

Evolutionary algorithms for automated drug design

M.Sc. thesis J.W. Kruisselbrink

LIACS, Universiteit Leiden

January 10, 2008

Abstract

This thesis presents an evolutionary algorithm for the automated generation of molecules that could be used as drugs. It is designed to provide the medicinal chemist with a number of candidate molecules that comply to pre-defined properties and can be used for further evaluation.

The algorithm proposed here is implemented as an extension to the so-called Molecule Evuator [10] which implements an interactive evolutionary algorithm. It implements a variable sized population and bases its search on target-bounds that are set for a number of molecule properties and implements a selection procedure based on the notion of Pareto domination.

The results show that it is indeed possible to apply the concept of evolutionary computation on molecule design using target-bounds for molecule properties as optimization goal. For practical usage, the algorithm presented here could serve as a starting point, but should be further improved with respect to diversity within the generated set of molecules.

Contents

1	Introduction	3
2	<i>De novo</i> drug design	5
3	Computer aided drug design	7
3.1	Drug design as an optimization problem	7
4	Evolutionary computation for drug design	10
4.1	Evolutionary algorithms for drug design	11
4.1.1	Molecule representation	11
4.1.2	Genetic operators	13
4.1.3	Evaluation	14
4.1.4	Solution diversity	17
4.2	Genetic Programming	18
4.2.1	Program representation	18
4.2.2	Population initialization	20
4.2.3	Genetic operators	21
4.2.4	Evaluation and selection	23
4.3	The Molecule Evaluator	23
4.3.1	Molecule representation	24
4.3.2	Population initialization	26
4.3.3	Genetic operators	27
4.3.4	Evaluation and selection	29
5	An automated evolutionary algorithm for drug design	33
5.1	Main evolution cycle	33
5.2	Genetic operators	34
5.3	Evaluation and selection	35
5.4	Variable sized populations	36
5.5	Parameter tuning	37

6 Experiments	38
6.1 Setup	39
6.2 Results	39
7 Conclusions and outlook	47

Chapter 1

Introduction

To aid the design and development of new drugs, automated computational methods can be used to generate molecules that could be used as lead compounds for possible new drugs. Lead compounds are chemical compounds that have pharmacological or biological activity of which the chemical structures can be used as a starting point for chemical modifications in order to improve potency, selectivity, or pharmacokinetic parameters.

By automatically generating libraries of molecules that comply to certain predefined physicochemical properties, the medicinal chemist can be provided with sets of promising molecules that are good candidates for further investigation. The particular use of automated methods is that they can speed up the search and cover a larger part of the search space because they are fast and unbiased.

In this thesis we will propose an evolutionary algorithm that will be an extension implemented in the so-called Molecule Evoluator [10]. The Molecule Evoluator implements an interactive evolutionary algorithm and can be used for both lead generation (generating new molecules with certain desired properties from scratch) and lead optimization (start from one or more lead molecules and optimize the molecule properties or generate derivatives).

In addition to the interactive evolutionary algorithm used by the Molecule Evoluator, we will propose an automated evolutionary algorithm for the generation of a diverse set of molecules with very specific physicochemical properties. Based on a set of strict targets that can be specified by the medicinal chemist, the algorithm will search for molecules that comply to these targets and therefore could be promising candidates for further investigation. We will focus on finding a method that can incorporate multiple objectives and that can generate a diverse set of molecules as output to offer the medicinal chemist a broad set of promising molecules.

Chapter 2 starts with a brief description of drug design and the difficulties of drug design. Thereafter, chapter 3 will discuss the use of computational methods in the field of drug design. Chapter 4 will discuss the application of evolutionary computation for drug design. In particular, it will discuss genetic programming and the implementation of the Molecule Evaluator. The newly proposed automated evolutionary algorithm will be given in chapter 5 followed by a presentation of the outcomes of the experiments in chapter 6. Chapter 7 will close with conclusions and future research recommendations.

Chapter 2

De novo drug design

De novo drug design and development is the process of finding and creating molecules which have a specific activity on a biological organism.

Generally, drugs are formed around small key molecules that can bind to a 'target' which, in most cases, is a protein such as a receptor, enzyme, transport protein, or antibody. These proteins can be found on the cell surface, in the cell, or in plasma (except for receptors) and by exhibiting or inhibiting the activity of these critical components, the drug molecules can change the behavior of the entire cell. Target cells can be either cells of the disease causing organisms or of the diseased patients themselves. For illustrative purposes, figure 2.1 shows two molecules that have been found to be very suitable as drugs; Zoloft (antidepressant) and Prozac (antidepressant).

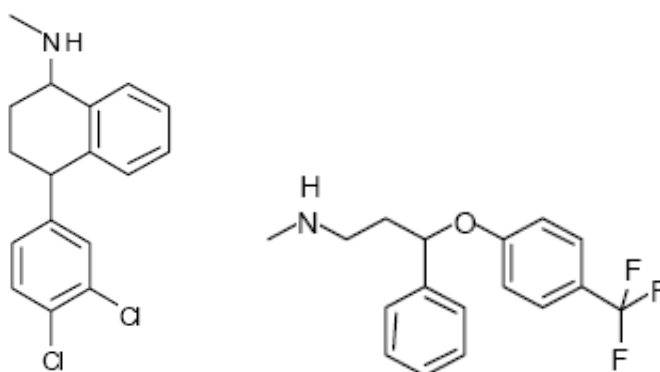


Figure 2.1: The molecular structures of the two well-known drugs Zoloft and Prozac.

Finding molecules that could be used as drugs is a tedious task. The main goal is of course to find molecules that exhibit the desired behavior on the targeted cells. In the very large search space of molecules with only a very small number of molecules that exhibit the desired behavior, finding the desired molecules is by no means trivial and cannot be done by simply considering every possible molecule.

Moreover, having found molecules that exhibit the desired behavior on the targeted cells does not assure usability as drugs. Molecules that are effective on the targeted cells but are badly soluble in water/blood fail to be suitable as drugs after all [11]. Also, it might happen that the molecules are broken down by the body before they reach the targeted cells. So, except for having the desired behavior on the targeted cells, the molecules should also have other pharmacokinetic properties that make them drug-like.

The notion of drug-likeness increases complexity of the search to a great extent. In the first place, it reduces the number of suitable molecules while it hardly reduces the size of the search space as the number of all drug-like molecules is still estimated to be at least 10^{60} [6]. Secondly, besides some prediction methods and heuristics [4], there is no clear way to determine the drug-likeness of a molecule.

In other cases, promising molecules could fail when the molecules exhibit a too small influence to be effective, or, when they are sufficiently effective, also have a damaging effect on other parts of the body. Matters like these also increase the difficulty of the search to a large extent.

The main bottleneck of the drug development process is the investigation of promising candidate molecules. The pharmaceutical industry spends a lot of time, effort and money investigating candidate molecules that eventually turn out to be unsuitable as drugs. In this process, especially the physical experiments are time and money consuming. To reduce the time and money spent on the development of new drugs, it is therefore required that unsuitable candidate molecules fail as early in the process as possible (fail-fast, fail-cheap). Or even better; unsuitable drugs will not be selected as candidates at all.

Computational methods offer excellent possibilities to reduce the time and money spent in the development of new drugs. By implementing simulation methods on virtual drugs, a good part of the unsuitable candidate molecules can be filtered out before going into the laboratory for real-world testing. Besides this, computers can be used to offer the medicinal chemist new and maybe less obvious possible alternative molecules that might be very promising. The latter is the focus of this thesis.

Chapter 3

Computer aided drug design

The main problem of drug design is that, in general, there is a clear view on the desired behavior of the molecules but it is the finding of molecules that exhibit this behavior and have the other necessary pharmacokinetic properties that proves to be difficult. The fraction of possible solutions for each search is generally very small, and the huge search space simply makes it impossible for a chemist to do a full search and consider every possible molecule by creating it and doing experiments on it. This will take too much time (in the order of many lifetimes) and will be too expensive. The use of computers to help with this search seems to be obvious and inevitable.

Computer aided drug design aids the drug design process by automatically searching for promising molecules in the space of all possible drug-like molecules. Having computers perform this search on virtual molecules and simulation and calculation based estimations molecule behavior allows for a much faster and cheaper search. Using computer based methods is therefore a very good way to find promising molecules for the process of drug design.

3.1 Drug design as an optimization problem

From the computational point of view, the search for molecules that can be used as drugs can be seen as yet another optimization problem. In this particular optimization problem, the search space is the set of all the drug-like molecules, and as optimization objective there is the desired behavior of the molecules. However, actually implementing and solving this optimization problem is a whole other story.

A first complication is that the search space is tremendously big. With the number of possible drug-like molecules estimated to be at least 10^{60} , also for computers, it

is simply impossible to solve this optimization problem by complete enumeration. With the knowledge that complete enumeration is not an option we are bound to resort to other, more clever, computational methods for solving the problem (e.g. evolutionary algorithms).

A second complication lies in the complexity of the molecular structures that make up the search space. Note that this search space is significantly more complex than the common search spaces. Although molecules can well be represented by graphs, matters like neighborhood, distance and similarity are not easily defined. Because of this, the design of variation operators and the analysis methods of the search space are difficult.

A positive notion about the representation is that drug molecules generally tend to have a molecular weight between 0 and 500 atomic mass units [11] and therefore are generally relatively small. Exploiting this general property and only focusing on small molecules would ease the search to a great extent. This simplification not only decreases the size of the search space, but working with relatively small graph-structures also allows for more complex (with exponential order time-complexity) graph operations to be used. Although a part of the search space will be lost (i.e. not covered by the search), this part is small enough for this simplification to be useful in practise.

Bigger problems arise when trying to come up with a suitable fitness function. Usually there is a clear view on the desired behavior that the molecules should exhibit. However, actually capturing this desired behavior in a mathematical scoring function is very difficult. The most ideal fitness function would be a function that could somehow express the closeness of a molecule's behavior to the desired behavior in a simple numerical fitness score. Obviously, the real-life case does not quite provide for such a scoring function.

What can be done is to use calculatable molecule properties and running computer simulations to estimate the behavior of a molecule. Frequently the medicinal chemist is capable of determining which calculatable properties a molecule should have in order to exhibit a certain behavior. Using targets for a number of (calculatable and simulatable) properties and trying to minimize the difference between the property values of the candidate solutions and the target values would then prove a very good help to computationally provide a number of promising molecules. With this, the problem becomes a multi-objective optimization problem.

Finally, there is the desire to generate a diverse set of molecules instead of finding one single solution. In the end, drug design will always remain a job for the medicinal chemist rather than the computer. Among a few suitable candidate molecules

it is probably the medicinal chemist that is better than a computer in deciding which of the candidates is the most suitable. So, ideally the optimization method should focus on providing the medicinal chemist a diverse set of possible molecules rather than only one.

