Universidade de Aveiro
Departamento de
Electrónica, Telecomunicações e Informática,
2010

Paulo Miguel
da Silva Gaspar

Optimização de genes para expressão heteróloga
Gene optimization for heterologous expression

Paulo Miguel da Silva Gaspar

Optimização de genes para expressão heteróloga
Gene optimization for heterologous expression

Tese apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários para a obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sobre a orientação ciêntifica do Dr. José Luís Oliveira (Professor Associado da Universidade de Aveiro e investigador no IEETA) e da Dr.ª Gabriela Moura (Professora Auxiliar na Universidade de Aveiro e investigadora no CESAM).

o júri / the jury

presidente / president

Joaquim Arnaldo Carvalho Martins
Professor Catedrático da Universidade de Aveiro

vogais / examiners committee

José Luís Oliveira
Professor Associado da Universidade de Aveiro (orientador)

Gabriela Moura
Professora Auxiliar na Universidade de Aveiro (co-orientadora)

Rui Pedro Lopes
Professor Coordenador do departamento de Informática e Comunicações do Instituto Politécnico de Bragança

agradecimentos / acknowledgements

Elaborar uma tese sobre um assunto algo complexo em cerca de cem páginas é de facto uma tarefa difícil. Compreender em alguns meros parágrafos o reconhecimento e a gratificação por quem me apoiou, é irrealizável.

Agradeço em primeiro lugar ao meu professor e orientador José Luís Oliveira, que me acompanhou, guiou e ajudou durante todo o progresso desta investigação, mostrando-se sempre presente e disponível, e cujos conselhos e apoio foram vitais.

À minha co-orientadora Gabriela Moura, ao investigador Jörg Frommlet e ao professor Manuel Santos agradeço a disponibilidade, paciência e entusiasmo para com o projecto, e também a preciosa ajuda que me forneceram aquando de dúvidas relacionadas com genética.

A todos os meus amigos em Aveiro, especialmente ao meu grupo mais chegado e aos amigos com quem vivo. Um obrigado também ao grupo de bioinformática do IEETA, sobretudo ao Carlos Ferreira, com quem mais proximamente partilhei a experiência de desenvolver uma tese.

Agradeço sobretudo ao meu pai, João, por permitir que eu esteja onde estou, e também ao meu irmão, Pedro, igualmente pelo suporte e apoio incondicional que me deram sempre. Sem eles, não seria o que sou, nem estaria onde estou.

Finalmente, agradeço e dedico esta tese à minha companheira e amiga, Esther del Castillo, pela paciência inesgotável, afecto e ajuda que foi. Foste quem mais me apoiou e mais esteve presente, animando-me e felicitando-me conforme necessário. Obrigado a ti.

Resumo

Com o uso de computadores para assistir investigadores na área da biologia na resolução de tarefas complexas, o seu potencial surgiu como uma ajuda preciosa para alcançar o que está para além das capacidades humanas. Para um biólogo, nos tempos que correm, lidar com um computador é uma tarefa tão trivial como realizar experiências em laboratório. Assim, a capacidade fornecida pela tecnologia computacional, juntamente com as centenas de aplicações e ferramentas de software que já existem, concedem à Biologia um apoio significativo para a investigação e desenvolvimento.

O ramo da Biologia Molecular tem testemunhado um uso crescente destas capacidades tecnológicas, sobretudo nos programas de sequenciação de genomas, que traduzem a informação genética de seres vivos para formatos digitais. Como fruto destes projectos, são gerados grandes volumes de dados de várias espécies, que são disponibilizados. Em consequência, muitos sistemas de bio-informática tem como objectivo analisar estes dados. Novas descobertas e avanços requerem novas ferramentas e técnicas.

Esta tese debruça-se sobre o problema das metodologias de redesenho de genes, estudando e reunindo várias características conhecidas dos genes e o seu impacto na criação de proteínas, na perspectiva das estratégias de manipulação de sequências de genes. Estas características e algoritmos de redesenho devem ser encaixados numa só ferramenta que permita aos investigadores estudar mais apropriadamente os genes e os factores que influenciam as suas sequências. Também objecto de estudo nesta tese é a capacidade de combinar esses factores de forma óptima, num só processo de redesenho.

Abstract

As computers started assisting biology researchers in complex tasks, their potential arose as a precious aid to achieve what was beyond human capacity. In modern times, for a biologist, dealing with a computer is as trivial as working with test tubes in the laboratory. Thus, the power provided by computational technology along with hundreds of software applications and tools that already exist, grant biology a significant support for research and development.

Molecular biology has witnessed an increased use of these technological capabilities, especially with the genome sequencing projects that translate the genetic information from living beings into digital formats. Large volumes of data from various species are, thus, generated and made available. Analyzing that data is now the goal of many bioinformatics systems. Consequently, new discoveries and advancements demand new tools and techniques.

This thesis lays on the problem of gene redesign methodologies, by studying and gathering the available known gene characteristics and its impact on protein production, from the perspective of their sequence manipulation strategies. These characteristics and redesign algorithms should be assembled into a single package tool, to allow researchers to better study genes and all factors that influence their sequence. Also a subject of study is the capacity to correctly and optimally combine those factors into a single redesign process.

Contents

List of Figures

List of Algorithms

Acronyms

AOF Aggregate Objective Function. 34–41, 43, 44, 46, 61

CAI Codon Adaptation Index. 15

DNA Deoxyribonucleic Acid. 3–5, 9, 16, 21, 48, 50

EBI European Bioinformatics Institute. 57

FASTN Fast-N. 21, 50, 67

JRD Joint Requirement Development. 19

JVM Java Virtual Machine. 54

mRNA messenger Ribonucleic acid. 4, 7, 12, 13, 15, 16, 51, 54

MVC Model View Controller. 56

NCBI National Center for Biotechnology Information. 51, 69

PDB Protein Data Bank. 57

RNA Ribonucleic acid. 3

RSCU Relative Synonymous Codon Usage. 16

SD Shine-Dalgarno. 12

Chapter 1

Introduction

1.1 Motivation

Discovering the underlying information of genes, describing protein functionality and interactions, and assessing the effect and significance of the genetic information for cells are some of the most important tasks in the field of Molecular Biology. Ultimately, progress in both genetics and informatics will walk hand in hand, as more developments in biology require even more technological capacity, which by turn will result in more developments.

Eventually, these advancements have a vast impact in human life. The study of genes, its functionality and the effect of the resulting proteins in organisms, are goals that in due course lead to human benefits. Particularly, understanding in detail how genes originate proteins and how to make them being expressed more rapidly or more accurately, may lead to many useful applications, like massive vaccine production. Other studies point to drug design investigation through the understanding of the pathologic organism and the translational machinery of bacteria and parasites. For instance, the Mephitis European project [1] deals with the comprehension of protein synthesis in Plasmodium sp.(the agent of malaria disease). This study will hopefully lead to the identification of novel drug targets to fight malaria. To achieve that objective, new methodologies to study the genetic influence of the malaria genes have to be developed, and thus, new tools to help in that task.

The developments already made in the field of gene decoding and protein synthesis show already a large conception of the factors that are involved in it. Such factors may vary widely and depend on diverse matters. This thesis lays on this problem, by studying and gathering the available known gene characteristics that impact on protein synthesis and the respective strategies for digital sequence manipulation strategies into a single package tool, allowing researchers to better study genes and the variables that influence their sequences, along with a strong capacity to combine these factors in order to assess its weight in the gene translation process. Furthermore, the resulting application shall be used by researchers to investigate the biological process of synthesizing functional proteins, and thus it serves as a motivational purpose.

1.2 Goals

To work with the immense quantities of available genetic data, and to extract detailed statistical information from genomes, sophisticated software tools are required. Moreover,

to manipulate this genetic digital data so that their biological properties become changed in a controlled way requires also mathematical approaches, and therefore even more specific software instruments are needed. Thus, the goal of this thesis is to tackle the problem of gene sequence redesign and optimization for heterologous expression, using algorithmic approaches. Consequently, the main targets include:

- The study and gathering of gene redesign algorithms
- The study and construction of a supportive platform to sustain the algorithms variety and to easily manage them (add, remove, and change)
- The optimization of the process of gene redesign, by allowing a flexible control over the redesign algorithms
- The study of algorithms to perform multiple gene redesign (several simultaneous redesign methods to manipulate a single gene sequence)

Such studies should be integrated into a single platform along with other specified requirements. Hence, the engineering of an application to incorporate all the redesign methods, algorithms, and supportive technology, is also a major goal for this thesis.

1.3 Thesis outline

The remaining chapters are organized according to the following:

Chapter 2 explains the biological context in which this thesis is inserted. A brief description about the biology behind genetic information and common biological concepts are presented. The process of protein synthesis - the main field on which this thesis is focused - is described in an overview of the translation process. Also, a presentation of gene redesign strategies is made along with several methods and algorithms to improve gene sequences in some order. Moreover, some software applications that perform gene redesign are detailed, showing their main advantages and drawbacks.

Chapter 3 details the requirements of the project. The user requirements are presented in a first part, by listing the needs of researchers. The redesign methods that are required in the application, the tools that must be present to assist investigation and usability issues are also described.

Chapter 4 presents the model that is proposed to address the referred problems and challenges. A plug-in system is presented as the solution to aggregate the several different functionalities; an extensive approach to optimization both in mathematical and algorithmic sense is made; the issue of opening and parsing genome files is also addressed. Finally, a global perspective over the developed application is presented.

Chapter 5 shows the conclusion of this research, and a small validation of the application is presented, along with a SWOT analysis and future work.

Chapter 2

Gene context and redesign

The process of building a protein is intricate and has been the target of much research and development in the biology field. As new discoveries are made, new requirements for further investigation emerge. As a result, researchers are continuously in need of new tools to help untying the process of protein biosynthesis.

Massive amounts of genetic data already exist in databases [2], allowing biologists to work with genes just by downloading them. Tasks such as searching for similar genes, comparing several genes or obtaining deeper information about their sequences, are made frequently and require specialized tools. Also, in order to better understand and study the inner-cell process of making a protein, researchers often resort to redesign systems to enhance some characteristics of those genes in order to enhance their expression, either in native or in heterologous cells.

In this chapter, an overview of the biology of protein synthesis, several important methodologies and algorithms, and some software tools for genetic redesign are presented, in order to illustrate the state of the art of gene redesign systems.

2.1 Genetic Information

Every living being is built based on an inherited genetic library that defines every characteristic of its organism, called Deoxyribonucleic Acid (DNA). This code has been created and perfected over millions of years in order to assure the survival of organisms to any environmental pressure.

As a result of this evolution, species gathered in their DNA (genome) all the information they needed to build and reproduce themselves, transmitting that information to their offspring. DNA molecules exists in almost every cell and the information they bear is written through a combination of four nucleotides (also called bases): Adenine, Thymine, Cytosine and Guanine, frequently referred to by their first letter A, T, C and G respectively. A DNA double helix is represented in figure 2.1.

The DNA comprises all the information by which species are defined, including genes. A gene is a fraction of the DNA (and therefore a sequence of nucleotides) whose information can be used by the cell to build a functional molecule such as a protein.

In order to use this coded genetic information, a group of the many machine-like entities in the cell (called enzymes) makes a copy of DNA sequence of the gene in another kind of nucleic acid, the Ribonucleic acid (RNA), in a process called transcription. The copied

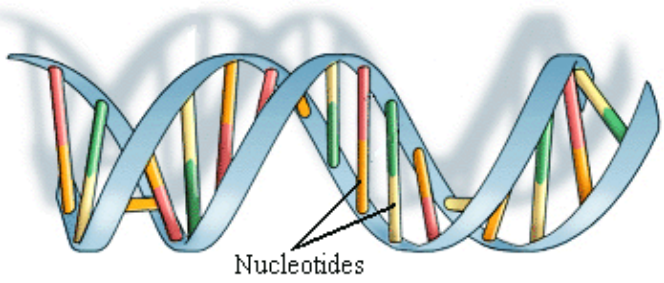

Figure 2.1: DNA double helix and nucleotides representation. Each nucleotide has a different colour, distinguishing the four available (A, T, C and G).

---

sequence, called messenger Ribonucleic acid (mRNA) is then used by another enzyme complex (named ribosome) that reads and decodes it in a process named translation. Each group of three consecutive nucleotides in the mRNA is called a codon, and the ribosome reads one codon at a time (independently) translating it as an amino acid. Those amino acids are added to a forming chain, producing what is called a polypeptide. The complete process is illustrated in figure 2.2.

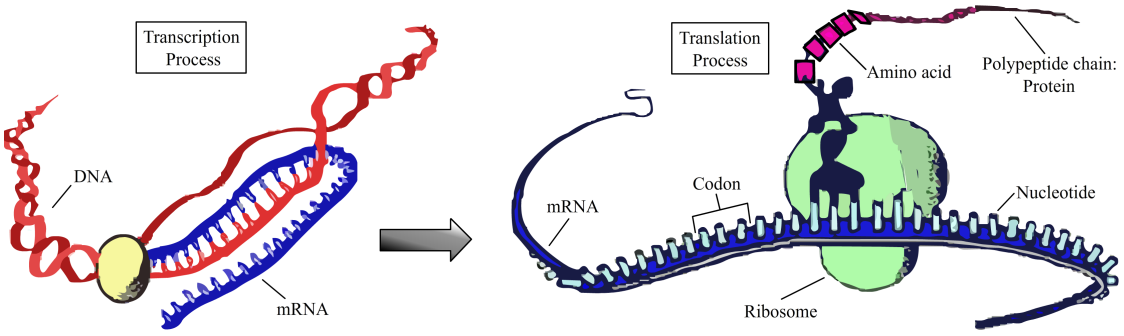

Figure 2.2: Transcription and Translation processes. First the DNA is copied into mRNA. Then a ribosome runs the mRNA, decoding each triple of nucleotides, called codons, into amino acids. The resulting chain will be the protein.

The nucleotide sequence of a valid mRNA sequence isn't in any way random. It always starts and ends with specific codons called start and stop codons that, with some exceptions, are the same in all species. Also, as a codon is a group of three nucleotides, the complete sequence has a number of nucleotides multiple of three. The same codon always codes for the same amino acid but, as there are four different nucleotides and a codon is made of three nucleotides, there are four to the cube (64) different codons and only 20 different amino acids. Therefore, there are different codons that code for the same amino acid (named synonymous codons), conveying redundancy to the genetic code. Stop codons are the only ones not coding for an amino acid, they just indicate the end of the sequence.

Normally, still during translation, the growing polypeptide is subject to a very important folding process, acquiring its final shape according to its amino acid properties. When this step is reached the protein is ready [3]. However, the incorrect folding of a protein may lead to its incorrect functioning, accumulation and aggregation with increased toxicity to the cell.

As an example, some human diseases, such as Alzheimer and Parkinson, are associated to protein aggregation in neurons.

Related species have evolved from common ancestors, and so they share much of their DNA sequences. When two genes from different related species share enough similarity it is reasonable to assume that they derived from a gene of their common ancestor. These genes are called orthologs and usually originate proteins that have the same function [3].

2.2 Gene Redesign

An optimizing system implies that a given object of study can be, in some extent, improved so that the result suits better a certain application. In the context of gene expression, optimization is the enhancement of a nucleotide sequence, by substitution of one or more codons by their synonymous, with the purpose of improving the protein biosynthesis process (increasing the amount of protein or its quality). For instance, when an unwanted group of codons is present in a sequence [4], one of those codons could be replaced by one synonymous, preserving the sequence codification but eliminating the unwanted region, as illustrated in figure 2.3.

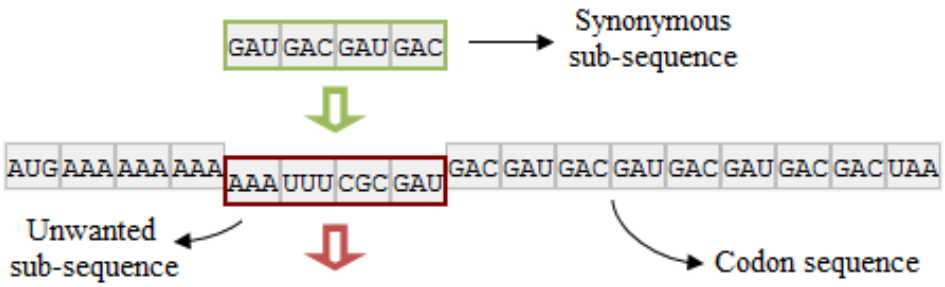

Figure 2.3: Sequence redesign example where an unwanted region is replaced by a synonymous sequence, according to the standard genetic code (see appendix A).

