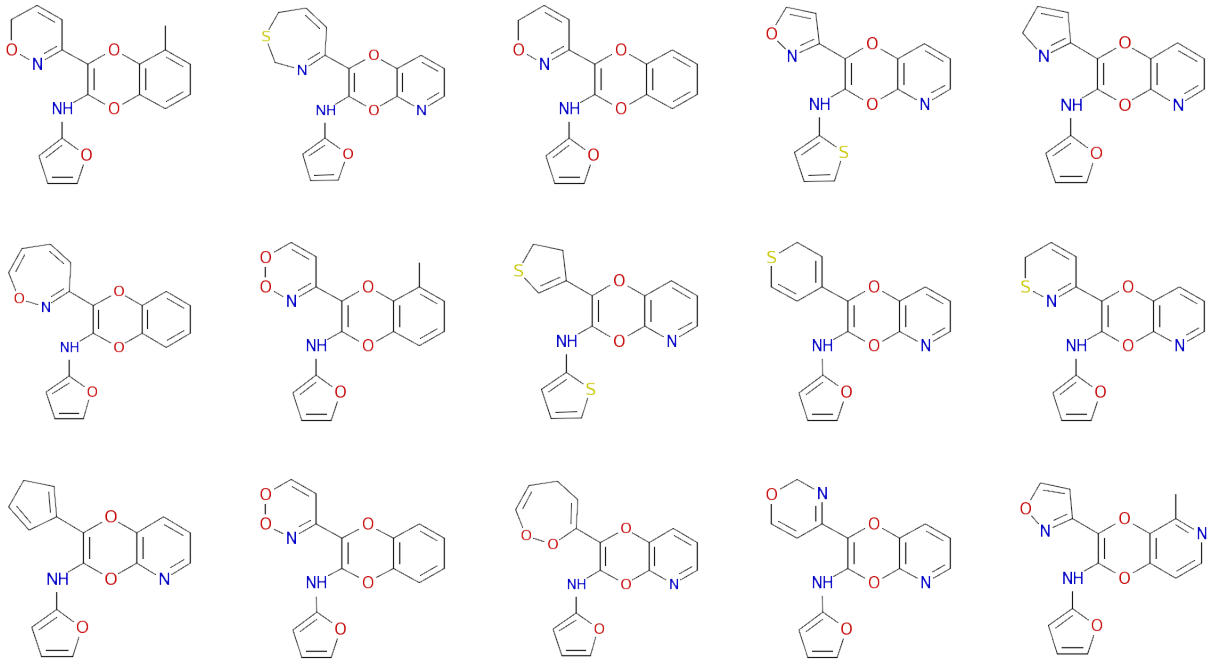


TABLE 1. Best scoring final population in the (μ, λ)-strategy with an average fitness of 0.50