The difficulty with this is two-folded. First, many optimization methods are limited when it comes to the generation of diverse solutions. Secondly, we are working with the graph-like structures of molecules. For these structures (as mentioned before) it is very difficult to even define a measure of diversity. Hence, the optimization method needs a way to create and ensure such a diversity in its set of final solutions.

To summarize, the search for molecules that could serve as lead compounds for the design of new drugs can be seen as an optimization problem for which the optimization method:

- has to be able to deal with the large search space
- has to be able to work with graph-like structures
- has to be able to optimize over multiple objectives
- can return a diverse set of solutions.

And of course, there also exists the implicit criterion that the optimization method has to do all the above within a reasonable amount of time. The faster, the better.

Chapter 4

Evolutionary computation for drug design

The size of the search space together with its complexity make it very hard (if not impossible) to use classical computational approaches to search for molecules. Evolutionary algorithms are specially suited for such large search spaces, and can also deal with the complexity of the problem, as they do not (strictly) require any knowledge about the search space. Therefore, evolutionary algorithms are a good choice to use to search for molecules that comply with certain pre-defined properties.

The idea of applying evolutionary computation methods to evolve drug molecules is not new. A patent exists that covers designing of molecules with desired properties by means of an evolutionary process [15]. Also, there is a number of publications that discuss the application of evolutionary algorithms on drug design, among which are [6, 7, 13, 9, 10].

This section will give an overview of evolutionary computation in the scope of drug design. It will start with an overview of applying evolutionary algorithms on molecules and drug design. Thereafter, an overview of Genetic Programming will be given as this is a field of evolutionary computation that deals with structures that are very suitable for representing molecules and can therefore be a good source of inspiration. Lastly, an overview will be given of the so-called Molecule Evaluator as the algorithm proposed in this paper is intended to serve as an extension of the Molecule Evaluator.

4.1 Evolutionary algorithms for drug design

An evolutionary algorithm can roughly be implemented as outlined in algorithm 1 [3]. Although the actual real-world implementations come with many different faces, this basic principle is always the same.

Algorithm 1 General outline of an Evolutionary Algorithm

```
1: t:=0;
2: initialize P(t);
3: evaluate P(t);
4: while not terminate do
5:   P'(t):= select-mates(P(t));
6:   P''(t):=variation(P'(t));
7:   evaluate(P''(t));
8:   P(t+1):=select(P''(t)∪P(t));
9:   t:=t+1
10: end while
11: return P(t);
```

To apply the concept of evolutionary computation to the problem at hand brings along a few problems that distinguishes it from the usual applications of evolutionary computation. These problems are the same as mentioned in section 3.1; the problems of dealing with the complexity of the molecular structures that make up the search space, the finding of a suitable fitness function and the creation and preservation of diversity among the population.

Of those three mentioned difficulties, the first two are encountered and discussed in every other attempt to apply the concept of evolutionary computation to drug design. Remarkable is, however, that most of these attempts focus very much on the genetic operators (and especially the design of the crossover operator), while less attention is spent on the construction of the fitness function. Practically no attention is spend on trying to generate a diverse set of solutions instead of just one.

4.1.1 Molecule representation

The choice of the representation is important because it largely influences the implementation of the genetic operators and the way in which the evolutionary algorithm crawls through the search space. A major difficulty is that the genetic operators should be designed such that, in principle, the whole search space is reachable, but also such that it is assured that the structures of the molecules will not be disrupted by the genetic operators.

Graphs are probably the most suitable representations that exist in computer science for modelling molecules. The similarities are very obvious, and as graphs are well known in computer science, taking this point of view is a logical choice.

However, although graphs are broadly used structures in today's computer science, using graphs in evolutionary computation is still difficult. The search space of graph-structures is much more complex than the usual vector representations that are used in Evolution Strategies (ES) and Genetic Algorithms (GA) [3], and even more difficult than the tree-structures that are used in Genetic Programming (GP) [8]. The major difficulty with a search space of graphs is that it lacks a certain triviality in the way that the search space is structured.

Graphs as molecules

Using graphs to represent molecules, each node represents an atom and each edge represents a bond between two atoms. The edges of the graphs are undirected and also contain a weight which represents the order of the bond (i.e. double bonds, or triple bonds are edges with respectively weights two and three).

The most important restriction that the graphs need to satisfy is that atoms in a molecule have a valence. Therefore, each node should have an exact number of edges bound to it. The number of bonds for each node is equal to the valence of the atom that is represented by that node.

Also, as mentioned earlier, drug molecules generally tend to be relatively small, and thus so are the graphs. This restriction is very useful as it limits the size of the search space, and it allows for more complex graph operations to be used.

It becomes more complicated when noting that graphs that satisfy the valence rules are not necessarily stable molecules (i.e. not makable/synthesizable). One can imagine that with graphs, the connections can bend in any way and very twisted graphs can be constructed. Atom bonds are limited in their connection strengths and therefore the flexibility of the molecular structures is limited as well. Matters like synthesizability in this sense are much more complicated.

Incorporating restrictions like these in the representation and the genetic operators could save a lot of time evaluating graphs that are unmakeable as molecules or unsuitable as drugs. However, for practical usage, these matters can only be taken into account to a certain extent. The restriction in size and the restriction of the valence are easily incorporated. For the more complex restrictions, one should consider the trade-off; it could well be that the cost of generating only makable molecules is much higher than the price of evaluating unmakeable solutions from time to time.

Implementing graphs

A last issue that is well worth mentioning is the choice of how to implement the graphs. A well chosen graph-representation could ease the implementation and improve the performance (speed) of the evolutionary algorithm and also a good representation might give inspiration for the use of different genetic operators.

Graphs are collections of vertices and edges that connect pairs of vertices. There are a few possibilities when it comes to the representation of graphs, each with its advantages and disadvantages. The most commonly used representations are:

- Node list / edge list: keep up two stacks; one with the nodes listed, and one containing the bonds that link two nodes together.
- Adjacency matrix: the rows and columns of a two-dimensional array represent source and destination vertices and entries in the graph indicate whether an edge exists between the vertices.
- Adjacency list: implemented by representing each node as a data structure that contains a list of all adjacent nodes.

From this, it can be seen that the choice of representation is very important when it comes to the ease of implementation of the genetic operators. For example, adding nodes and removing nodes would be something of a job for the adjacency matrix structure, but less for the adjacency list. However, the general maintenance and bookkeeping of an adjacency list is much more difficult.

4.1.2 Genetic operators

The main problem with molecules is the way to deal with the graph-structures and the finding of good genetic operators. The importance of the representation of the individual is due to the fact that a good representation can ease the implementation of the genetic operators and can be used as inspiration for the design of new genetic operators. The genetic operators decide how the population of an evolutionary algorithm 'moves' through the search space.

For the choice of the genetic operators one should focus on the following properties:

- Any given solution in the search space should be reachable from any given population within a finite number of genetic operations.

- Applying the main genetic operator should more likely result in a small change than in a big change. That is, a small mutation of a solution should lead to a small change in the fitness. The more this applies, the smoother the fitness landscape will be and the better the evolutionary algorithm will perform.
- One should try to prevent that applying a genetic operator has a disruptive effect on the structure of the solution. In this particular case, graphs are very easily disconnected with the removal of an edge or a node, or disrupted by a misfortunate crossover. One could save precious evaluation time if applying the genetic operators would only result in graphs that are worthwhile evaluating.

In this case this means that the genetic operators should at least have to be able to add and remove nodes, and to add and remove edges. But with this, the genetic operators should also be chosen such that they yield a smooth fitness landscape, and it has to be prevented that the graph gets disconnected or becomes invalid in some other way.

4.1.3 Evaluation

The most accurate way of determining a drug's behavior is doing real-life experiments. Unfortunately, with such an approach, it is impossible to evaluate a large set of candidate molecules because it is time consuming and very expensive.

Another option would then be the use of expert knowledge. This is the way it is done by the Molecule Evaluator as will be described in section 4.3. Experts are very well capable of determining "good" molecules for the search they are performing and can make the decision already a lot faster. With this, the number of candidate molecules that can be considered is already much bigger than with the real-life experiments but also this approach is still too limited to evaluate larger sets of molecules (especially when looking for molecules with very specific properties).

The third option is the one that is more suited for the evaluation of a huge number of molecules; making use of calculatable fitness function. This option is also the only way to implement a 'real' evolutionary algorithm.

Calculatable fitness functions

Molecule properties come in many different flavours, varying from the similarity to a certain target molecule to the use of fitness measures based on the molecule activity or even on complex simulations. Some are easily calculatable but of little importance with respect to the estimation of the molecule's behavior. Others could

be very useful but are very complex and therefore take a lot of computing time to calculate. Having those kinds of properties and everything in between makes the choice of the properties that will be used a delicate matter which can influence the performance of the evolutionary algorithm very drastically.

Most other applications of evolutionary computation for drug design spend very little attention to the construction of the fitness function. The major reason for this is that using the concept of evolutionary computation on drug design is fairly new and the focus mainly lies on proving that the concept of evolution is usable to reach the whole search space. As a result, typically some kind of similarity measure that measures the difference between the structure of the solutions to the structure of a certain target molecule is used.

One might very well argue that measuring the structural difference between the solution and a certain target molecule is somewhat unrealistic or even unusable as objective function since we obviously already know the target molecule. It is usable as a theoretical measure for testing the quality of the genetic operators, but a more realistic approach would be to have a certain desired behavior as target and the search algorithm searching for structures that exhibit that behavior.

A more realistic approach would be that the fitness of a solution is somehow expressed in the form of calculatable molecule properties that together can express the desired behavior. In this case, targets can be set for the calculatable molecule properties that together can express the desired behavior of the molecule and the objective becomes to minimize the difference between the property values of the molecules and the desired property values. With this, the optimization problem becomes a multiple objective optimization problem [5].

Taking the multiple objective point of view, we assume to optimize over a number of numerically expressible molecule properties with target values each property. So, each property i with target t_i can be expressed numerically by a function $g_i(x)$ with the input x being any instance of the set of all possible molecules. With that, we can define for each molecule property i an objective function f_i as:

$$f_i(x) = |g_i(x) - t_i| \quad (4.1)$$

Or, when the target is determined to be an interval $[a_i, b_i]$ rather than a single value, among more exotic functions, one could take:

$$f_i(x) = \begin{cases} |g_i(x) - a_i| & \text{if } g_i(x) < a_i \\ |g_i(x) - b_i| & \text{if } g_i(x) > b_i \\ 0 & \text{otherwise} \end{cases} \quad (4.2)$$

Minimizing over all objective functions f_i is then the goal of the optimization process.