Ultimately, manipulating a sequence involves redesigning it without changing the original outcome. It is possible to optimize a sequence of a gene by simply replacing its codons by synonymous ones, because this still allows eliminating or adding characteristics to the sequence that will improve its translation, without changing the amino acid sequence. On the other hand, genes are expressed in the laboratory using organisms different from those that originated the genes in the first place, in a process called heterologous expression. The main reason for this is to use species that are well adapted to laboratorial conditions and for which there are techniques for genetical manipulation. For instance, cloning a human gene for its expression using human cells is not trivial, so it is typically transformed into other organisms like bacteria to ease its manipulation. However, the frequency of each codon on the entire genome of a gene in the expression host is rarely the same as their natural host, because different organisms often have different preferences for codons to code each amino acid. Therefore, placing a gene in a heterologous host for its expression could result in a malformed protein as a consequence of a translational pace different from that of the natural host, since codons are interpreted at rates that differ from those of the gene in the native expression system [5, 6].

Therefore, improving the characteristics of a gene by codon synonymous replacement must take into account which synonymous exists for each amino acid in the expression host and not the natural host. Also, to make the translation of a gene in a heterologous organism more accurate (and thereby improving the chances to have a correctly folded, functional protein), the rates of translation of each codon in the expression host should closely match the ones in the natural host, and therefore, codons that code for the same amino acid and with frequencies close to the natural host ones should be used [5]. Of course this choice of codons would prevent other optimizations, since any other change in the sequence would interfere with the previous codon selection.

Hence, several characteristics of the genes can be manipulated to improve expression without compromising the amino acid structure, thus the redesigned gene becomes a synonymous sequence of the original one, by using synonymous codons available in the expression host. Such characteristics are presented in the following sections along with algorithms that have been related to accomplish gene optimization.

2.2.1 Codon Usage

As described before, different organisms frequently have different preferences as to codon use, with some codons being more used than others. This distinction is employed by the organism when producing proteins, by having codons in the gene sequence that range from very common to very uncommon. There is a relationship between the frequency of each codon and the speed of its decoding: the more frequent codons are decoded into amino acids faster than the less frequent codons. Therefore, when in translation, regions of the coding sequence with infrequent codons take longer to translate, allowing the protein to assume its shape [7]. Also, more frequent codons are commonly found in highly expressed genes, as a strategy to optimize efficiency by granting a faster synthesis of proteins in periods of high growth rate. In more complex proteins, codon usage serves as a speed regulator to ensure correct folding by using clusters of codons that have slow or fast decoding rates for slower or faster translations, granting the protein the necessary time to fold correctly [8–10].

The codon usage of a genome can be determined by counting the number of times each of the 64 codons appears. This shows how frequently each codon is used, particularly in relation to other codons that code for the same amino acid. With this codon usage table, a sequence can be mapped according to the usage of its codons, revealing which zones are decoded faster or slower.

Redesigning a sequence according to the codon usage is essentially the process of replacing each codon by the synonymous codon that has the best target frequency. Hence, to maximize the codon usage of a sequence, choose among the synonymous of each codon the one that is more used in the whole genome of the species. This increase in codon usage would lead to faster decoding in the ribosome, generating the protein faster. As a consequence, however, the protein might not fold correctly, since some zones could need to be translated more slowly. Moreover, when redesigning for codon usage improvement, one has to take into account that the gene will be expressed in a heterologous organism, and therefore the frequency of each codon and which codon codes for which amino acid might differ.

For instance, redesigning a human gene for increased speed of translation when expressing it in Escherichia coli (bacteria), is replacing the codons of the gene sequence by synonymous codons that code for the same amino acids in the bacteria, with the higher codon usages. For slower translations, similarly, the synonymous codons with lower codon usages should be

used. An example of the codon usages of an organism can be seen in table 2.1, where the frequencies of each codon in the genome of that species are presented.

Table 2.1: Codon usage table (with frequencies per thousand) for Escherichia coli.

UUU	22.1	UCU	8.7	UAU	16.5	UGU	5.3
UUC	15.8	UCC	9.0	UAC	12.3	UGC	6.4
UUA	13.8	UCA	8.2	UAA	1.9	UGA	1.1
UUG	12.9	UCG	8.8	UAG	0.3	UGG	15.3
CUU	11.4	CCU	7.3	CAU	12.8	CGU	20.3
CUC	10.5	CCC	5.6	CAC	9.3	CGC	20.9
CUA	3.9	CCA	8.4	CAA	14.6	CGA	3.9
CUG	50.9	CCG	22.5	CAG	29.5	CGG	6.4
AUU	29.4	ACU	9.1	AAU	19.1	AGU	9.4
AUC	23.8	ACC	22.7	AAC	21.6	AGC	16.0
AUA	5.6	ACA	8.2	AAA	34.0	AGA	3.0
AUG	27.1	ACG	15.1	AAG	11.1	AGG	1.9
GUU	18.0	GCU	15.4	GAU	32.8	GGU	24.1
GUC	14.7	GCC	25.2	GAC	19.2	GGC	27.9
GUA	10.9	GCA	20.7	GAA	39.2	GGA	9.0
GUG	26.1	GCG	32.2	GAG	18.9	GGG	11.9

2.2.2 Codon Context

Likewise codon usage, other studies show that many organisms also have preference for specific pairs of consecutive codons [11]. This preference means that some pairs of codons are more frequent in the genome than others, and thus some codons ought to be followed more often by specific codons. Therefore, the number of times a pair of consecutive codons appears in a genome can also be counted to build up a two-codon context table. This table would reflect how many times each codon appears followed by every other codon. A pair of codons with a higher frequency means that that pair is present more often in the genome than pairs with lower frequencies.

This frequency reflects the predilection of some pairs by organisms with the purpose of regulating the process of protein biogenesis. It is believed that the nucleotide bases around each codon affects mRNA translation, for instance, several phenomenon have been reported to be influenced by the codon context of genes, such as the termination efficiency (see section 2.2.5) [12] and decoding accuracy. Unlike codon usage that reflects the selection of some codons to enhance efficiency in translation (the speed of translation), codon context is believed to influence the accuracy with which the translation is made [13]. When visualizing the matrix of pairing between each of the 64 codons, if the frequencies of each pair are colour-coded according to its value, some patterns become visible in the matrix, indicating that codon pair preference is not random. Some applications already offer the ability to show this colour-matrix, like Anaconda[1] (illustrated in Figure 2.4).

Moreover, when this colour-code is used to map a gene sequence, one can easily identify

[1]Software package developed by the University of Aveiro bio-informatics lab to analyse genomes open reading frames. The software can be found at http://bioinformatics.ua.pt/applications/anaconda [14]

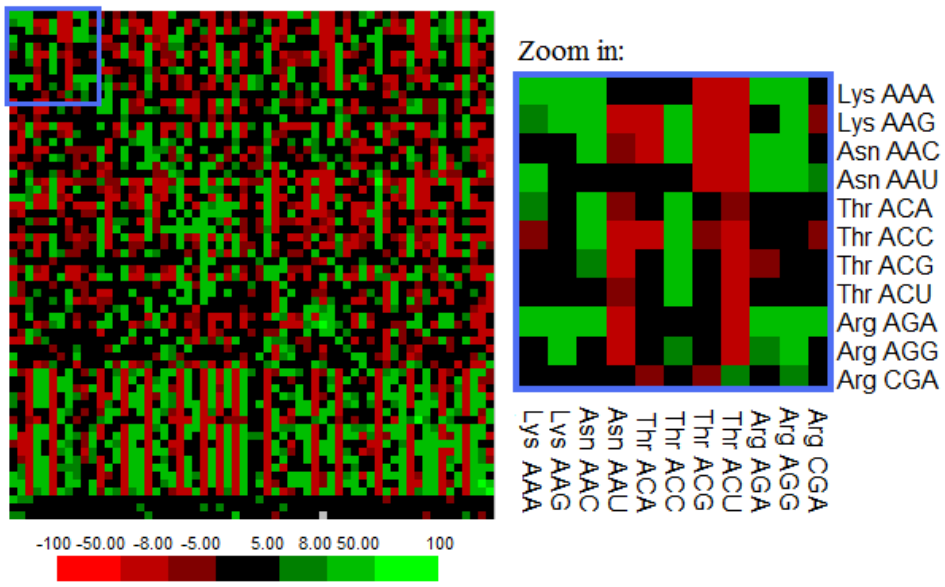

Figure 2.4: Colour-coded codon context matrix of an Homo sapiens taken from Anaconda, where each row and each column represents a codon, and the intersection between a row and a column is the matrix point that represents the codon context value of that pair of codons. Therefore, there are 64x61 points in the matrix, representing each one of the codons (in the 61 columns, since the stop codons are the last and so cannot be followed by another codon) followed by another codon (in the 64 lines). The context is represented by a colour scale ranging from green to red, where green represents a high codon context, and red represents a low codon context. The matrix colour distribution is clearly non random.

---

zones of consecutive codons with the same colour. For instance, a zone of green codons (as shown in Figure 2.5) indicates a streak of high context codon. As a result, increasing or decreasing the codon context in a gene might prove interesting and therefore, used to optimize gene sequences for accuracy improvement.

Figure 2.5: Hypothetical gene sequence coloured according to the codon context matrix. If one codon and its following codon are a very frequent pair in the genome, the first codon appears in green. This colouring allows a context analysis of the gene, inferring zones of high or low codon context. In this illustration there are clear zones of high codon context (a few consecutive green codons) and of low codon context (a few consecutive red codons).

Thus, optimizing according to codon context is finding the synonymous codons that maximize the frequencies of all pairs of codons in the sequence. However, unlike codon usage, a simple algorithm that runs through the sequence choosing for each pair the best synonymous pair that would maximize the codon context, wouldn't always return the best result since changing one codon influences two pairs. Hence, dynamic programming is a very good approach to obtain the sequence with an optimized codon context, since it can deterministically

find the best sequence, and accomplish that in linear time. The algorithms 2.1 and 2.2 are presented as a possible solution.

Algorithm 2.1 Codon Context Optimization

$MaxCodonContexts \leftarrow CalculateBestCodonContexts(Sequence)$ {See Algorithm 2.2}
$N \leftarrow GetSequenceLenght(Sequence)$
$Max \leftarrow \max\{MaxCodonContexts[N][k]\}$, for all k
$OptimizedSequence[N] \leftarrow c \leftarrow j$, such that $MaxCodonContexts[N][j] = Max$
for $i = N - 1$ to 1 **do**
  for each synonymous j coding Sequence[i] **do**
    if MaxCodonContexts[i][j] + GetCodonContext(j, c) = Max **then**
      $OptimizedSequence[i] \leftarrow c \leftarrow j$
      $Max \leftarrow Max - GetCodonContext(j, c)$
      break
    end if
  end for
end for
return OptimizedSequence

Algorithm 2.2 Auxiliar Function: Calculate Best Codon Contexts for each codon

$CC[1][j] \leftarrow 0$, for each synonymous j that encodes Sequence[1]
for $i = 2$ to SequenceLenght **do**
  for each synonymous j that encodes Sequence[i] **do**
    $CC[i][j] \leftarrow \max\{CC[i-1][k] + CodonContext(j, k)\}$, for all synonymous k of Sequence[i-1]
  end for
end for
return CC;

The basic mechanism of this algorithm is calculating the best codon context between every combination of two codons (among all synonymous possibilities), and summing the best results obtained with the previous pair. Then, when reaching the end of the sequence, the algorithm will run the sequence again, backwards, choosing only the codons that maximized the sequence for codon context in the first run. With this method, the algorithm avoids verifying all possible combinations of synonymous sequences by recording only the best results in each step, and then choosing the combination that maximizes the overall codon context.

2.2.3 GC Content

Two of the four nucleotides of DNA molecules are Guanine (G) and Cytosine (C). The GC content of a gene is the proportion of Guanine and Cytosine nucleotides in its sequence. DNA molecules with high GC content are believed to be less unstable, because the AT (Adenine and Thymine) pairing in DNA molecules is destabilizing [15]. Also, higher GC contents lead to the destruction of the cell by its own mechanisms [16]. Moreover, when expressing a gene in a heterologous host, it might be useful to harmonize the GC content of the gene with that

of the expression host (making it similar) in order for the gene to become more similar to the genome of the species used to produce the protein (Figure 2.6).

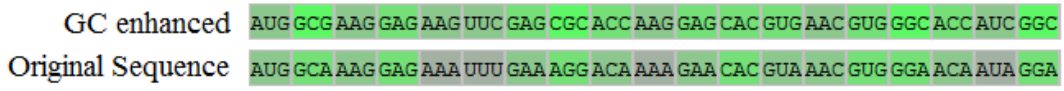

Figure 2.6: Comparison between a native and GC enhanced synonymous sequences. The original sequence codons have less G and C than the enhanced sequence. The colour code shows greener codons when they have more G and C nucleotides.

Calculating the percentage of GC content is the trivial operation of summing the number of G and C nucleotides in the sequence and dividing by its total number of nucleotides. So, for instance, the enhancement of a gene to maximize its GC content can be achieved by replacing each codon of its sequence by the synonymous codon that has the highest amount of G and/or C. As this optimization only depends on each codon, the simple replacement of every codon would do the job. On the other hand, if the goal is to redesign the sequence to have a GC content that matches some other genomic GC percentage, the optimization relies on all the codons of the sequence to guarantee the best result, that is, the synonymous sequence whose percentage of GC is closer to the one of the host genome. Achieving that requires a more complex algorithm (proposed in chapter 3). However, a rough estimation can also be made iteratively, by successive approximation of the GC content when selecting the codon synonymous: for the first codon, select the synonymous codon that more closely matches the required GC content, and record that GC content; for the next codons, do the same, always taking into account the recorded GC content.

2.2.4 Codon usage harmonization

As mentioned in the beginning of the chapter, genes are usually expressed in organisms different from the native ones, for practical purposes, since tools and methods are available for manipulating a set of organisms in the laboratory more easily than others, and also because some species are well adapted to laboratorial conditions. For instance, one species that is frequently used as an expression host is the bacteria *Escherichia coli*, as it can be easily manipulated to produce proteins from other hosts and thus allowing the mass production of heterologous proteins, such as insulin. Therefore, there is interest in manipulating genes so they can be correctly expressed in an heterologous expression host, that is, the proteins formed from that gene are correctly formed and folded in a functional way. However, codon usage tables of different organisms are often different, meaning that codons that are frequent in some species (and thereby decode faster) might not be frequent in other. Hence, expressing a gene in a heterologous host without modifying it to better suit the codon usage of the native host might lead to non-functional proteins due to dissimilar translation rates. For instance, image 2.7 illustrates three different synonymous sequences of a gene from the species *Aquifex aeolicus*, two of them expressed in a heterologous host, showing the differences in codon usage (illustrated with colours).