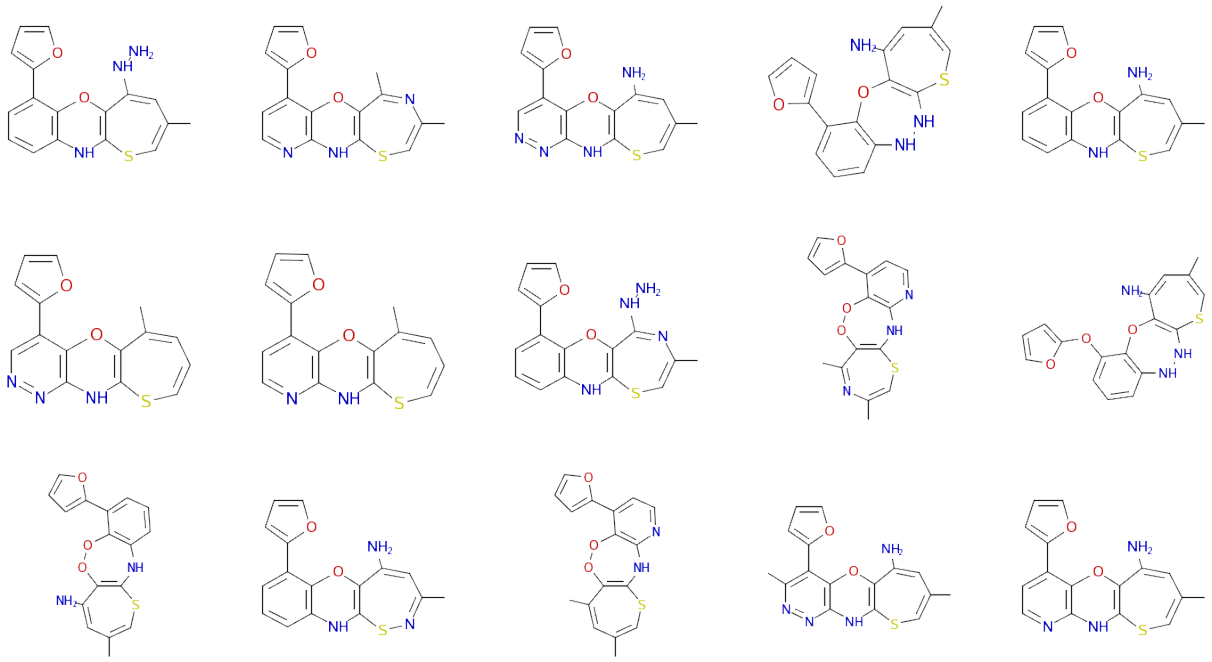


TABLE 2. Best scoring final population in the $(\mu + \lambda)$-strategy with an average fitness of 0.55

Mutation	Mean	Standard deviation
AddAtom	0.00134	0.00425
RemoveAtom	0.00112	0.00340
AddGroup	0.00212	0.00686
RemoveGroup	0.00233	0.00700
MakeRing	0.00121	0.00345
BreakRing	0.00163	0.00510
IncreaseBondOrder	0.00143	0.00442
DecreaseBondOrder	0.00137	0.00438
InsertAtom	0.00133	0.00425
UninsertAtom	0.00120	0.00330
MutateAtom	0.00133	0.00396

TABLE 3. Mean and standard deviation of the mutation impacts

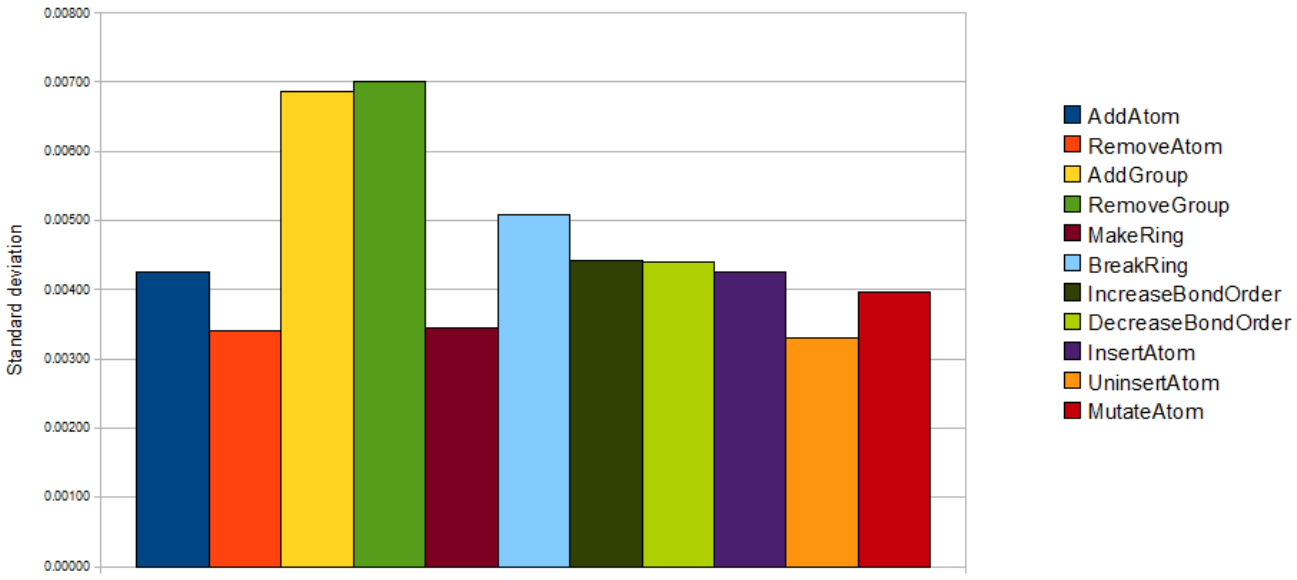


FIGURE A.0.1. Standard deviation for the mutation impacts