When searching for solutions within a search space where multiple different fitness functions are used, one is performing a multiple objective search. The difficulty when dealing with multiple objectives is that most of the time, these objective are (at least to a certain extend) conflicting. Because of this, it is not always possible to compare different solutions and call one of them the "best", so making an ordered ranking in the straightforward manner is not an option. There are basically two ways that are in this case applicable to deal with the set of multiple objectives.

The first manner is to create a new fitness function that is the weighted sum of all the objective functions and search for the optimal solutions that minimize this new function:

$$f(x) = c_1 \cdot f_1(x) + \dots + c_n \cdot f_n(x) \quad (4.3)$$

With:

$$\sum_{i=0}^n c_i = 1 \quad (4.4)$$

Here, f_i denotes the i^{th} objective function, and c_i denotes the amount in which the objective function is taken into account (weight).

In principle this method could work fairly well, but one should note that with the weighted sum approach, the different weights for the different objectives are chosen subjectively and it is very difficult to find suitable weights when dealing with objective functions that output completely different values. One could imagine one objective function to have a discrete and finite output set (e.g. $\{1,2,3,4,5\}$) and another objective function that has as output a real number. So, much effort should be spend on the tuning of the weights when choosing such a solution.

Also, even if much time is spend on the tuning and one assumes to have a nice set of weights, it is not guaranteed that the algorithm will find good local optima. Using a weighted sum will push a search towards a certain direction and it will only look for optima in a certain direction instead of considering a wide range of directions. The search direction is therefore misleading and very good compromise solutions could just be missed.

The second method to deal with multiple objectives is a method that is capable of keeping the search focused on a broad range of directions and solutions. This method bases the fitness assignment on the notion of Pareto domination [5]. Pareto domination is defined as follows:

A solution x_1 is said to dominate x_2 iff:

$$\forall i \in 1, \dots, n : f_i(x_1) \leq f_i(x_2) \wedge \exists i \in 1, \dots, n : f_i(x_1) < f_i(x_2) \quad (4.5)$$

With the notion of Pareto domination, there is a way to compare the fitness in a fairer manner than is possible with the usage of a weighted sum.

The most simple form of the use of the notion of Pareto domination is making two sets: the set of dominated solutions and the set of non-dominated solutions. The non-dominated solutions of a given set of solutions consists of the solutions that are not dominated by any other solution of the population.

Another way to use the notion of Pareto domination is to count for each individual in the population the number of individuals that it dominates. Good solutions will dominate many other solutions, and bad solutions will dominate only a few or even zero solutions. This way, it is possible to get an even better ordered list from which we can select the "better-fit" solutions easily.

With the notion of Pareto domination there are still many variants to chose from, but the important notion is that making use of Pareto domination allows for a fairer comparison of solutions and a broader search. The only downside is that it requires many more solutions (i.e. larger population sizes) to form a good and nicely distributed Pareto front.

4.1.4 Solution diversity

A big issue is the desired molecule diversity in the set of molecules that the algorithm should generate. How do you keep the search both explorative and exploitative at the same time and how do you find optima in multiple parts of the search space at the same time? This is a very delicate job, especially since we are dealing here with graphs. With graphs, distance and measures of similarity are not easily defined, and therefore niching techniques (as used in evolutionary algorithms) are not easily implemented.

Evolutionary algorithms naturally tend to converge to one single part of the search space. Therefore extra attention should be paid on keeping the population diverse. Methods to keep the population diverse in evolutionary algorithms are the so-called niching methods [12]. Implementing niching methods for the problem at hands would complicate the design even more, but could also prove to be very worthwhile.

4.2 Genetic Programming

A large part of the inspiration of the attempts to apply evolutionary algorithms on molecule design comes from genetic programming [8]. Genetic programming is the branch of evolutionary computation that aims to apply the principles of evolution as a way of automatically generating computer programs.

Just like the problem currently at hands, the difficulty with genetic programming is that the search space is huge, consists of complexly structured instances and is non-trivial to apply in a straightforward manner in an evolutionary algorithm. Hence, genetic programming requires a different way of handling with the complex structures of programs that are used as individuals, as well as it needs to determine the fitness in a non-straightforward manner. Especially the representation and the genetic operators that are used in genetic programming can be used as inspiration for applying evolutionary computation on drug molecules as the commonly used tree-representation in genetic programming is very similar to the graph structures of molecules.

Most commonly, applications of genetic programming adopt the flowchart of evolution as given in figure 4.1 (from [8]). In principle, this is just another implementation of the general evolution cycle of algorithm 1, and in that sense not very different. However, it does place genetic programming within its scope and it is therefore included in this thesis.

In general, genetic programming works with large population sizes and with an implicit '+'-strategy. Both are usually chosen because of the complex nature of the search space and because the probability of creating better-fit offspring is somewhat smaller than with the usual evolutionary algorithms.

4.2.1 Program representation

The Most interesting aspect of genetic programming is, as mentioned, the program representation. In genetic programming, the most broadly used representation of programs are parse-tree representations. Although the graph structures of molecules are more complex than trees, the principles of genetic programming can still be used as inspiration to applying evolutionary algorithms on graphs. Besides the difference of cycles versus no cycles, graphs and trees are very similar and the genetic operators of genetic programming could be adopted to a great extend.

In the parse-trees used in genetic programming, every inner node in the tree represents an operator function and every leaf node represents an operand. Using

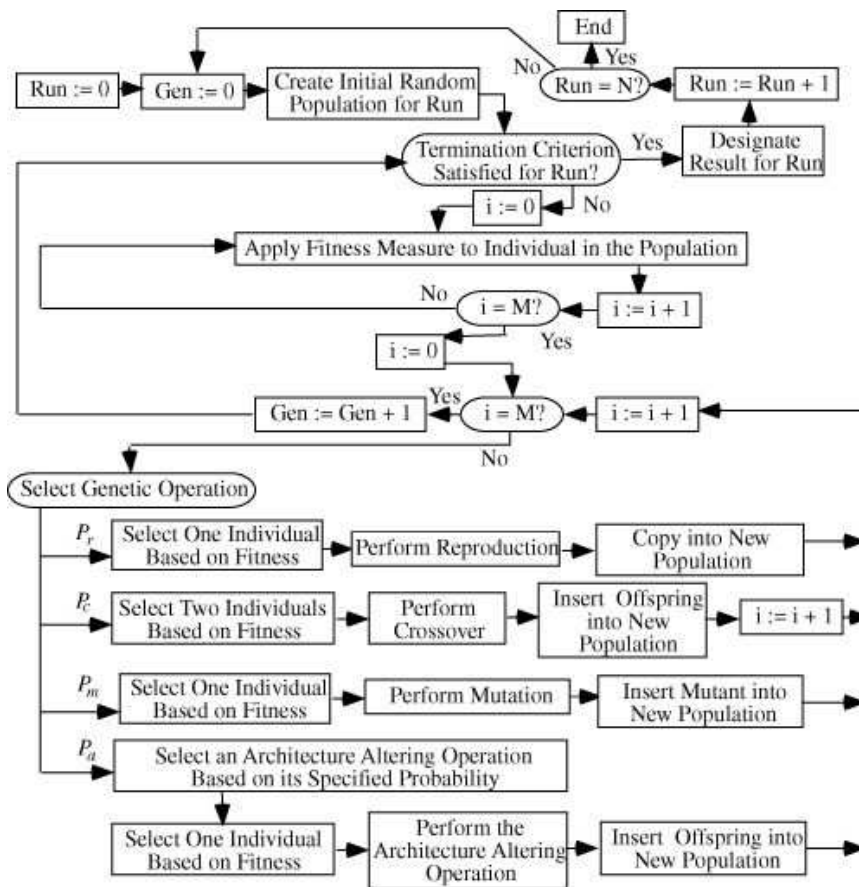


Figure 4.1: The flowchart of genetic programming (from [8]).

this representation makes programs both easy to evolve and easy to evaluate. In effect, genetic programming favors programming languages that naturally embody tree structures (e.g. Lisp). Figure 4.2 shows an example of such a parse tree representing the prefix expression $(+ (* (\text{sqrt } y) 2) (/ y (\text{if } (= y 0) 1 y)))$.

An interesting observation is that the nodes (both terminal nodes and function nodes) have a restriction when it comes to connections with other nodes. Typically, each function has an input arity which has to be preserved to keep the parse trees valid. E.g. the simple `if`-statement has an arity of 3, the operators `+`, `-`, `*` and `/` all have an arity of 2, and terminal nodes have an arity of 0. These restrictions are similar to the valence constrictions that atoms are bounded to.

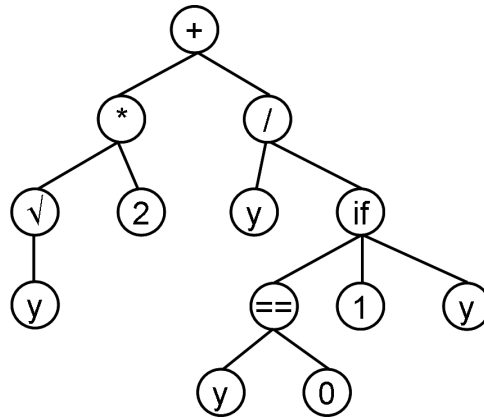


Figure 4.2: A tree representation as used in genetic programming representing the prefix expression $(+ (* (\text{sqrt } y) 2) (/ y (\text{if } (= y 0) 1 y)))$.

4.2.2 Population initialization

Before starting the evolution, one first needs an initial population. The method of creating the initial population determines the starting point of the evolution and therefore the whole evolution cycle and is therefore another point worth mentioning. In principle, generating the initial population comes down to simply randomly generating the designated number of trees. A random tree can be build recursively by starting with a random root node, and continuing to add child-nodes until all ends of the tree consist of terminal nodes (thus the real leaf nodes). However, there are some issue concerning the size and the shapes of the initial trees.

Concerning the size, as trees can vary in length from just a terminal as a root node, to an infinite tree length, there are infinite possibilities for the tree sizes of the initial population. The usual method is to determine a maximum length. The maximum dept normally depends on the problem at hand and the function set.

For the regularity, there is a choice to create solely regular trees with equal length branches (the full method) or creating trees with random length branches (the grow method). With the full method, the length of these branches is obviously the pre-defined maximum size. In the grow method, typically also a maximum tree-size is added to prevent the trees from growing out of control. In genetic programming, the so-called 'ramped half-and-half' method is commonly used. This method uses the full method and grow method together and each method delivers half of the initial population.

To summarize:

- Maximum initial depth of trees D_{max} is set

- Full method:
 - nodes at depth $d < D_{max}$ randomly chosen from function set F
 - nodes at depth $d = D_{max}$ randomly chosen from function set T
- Grow method:
 - nodes at depth $d < D_{max}$ randomly chosen from function set $F \cup T$
 - nodes at depth $d = D_{max}$ randomly chosen from function set T
- Ramped half-and-half method:
 - grow and full method each deliver half of the initial population

4.2.3 Genetic operators

Genetic programming uses four types of genetic operators as can be seen in figure 4.1. Besides mutation and recombination, genetic programming also uses a reproduction operator and architecture altering operations. By using reproduction as a genetic operator, genetic programming implicitly allows good solutions to be preserved in the next population (it could be viewed as an implicit implementation of a '+' strategy). The architecture altering operations are operations that are used to change the internal structure of an individual but are preserving the functionality.