Gene redesign through harmonization is the process of approximating the usage preferences of each codon to that of the host species, minimizing the total difference in codon usage between the gene in the native organism and the gene in a heterologous host. In order

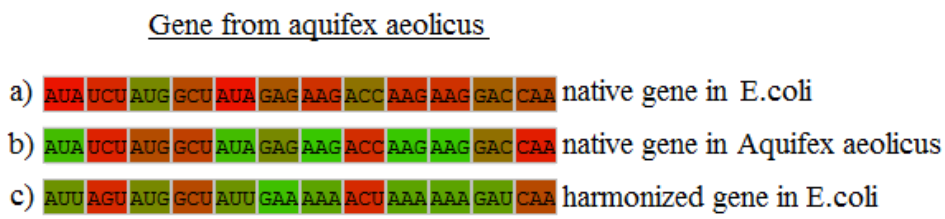

Figure 2.7: Illustration of three synonymous sequences of a gene from *Aquifex aeolicus*, coloured according to the codon usage. In a) one can see the original sequence mapped with codon usage preferences from another organism (the *Escherichia coli*) without being harmonized. In b) the original sequence is shown in its native host, and thus, its colouring illustrates the native codon usage. Finally, in c) the sequence was harmonized using the *Escherichia coli* codon usage table, to match its original codon usage (in the native genome).
There is a clear difference between the codon usages of the sequences in a) and b), meaning that the gene wouldn't decode in *Escherichia coli* in the same rate as it decodes in its native organism. However, in c) the gene sequence is redesigned so that its codon usage is more close to the original one in b), thus harmonizing the gene codon usage.

to achieve that compatibility between foreign gene and host species, each codon of the sequence must be replaced by its synonymous whose codon usage (in the expression host) most closely resembles the one of the original codon usage (in the native host). The algorithm 2.3 demonstrates one approach to accomplish harmonization of a gene to an expression host.

Algorithm 2.3 Optimization through Harmonization

for each codon i in the Sequence **do**
  $BestApproximation \leftarrow \infty$
  for each synonymous j of i **do**
    if $|CodonUsageNative(i) - CodonUsageHost(j)| < BestApproximation$ **then**
      $BestApproximation \leftarrow |CodonUsageNative(i) - CodonUsageHost(j)|$
      $SelectedCodon \leftarrow j$
    end if
  end for
  $OptimizedSequence[k : Sequence[k] = i] \leftarrow SelectedCodon$
end for
return OptimizedSequence

2.2.5 Out-of-Frame Stop Codons

The ribosome (the enzymatic complex that catalyses the translation) sometimes slips a few nucleotides of the coding sequence for several reasons. This slippage provokes errors in translation as the ribosome proceeds the decoding in the wrong 3 letters frame, and so the codon sequence becomes shifted and the message misinterpreted. Not aware of this problem, the ribosome only terminates translation when the next stop codon is reached. One way to avoid this issue is placing stop codons in out-of-frame positions through out the sequence, so that any eventual out-of-frame translation will stop as soon as possible (see figure 2.8) [17].

Normally, this procedure is accomplished through the substitution of some two codons for

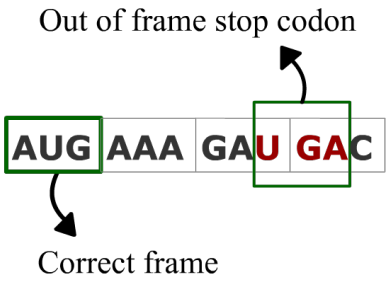

Figure 2.8: The correct frame sequence is the one the ribosome should translate. However, sometimes the slippery nature of the mRNA causes the ribosome to frameshift and thus decode in the wrong frame. Replacing codons so that there is an out-of-frame stop codon between them is a strategy to avoid useless resource consumption in the cell.

---

synonymous that would form an out-of-frame stop codon (also known as hidden stop codons). Not all pairs of codons can be replaced to introduce a hidden stop codon, but maximizing the number of hidden stop codons can be achieved with an algorithm similar to that already explained for codon context, since the optimization of each codon depends on its following and preceding codons, and therefore requires an analysis of all synonymous possibilities for each pair [18].

2.2.6 Repetitions Removal

One of the causes of ribosome slippage is the existence of long chains of repeated nucleotides, or repeated patterns of nucleotides. Such sites can become deleterious to the cell if the ribosome tries to translate them and a frame shift occurs. Therefore, zones with nucleotide repetitions should be avoided by replacing one or more codons inside the repeated region by synonymous that would break the repetition string [19]. Preventing repeated nucleotides can be easily attained by counting repeated consecutive nucleotides, and replacing them, for instance, when finding more than five repeats. The algorithm 2.4 serves as an example. It might not be the best algorithm, and lacks other stopping criterion, but otherwise states a possible solution.

Also of interest is to prevent the use of the same codon to code a consecutive repeated amino acid (repeated codon sequence). The codon repetition optimization can be achieved likewise, counting codons instead of nucleotides.

2.2.7 Deleterious Motifs Removal

Analogous to the repetitions, other sequences need to be avoided due to their deleterious nature. Several of such sequences are well defined as specific strings of nucleotides. For instance the Shine-Dalgarno (SD)[1] sequence is UAAGGAGGU, and the Kozak sequence is GCCACCAUGG. SD and Kozak sequences are known to interact with the ribosome outside of the reading slot, and to interfere with the way the mRNA is read by the ribosome, eventually

[1]Shine-Dalgarno are specific nucleotide sequences in a gene. They have been shown to reduce translation of a sequence, and thus, should be avoided [20].

Algorithm 2.4 Optimization by Repetition Removal

```
repetitionBeginIndex ← 1
repetitionEndIndex ← 1
replacementOccured ← true
while replacementOccured do
  replacementOccured ← false
  for i = 2 to SequenceLength do
    if Sequence[i] = Sequence[i-1] then
      repetitionEndIndex ← repetitionEndIndex + 1
    else
      repetitionBeginIndex ← i
      repetitionEndIndex ← i
    end if
    if repetitionEndIndex − repetitionBeginIndex ≥ 5 then
      ReplaceOneCodonInside(repetitionBeginIndex, repetitionEndIndex)
      replacementOccured ← true
    end if
  end for
end while
```

delaying it and causing increased error rates. Thus, these sequences should be avoided in all three possible frames of the mRNA [20, 21].

Taking an approach like the one used for the repetition removal would suffice to remove any type of these sequences. Removing several types simultaneously requires a more complex algorithm, that will be discussed later.

2.3 Software available for gene optimization

Aside from studying the biological process involved in translation and its regulation by the cell the fundamental objective of gene redesign for optimized translation is the enhancement of protein expression. That means synthesizing proteins faster, in higher amounts and without compromising its functionality. Research fields benefiting from this improved expression include industrial protein production for therapeutical use, higher quality reagent production for research, biopharmaceutical and biochemical studies, and even vaccine production [22, 23]. Consequently, a large number of gene optimization systems have been built to deal with that demand, many of them from commercial companies [24]. The major focus of some of these systems is to increase protein expression, despite the knowledge that other factors might influence its accuracy, with profound effects on the usability of the final result of protein biosynthesis [5, 18, 25].

In this section, relevant gene optimization systems will be presented and discussed. Such systems implement some of the previously presented algorithms and methods for gene redesign along with other optimization characteristics. Furthermore, their main limitations will be addressed as a mean to avoid them.

2.3.1 Gene Composer

Gene Composer offers a gene development studio with a modular work-flow design and with several user friendly graphical interfaces (see figure 2.9) [26]. Though it is more than a gene optimizer, it accommodates some sequence redesigning features such as removal of nucleotide and codon repetitions, removal of several kinds of unwanted sub-sequences, introduction of hidden stop codons, GC content personalization, among others. Other useful features are brought together in Gene Composer that make it a generalized gene redesign software, like allowing sequence alignments[1], showing the protein secondary structure[2], and opening a gene in several file formats [26]. Furthermore, several codon usage tables from various species are brought along with the application, but others than those must be calculated elsewhere and introduced manually, since it isn't allowed to open a complete genome and make the calculations from there. Gene redesign and back-translation (building a sequence of codons starting from the amino acids sequence) tasks are made resorting to the common method of generating several thousands of synonymous sequences and then choosing one that best fits all requirements (GC content of the host species, etc.).

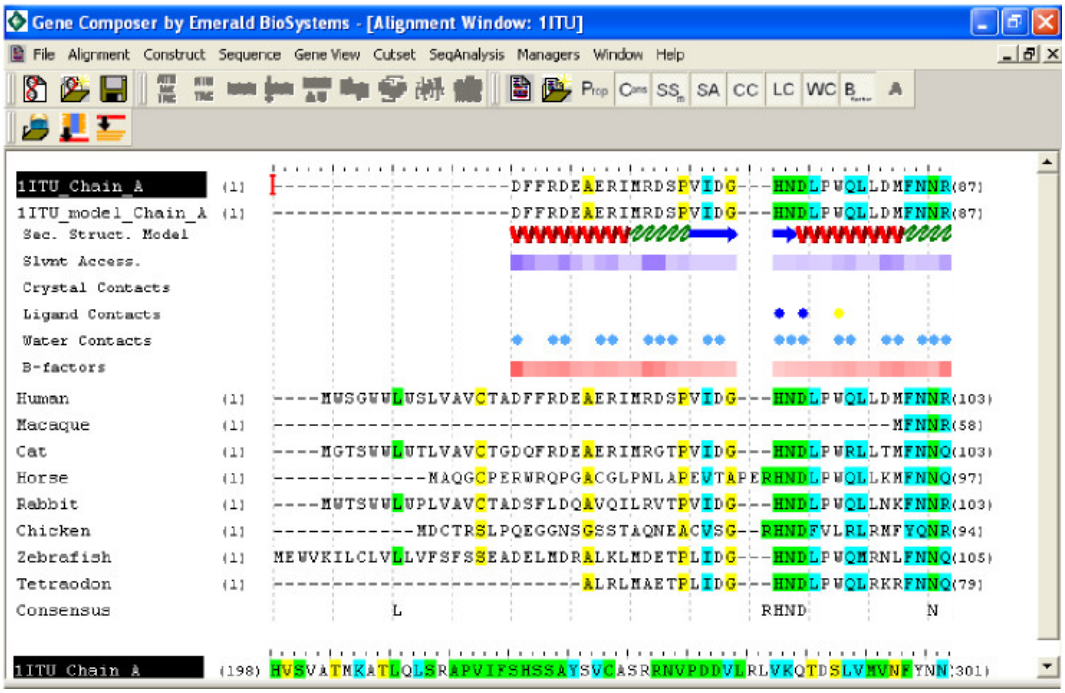

Figure 2.9: Screen-shot of the gene composer main window. A protein is aligned with several orthologs from other species. Also other informations about the amino acids are shown, like chemical properties. Image taken from [26].

Some of these features, however, show little flexibility to the user's needs, such as not

[1]Consists of a process where several sequences are aligned introducing gaps between codons so that the number of common codons in the same position is maximized. Large common zones are discriminated thereby.

[2]When the polypeptide (sequence of amino acids synthesized by the ribosome) starts the folding process to become a complete protein, the chain shapes itself assuming three-dimensional forms called structures. The secondary structure is the local 3D pattern of a segment formed by the interactions between amino acids of the sequence.

allowing the complete customization of more parameters when back-translating the amino acid sequence. Also other gene redesign techniques like codon context optimization are not available. Although other redesign techniques are present, they miss some more complex functionality, for instance, when removing nucleotide repetitions, the removal process may generate other nucleotide repetitions, since it does not account for the possible outcome of their change in the first place. Overall, Gene Composer is a very complete gene manipulation studio, not only directed to sequence optimization but integrating many commonly used tools for gene analysis. Nevertheless, some features do not give the user many possibilities for parameter input and overall transparency, and since it doesn't use separate processes, the application blocks when performing time consuming tasks until they finish.

2.3.2 Gene Designer

Gene Designer aims at bridging over the faults in other optimization systems while showing a robust user-friendly interface. By implementing a multiple parameter optimization algorithm based on codon usage tables, the Gene Designer is able to produce results based simultaneously on Codon Adaptation Index (CAI), codon usage, unwanted motif removal, repeats removal, mRNA secondary structure filtering, restriction sites introduction or removal, orthologs comparison, etc [27].

Gene Designer bases its optimization in a simple algorithm that creates several synonymous sequences based on favourable codon usages and then, for each sequence, the algorithm manipulates the sequence to meet other optimization requirements, accepting changes upon finding a better sequence. Additionally, the algorithm takes orthologs into account, giving the option to optimize a sequence to approximate it to another sequence. However, the flow of the algorithm doesn't keep record of anterior changes, meaning that after each redesign the resulting sequence serves as input for the next optimization method, disregarding the changes of the first when making further codon manipulations. The same happens for the next optimization steps in the work-flow, as illustrated in figure 2.10 [27].

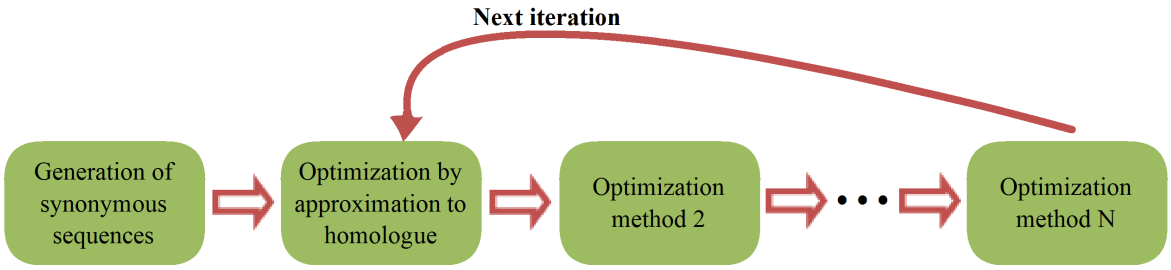

Figure 2.10: Gene Designer Optimization process showing every optimization method acting upon the result of the previous one.

The algorithm [4] eventually converges to sequences that more closely match the optimization requirements than the original one when applying numerous iterations. Yet, as each optimization method can "destroy" the previous one, the last methods always have priority over the first ones. Furthermore, the application is not flexible to other kinds of optimizations, requiring the rebuild of the algorithm if another optimization strategy is needed.

Generally, this application offers many tools to construct and optimize sequences, but lacks other tools like sequence alignment or the visualization of protein secondary structures.

The major illation to take is the need of an expansion support for the optimization process, since new gene optimizing methodologies might be required that the application does not support, and remaking the algorithm for each new method is not a good option.

2.3.3 GeneDesign 3.0

Some applications are available online that facilitate the access without the need for local installation. Such applications (also called web applications) have some advantages like remote processing instead of relying on the local machines of each user, thus usually supplying more processing power. Also, they are easier to update without having to redistribute the application (as with desktop applications) and the set of available genes and genomes is generally larger and more up to date.

GeneDesign[1] is one example of such service, which encompasses a set of web applications to manipulate gene sequences (figure 2.11). The 3.0 version of this software offers some tools like adding and removing restriction sites[2] from a given protein coding nucleotide sequence, making reverse translation from an amino acid sequence, or displaying a chart of the Relative Synonymous Codon Usage (RSCU) average values of a codon sequence. Another useful tool is the gene analysis which examines a nucleotide sequence and returns information like its GC content and existing restriction sites. Also, many tools offer the opportunity to take the resulting sequence from that tool into another one, allowing the creation of a work-flow of tools. Moreover, as a web site, GeneDesign becomes an attractive application as it can be accessed from any Internet connected computer and thus supplying means to rapidly redesign genes. [29]

However, being a web application, it only allows uploading and downloading small parts of information, like genes. If whole genome relative information is needed, for instance to redesign a gene sequence to enhance its RSCU values, one has to rely on the available RSCU tables of some species present on the application. Also, in the reverse translation feature, equal amino acids are always reverse translated by the same selected codon for that amino acid. Many other tools lack features and flexibility to control the processes and the possibility to save the progress is not available, and so the application relies on the internet connection stability to avoid losing work. Furthermore, the interface is very simple and easy to manage, but otherwise requires more options, like the common "undo" feature. Finally, almost no sequence enhancement and manipulation methods are present, such as redesigning a gene to improve codon context or remove repetitions or deleterious sites.

2.4 Summary

In this chapter a brief introduction to the biology of protein synthesis was presented, defining the DNA as a code map that contains genes, that have the information to build proteins. Reading this chapter should transmit that genes are transcribed (copied) from DNA into another molecule called mRNA, which in turn is translated by an enzimatic complex that reads three nucleotides of the mRNA at a time (codons) decoding them into amino acids that together form a polypeptide chain (becoming a protein after folding).

[1] http://www.GeneDesign.org/

[2] Some enzymes, called restriction enzymes, cut DNA sequences in specific sites, called restriction sites. Thus, these sites are useful for genetic manipulation, as they are used, for instance, in the large scale production of insulin [28].