The more interesting genetic operators from our point of view are recombination (crossover) and mutation. Of those two, the crossover operator is the most important one and is used as the main genetic operator. The mutation operator is used as an explorative operator. For both operators, there exist multiple variants all implementing the operator in a slightly different way.

For crossover, the basic principle is that the genetic programming algorithm somehow exchanges subtrees among the individuals in the population. Table 4.1 shows the mainly used crossover types in genetic programming. Figure 4.3 shows a visualization of three of these commonly used crossover types. This operator basically recombines individuals of the population in the hope that even better individuals will emerge.

As with crossover, also the implementations of the mutation operator vary. Table 4.2 shows the many mutation types. With the mutation operator, one should however keep in mind that the mutation operator together with the crossover operator should always provide a way for the population to reach every part of the search space. Not all the mutation operators shown in Table 4.2 are single-handedly able to comply to this criterion.

Crossover type	Effect
subtree exchange crossover	exchange subtrees between individuals
self crossover	exchange subtrees between an individual and itself
module crossover	exchange modules between individuals
context preserving crossover (SCPC and WCPC)	exchange subtrees if coordinates match exactly (SCPC) or approximately (WCPC)

Table 4.1: Crossover types used in genetic programming.

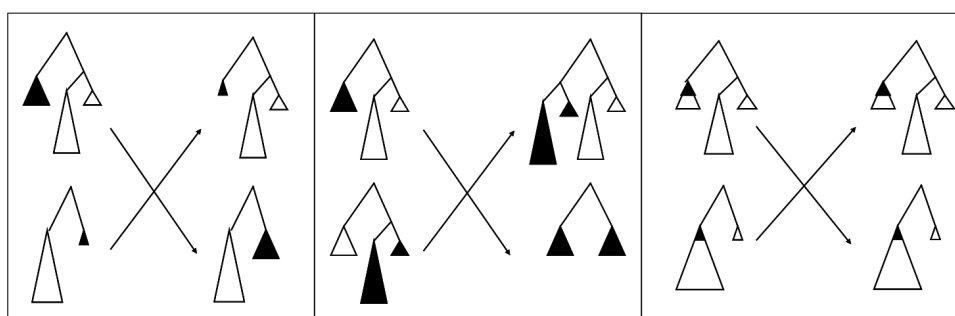


Figure 4.3: Three commonly used crossover types, from left to right: subtree exchange crossover, self crossover and module crossover.

Mutation type	Effect
point mutation	single node is exchanged with a different node with the same arity
permutation	arguments of a node are permuted
hoist	new individual generated from subtree
expansion permutation	terminal exchanged against a random subtree
collapse subtree mutation	subtree exchanged against a random terminal
subtree mutation	subtree is exchanged against a random subtree
gene duplication	subtree substituted for a random terminal

Table 4.2: Mutation types used in genetic programming.

4.2.4 Evaluation and selection

In general, the programs that are evolved should work for a large (or even infinite) set of input-output pairs. This makes it impractical or even impossible to evaluate the population over the whole input-output set. Because of this, the determination of the fitness of each program in the population is usually taken over a number of runs with different inputs. The objective function for genetic programming to minimize is then usually the mean squared error over some number of runs, or something similar. The error is calculated as the measure between the output o_i and the desired output t_i .

$$f = \frac{1}{n} \cdot \sum_{i=0}^n (o_i - t_i)^2 \quad (4.6)$$

Of course, as most problems require some slight modification, other variants are being used as well. However, the main principle of the evaluation is that for each individual the fitness is determined over a number n of input-output combinations.

As can be seen in the flowchart of figure 4.1, selection in genetic programming is done in an implicit manner. The better fit individuals have a higher chance of being selected for the creation of each new offspring.

4.3 The Molecule Evuator

The algorithm that will be presented in this paper is an extension to the Molecule Evuator and is integrated in a command line version of the Molecule Evuator. This section will describe the working of the Molecule Evuator.

The Molecule Evuator implements an interactive evolutionary algorithm which can be used for both lead generation (generating new molecules with certain desired properties from scratch and lead optimization (start from one or more lead molecules and optimize the molecule properties or generate derivatives)).

The most remarkable fact about the evolutionary algorithm implemented by the Molecule Evuator is that it is user-involved. The Molecule Evuator uses the knowledge of the expert in a direct way by letting the user guide the evolution process towards desired molecules. Moreover, the user-involvedness also allows the user to test every inspiration perceived during the evolution process directly which makes the Molecule Evuator especially powerful as an interactive design tool.

The Molecule Evuator has implemented its evolutionary algorithm as shown in algorithm 2. As can be seen, the user decides each iteration on the number

of offspring that will be created and selects the parents for the creation of each next generation. Also, the user decides the termination condition of the evolution loop (or rather, the user decides each step whether or not another evolution step is performed).

The procedure **Allowed(o)** is a function that determines if the created offspring complies with the calculated physicochemical filters. These filters will be described below.

Algorithm 2 General outline of the Evolutionary Algorithm implemented by the Molecule Evaluator

```
1: for each evolution step do
2:   filters := let user set filter bounds;
3:   constraints := let user set filter bounds;
4:    $\lambda$  := number of offspring selected by user;
5:    $P'(t)$  := parents selected from  $P(t)$  by user;
6:    $P(t + 1)$  :=  $P'(t)$ ;
7:    $n$  := 0;
8:   while  $n < \lambda$  do
9:     select crossover or mutation;
10:    if crossover then
11:       $p_1, p_2$  := randomly select two parents from  $P'(t)$ ;
12:       $o_1, o_2$  := recombine( $p_1, p_2$ );
13:       $o$  := randomly select between  $o_1$  and  $o_2$ 
14:    else
15:       $p$  := select one parent randomly from  $P'(t)$ ;
16:       $o$  := copy( $p$ );
17:       $o$  := mutate( $o$ );
18:    end if
19:    if Allowed( $o$ ) AND  $o \notin P(t + 1)$  then
20:       $P(t + 1)$  :=  $P(t + 1) \cup o$ ;
21:       $n$  :=  $n + 1$ 
22:    end if
23:  end while
24:   $t$  :=  $t + 1$ ;
25: end for
```

4.3.1 Molecule representation

The representation that is used in the evolutionary algorithm part of the Molecule Evaluator is the so called TreeSmiles representation [10]. The TreeSmiles notation

is an extended version of the by chemists commonly used SMILES notation [14].

With SMILES and TreeSmiles, the molecules are represented by a string (similar to the Lisp-notation used by genetic programming) that is human-readable and can easily be transformed to a 2D-structure by a chemist. The TreeSmiles extension to SMILES lies in the fact that it is less readable by humans, but more complete and thus very well suited for use by computers.

SMILES

SMILES represents molecules with character strings in which atoms are represented by their atomic symbols. To simplify the reading, hydrogen atoms and single bonds are usually left out of the notation. The hydrogen atoms can be left out of the representation as their presence can be deduced from the valence rules of the atoms.

Bonds are implied by the adjacency of atoms in the SMILES-strings. Atoms that are adjacent are assumed to be connected and single. Double, triple and aromatic bonds are indicated by the symbols =, # and :. Connections between non-adjacent atoms, which is the case when representing ring structures, are denoted by ring-number labels.

Subgroups or branches can be denoted by enclosing them in parentheses. Branches can be nested (i.e. branches can contain branches as well) or stacked (one atom can have two separate branches). In all cases, the connection to a parenthesized branch is to the left.

The above rules give a rough outline of the SMILES notation. For a more thorough overview that also covers the more exotic options like aromaticity, stereochemistry and isotopes, we refer to [1].

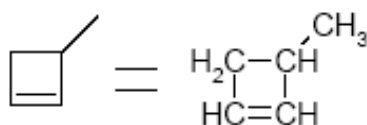
TreeSmiles

The only difference between SMILES and TreeSmiles is that the TreeSmiles notation does not allow hydrogen atoms to be defined implicitly, but requires every hydrogen atom to be notated in the string as a branch. The reason for requiring this is that this makes the implementation of the genetic operators a lot easier.

Figure 4.4 shows a molecule with its SMILES and TreeSmiles representation.

TreeSmiles in the Molecule Evuator

The TreeSmiles notation finds its use because of the fact that string and vector representations allow for an easy implementation of nearly all genetic operators



SMILES: C1C(C)C=C1

TreeSmiles: (C(1)(H)(H)(C(C(H)(H)(H)))(=C(H)(C(1)(H)(H))))

Figure 4.4: An example of the SMILES and TreeSmiles representation of a molecule (from [10]).

that are used by the Molecule Evaluator, but has one major drawback; there are molecules that can be represented in multiple ways by the TreeSmiles notation. This is obviously not desirable as it complicates the testing whether two molecules are identical.

Also (if one does not pay attention with the implementation of the genetic operators) this could have the effect that two identical molecules that are differently represented could have different genetic neighborhoods.

A good example of this is the "break ring" mutation that will be described in section 4.3.3. For the "breakring" mutation it should be possible to break the ring at all places (instead of only at the places in the TreeSmiles that denote the ring).

A solution to this problem is to compare the molecules to the so-called "unique SMILES" notation. This is a more strict version of the SMILES notation in which each molecule can be represented in only one way. However, attention and care have to be taken and one should always keep in mind that the TreeSmiles notation is in this sense problematic. Also, in general, switching between representations is very error-prone.

4.3.2 Population initialization

The Molecule Evaluator also allows the user to provide the initial population of molecules from which the evolution can start. If no initial population is provided by the user, an initial population is generated from scratch.

For the creation of an initial population from scratch, two different molecule generation methods are implemented. The first method to generate molecules takes for each generation a simple Methyl molecule and applies a number of mutations on it. The second method is a bit more sophisticated and implements a fragment-based generation of molecules described in [2]. The balance between the number of molecules generated by both methods can be set by the user.

4.3.3 Genetic operators

The Molecule Evuator provides one crossover operator and eleven mutation operators as genetic operators for the evolutionary algorithm. To allow for a certain measure of control to the structures of the mutated molecules, the user is allowed to decide which of the eleven mutation operators are actually used (i.e. each mutation operator can be turned on and off). Also, the user is allowed to decide which genetic operator is the primary operator as the Molecule Evuator provides the option to adapt the balance between crossover and mutation.

The implementation of the crossover operator is inspired by the subtree crossover as used in genetic programming. As the Molecule Evuator deals with graphs rather than with trees, the only complication lies in the cases where the nodes that were picked as root nodes of the subtrees are part of a cycle. This is simply resolved by only allowing nodes that are not part of a cycle to be picked as root nodes of the subtrees.

The mutation operator is the more complicated genetic operator. For the mutation, the Molecule Evuator randomly chooses between the available mutation operators. The eleven mutation operators are listed below:

- Add atom: Replaces a hydrogen atom in the molecule with a non-hydrogen atom. The remaining bonds of the added atom are filled with hydrogens.
- Delete atom: Removes an atom that is attached to only one non-hydrogen atom (with a single bond) by first deleting the hydrogen atoms attached to it, and renaming the atom to a hydrogen atom.
- Add group: Replaces a hydrogen atom in the molecule with a larger chemical group such as a phenyl group.
- Delete group: Removes a larger chemical group that is attached to only one non-hydrogen atom (with a single bond) and replaces it by a hydrogen atom.
- Insert atom: Adds an atom by inserting a new atom with a valence of two or higher into a bond. The remaining bonds of the new atom are fulfilled by adding hydrogens to it.
- Uninsert atom: Removes an atom that has exactly two non-hydrogen neighbours. It removes the atom and its hydrogens and subsequently creates a bond between its two neighbouring non-hydrogen atoms.

- Mutate atom: A non-hydrogen atom is changed into another non-hydrogen atom which has a valency of at least the number of bonds of the original atom with other non-hydrogen atoms.
- Increase bond order: If two atoms which are bonded to each other both have at least one hydrogen atom, those hydrogen atoms are removed and an extra bond is created between the atoms (the bond order is increased from single to double, or from double to triple).
- Decrease bond order: If two atoms which are connected by at least two bonds, one bond is broken and hydrogen atoms are attached to the loose ends.
- Create ring: Similar to increase bond order, but works between two atoms which are not bonded to each other. These atoms are connected using a single bond (the two hydrogen atoms are changed into ring indices).
- Break ring: Chooses a single bond in a ring, breaks that bond and adds hydrogen atoms to correct the valences. This mutation is the most difficult to implement. The algorithm should be able to break any bond in the ring, which is not easy to do with a graph structure.

Figure 4.5 is taken from [10] and shows the effect of the different mutations on molecules. In this figure, the "addgroup" and "removegroup" mutations are not shown, but are from that point of view of course similar to the "addatom" and "removeatom" mutations.

The use of eleven mutation operators is a remarkable choice. With eleven mutation operators, there are many different molecules possible for a single mutation of one single molecule. Thus, each molecule has many neighbors. Although this does allow the genetic operators to reach the whole search space, there is very much to choose from when mutating an individual.

Also, little time and thought have been spent on the tuning of the parameters. The user is provided an option to turn mutation operators on or off, but if a mutation operator is set, it has equal probability of being picked compared to the other mutation operators that are set. It might also be good to give certain mutation operators a higher probability compared to others. For example, the "make ring" operator could be picked less frequently than the "mutate atom".

Besides this, one might ask whether it is such a good idea to allow the user to set such parameters. Allowing the user to select which mutation operators are set and allowing the user to decide on the balance between mutation and recombination gives the user a lot of flexibility, but this will also allow the user to tune the


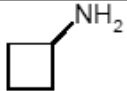


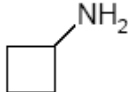













Mutation name	Initial structure	Final structure	Initial TreeSmiles	Final TreeSMILES
Add atom			...(C(H)(H)(C...	...(C(H)(N(H)(H))(C...
Insert atom			...(C(H)(H)(C...)...	...(C(H)(H)(N(H)(C...))...
Delete atom			...(C(H)(N(H)(H))(C...	...(C(H)(H)(C...
Uninsert atom			...(C(H)(H)(N(H)(C...))...	...(C(H)(H)(C...)...
Increase bond order			...(C(H)(H)(C(H)(H)(...	...(C(H)(=C(H)(...
Create ring			(C(H)(H)(H)(C(H)(H)(C(H)(H)(H)))	(C(1)(H)(H)(C(H)(H)(C(1)(H)(H)))
Decrease bond order			...(C(H)(=C(H)(...	...(C(H)(H)(C(H)(H)(...
Break ring			(C(1)(H)(H)(C(H)(H)(C(1)(H)(H)))	(C(C(H)(H)(H))(H)(H)(C(H)(H)(H)))
Mutate atom			...(C(H)(H)(C(H)(H)(C...	...(C(H)(H)(S(C...

Figure 4.5: The effect of the mutation operators of the Molecule Evaluator on molecules.

parameters to (seen from the evolutionary computation point of view) unrealistic and unwise settings. It might be best to find suitable parameter settings that, on average, perform very well and keep the user of the system from this parameter tuning.

4.3.4 Evaluation and selection

As mentioned above, the Molecule Evaluator uses the expert user as a fitness function. The user decides each iteration which molecules can be used as parents for the next generation. With this, the user forms the fitness function and selection function in one, and the performance of the evolutionary algorithm depends very much on the intuition, knowledge and luck of the user.

Besides this, to assist the users in their choice, the Molecule Evaluator also supplies several chemical descriptors for each molecule. These descriptors are especially useful for the user to estimate drug-likeness (Lipinski's rule of five makes use of these properties to estimate drug-likeness [11]) or the way that the drug molecule is to be taken in and can be used in addition to the visual representation to estimate how good each molecule is. The descriptors are:

- The number of hydrogen donors (HD)
- The number of hydrogen acceptors (HA)
- The molecular weight (MW)
- The Logarithmic value of the octanol/water partition coefficient (logP)
- The Logarithmic value of the aqueous solubility in mol/l at 25°C (logS)
- The polar surface area (PSA)
- The number of rotatable bonds (RB)
- The number of aromatic systems (AR)
- The number of aromatic substituents (AS)

In order to allow the user to force the evolution towards molecules with certain descriptor values, the Molecule Evaluator provides a filter mechanism (chemical filters). When one or more filters are set, the Molecule Evaluator only returns offspring that have their property values within these filter bounds.

In addition to the chemical filters, the Molecule Evaluator also allows the user to set constraints and enforce the molecules to have certain structures (called physical filters). If these filters are set, then the Molecule Evaluator will reject molecules that do not comply to these constraints. These constraints are listed below:

- Bredt's rule: no double bond with one end at a bridgehead of a bridged ring system.
- Acetals: allow molecules to contain a functional group of a Carbon bonded to two OR groups.
- CH₂-Imines: allow molecules to contain Imine groups (R-N=CR₂) or CH₂-Imine groups (R-N=CH₂).

- Ortho-, meta-, or paracyclophane: ortho / meta / paracyclophanes are benzene rings that have a short bridge forming a second ring over, for example, the meta-atoms.
- Common Ring System: Filters out all ring systems that do not occur in the NCI. By checking the Also Consider Atoms box, the comparison will also consider the atoms in the ring besides comparing the bonds.

In contrast to what is suggested in [10], the search for offspring that have valid property values is not done by another evolutionary algorithm. Instead, the current version of the Molecule Evuator (version 4.0.0.0) simply keeps on creating offspring from the selected parents until it has the targeted amount of valid offspring. In algorithm 2, the function `Allowed(o)` determines whether a molecule has its properties lying within the filter bounds. Note that this paper proposes an extension that will implement the other evolutionary algorithm that was suggested by [10].

The expert as selection and fitness function

Using the expert user to select the parents for each next generation is the major strength and main distinction from other implementations for *de novo* drug design. The advantage of this is that an expert is capable of basing the selection on heuristics and intuitive knowledge that cannot be modelled by any fitness function. In effect, the algorithm will waste much less time on obviously worthless solutions and the expert can guide the algorithm towards the parts of the search space that are more of interest. This reduces the number of evolution iterations the algorithm needs to find good results as there is less time wasted on bad solutions and less interesting parts of the search space.

However, this strength is also a main weakness. There are two big disadvantages of solely using experts for the selection of the parents in evolutionary algorithms.

First, humans interacting with algorithms make algorithms slow. It generally takes much more time for a human to evaluate and select suitable solutions for further evolution than it does for a computer. Therefore, it will take much more time to run the algorithm for a higher number of generations with experts selecting the parents.

Also, with the use of heuristics and intuitive knowledge in the search process, the algorithm risks performing an incomplete search. It is likely that there are parts of the search space that do not seem interesting at first sight but yield pretty good results when evaluated. Especially those parts could be very interesting, but will probably be overlooked in a search guided by an expert.

These two disadvantages call for yet another attempt to find an automated selection procedure.

Molecule descriptors and filters

The molecule descriptors offer the expert user extra information which can be used to better predict the usefulness of a molecule for further evaluation. Besides that, the descriptors can be used in the evolution process to generate offspring with certain properties.

Although the idea of this is in principle a good one, its implementation in the Molecule Evaluator is a candidate for improvement. The random search for offspring that by chance have the desired properties is very slow and makes it in some cases impossible for the algorithm to produce valid offspring even when in principle, these offspring could exist.

Consider the case in which the property values of the molecules in a certain generation differ very much from the target property values of the filters. In this case, the probability of creating offspring with the desired property values could be very small as such molecules would probably lie in a whole different part of the search space that is simply not reachable in one evolution-step from that set of molecules.

Then also consider the cases where the filter bounds are really strict. In these cases, there are probably one or a few molecules that have the desired properties. The probability of getting there from a population with one genetic operation is very small.

Even worse is the case when the expert user decides to create molecules from scratch. In this case, there is no set of parent molecules and the new molecules are created randomly. When certain filters are set strictly, the search is just a random search and could take very very long and has nothing to do with evolutionary computation.

This also calls for improvement.

Chapter 5

An automated evolutionary algorithm for drug design

In this section, an automated evolutionary algorithm for drug design will be proposed that is based on a multiple objectives fitness function based on the principle of Pareto domination. To make it as flexible as possible, the algorithm is designed to work for any set of property functions, but for testing, it makes use of the chemical filters of the Molecule Evaluator. Moreover, the algorithm will serve and be presented as an extension to the Molecule Evaluator and will add a fully automated evolutionary algorithm as functionality to the user-controlled evolutionary algorithm of the Molecule Evaluator.

5.1 Main evolution cycle

To improve the Molecule Evaluator, the main evolution-loop as given in algorithm 2 could be replaced by algorithm 3. As can be seen, the main structure can be kept about the same. The only change actually is that a big part of the body is replaced by the function `evaluate($P'(t)$, filters, constraints, λ)`. It is this function that contains the automated evolutionary algorithm within the manual evolutionary algorithm of the Molecule Evaluator which replaces the purely random search for molecules with the property values within the filter bounds.