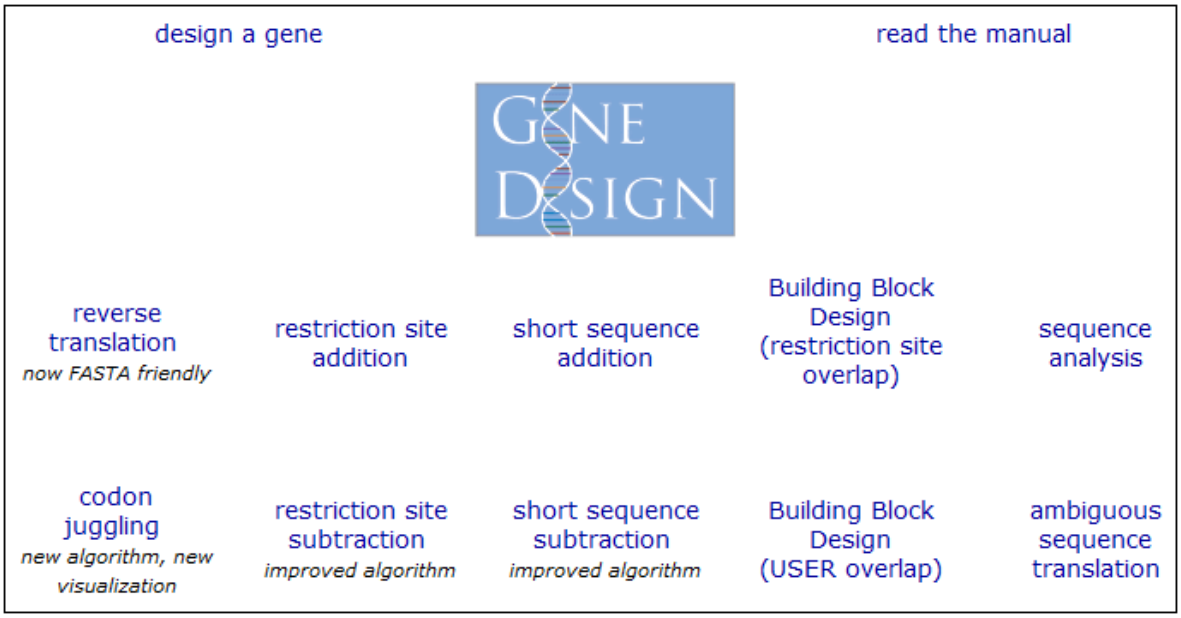

Figure 2.11: Main page of the GeneDesign 3.0 web application. Tools are available immediately by clicking on the respective links.

Genes, however, can be redesigned substituting codons by others that code for the same amino acid (called synonymous codons). Several redesigning methods based on codon substitution were presented as well as means to enhance genes in varied forms. Also, some software applications that already integrate some of these methods were shown to characterize the state of the art in this field. These applications gather many useful tools and methods to deal with gene redesign, but remain limited in some aspects, offering few methods, not many customizations, algorithms that do not guarantee best solutions in all cases, and the lack of extensibility for new - frequently emerging - redesign methods. Furthermore, even though many tools have been incorporated into these software packages, some commonly used tools are not present, or are poorly usable.

Chapter 3

Requirements

The exploitation of new approaches in gene redesign and gene optimization for heterologous expression was the main motivation behind this work and served as the locomotion for building a new application. Several joint requirement development sessions[1] led to the formation of many requirements that would improve research in this area.

In this chapter, more detailed information about the requirements that sculpted the project is presented.

3.1 User requirements

The underlying need for a software tool accommodating several requirements was motivated by the fact that, despite the amount of available software tools, the lack of cohesion between them makes researchers resort to time-consuming procedures, from application to application, in order to execute a simple task. That is - for instance - getting orthologs for a gene, aligning them, and optimizing the gene, implies the use of at least three different applications. Some software packages already gather many of these requirements as described in 2.3. However, the lack of ease of usability, extensibility, other tools, and adequate optimization algorithms, suggests the need for an integration of those dispersed elements.

Overall, joining several distinct tasks in one time-saving process is just one approach to creating short-cuts. It became clear that researchers (the main stakeholders) need efficient ways to enhance their work without having to access several independent systems to achieve a unique goal, in a human work-flow fashion. Other requirements focus on performance, portability or efficiency needs for the application, or ease of usability when managing sequences, graphical pleasant appearance yet tuned to biological issues, and showing several kinds of information regarding genes and genomes.

The following sub-sections describe two of the most important user requirements.

3.1.1 Mission

The most important requirement is the ability to redesign genes according to different redesign parameters and algorithms. This implies being able to modify a given gene codon sequence to meet one or more requirements while maintaining its corresponding amino acid

[1] Joint Requirement Development (JRD) sessions are meetings where stakeholders of the project gather to extract requirements and details necessary to the development of the project.

sequence (the result of the translation process). Thus, the application must support several different algorithms to perform different gene sequence redesigns. Also of major importance is the ability to merge several redesign methods into a single optimization process where the gene sequence is manipulated to match all the requirements as close as possible.

3.1.2 Interface

Usability is a main target of attention when building the interface, as it entails that current common tasks and methodologies performed by researchers are improved to increase efficiency and organization. Thus, making a graphical user interface is a process that cannot be accomplished without the user participation itself.

The joint construction of a graphical draft where the user specifies particular issues that must be attended culminated into an interface model of what is needed and how to reach it. A sample layout of the intended final look was thereby extracted from debate, resulting in the mockup shown in figure 3.1.

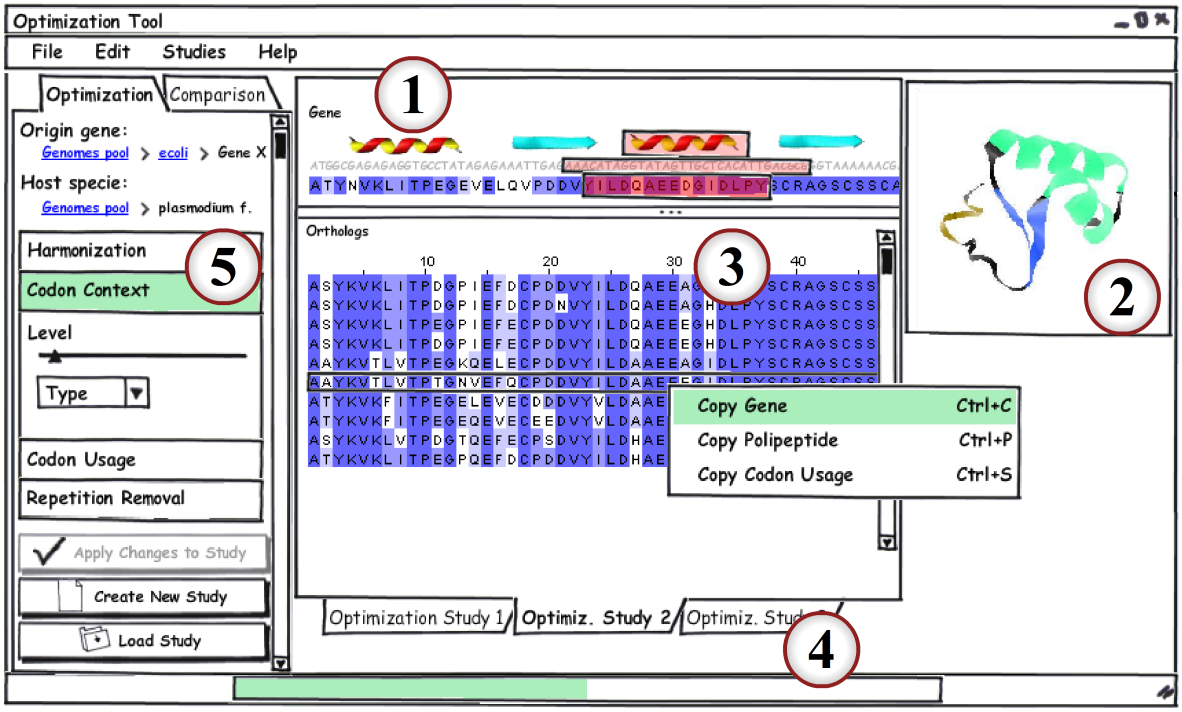

Figure 3.1: Resulting application mockup as retrieved from user requirements

Several zones in the illustration (represented by numbers) characterize the main parts of the application layout. The first region (1) marks the gene representation, where its codon and amino acid sequences, and correspondent protein secondary structure are shown. Those sequences were specifically asked to be represented in an aligned fashion where each codon is associated with its respective amino acid, and the respective secondary structure to which it belongs. Another representation of the protein is shown in 2, where its three dimensional tertiary structure is plotted, and where the secondary structures highlighted in 1 should be clearly distinguishable. The sketch shows that colours can be used to differentiate differ-

ent structures at the 3D visualization, and also to clarify the identity between the orthologs genes pointed in 3. These are usability requirements that facilitate gene manipulation and understanding. The zone pointed by 4 shows distinct projects separated by tabs (to support different simultaneous studies), and also the progress-bar that represents the state of an optimization process running (that, by experience from other applications could take some time). This single progress-bar was transformed into several progress-bars in the final application, representing several running tasks (not only optimizations). Lastly, gene manipulation should be achieved by addressing an accordion menu - like the one in zone 5 - where all optimization methods are made available. One should be able to select a redesign option and see all its customizable parameters before applying changes to the gene of interest. Applying more than one redesign method should also be an available option, by selecting several methods in the accordion menu. Other illustrated details include the availability of options when right-clicking on top of a gene or its structures (see an example in zone 3).

The mockup only represents a draft of what was needed. Several details were changed, added or removed afterwards due to user re-specification, technical issues, or usability tests performance.

3.2 Functional requirements

Some specific requirements pointed out the need to retrieve orthologs of a gene of interest, align several genes, show the secondary and tertiary structure[1] of the resulting protein, and diagnose a gene sequence by identifying several pre-defined characteristics. Furthermore, using on-line databases to get the most up-to-date information was also requested. However, the most important functional requirements are those involved in the processes of opening a genome, showing a gene, and redesigning it:

- Opening/Parsing/Validating a genome.
- Displaying the gene (codon sequence) and protein (primary, secondary, tertiary) structures.
- Redesigning genes according to a pre-defined method.
- Redesigning genes according to a combination of more than one method.
- Identifying different characteristics in a gene sequence, such as out-of-frame stop codons, or codon usages.

Those are the fundamental functional requirements, and establish the basic operations that the application should address.

Moreover, the main requirements fit into five different function-oriented packages:

- Opening/parsing package, in which requirements such as filtering the correct genes from a genome, or reading Fast-N (FASTN)[2] format files are answered;
- Interface, where user interactions, aesthetics, and visual aid requirements are listed;

[1] The global three dimensional conformation of the protein.

[2] Fast-N format refers to the text based format that is used to represent strings of DNA nucleotides.

- Sequence redesign requirements, where gene redesign and enhancement methods are stated;
- Algorithms, where requirements of algorithmic nature are listed;
- Tools, in which the external tool requirements that are needed to obtain information about genes, proteins or orthologs are plotted.

Those packages and its requirements are resumed in table 3.1.

Requirement	Description
Opening/Parsing	
Open FASTA files	Open genome files in FASTA format.
Select genetic code table	Select the genetic code to use in a genome
Filter genes	When opening genomes, load only valid coding genes1
Open several genomes	Allow opening more than one genome file at once
Open in background	Allow opening the genomes in a background process
Interface	
Display genomes	Display the loaded genomes in a list
Display genes	Display the genes of the selected genome
Show gene codons	Display the gene codon sequence
Show gene amino acids	Display the amino acid sequence of the gene resulting protein
Show gene/genome names	Show the name of the selected gene, and respective species
Show GC content	Display the GC content percentage of the selected gene
Display 3D Structure	Show the tertiary structure of the resulting protein
Show secondary Structure	Show the secondary structure of the resulting protein
List redesign methods	Display available redesign methods to manipulate sequences
Show task progress	Show the progress of each running/terminated process
Select colour scheme	Allow selecting colour-code to paint gene, like codon usage
Show/hide alignments	Display ortholog genes aligned with the selected gene
Display comparison charts	Show charts comparing genes characteristics, like codon usage
Show analysis of the gene	Show the selected redesign methods analysis of the gene
Load/Save parameters	Load or Save the parameters chosen in each redesign method
Sequence Redesign	
Redesign by Codon usage	Described in section 2.2.1
Redesign by Codon Context	Described in section 2.2.2
Redesign for GC content	Described in section 2.2.3
Harmonize with species	Described in section 2.2.4
Insert hidden stop codons	Described in section 2.2.5
Remove nucleotide repeats	Described in section 2.2.6
Remove deleterious sites	Described in section 2.2.7
Algorithms	
Analyse a gene	Make a gene analysis according to some redesign methods

Redesign sequence	Find the best synonymous sequence for the chosen parameters
Multi-redesign sequence	Redesign a gene sequence according to several redesign methods
Tools	
Calculate usage/context	Calculate the codon usage/context of a genome upon opening it
Calculate secondary structure	Calculate the selected gene secondary structure
Fetch orthologs	Use internet resources to fetch orthologs of a gene
Fetch tertiary structure	Use internet resources to fetch the tertiary structure of a gene
Auto-discover gene	Use internet resources to discover information about a gene
Align orthologs	Align the fetched orthologs into maximum similarity

Table 3.1: List of main requirements

Furthermore, in the next part of this section, some of the tasks, actions, tools and activities that are required will be presented in detail.

3.2.1 Redesign Genes

Specific requirements of redesign methods encompass enhancing genes to improve codon usage and codon context, changing the quantity of G and C nucleotides, removing sites in the sequence that might prove deleterious to the translation process such as nucleotide and codon repetitions or Shine-Dalgarno sequences, and introducing out-of-frame stop codons. All the redesign parameters must have in consideration that the gene will be expressed in a heterologous system, thus having a different codon usage, translation table[1] and codon context preference, and therefore needing harmonization in the process. Also, each redesigning method should provide parameters to customize its constraints when changing the gene codon sequence. For instance, the GC content redesign method can optimize a gene to have an increased or decreased amount of G and C nucleotides, or a specific GC percentage as introduced by the user. This parameters can also vary widely from method to method.

3.2.2 Gene analysis

One fundamental feature is allowing the analysis of a gene to identify several characteristics of a gene sequence in relation to the redesigning methods to be used. For each selected redesigning method, an analysis to the gene should be performed by highlighting the issues related to that specific redesigning method. For instance, given a gene codon sequence, analysing it for out-of-frame stop codons would show where those out-of-frame stop codons are in the sequence; analysing for codon usage would illustrate the degree bias for usage of each codon, etc. Figure 3.2 shows an example of an analysis of a gene using several different redesign requirements.

[1]The translation table lists which codons decode for which amino acids in an organism.

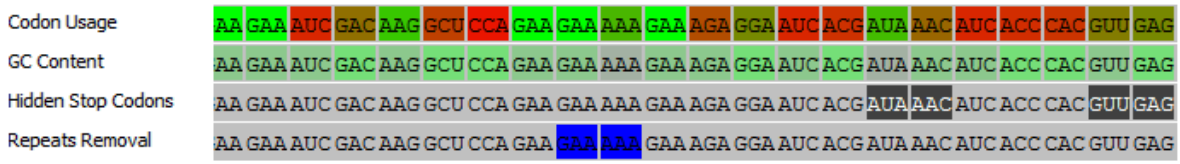

Figure 3.2: Gene analysis example. A gene is analysed focusing on several different redesign methods. Each redesign method highlights the gene according to the variable it deals with. For instance, hidden stop codons are highlighted for the out-of-frame stop codons redesign method, thus showing where this codons are in the sequence. The same for repeat removal. Codon usage and GC content show different intensities of usage and presence of GC nucleotides in a colour code. This kind of diagnosis should be present in the application.

3.2.3 Tools

Performing alignments

Genes alignment is the process of arranging their nucleotides (or amino acids) sequences in order to visually identify related similar zones. Alignments are usually presented having each sequence in a row, starting at the same place, with gaps inserted so that identical characters (nucleotides or amino acids) are aligned vertically. An example of an alignment is depicted in image 3.3.

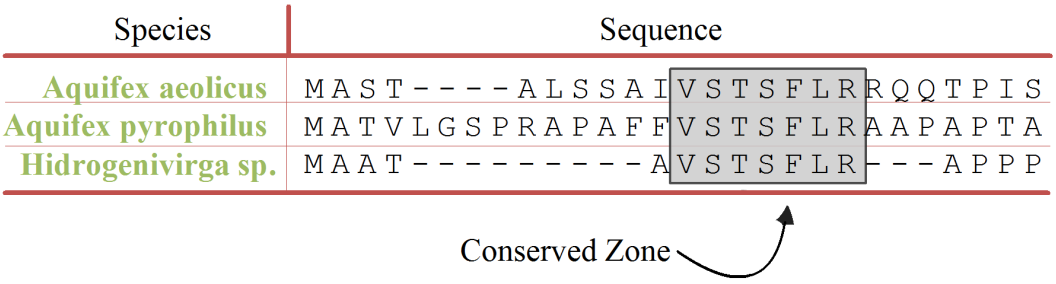

Figure 3.3: Example of an alignment from three genes of different species. A conserved zone is highlighted.

Many algorithms exist to perform multiple sequence alignment, being ClustalW the most widely used for many years. More recent tools show results outperforming ClustalW in average accuracy, namely MUSCLE, MAFFT, T-COFFEE and PROBCONS. Also, for application usability, MAFFT and MUSCLE offer the ability to make good alignments at high speeds, for a large number of sequences [30].

Secondary Structure

The secondary structure of a protein can be obtained from its amino acid sequence using a proper algorithm. When the polypeptide (sequence of amino acids forming in the ribosome) starts the folding process to become a complete protein, the chain shapes itself assuming three dimensional forms called structures. The amino acid sequence is also called the primary structure. The secondary structure is the local 3D pattern of a segment formed by the

interactions between amino acids of the sequence. Generally they can assume three different forms: the Alfa-helices, Beta-sheets, and random coils (not actually a structure, but instead the lack of structure), as seen in figure 3.4. The secondary structure might tell information about zones of an amino acid sequence, particularly functional zones and especially after an alignment where the purpose of each amino acid is exposed.

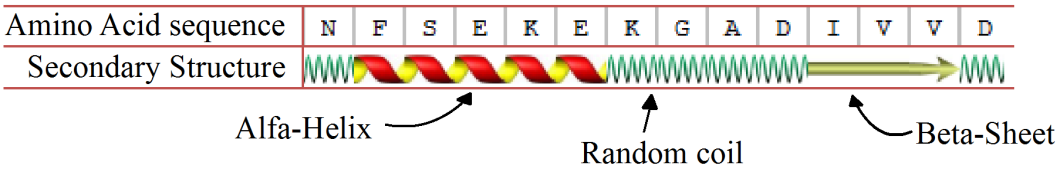

Figure 3.4: Illustration of the secondary structure of the resulting protein from a gene. An Alfa-helix, Beta-Strand and coil structures are shown.

Determining secondary and tertiary structures are not very accurate operations due to the lack of knowledge about protein structural stability and the large number of possible structures. There are many algorithms for secondary structure prediction such as JPred, PSIpred, GOR and PROFsec, and most of them are continuously benchmarked to maintain a basis comparison [16]. The algorithm that distinguishes the most is PSIpred, with average superior accuracies, reaching about 80 % in best cases [31].

Fetching orthologs

Fetching orthologs can be done with a process called sequence searching. Searching for sequences that resembles a base sequence is a step of extreme importance when studying a sequence. Finding similar sequences serves several purposes like determining if other similar genes exist following the discovery of an unknown gene to assess its functionality. The most widely used algorithm to perform sequence similarity searching is BLAST (Basic Local Alignment Search Tool) and is also one of the most used tools [32]. Accessing online databases (that usually supply a web-based BLAST tool) and use this tool is a common task to fetch orthologs. Nonetheless, this tool can also be accessed in many databases using programmatic means, like web-services.

3.3 Non-Functional requirements

3.3.1 Performance and Effectiveness

Some non-functional requirements rely on performance issues, such as completing the tasks the more quickly possible, thus implying that any processing that is not essential to the moment might be done on background. For instance, tasks like codon usage and codon context calculation should not be computed while opening a genome file, but instead provide the genome immediately after reading and filtering it, and start the usage/context calculations in the meanwhile. Thus, all tasks that allow the user to proceed with their work should run in background. This requirement allows the user to use the application in a more fluid manner, as illustrated in diagram 3.5.

Also, all tasks should be completed in an efficient way, meaning that, for instance, redesign methods should always use algorithms that are both the most efficient and quick.

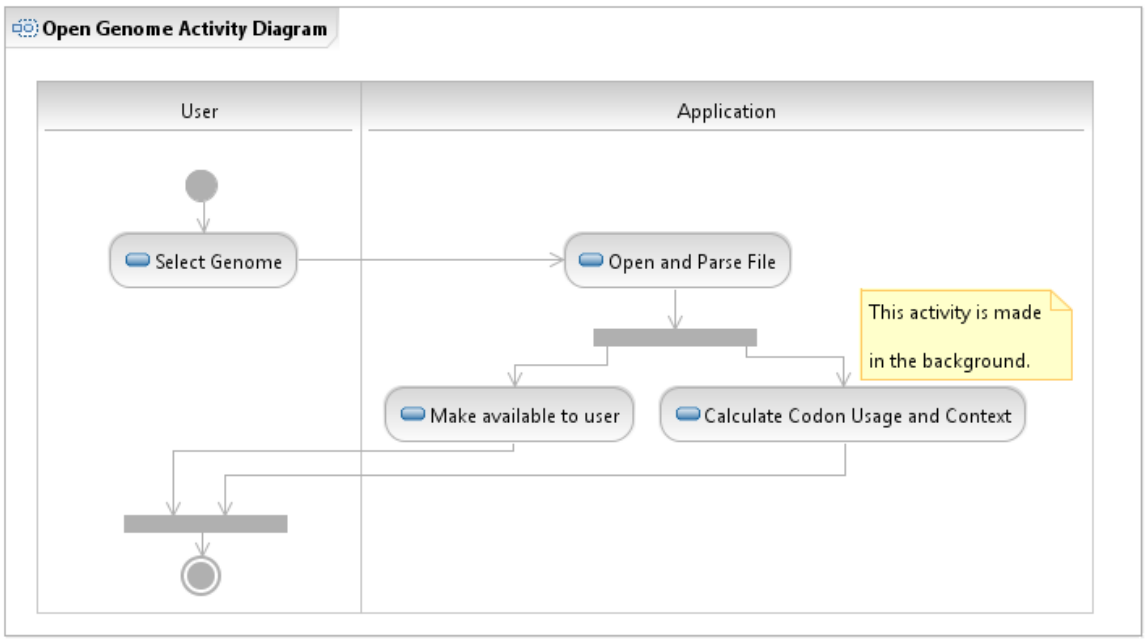

Figure 3.5: Activity diagram illustrating the strategy of placing non-essential processing into the background processing. After opening and parsing the file, the genome is made available (instantly) to the user. However, the calculation of codon usage and context might take same time. Nonetheless, the user can use the genome after opening it, without waiting for the usage/context calculation.

Furthermore, the redesign algorithms should return the best possible results. This implies using deterministic algorithms whenever possible in order to obtain the synonymous sequence that best matches the redesign requirements. However, one main concern also present as a requirement is the ability do collude several redesign methods simultaneously, and this raises the problem of multi-parameter optimization whereas a synonymous sequence must respect several requested redesign requirements. When carrying out this multiple redesign scheme, the resulting optimization cannot - in majority of the cases - maximize the sequence to agree with all the chosen redesign methods, since choosing one codon to benefit one redesign method to the most may not benefit another. However, the resulting optimization must find the middle term that maximizes the most all the redesign specifications (further development and solutions to the optimization process are discussed in section 4.2). Thus, given the commonly large number of possible synonymous sequences and the dissimilarity between redesign algorithms, finding a global deterministic approach might be very difficult or impossible, and consequently heuristic approaches should be used to find solutions that closely match the best one.

3.3.2 Life-cycle support

The more complex and important requirements rely on gathering many optimization algorithms, most of them very dissimilar. Culminating those into a single application brings the extensibility issue: constant evolving areas like biology come up with new algorithms in a regular base, suggesting the support for frequent changes and additions in optimization algo-

rithms. Also, such algorithms may rely on a large range of gene and genome characteristics, like the amino acid sequence, the codon usage table, decoding table of the genome (which codons code to which amino acids), codon sequence, etc. Hence, that information must be available to the algorithms.

This leads to a new perspective over the architectural design of the application, since it must sustain under constant updates. Consequently, each redesign method should behave as an independent element, to allow the ease of separation of purposes, and to make it work as a disposable unit that can be changed independently of the other methods. This subject will be discussed in section 4.1.

3.3.3 Portability

The system should support several different operating systems, since researchers might have different preferences in this area. Thus, the application has to support being ported to other systems, with other file systems, interfaces, internet accesses, and usability.

3.4 Summary

In this chapter, the user requirements that shaped the investigation were presented. Given that researchers use many tools and algorithms to accomplish one final purpose, gathering them in a time-saving process becomes one of the requirements. Also, much information used is available on-line in updated databases that are commonly used to fetch data from. Thus, those databases could be used in an automated fashion to obtain up to date data. However, the main requirement is gene sequence redesigning. When studying genes, it is a common task to perform sequence redesigning by using different sequence redesign methods that exist (and are frequently created). These techniques and algorithms should be gathered in some system with the ability to expand to new redesign methodologies. Another important requirement is the capability of redesigning a gene sequence for several simultaneous constraints, finding the synonymous sequence that more closely respects them all.

Taking into consideration the target user and the fact that the application should ease the research of genes, usability becomes a factor that cannot be ignored when building the application. A sketch of a possible application was constructed along with researchers to help the visualization of the final interface and better shape its appearance to adjust to the requirements.

Chapter 4

Model proposal and Implementation

The requirements listed in the previous chapter were tackled by creating an appropriate architecture to support the redesign methods changing frequency. Also, to achieve the best results when redesigning a gene sequence using several redesign methods simultaneously, a study about optimization algorithms and strategies is presented along with some solutions. Moreover, to deal with the problem of genome opening, parsing and storing in memory, some structures and active entities were created. This chapter explores the foundations of the application, the problems found while building it and the engineered solutions along with the path that took to them.

4.1 Plug-in System

To tackle the need for extensibility and support for heterogeneous algorithms and methods, a supportive architecture must be created. It should focus on having a structural design to sustain the use of different modules under the same contract, thus suggesting having independent separated units with different concerns, but the same entrance points. This design is explored in the following subsections.

4.1.1 Architecture

Having modular design architecture is dividing a system into smaller units - usually independent - that can serve different purposes inside the system. Such architecture offers several advantages, particularly in a complex system, like separation of concerns, where each component is given some responsibility, and flexibility of augmentation, allowing the easy expansion of the system by plugging new modules without changing the main system. Moreover, unit separation provides low coupling, and the creation of new abilities becomes the simplistic task of creating a new unit (or several) instead of stretching the system to accommodate it.

Adopting other strategies has many disadvantages. For instance, disregarding modularity and accommodating functionality into the application as needed results in an unstructured, highly coupled, not scalable architecture, where what should be new modules becomes spread code along the application in a try-to-fit fashion. The stretching of the application analogy is shown in figure 4.1.

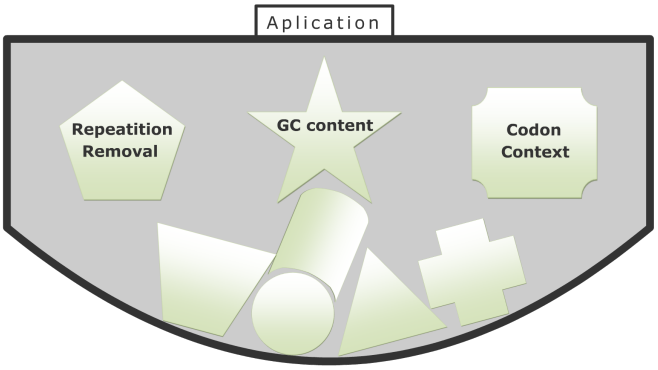

Figure 4.1: Illustration of a typical application where no structured architecture is present.

Even though a modular architecture is an obvious choice, there are still some disadvantages in using one: if many modules are present, the application becomes heavy; besides, any addition or change on modules implies the reconstruction of the application, redistribution to whoever is using it, and only the responsible for the application can make these changes.

However, in order to support the required extensibility of redesign algorithms, a modular architecture approach can be taken, where each optimization method is constructed in an individual unit, exterior to the application (leaving it with basic functionality only and therefore lightweight). This way, adding new gene manipulation methods can be achieved by building a module that respects a predetermined contract and plugging it to the application by leaving it in a plug-in site (e.g. a folder). Therefore, a plug-in architecture is proposed as a model to design the extensible part of the system, as shown in figure 4.2.

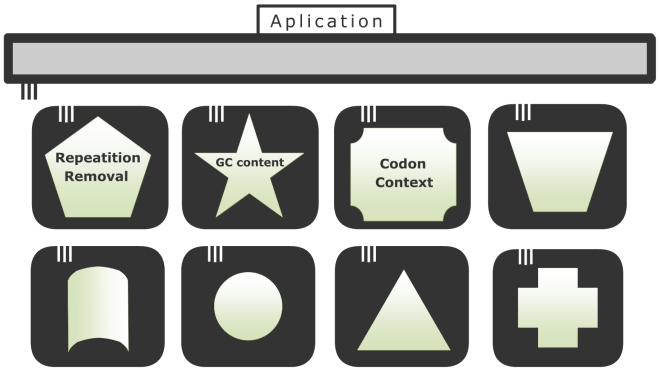

Figure 4.2: Proposed modular concept illustration by using plug-ins outside the application. Each plug-in is an independent module that uses an interface to allow the interaction of the application with it.

4.1.2 Implementation

A more detailed view of the proposed implementation is illustrated in figure 4.3, where the application defines several interfaces (like *IOptimizationPlugin*) to be respected by the

outside plug-ins. A plug-in loader is called to read the available plug-in modules that implement the contracts. Once the plug-ins are loaded, an optimization process can use one (or several) module by addressing the contract it implements. Also, new modules can be added by creating a unit that implements the given interface, and placed in the plug-in store zone.