Algorithm 4 shows the outline of `evaluate($P'(t)$, filters, constraints, λ)`. The population that was selected by the user is taken as initial population of this evolution loop (or is generated randomly if the user did not select any initial population). Then, the evolution loops a number of time until there are λ offspring that comply to the user-set filter-bounds and constraints.

For each iteration, k offspring are generated and added to the population.

Algorithm 3 Adapted structure of the Molecule Evaluator

```
1: for each search do
2:   filters := let user set filter bounds;
3:   constraints := let user set filter bounds;
4:    $\lambda$  := let user select number of offspring;
5:    $P'(t)$  := parents selected from  $P(t)$  by user;
6:    $P(t+1)$  := evaluate( $P'(t)$ , filters, constraints,  $\lambda$ );
7:    $t := t + 1$ 
8: end for
```

Note that the algorithm distinguishes between constraints and filters. Here constraints are hard constraints that every offspring needs to satisfy at all times. A new offspring is accepted if it passes `Accept(o)`. The function `Accept(o)` accepts molecules that comply to the constraints. The function `Accept(o)` is similar to the function `Allowed(o)` of algorithm 2, but it does not take the filters into account.

After the generation of the offspring, the property values of the offspring are evaluated and compared with the user-set filter values. Based on this evaluation, the parents for the next generation are selected by the multi-objective selection procedure `select($P(t) \cup O(t)$)` given in algorithm 5. This selection procedure bases its selection on the filter bounds that were set by the user as explained in section 4.1.3. It selects at least μ individuals from the population P based on the number of individuals that dominate it (i.e. good individuals are dominated by few/zero others, so the less individuals that dominate it, the better an individual is). By keeping at least a population size of μ , the selection procedure should prevent the population to collapse to one or a few individuals (which would disrupt diversity).

5.2 Genetic operators

For the genetic operators, the same operators were chosen as the Molecule Evaluator (section 4.3.3). As this set of genetic operators allows the algorithm to crawl through the whole search space, this set of genetic operators seems to be very fit for our purpose. Especially the broad set of mutation operators is very helpful in this case, as we want to preserve diversity in the population. Using solely recombination would not be very beneficial as this operator tends to disrupt diversity.

Algorithm 4 Automated inner evolution cycle for the Molecule Evoluator

```
1: procedure EVOLUATE( $P$ , filters, constraints,  $\lambda$ )
2:    $t := 0$ ;
3:    $P(0) := P$ ;
4:   while not terminate do
5:      $n := 0$ ;
6:     while  $n < k$  do
7:       select crossover or mutation;
8:       if crossover then
9:          $p_1, p_2 :=$  randomly select two parents from  $P(t)$ ;
10:         $o_0, o_1 :=$  recombine( $p_1, p_2$ );
11:         $o :=$  randomly select between  $o_0$  and  $o_1$ 
12:       else
13:         $p :=$  select one parent randomly from  $P(t)$ ;
14:         $o :=$  copy( $p$ );
15:         $o :=$  mutate( $o$ );
16:       end if
17:       if Accept( $o$ ) AND  $o \notin P(t + 1)$  then
18:          $P(t + 1) := P(t + 1) \cup o$ ;
19:          $n := n + 1$ 
20:       end if
21:     end while
22:     evaluate( $P(t + 1)$ );
23:      $P(t + 1) :=$  select( $P(t + 1)$ );
24:      $t := t + 1$ ;
25:   end while
26:   return  $P(t)$ 
27: end procedure
```

5.3 Evaluation and selection

As mentioned, molecules that comply to the physical filters (constraints) are accepted as offspring. We only accept molecules complying to the physical filters as these filters can filter out many unmakeable molecules and we do not want to waste time on those molecules. For the offspring that are accepted, the selection method is based on the notion of Pareto domination and shown in algorithm 5.

Besides that using the notion of Pareto dominance is probably fairer when it comes to the determination of a performance order, it also is in this case a way to maintain a certain level of diversity. Note that with Pareto domination, two dis-

Algorithm 5 Automated multi-objective selection procedure

```
1: procedure SELECT( $P$ , filters, constraints,  $\mu$ )
2:    $l := 0$ ;
3:   while  $k < \mu$  do
4:     for each  $p \in P$  do
5:        $d_p :=$  number of solutions that dominate  $p$ ;
6:       if  $d_p == l$  then
7:          $Q := Q \cup p$ ;
8:          $k := k + 1$ ;
9:       end if
10:    end for
11:     $l := l + 1$ 
12:  end while
13:  return( $Q$ );
14: end procedure
```

tinct solutions have a higher chance of being incomparable than similar solutions have. With this, it is therefore more likely for similar solutions that they will be dominated (by other similar solutions) than distinct solutions. And thus we hope that this selection method will select a more diverse set than using a weighted sum as fitness function. For further research, it would be nice to test if this effect indeed occurs.

5.4 Variable sized populations

What makes this implementation stand out compared to other evolutionary algorithms is the fact that this allows the population to grow without any limits. No matter how many non-dominated solutions there are, this implementation will not choose between any non-dominated solutions, but keep them all.

As it is up to the user to select the filters that should be taken into account for the fitness, the search can be an optimization of only one objective function, but it could also be more. With such a varying number of objectives, it is impossible to determine a good population size yourself. By letting the algorithm grow the population size, it can grow to the number of solutions needed for a nice Pareto front. The advantage is therefore that the algorithm can keep a nice broad Pareto front and thus there is a big chance that the algorithm ends up with a nice diverse set of molecules with the desired properties.

One might argue that this method will eventually result in huge populations, but we know from experience that this is not the case. In the first few iterations,

the population will indeed grow rapidly, but after a while, offspring will be created that are fit on multiple of the objectives. It is the creation of those offspring that reduces the number of solutions in the population, as they are likely to dominate multiple solutions of the population.

5.5 Parameter tuning

The main parameters that should be tuned are the minimal population size μ and the number of offspring k that will be generated each iteration.

The minimal population size should not be too small, as we do not want the population to shrink entirely when suddenly one non-dominated solution arises. This would ruin the diversity of the population. We do not want to have μ too large either, because that would decrease the selection pressure.

Ideally, the number of offspring k should be chosen such that for each generation, an offspring is created that is non-dominated or (even better) the offspring dominates a solution from the population. This is not an easy parameter to tune, as the number of offspring needed to create such an offspring is a stochastic number which depends also on the state of the evolution. The more evolution proceeds towards the target, the more difficult it will become to create such offspring.

Chapter 6

Experiments

For the experiments, the property values of already existing and well known drugs were used to construct the targets for the test-runs. The goal of the test-runs was of course not only to find the original molecules, but also to find alternative molecules having about the same property values as the drug molecules. Table 6.1 shows the molecules used for testing and table 6.2 shows the property values of the molecules of the test-set.

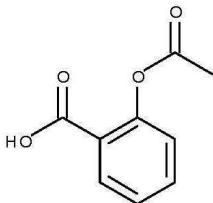
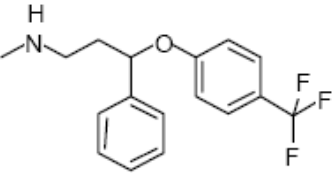
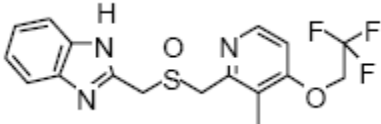
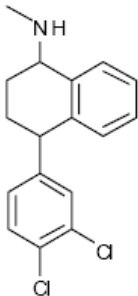
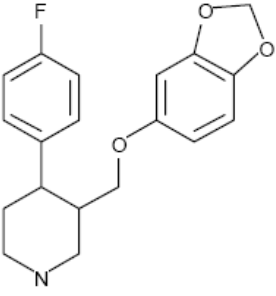
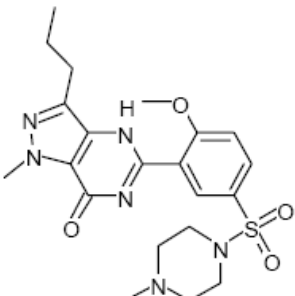
 1. Aspirin (Acetosal)	 2. Prozac (Fluoxetine)	 3. Prevacid (Lansoprazole)
 4. Zoloft (Sertraline)	 5. Paxil (Paroxetine)	 6. Viagra (Sildenafil)

Table 6.1: Set of test molecules

	Aspirin	Prozac	Prevacid	Zoloft	Paxil	Viagra
HD	1	1	1	1	1	1
HA	4	2	5	1	4	10
MW	180.16	309.33	383.39	306.23	329.37	460.55
logP	1.31	4.44	3.61	5.18	3.33	1.98
logS	-2.14	-4.43	-4.03	-5.83	-3.34	-3.58
PSA	63.60	21.26	76.25	12.03	39.72	126.12
RB	3	6	6	2	4	6
AR	1	2	2	2	2	2
AS	2	3	4	5	5	7

Table 6.2: Property values of the test molecules

6.1 Setup

The experiments were run on a Pentium 1.73GHz computer. For each of the test molecules, the algorithm was tested with 10 runs. Each of these runs, the targeted number of molecules was set to 5, the minimal population size was set to 40, and each generation 10 offspring were produced. The evolution process stopped when the desired number of molecules were found, or at generation 1000.

The relatively small number of molecules in the output set was chosen to save time. For the creation of larger output sets, the algorithm would need a larger minimal population size and more generations which would take longer. With the capacity at hands, taking 5 would give a good view of the performance in relatively little time.

For the runs of the experiments, the tables 6.3 to 6.8 show the filter bounds that were used. The filter bounds were set up not to be too strict to also allow molecules other than the target molecules to be outputted. Also, not all filter bounds were set in a similar way for the different molecules, e.g. the MW-filter of Aspirin was set much stricter than the other molecules.

6.2 Results

Table 6.9 shows a summary of the outcomes of the test runs. From this table it can be seen that only one test run managed to find the targeted molecule (Aspirin). Although this is somewhat disappointing, a positive notion is that of the in total 60 test runs in total, only 8 runs failed to find 5 molecules complying to the targeted filter bounds. Moreover, of those failed runs, most did find at least

	Aspirin	Lower bound	Upper Bound
HD	1	1	1
HA	4	4	4
MW	180.16	175.16	185.16
logP	1.31	0.81	1.81
logS	-2.14	-2.64	-1.64
PSA	63.60	58.60	68.60
RB	3	3	3
AR	1	1	1
AS	2	2	2

Table 6.3: Filter settings for Aspirin-like molecules

	Prozac	Lower bound	Upper Bound
HD	1	1	1
HA	2	2	2
MW	309.33	304.33	314.33
logP	4.44	3.94	4.94
logS	-4.43	-4.93	-3.93
PSA	21.26	16.26	26.26
RB	6	6	6
AR	2	2	2
AS	2	2	2

Table 6.4: Filter settings for Prozac-like molecules

	Prevacid	Lower bound	Upper Bound
HD	1	1	1
HA	5	5	5
MW	383.39	378.39	388.39
logP	3.61	3.11	4.11
logS	-4.03	-4.53	-3.53
PSA	76.25	71.25	81.25
RB	6	6	6
AR	2	2	2
AS	4	4	4

Table 6.5: Filter settings for Prevacid-like molecules

one solution. Thus, the algorithm is indeed capable of finding molecules that have property values similar to the targeted molecules.