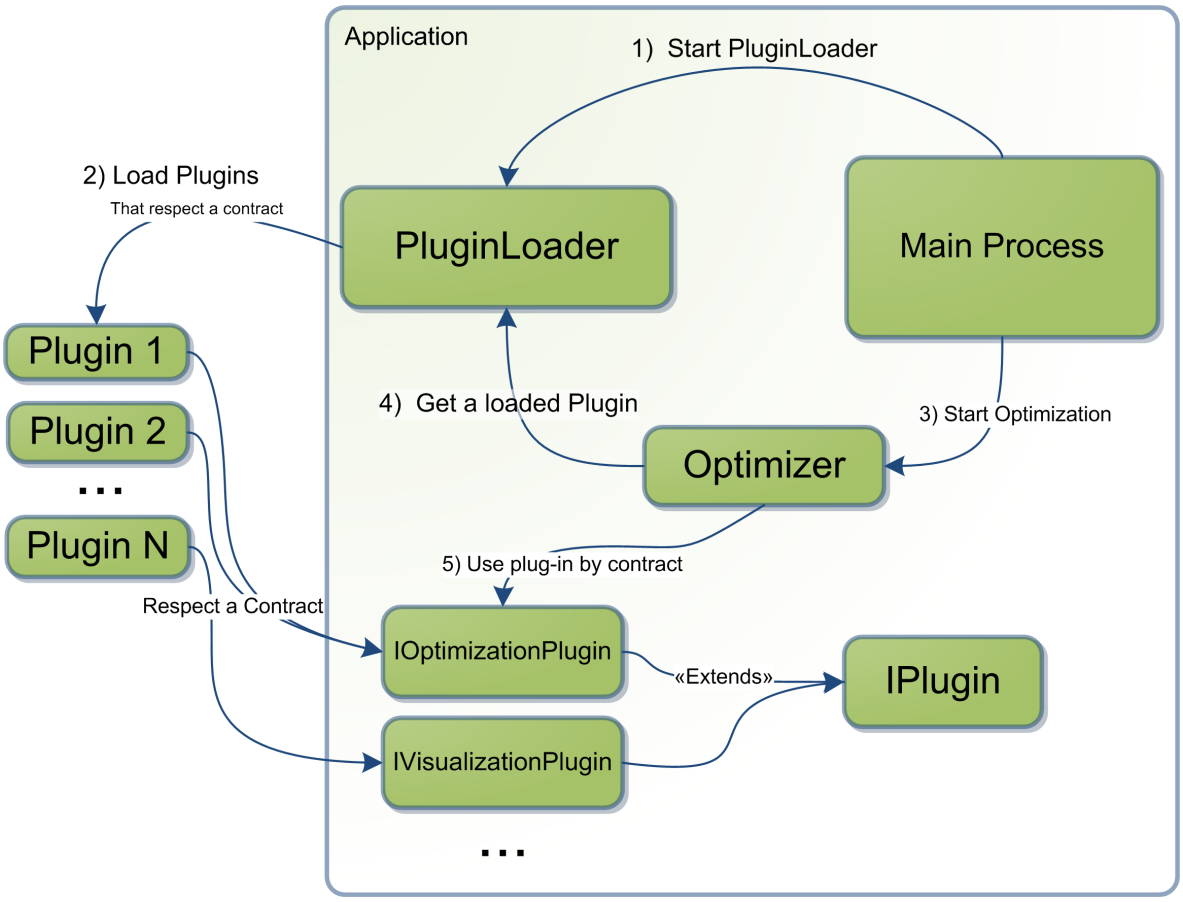

Figure 4.3: Plug-in architecture work-flow. Plug-ins outside the application must respect a contract in order to be used as modules.

Furthermore, the $IPlugin$ interface contains the global requirements that must be met by all kinds of plugins, and it cannot be directly used as it does not describe any specific functionality other than essential functions. For instance, all plug-ins should have a name, and a version, so the $IPlugin$ interface defines contracts like $getPluginName$ and $getPluginVersion$. Thus, when plug-ins use contracts that extends $IPlugin$, they must implement those functions in order to be accepted by the plug-in loader. On the other hand, interfaces that extend $IPlugin$ should be more functionality specific, and therefore declare contracts more precise for a purpose. An example is shown in the class diagram of figure 4.4.

Plug-in loader

According to the proposed architecture, the plug-ins are stored in a single place, which should be dedicated to them (in order to avoid the overhead of trying to load files that are not

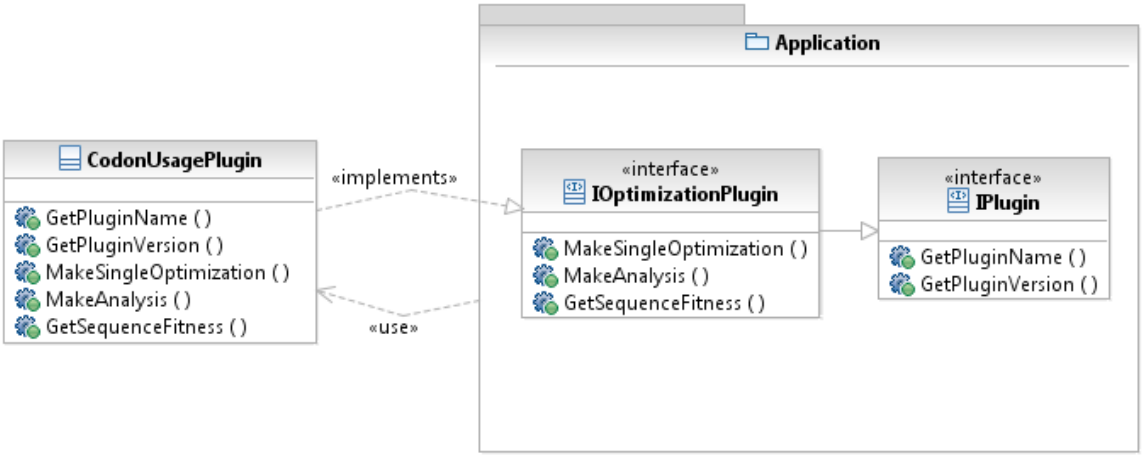

Figure 4.4: Plug-in interface class diagram. An example plug-in (CodonUsagePlugin) is used by the application through the interface IOptimizationPlugin, that extends IPlugin.

plug-ins). It was also stated that only plug-ins that correctly respected some interface should be loaded. Thus, to elaborate this scheme, an entity called plug-in loader (*PluginLoader*) was developed.

When this entity is started, it scans a given directory (the plug-in directory) for .class files, given that the plug-in were written in the Java language. For each .class file, its bytes are entirely loaded into memory and then transformed into the Java class object. Then the verification of contracts is made by listing all the interfaces that the class is implementing and comparing them to the list of valid accepted interfaces[1]. If any contract is being respected by the class, it is added to the list of plug-in, and the application is notified that a new plug-in is available. Such implementation is illustrated in the sequence diagram of figure 4.5.

4.2 Optimization

Optimization, in the mathematical sense, refers to the problem class that deals with searching and finding of the best available elements among a given domain. This typically implies seeking the global minimum or maximum value of a real function. A formulation of this problem can be understood by admitting some non-empty subset S of a real linear space, and let $f : S \rightarrow \Re$ be the objective functional of which the minimum is to be found. The minimum value x' of f shall respect the following formulation: [34]

$$f(x') \leq f(x), \forall x \in S \equiv \text{ finding } x' \text{ such that } f(x') = \min_{x \in S} f(x) \tag{4.1}$$

This function f often represents some fitness measure in-between the problem, and is the target of optimization whereas its input parameter that satisfies formulation 4.1 is the

[1] Actually, the definition of contracts is a subject of much discussion. An interface is not directly a contract but instead a representation of a contract, as it obligates a class to implement all signatures and use the input and output types, and by turn agrees that the interface will never be changed. However, interfaces cannot define pre-conditions, invariants or post-conditions, which are believed to be essential to define abstract data types, according to the Programming by Contract approach [33].

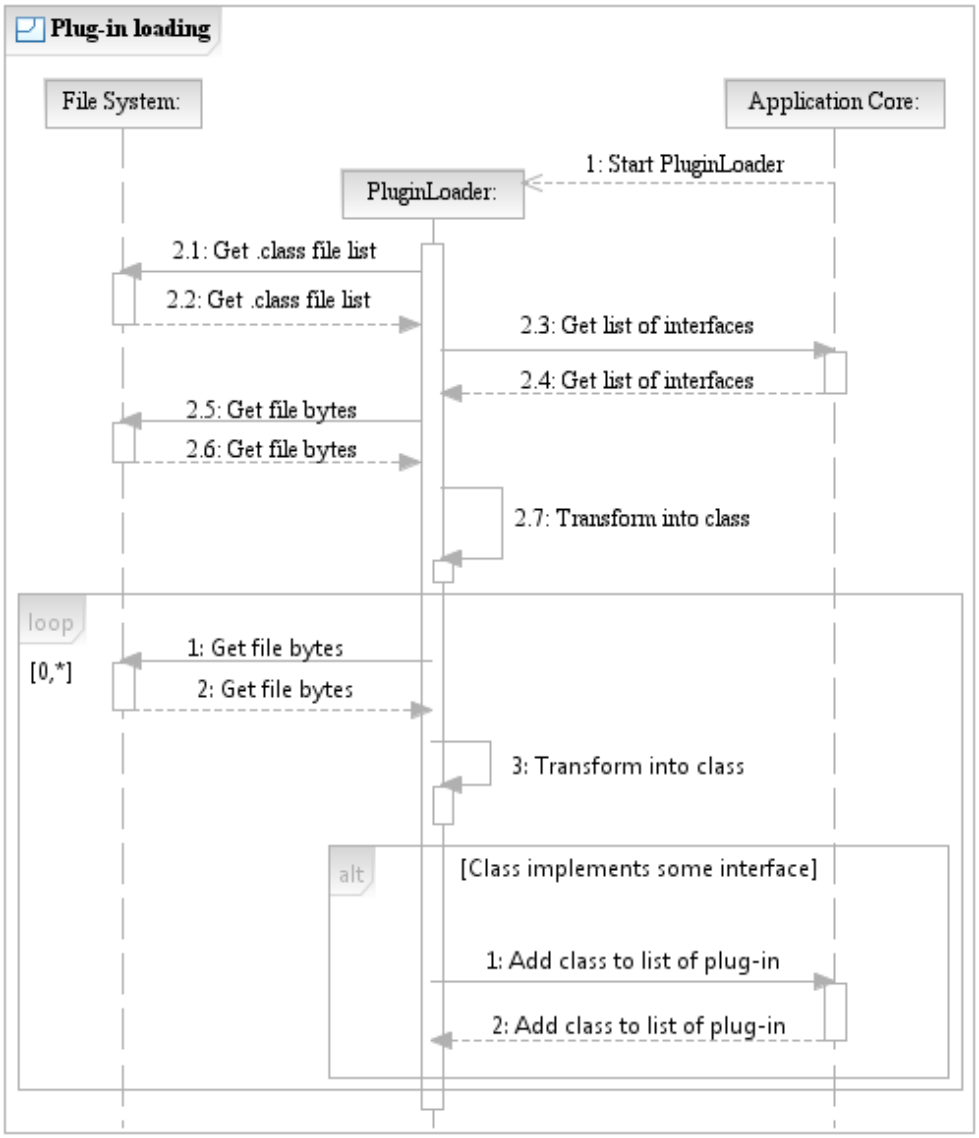

Figure 4.5: Plug-in loading sequence diagram. The application starts the PluginLoader to read all class files in a folder and the classes that implement some accepted interface are added to the application list of plug-in.

---

solution. Accomplishing this means exploring the search space S until a feasible solution that minimizes the fitness function is found. Therefore, a general optimization problem can be formally presented as the tuple (I, f, m, g), where I represents the set of instances of the problem, each set of feasible solutions (also known as candidate solutions) of an instance can be obtained from $f(x), x \in I$, the measure of each solution is retrieved from $m(x, f(x)), x \in I$, and the objective of the optimization is given by the goal function g that can either be min or max. The problem is thereby solved for an instance x by finding the optimal solution among the feasible solutions y such that: [35]

$$m(x, y) = g\{m(x, y') \mid y' \in f(x)\} \tag{4.2}$$

In a computational perspective, finding the optimal solution will generally take more time the bigger is the space of feasible solutions, thus testing one by one to find the best might not always be a practical approach. The use of optimization to deal with real world problems usually has more than one objective, hence having several functions to assess a measurement of a candidate solution, in a so called multi-criteria optimization process. In the above formulation, one can describe a multi-objective support by defining the measure function m as the following:

$$m(x, y) = [m_1(x, y), m_2(x, y), \ldots, m_n(x, y)] \tag{4.3}$$

Where m_i states the measure function of the i^{th} criteria. The optimal solution to a multi-criteria problem follows the same scheme of finding the y that fulfils the goal g, and thus minimizing (or maximizing) every m_i. Additionally, in a well formed multi-criteria optimization problem, improvement in one measure should worsen other measure(s), creating a trade-off between the criteria. Finding the configuration that maximizes the overall improvement (even sacrificing some criterion in favour of more compensating advancement) of m is commonly the target of multi-objective optimization [36]. For instance, improving the performance of a car while reducing its consumption is a problem with two criteria that have conflicting objectives. The most intuitive form of expressing the overall performance of m is to combine every m_i into a single function (Aggregate Objective Function (AOF)) as a weighted sum of all measures. Then the problem can be approached as if with a single objective optimization problem, using the AOF as its measure function.

$$AOF(m) = \sum_i m_i w_i \tag{4.4}$$

In gene optimization, the criteria are represented by the several redesign methods, and the gene sequence represents the problem instance. Each redesign method should also present a measure m_i of a sequence according to its objective, thus making m their aggregation. Every synonymous sequence of the original gene constitutes the feasible solution space, and is therefore the target of exploration when performing an optimization. The goal remains minimizing or maximizing m, now being the AOF. A measure example can be seen in figure 4.6.

The following algorithms and methodologies address solutions to optimization problems. They are presented along with their characteristics, advantages and faults. Furthermore, in the proceeding sub-sections, the proposed model to this project and its implementation is presented. All methodologies are approached in a gene optimization perspective to better fit the problem.

Exploring the search space

There are 64 possible different codons (4^3), and there are only 21 different amino acids, resulting in an average of 3 codons that encode each amino acid. Despite depending much on the species, one gene can have thousands of codons. However, a hypothetical gene with 500 codons would have about 3^{500} different synonymous sequences (see figure 4.7). Even admitting that some system could compute billions of synonymous sequences per millisecond, computing all the sequences would take an unachievable amount of years, and thus, running through all of them to find which one maximizes the AOF is not a practical solution.

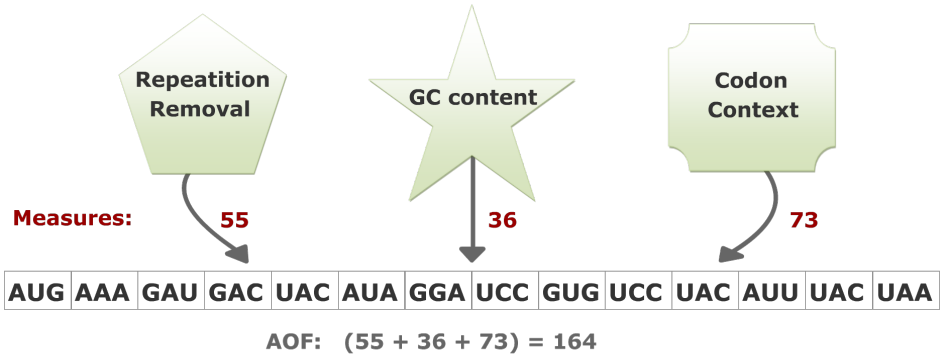

Figure 4.6: Multi-objective optimization concept: measurement from several criteria aggregated into an AOF function.

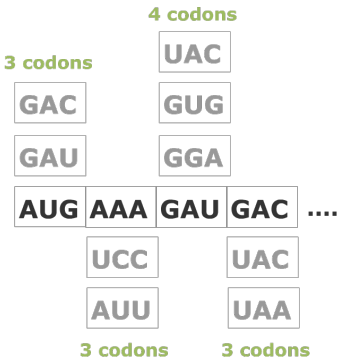

Figure 4.7: Slice of a codon sequence representation with synonymous of every codon. Big sequences result in numerous possible combinations.

Another approach to explore the search space is to iteratively pursuit solutions that are better than the previous one. The Hill Climbing algorithm starts with a given possible solution (for instance the original sequence to be optimized) and chooses a neighbour solution that is better for the next iteration. Neighbours can be defined - for the example sake - as sequences similar to the current one, only differing in one codon. Algorithm 4.1 presents the pseudo code of a gene optimization specific hill climbing.

Although the algorithm is fairly simple, it carries a critical problem: while always accepting only better changes, it easily gets lost in local maximum states (unless *GenerateNeighbour* allows changing the whole sequence, and in such case the algorithm would be worse than running through all possible synonymous). For instance, if *GenerateNeighbour* only changes one random codon, it is possible that all feasible changes do not result in a sequence with better fitness, and the current sequence is not the global optimum, like exemplified in figure 4.8.

If no local maximum exists in the search space, the algorithm statistically converges to the global maximum after some time. But even so, it would require a large amount of time that could extend to more than exploring all possible sequences one by one due to the termination

Algorithm 4.1 Hill Climbing

```
CS ← OriginalCodonSequence
AOF ← Evaluate(CS)
while EndConditionNotMet do
  NS ← GenerateNeighbour(CS)
  AOF2 ← Evaluate(NS)
  if AOF2 > AOF then
    CS ← NS
    AOF ← AOF2
  end if
end while
return CS
```

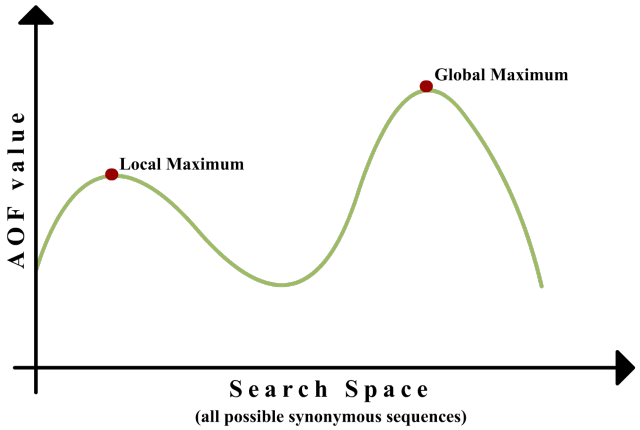

Figure 4.8: Graphic representing hypothetical AOF values of the search space. Local and Global maximum are distinguished.

condition that usually states that the algorithm only stops when no neighbour is a better choice.

There are, however, several ways to improve this algorithm, one of them being the Simulated Annealing algorithm [37], which is based in the real procedure of making the strongest crystal possible: heating the glass so that it turns liquid and the atoms can move freely, and then cooling it slowly giving the atoms the chance to find the most stable configuration. The same process can be applied as a global optimization solver (though it does not guarantee the optimum solution) by initially allowing the algorithm to choose worst solutions than the current, and as the iterations go by, increasingly choose only best solutions (simulating the cooling process). Allowing the choice of worst solutions avoids the algorithms from getting stuck in local maximum points, and therefore an advantage comparing to the hill climbing. Changing the Hill Climbing algorithm to support Simulated Annealing results in the algorithm 4.2.

Therefore, the resulting solution is an exchange between the computational time and the quality of the solution: slower cooling (T decreases slower, but the process takes more time)

Algorithm 4.2 Simulated Annealing

$CS \leftarrow OriginalCodonSequence$
$Temperature \leftarrow MaxTemperature$
while EndConditionNotMet **do**
  $NS \leftarrow GenerateNeighbour(CS)$
  if $Evaluate(NS) > Evaluate(CS)$ **then**
    $CS \leftarrow NS$
  else
    $CS \leftarrow AcceptanceCriterion(CS, NS, Temperature)$
  end if
  $Temperature \leftarrow CoolingSchedule(Temperature)$
end while
return CS

generally results in better solutions. When a neighbour's AOF is worse than the current one, it can still be accepted according to a criterion. This criterion defines the probability of choosing a worse solution, taking into account the current and next solutions and also the current temperature. Following the Boltzmann distribution [37], the acceptance criterion probability can be defined as:

$$P_{acceptance} = e^{\frac{Evaluate(CS)-Evaluate(NS)}{Temperature}} \tag{4.5}$$

Also, the cooling schedule defines how slowly the temperatures decreases, and therefore having a major impact in finding good solutions:

$$Temperature_{i+1} = \omega * Temperature_i, \omega \in]0, 1[\tag{4.6}$$

The trade off between computational time and cooling schedule represent the limitation of this process. Since the synonymous sequence search space is big, to ensure good solutions and avoiding local maxima, a lot of computational time may be taken.

Iterating through the redesign methods

Another approach to optimize a sequence is to take the original sequence, redesign it with some method, then take the result and redesign with another method, and so on in a pipeline fashion (illustrated in figure 4.9). This way, each method chooses which synonymous best match its criteria and the next method take the previous method choices as an input. Consequently if a method's choice conflicts with a previous one, the first method's choice will not prevail, and therefore the latest methods always have preference. Thus, to avoid disregarding the previous changes, the main redesign methods should be placed in last to get preference.

Furthermore, if each redesign method is not enclosed in a module like suggested in section 4.1, adding redesign methods would require writing them in the algorithm as the next step in the work-flow instead of plugging-in the module and allowing the algorithm to use it. This limitation is present in the software Gene Designer as already described in section 2.3.2.

The main drawback in this algorithm is precisely the work-flow manner in which the sequences run, allowing for each method to completely change the previous choices, resulting

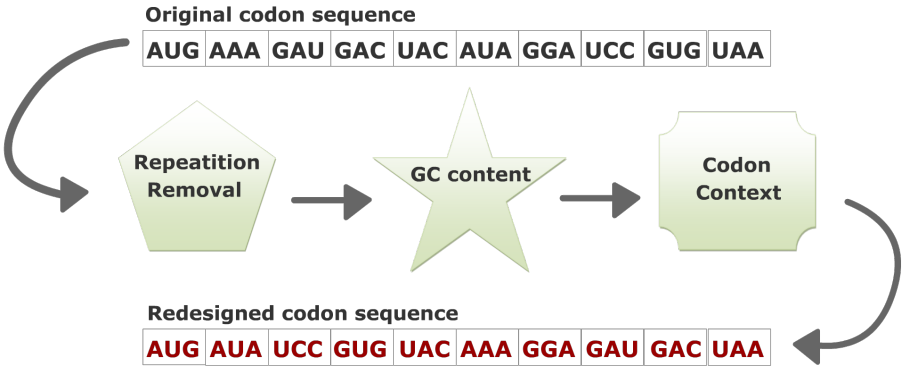

Figure 4.9: Illustration of the work-flow when iterating the original sequence through the redesign methods.

---

in no influence from the first. Thus implying that the last methods have greater impact than the first ones, and thereby not all the methods equally influence the sequence. This reflects one of the main problems in the optimization and redesign processes: if several redesign methods want to change the same codon for different synonymous, not all the methods can be simultaneously pleased. The target of optimization is therefore finding which combination of synonymous (for each codon of the sequence) maximizes the AOF.

Iterating through the sequence

Like mentioned before in this chapter, if several redesign methods need to change the same codon to different synonymous codons, there isn't a single choice that satisfies all the methods simultaneously. Thus, one way to optimize the multiple redesign process is to run through each codon of the sequence and list all their available synonymous codons in order to choose which replacements result in a better choice for the most redesign methods. The algorithm for this process is very simple: list the synonymous for the first codon, then for each synonymous get a punctuation from each of the redesign methods (or better: from the AOF), and finally choose the synonymous that returns the maximum score. Repeat the process for each codon of the sequence. The algorithm is illustrated in figure 4.10.

The main limitation with this algorithm is that it only makes local choices, regarding only one codon at a time. If all redesign methods only take one codon a time into consideration when redesigning a sequence, iterating the sequence with this algorithm would suffice, and the global optimum of the AOF would be guaranteed. However, having no control over the redesign method of each module (assuming a plug-in architecture) implies that it is possible that some method takes simultaneously several codons into account. As a result, an optimization that only accounts for one codon at a time does not guarantee a good solution when methods that need – for instance – two codons at a time are present.

One example of a redesign method that uses more than one codon at a time is the codon context that counts the frequency of each pair of consecutive codons and then replaces the codons in the sequence in order to get a synonymous sequence with the more frequent pairs. Thus, optimizing one codon at a time will not take into account a pair of codons, and consequently not working for codon context.

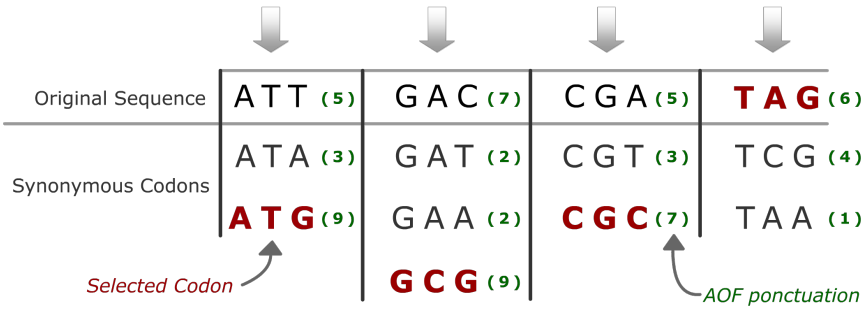

Figure 4.10: Illustration of the sequence iteration algorithm. Each step chooses the synonymous codon that gets the best AOF score.

Overall, this algorithm cannot be used in a system that makes no limitations about the redesign methodologies (and making them would not make the application scalable to many redesign methods) because it only is concerned about one codon at a time. Nevertheless, dealing with more codons simultaneously could overcome some limitations. Accounting for two consecutive codons simultaneously can ensure the global optimum for several redesign methods like codon context and hidden stop codons (that use two consecutive codons at a time), and also for single codon redesign methods, like codon usage. However, finding the best combinations of two consecutive codons for a whole sequence involves a far more complex algorithm than simply running the sequence and choosing the best synonymous. Every two consequent pairs share one codon (like shown in figure 4.11), thus that codon must be chose in order to benefit the most the two pairs, and this rule applies to every two pair in the sequence.

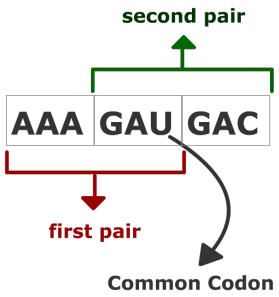

Figure 4.11: Illustration of the problem of optimizing two codon at a time: there is one codon that is shared by every two consequent pairs of codons. The optimization has to take that into account.

Hence, dynamic programming[1] and memoization[2] can serve as a solution to find which configuration of pairs maximizes the AOF. The algorithm 2.1 (in section 2.2.2) exemplifies

[1]Dynamic programming is a very powerful algorithmic paradigm in which a problem is solved by identifying a collection of sub-problems and tackling them one by one, smallest first, using the answers to small problems to help figure out larger ones, until the whole lot of them is solved [38].

[2]Memoization is a well-known method which makes use of a table of previously-computed results in order to ensure that parts of a search (or computation) space are not revisited [39].

how to obtain the best sequence for codon context (that, recalling, uses pairs of codons). To generalize the process for every method, the auxiliary algorithm 2.2 (also in section 2.2.2) is the fitness function and thereby should take an AOF into consideration (particularized to receive only two codons). This algorithms guarantees that the best possible sequence is found, since it runs the sequence a first time to record the best AOF scores for each synonymous choice, and then runs a second time to replace the codons for the ones that resulted in the best score sequence.

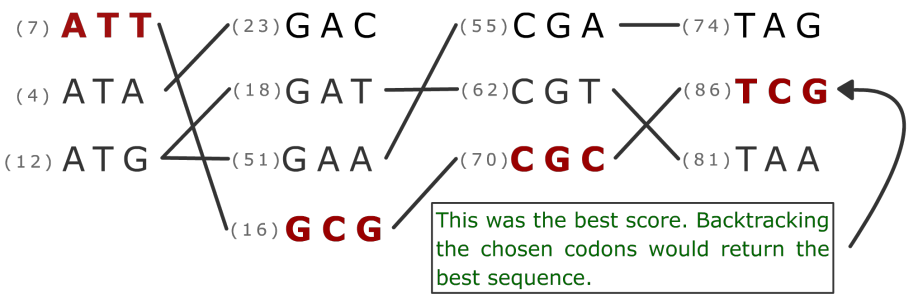

Figure 4.12: Illustration of the dynamic programming algorithm: each codon is tested with all the previous codon synonymous, and the best choice is recorded. Testing each pair involves summing the score of the previous codon it the score obtained with this pair (previous codon and current codon).

Nonetheless, the main limitation remains: the algorithm only takes into account a limited number of codons, and even though the majority of the redesign methods don't take more than two codons into consideration, imposing a limitation of two codons would be limiting the application scalability. Of course one could generalize the algorithm to take more than two codons into account, but even then, when the number of codons taken into account tends to the sequence length (in codons), the algorithm tends to the complete search space exploration, and thus unfeasible. This means that a redesign method cannot take large amounts of codons simultaneously into consideration, and thus limiting the application.

4.2.1 Genetic Algorithm

Inspired in nature, this class of algorithms use natural selection as scheme to obtain a global search optimization method. Species evolve by reproducing themselves in a non-random manner: better fit organisms (more adapted to the environmental pressure and more likely to survive and reproduce) tend to reproduce easily than other as a consequence of its fitness. Hence, over generations, best fit organisms tend to become more common in their population, assuming their advantage in natural selection. This is a specialization of the population to adapt to their usual environment [40].

Hence, genetic algorithms are a computational simulation of the natural selection and evolution of species. The first step in the algorithm is to create a set of individuals, the population. The objective is then to evolve this population to obtain individuals best fitted to some purpose. Each step of the simulation creates a new generation of individuals from the previous population. The generations and the evolution of the individuals can be obtained using three basic operators: selection, when the best fitted[1] organisms are selected to

[1] Actually, not always the best fitted are chosen, but instead they are most likely to be chosen, allowing for

reproduction; crossover, performing the actual reproduction taking two or more individuals to create other individuals; and mutation, to apply random variations to the created individuals, ensuring genetic disparity between parents and offspring. After reproduction, usually the best offspring replace the worse individuals in the population [40, 41].

As each individual represents a possible solution to an optimization problem, there must be a fitness function that measures how fit he is, in order to assess its value in reproduction for the next generation. Multiple criteria problems frequently use genetic algorithms to determine optimal or near optimal solutions using an AOF as the fitness function to evaluate individuals. An outline of a specific genetic algorithm for this problem is detailed in algorithm 4.3 [40].

Algorithm 4.3 Genetic Algorithm

```
Population ← CreateRandomPopulation()
EvaluateFitness(Population)
while EndConditionNotMet do
  Parents ← MakeSelection(Population)
  Children ← MakeCrossover(Parents)
  Children ← MakeMutation(Children)
  EvaluateFitness(Children)
  if Fitness(Children) > Fitness(Population[Weakest]) then
    Population[Weakest] ← Children
  end if
end while
return Population[BestIndividual]
```

In the gene redesign context, one can assume that each individual of the population is a synonymous sequence of the original sequence to be redesigned. This way, all individuals are synonymous with each other, but all different in their codon sequence. The multiple criterion are the redesign requirements that are chosen, and thereby they assign the punctuation of each individual (all the punctuations should be in the same scale so that no redesign method is privileged). The sequences that get the best score are more likely to be chosen to create offspring, since they represent more significant variations of the original sequence, from the perspective of the redesign methods. Hence, the synonymous codons used in the best fit sequences are more likely to create other sequences that have equal or higher punctuation (and therefore being improvements of them).

The crossover operator can be applied to the selected individuals in a multi-point strategy, meaning that the resulting sequences codons will have split probability of coming from one of the parents. For instance, if two parents are used (two codon sequences), the two children will be opposed (if one has the codon of one parent, the other will have the codon of the other parent), with each codon having a 50 percent probability of coming from one parent or another (as illustrated in step c) of figure 4.13).

In the end, each child will have codons from both parents, with equal probability[1].

individuals with other fitness to also reproduce. This is a process called fitness proportionate selection, where each individual is given a probability of reproducing according to its fitness. Other selection methods exist, like the tournament selection [40].

[1]One could benefit the parent with higher fitness, attributing to its codons a higher probability of being chosen than those of the other parent(s). This strategy wasn't used in order to maintain fidelity to the natural selection principle, and avoid that the genetic variety would narrow too quickly (giving too much advantage

To simulate the errors and casualties that also lead to evolution, mutations are applied to the offspring in a random fashion, with a pre-determined probability. Mutations allow sequences to have new codons that did not derive from its parents, providing an opportunity to evolve into a better sequence with features not inherited. Of course they could also mutate into worse sequences (sequences with lower fitness from the redesign methods point-of-view), but they wouldn't be as likely to reproduce in the next generations.