	Zoloft	Lower bound	Upper Bound
HD	1	1	1
HA	1	1	1
MW	306.23	301.23	311.23
logP	5.18	4.68	5.68
logS	-5.83	-6.33	-5.33
PSA	12.03	11.03	13.03
RB	2	2	2
AR	2	2	2
AS	5	5	5

Table 6.6: Filter settings for Zoloft-like molecules

	Paxil	Lower bound	Upper Bound
HD	1	1	1
HA	4	4	4
MW	329.37	324.37	334.37
logP	3.33	2.83	4.83
logS	-3.34	-3.83	-4.84
PSA	39.72	34.72	44.72
RB	4	4	4
AR	2	2	2
AS	5	5	5

Table 6.7: Filter settings for Paxil-like molecules

	Viagra	Lower bound	Upper Bound
HD	1	1	1
HA	10	10	10
MW	460.55	450.55	470.55
logP	1.98	1.48	2.48
logS	-3.58	-4.08	-3.08
PSA	126.12	121.12	131.26
RB	6	6	6
AR	2	2	2
AS	7	7	7

Table 6.8: Filter settings for Viagra-like molecules

Also from the time-perspective, the algorithm does not perform bad at all. Rang-

ing from an average running time of a little more than 7 minutes (Aspirin) to an hour (Viagra) is not a bad point to start with. Especially when considering the enormous search space. Also from the practical point of view, such a running time is acceptable. However, one should keep in mind that these runs only returned 5 molecules. Further work to speed the algorithm up some more would still be advisable.

There is a clear relation between the complexity/size of the targeted molecule and the running time, maximum population size and the number of generations of the evolutionary algorithm. This is not really very surprising, as the search space (and with that the complexity) grows exponentially with the molecule size. The results of Viagra give however a nice indication of the upper bound of the running time of the algorithm as this molecule has a molecular weight very near to the maximum of 500.

Another interesting notion is that in this case the dynamic population size method does work and that the population size stays relatively low. Figure 6.1 and figure 6.2 show the population size development of a successful run and of a failed run (both are runs of the Aspirin test). In both cases, the population size does not grow beyond control.

Lastly there is the population diversity and the molecules that were returned. Figure 6.3 to 6.8 show the outcomes of three of the test runs. For simple molecules like Aspirin, the algorithm is capable of finding good and diverse solutions. When more complex molecules (like Viagra) are used, this becomes harder. The algorithm tends very much to converge the population to a particular area in the search space, and diversity is lost. Also with more complex structures, the amount of unmakeable solutions increases.

	Aspirin	Prozac	Prevacid	Zoloft	Paxil	Viagra
Runs failed	1	2	1	0	2	1
Runs found target	1	0	0	0	0	0
Avg. generations	331	362	366	315	428	574
Avg. time (h:mm:ss)	0:07:30	0:30:17	0:25:20	0:14:06	0:28:58	1:09:24
Avg. max. pop-size	108	167	207	145	243	463

Table 6.9: Results

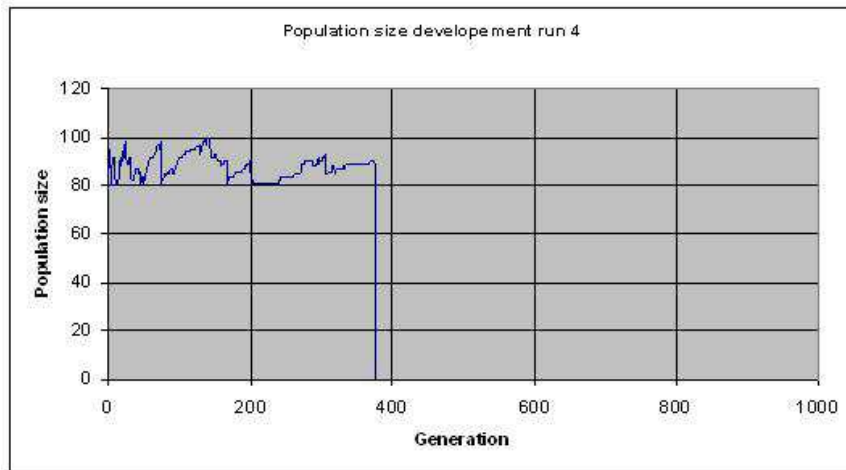


Figure 6.1: Population size development of a successful run.

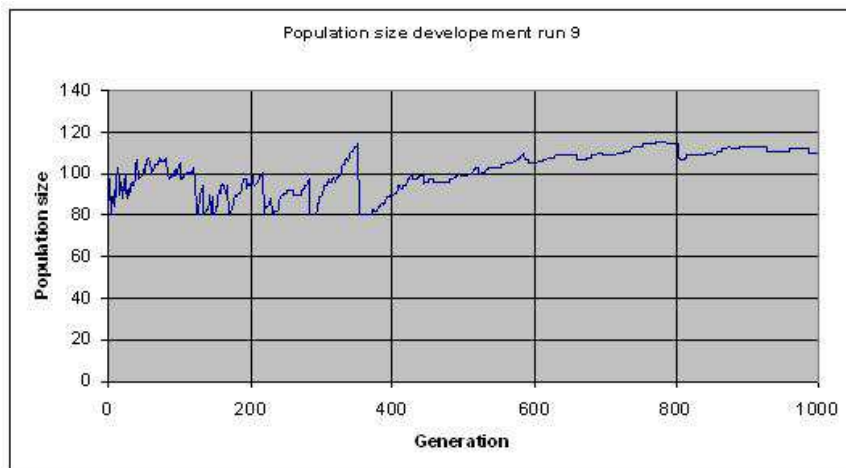


Figure 6.2: Population size development of an unsuccessful run.

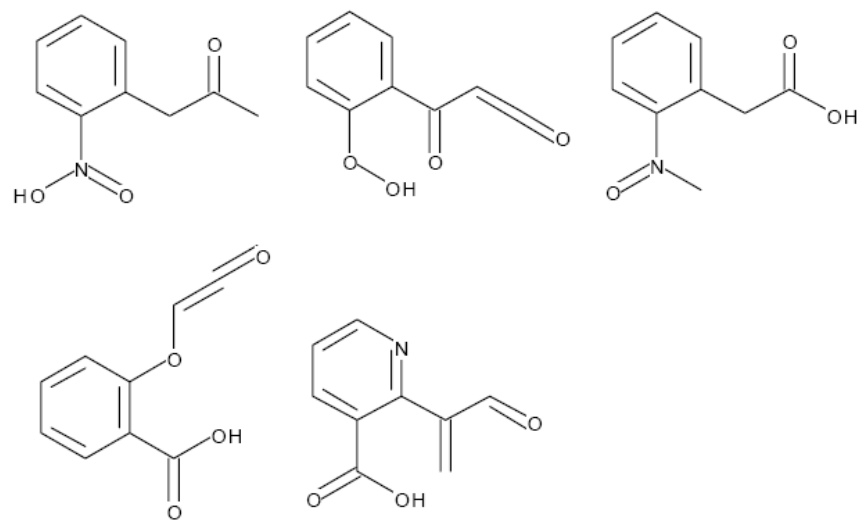


Figure 6.3: Molecules found in run 1 of the Aspirin tests.

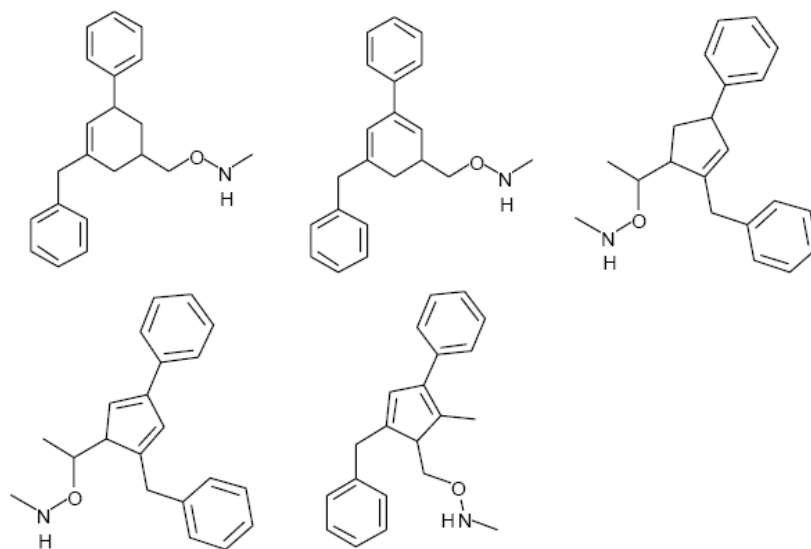


Figure 6.4: Molecules found in run 1 of the Prozac tests.

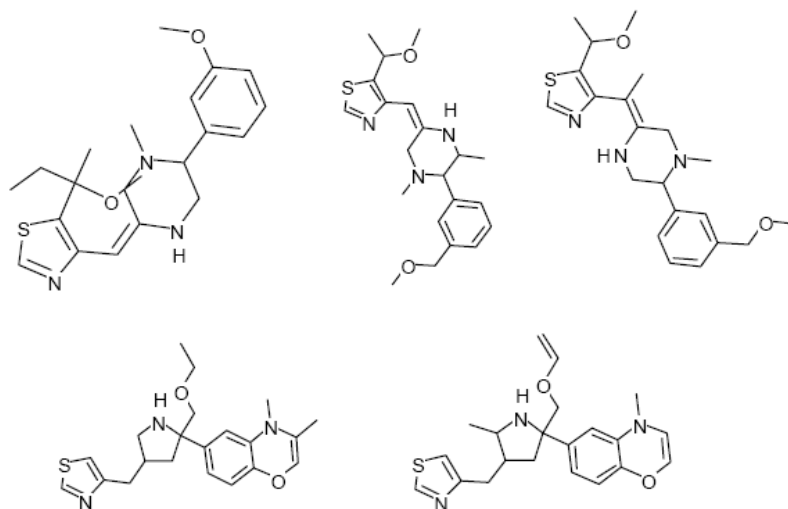


Figure 6.5: Molecules found in run 1 of the Prevacid tests.

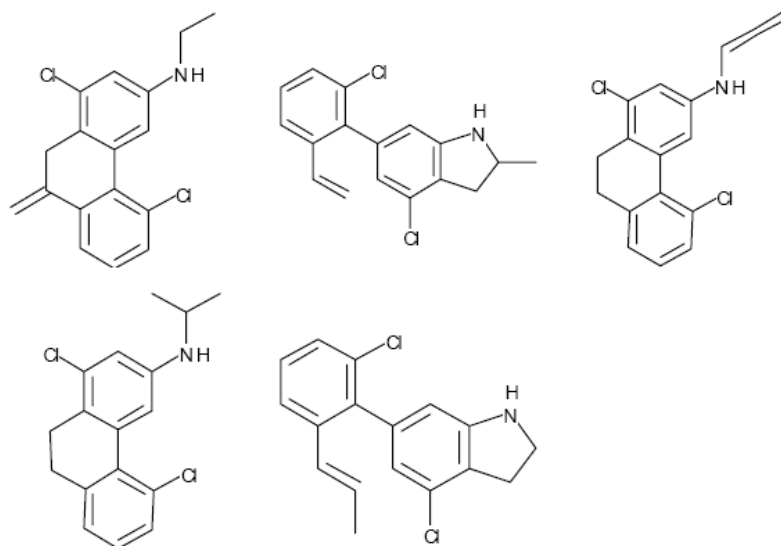


Figure 6.6: Molecules found in run 1 of the Zolofit tests.

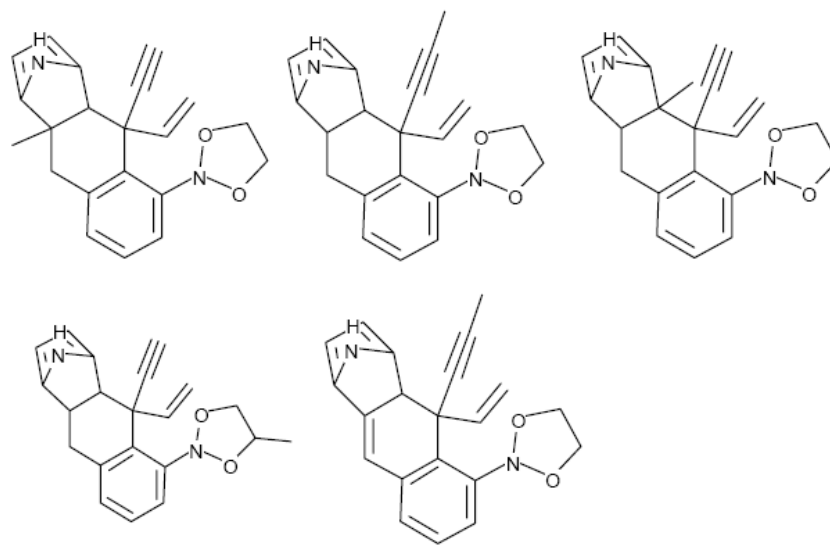


Figure 6.7: Molecules found in run 1 of the Paxil tests.

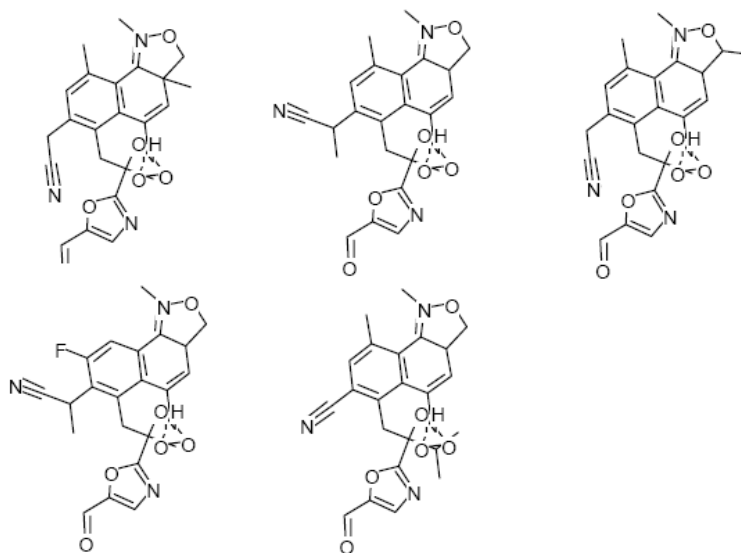


Figure 6.8: Molecules found in run 1 of the Viagra tests.

Chapter 7

Conclusions and outlook

The results show that it is indeed possible to apply the concept of evolutionary computation to find molecules with specific properties. Moreover, the results show that this can be done in a fully automated manner and by using targets for multiple molecule properties to determine the fitness rather than using expert knowledge or similarity measures to determine the fitness.

Multi-objective approach

The particularly new thing about the algorithm proposed in this thesis is the automated selection procedure based on the notion of Pareto domination for the selection of molecules in a multi-objective search. By taking the multi-objective approach, the algorithm should in principle be capable of implementing more or other fitness functions as well and is therefore more flexible. Especially from the practical point of view, this could prove to be very useful as it allows the algorithm also to be used by pharmaceutical companies that implement their own molecule property functions and simulation programs.

Basing the fitness determination on the notion of Pareto dominance is also a first attempt to generate population diversity. By focusing on population diversity, the hope is that a broader part of the search space is covered, and it provides the medicinal chemist with more possible solutions in the ultimate output. The latter is very desirable with such a complex search space, as the expert knowledge will probably always remain necessary in the drug design process.

Variable sized populations

Another matter that makes the evolutionary algorithm presented in this thesis different from the usual applications is the fact that it implements a variable sized population. It is remarkable to see that the population size does not grow beyond

control which is in this case very useful as no choices have to be made between (from the point of view of Pareto domination) equal solutions in the selection procedure. The cause of this is probably that the fitness landscape is very rugged.

Molecule diversity

Although the algorithm is very well capable of finding molecules that comply to pre-defined properties, the population diversity is still a problem and could also use some speed up. The population diversity decreases especially in the cases where more complex molecules were sought and the number of generations was higher.

It seems inevitable to resort to niching techniques to improve the population diversity. This will probably also have effects on the performance of the algorithm (both time and quality). This is something very worthwhile investigating.

Genetic operators

For further research, it would also be very worthwhile to further research the genetic operators, and their effect on the fitness landscape of different property functions. The algorithm presented here has simply adapted the mutation operators of the Molecule Evaluator, but although the genetic operators serve their purpose very well for the Molecule Evaluator, there might be more suitable genetic operators for an automated evolutionary algorithm. Testing various combinations of genetic operators could be very interesting.

Fine-tuning

It would be interesting to see the effect of different minimal population sizes and different numbers of offspring created every iteration (respectively μ and k in algorithm 5). The numbers that were chosen here were picked with only a few tests and heuristical knowledge. A good investigation could prove very beneficial.

List of Figures

2.1	The molecular structures of the two well-known drugs Zoloft and Prozac.	5
4.1	The flowchart of genetic programming (from [8]).	19
4.2	A tree representation as used in genetic programming representing the prefix expression <code>(+ (* (sqrt y) 2) (/ y (if (== y 0) 1 y)))</code>	20
4.3	Three commonly used crossover types, from left to right: subtree exchange crossover, self crossover and module crossover.	22
4.4	An example of the SMILES and TreeSmiles representation of a molecule (from [10]).	26
4.5	The effect of the mutation operators of the Molecule Evuator on molecules.	29
6.1	Population size development of a successful run.	43
6.2	Population size development of an unsuccessful run.	43
6.3	Molecules found in run 1 of the Aspirin tests.	44
6.4	Molecules found in run 1 of the Prozac tests.	44
6.5	Molecules found in run 1 of the Prevacid tests.	45
6.6	Molecules found in run 1 of the Zoloft tests.	45
6.7	Molecules found in run 1 of the Paxil tests.	46
6.8	Molecules found in run 1 of the Viagra tests.	46

List of Tables

4.1	Crossover types used in genetic programming.	22
4.2	Mutation types used in genetic programming.	22
6.1	Set of test molecules	38
6.2	Property values of the test molecules	39
6.3	Filter settings for Aspirin-like molecules	40
6.4	Filter settings for Prozac-like molecules	40
6.5	Filter settings for Prevacid-like molecules	40
6.6	Filter settings for Zoloft-like molecules	41
6.7	Filter settings for Paxil-like molecules	41
6.8	Filter settings for Viagra-like molecules	41
6.9	Results	42

Bibliography

- [1] Smiles - a simplified chemical language. Website. <http://www.daylight.com/dayhtml/doc/theory.smiles.html>.
- [2] D. Acohen. Computational fragment based optimization of leads. Unpublished work. Masterstage Sept. 2004 - Jan. 2006.
- [3] T. Back. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford University Press, Oxford, UK, 1996.
- [4] D.E. Clark and S.D. Pickett. Computational methods for the prediction of 'drug-likeness'. *Drug Discovery Today*, 5(2):49–58, February 2000.
- [5] K. Deb. *Multi-Objective Optimization Using Evolutionary Algorithms*. John Wiley & Sons, Inc., New York, NY, USA, 2001.
- [6] V.J. Gillet. De novo molecular design. In D.E. Clark, editor, *Evolutionary Algorithms in Molecular Design*, pages 49–69. Wiley-VCH, 2000.
- [7] A. Globus, J. Lawton, and T. Wipke. Automatic molecular design using evolutionary techniques. In Al Globus and Deepak Srivastava, editors, *The Sixth Foresight Conference on Molecular Nanotechnology*, November 1998.
- [8] J.R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.
- [9] E.-W. Lameijer, J.N. Kok, T. Back, and A.P. IJzerman. Evolutionary algorithms in drug design. *Natural Computing: an international journal*, 4(3):177–243, 2005.
- [10] E.-W. Lameijer, J.N. Kok, T. Back, and A.P. IJzerman. The molecule evaluator. an interactive evolutionary algorithm for the design of drug-like molecules. *Journal of Chemical Information and Modeling*, 46(2):545 – 552, 2006.

- [11] C.A. Lipinski, F. Lombardo, B.W. Dominy, and P.J. Feeney. Experimental and computational approaches to estimate solubility and permeability in drug discovery and developments settings. *Advanced Drug Delivery Revises*, 46:3–26, 2001.
- [12] Samir W. Mahfoud. *Niching methods for genetic algorithms*. PhD thesis, Urbana, IL, USA, 1995.
- [13] R.B. Nachbar. Molecular evolution: A hierarchical representation for chemical topology and its automated manipulation. In J.R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D.B. Fogel, M.H. Garzon, D.E. Goldberg, H. Iba, and R. Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 246–253, University of Wisconsin, Madison, Wisconsin, USA, July 1998. Morgan Kaufmann.
- [14] D. Weininger. Smiles, a chemical language and information system. 1. introduction to methodology and encoding rules. *Journal of Chemical Information and Computer Science*, 28(1):31–36, 1988.
- [15] D. Weininger. Method and apparatus for designing molecules with desired properties by evolving successive populations. United States Patent US5434796, 1995.