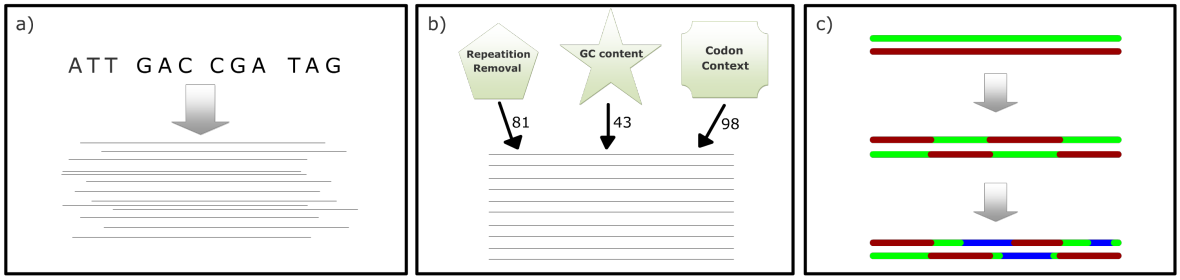

Figure 4.13: An illustration of a genetic algorithm process applied to the gene multi-redesign problem. **a)** Thousands of synonymous sequences are created from the original one, forming the initial population. **b)** Each redesign method assigns a punctuation to every individual of the population. The population is then ordered by score. **c)** The best individuals (those with greatest punctuation) are chosen to crossover. After creating new offspring, the children are subject to a random mutation, and the result is evaluated by the redesign methods to obtain a score.

The new individuals resulting from a reproduction are then inserted into the population according to some rule to assure both genetic diversity and increasingly converging to higher scores population. One possible technique is to replace the worst individuals in the population with the newly created ones, if the last ones are fittest. This guarantees a fast converging of the population since the lower scores rise rapidly, but on the other hand, individuals with low punctuation might not have the chance to reproduce and they could carry important variations that would lead to a greater score in a long term. Therefore, maintaining the genetic variety is essential to obtain results that aren't just local maxima. Another technique to insert the offspring into the population, is replacing the parents by its children if the last have higher scores. The advantage in this method is that the genetic variety is kept, since parents and offspring share the same information (before the mutations), as it can be seen on step c) of image 4.13. After inserting the offspring into the population, the generation is complete, and the process is repeated again until some termination condition is found. Usually, when the best individual of the population is the same for many generations, the algorithm might be considered to converge since no other individual has outranked it.

To guarantee the genetic variety during generations, and avoid convergence to local maxima, a combination of genetic algorithms and simulated annealing could be used. The genetic algorithms already try to avoid local maximum by applying mutations to newly created individuals. However, those mutations usually have a fixed rate in genetic algorithms, but the simulated annealing technique shows that starting with a very loosely approach (many variations) and slowly becoming more tight (accepting only better results) is a good method of avoiding local maxima. Thus, applying the annealing technique in genetic algorithms could assist the optimization in circumventing convergences to local maxima and getting stuck

to the fittest).

there. Making the mutations sensitive to the generation is one way of achieving that: in the first generations, the rate of mutations used in offspring is high, and when approaching the last generations the probability of making a mutation decreases to zero, as described in the following equation.

$$P_{mutation} = \frac{(generation_{max} - generation)}{k * generation_{max}}, k \in \Re^{+}, 0 \leq generation \leq generation_{max} \quad (4.7)$$

Applying the right number of mutations on any sequence could transform it in any other synonymous sequence, and therefore allowing the algorithm to escape from zones of the search space of local maxima when the mutation rate is high. However, the mutation rate of the above formula is highly dependent on k, leading to high mutation rates when k is low and low mutation probability when k is high. The higher the mutation rate, the more difficult it is to converge to local maxima, but on the other hand the convergence to global optimum is slower. Plotting different variations of k illustrates how convergence happens with different mutation rates (Figure 4.14).

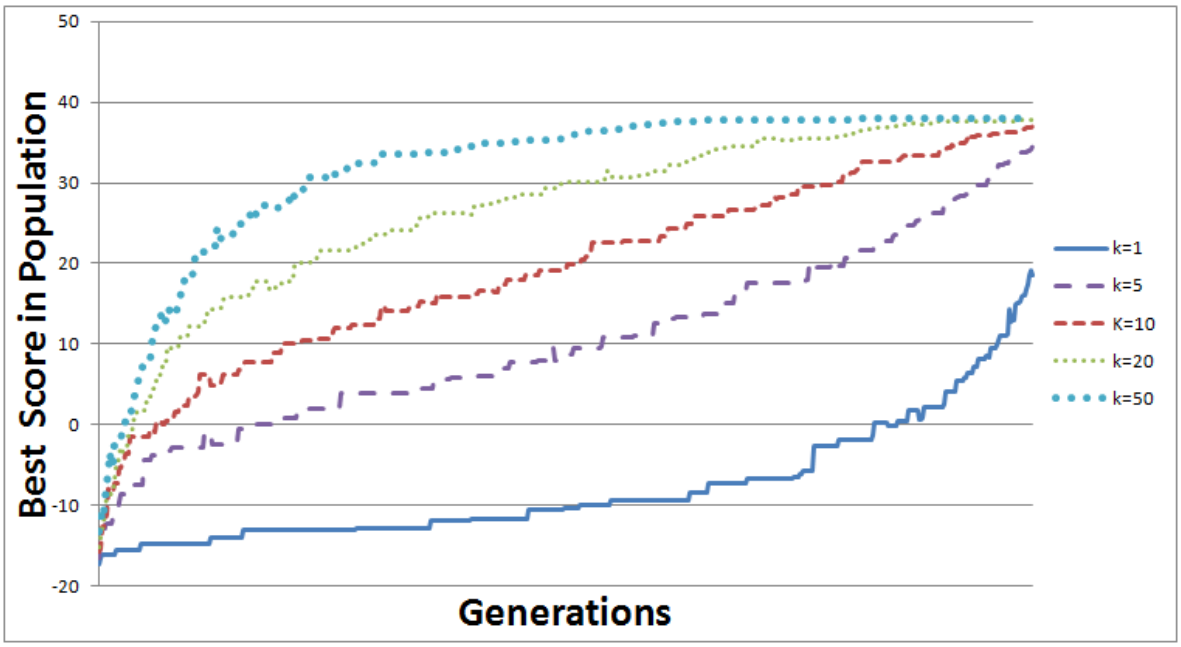

Figure 4.14: Plot of the different convergence rates when using different mutation rates. Each line represents the value of the AOF function of the fittest individual in the population in each generation. The lower is the k value, the longer it takes to the genetic algorithm to converge to a global optimum, but the variability is greater due to the high mutation rates, becoming harder to converge to local maxima. Higher values of k indicate less mutations (lower probability to mutate), and the best individual quickly converges. This could lead to the misinterpretation that mutation only slows the process, but instead it should be seen as a technique of avoiding local maxima by inducing randomization in each step, making it easy to jump through the search space. A 400 codon sequence was used during 600 generations, with 1000 different individuals as the population, to optimize for codon context.

The termination condition of a genetic algorithm usually relies in finding a good enough solution to the problem, or reaching a certain number of generations. Also, another way

to assume the end of the algorithm is to wait for a convergence to happen, assuming that when an individual stays the fittest of the population for a long number of generations no more individuals will outscore him, and thus he represents the global maximum. Though this approach is rough, it strongly depends on the number of generations with no evolution to assume convergence, and even admitting that an individual chosen with this method might not be the global maximum, since no other outscored it for too long (and scores in late generations do not rise much, as can be seem in the line of $k = 50$ of figure 4.14), it's reasonable to assume that waiting for a better individual will take many generations and its score won't represent a significant improvement.

Therefore, the important parts of a genetic algorithm are the AOF function to correctly assess which individuals are the fittest, the selection method to guarantee the population evolves into the right direction, the crossover to properly descend information from parents to offspring, and the mutation rate to maintain genetic variety and avoid convergence to local maxima. However, as a heuristic method, genetic algorithms are not certain to find the optimal solution, but they do offer some advantages in comparison to other algorithms. For instance, genetic algorithms take into account past experience of exploring the solution space, while simulated annealing only depends on the current individual and its neighbours. Also, managing several individuals simultaneously increases the chances of finding the global optimum, since more space is searched (in the solution space). Furthermore, optimum solutions or at least very good solutions can be found in linear time (highly dependent on the fitness functions), even with long codon sequences.

Genetic algorithms also offer high scalability, given that the punctuation may come from an AOF, and therefore support a modular architecture. As methods give scores to individuals instead of applying some kind of redesign, the flexibility of parameters and requirements may be very high and specific, much like searching directly into the solution space looking for a sequence that meets all conditions and requirements.

4.2.2 Implementation

To employ the discussed optimization process, two components were built. The first, called *GeneticAlgorithm*, is responsible for the genetic algorithm operations, namely the crossover, selection, fitness assessing, population generation and insertion of offspring. The component was made to deal specifically with gene sequences, and is used by the second component, the *OptimizationRunner*, which is responsible for choosing whether the genetic algorithm should be used or not, and deals with the process of evolving the genetic algorithm population (in case it is used).

Single Method Optimization

When performing optimization, one of the requirements stated that algorithms that guarantee the best results should always be used. Thus, if only one redesign method is chosen, no multi-optimization is needed, and the redesign should be achieved by employing an appropriate deterministic algorithm for that redesign strategy. That is, when redesigning a gene sequence for one specific redesign method like the ones presented in section 2.2, the application should not resort to heuristics like genetic algorithms but instead use its own algorithm (that should be the more appropriate, effective and fast form of redesigning the gene). Also, if there is no specific algorithm available for some redesign method or its implementation is

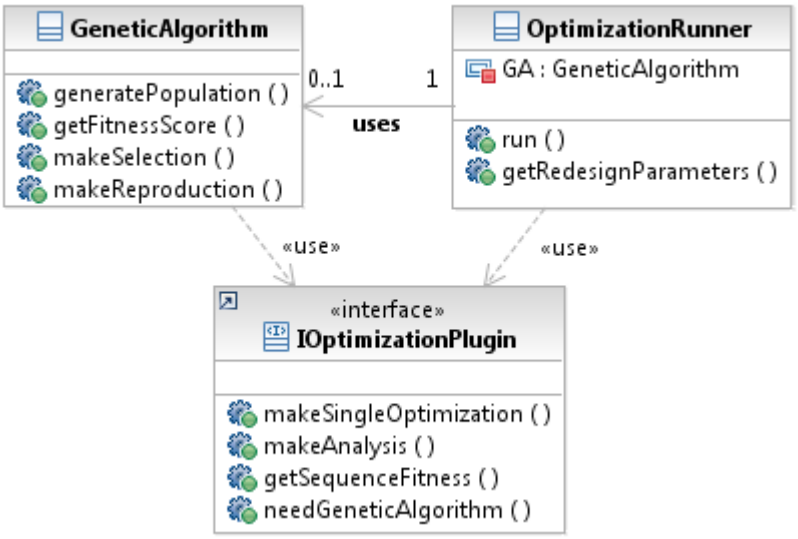

Figure 4.15: Class diagram of the optimization process. The *OptimizationRunner* is the active entity in the process, responsible for the optimization. The *geneticAlgorithm* class displays the needed operations to perform a genetic evolution. To use the plug-ins, the interface *IOptimizationPlugin* provides the necessary operation contracts.

---

not cost/performance-effective, then the genetic algorithm should be applied to find the best approximation.

Thus, to ensure that this can be achieved with each redesign method, a contract specification was stated by introducing two methods that gene redesign plug-in must implement. The first allows the redesign method to choose if it immediately needs to use the genetic algorithm to redesign the gene sequence, since there may not be a specific algorithm. This method is called *needGeneticAlgorithm*, and obligates the plug-in to return a Boolean value indicating the choice. The second contract is *makeSingleOptimization*, which receives the original gene sequence and other genetic information, and returns another gene sequence, supposedly the redesigned one.

Genetic Operations

On the other hand, when the genetic algorithm is needed, either by plug-in choice, or by using several redesign methods, the genetic operations implemented in the *GeneticAlgorithm* component are applied. To begin using the genetic algorithm, a population is needed to induce the evolution process. Hence, given an initial gene codon sequence, the *generatePopulation* method creates a fixed number of different synonymous sequences. To do this, an algorithm runs through all the sequence, choosing random synonymous codons for each position. The generated sequence is introduced into a set, and the process is repeated until there are enough sequences. This set will constitute the population. The algorithm 4.4 shows the pseudo-code of the implemented function.

One of the more important features of a genetic algorithm (if not the most important) is the calculation of fitness. The method to calculate the fitness regulates the efficiency of the

Algorithm 4.4 Generate Population

```
while Not enough individuals do
  for each Amino Acid position k in original sequence do
    selectedCodon ← randomSynonymousCodon(aminoAcid(k))
    newSequence[k] ← selectedCodon
  end for
  if newSequence doesn't exist in population then
    Population ← newSequence
  end if
end while
return Population
```

solution, the speed of the algorithm, and the reliability of the solution. It is this function that assesses the value of an individual in the population (the punctuation of a sequence), and as it uses the redesign methods to evaluate them (as an AOF function), the plug-ins must respect another contract: the $getFitnessValue$.

Starting from the redesign methods, they may implement very different approaches to redesign, and therefore, the punctuation given by all the methods must respect a common scale in order to ensure equal impact when evaluating a sequence. To do so, a scale based on the improvement in relation to the original sequence was created. Also, the sequence being evaluated is compared with the worst and best possible sequences, to establish a percentage of how good (or how bad) is the sequence in the universe of possible sequences, for that redesign methodology. This way, each redesign method can punctuate sequences with any technique or values, since in the end it returns a ratio to the best/worst sequences. The following function formalizes this concept:

$$Fitness(sequence) = 100 - \frac{sequenceValue - bestValue}{worstValue - bestValue} * 100 \tag{4.8}$$

The function returns a percentage value, that varies from zero (when $sequenceValue$ equals $worstValue$) to one hundred (when $sequenceValue$ equals bestValue). The $bestValue$ and $worstValue$ are calculated by each redesign method, since it depends on that method's functionality. For instance, if the redesign method evaluating the sequence is the Codon Usage, then the $bestValue$ would be the value of a synonymous sequence where all the codons are those with higher codon usage, and the $worstValue$ is the opposite. The value itself could be the sum of the usage of all codons. However, the function does not represent a comparison to the original sequence and it should be subtracted with the percentage value of the original sequence. As a result, the function returns a percentage value between -100 and 100, where positive values mean improvement to the original sequence.

Then, the AOF function of the genetic algorithm component (called $getFitnessScore$) sums all the scores obtained from each redesign method, and weights them as already discussed in section 4.2. Those weights are defined by the user through the user interface, allowing the prioritization of some methods in relation to another. The result becomes the final score of a sequence.

Another important operator is the $makeSelection$, which is responsible for choosing which individuals will get the chance to reproduce. As discussed previously, better fit sequences should be more likely to be picked up, but other sequences should have a chance as well. To

achieve that, a commonly used system called fitness proportionate selection was applied. Each individual has a probability of being selected proportional to its fitness value in comparison to the whole population. Accordingly, an individual with double fitness value has double chance of being selected in relation to the other. The following probability function was applied.

$$P_i = \frac{AOF_i}{\sum_{k=1}^{N} AOF_k} \tag{4.9}$$

Where N is the cardinality of the population and AOF_i is the fitness of the individual i. To achieve this algorithmically and in an efficiently way, the following strategy was used: first sum all the individuals fitness scores; then generate a random number between the smaller fitness value and the previously calculated sum; then run through the population, summing their fitness values until it reaches the random number(or above); select that individual. The following algorithm illustrates that strategy:

Algorithm 4.5 Fitness Proportionate Selection

$TotalScore \leftarrow \sum_{k=1}^{N} AOF(Population[k])$
$rand \leftarrow random(AOF(Population[1]), TotalScore)$
$counter \leftarrow 0$
for $k = 1$ **to** N **do**
  $counter \leftarrow counter + Population[i]$
  if $counter \geq rand$ **then**
    return Population[k]
  end if
end for

This algorithm has complexity $O(n)$, and so it depends linearly on the size of the population to choose an individual. Moreover, the algorithm is run several times to choose a fixed percentage of the population to reproduce.

Finally, to perform the crossover (and also mutation) procedure, a method was constructed, that receives two parent individuals and returns two children sequences. To simultaneously realize the crossover of the two parents and the mutation of the offspring, the following scheme was implemented: for each amino acid of the original gene, choose with a variable probability (as suggested in section 4.2.1) if the codon will have a mutation; then, if mutation is chosen, two random synonymous codons are chosen, one for each of the children; also, if no mutation should happen, then each of the offspring receives the codon from one of the parents, in an arbitrary manner. The algorithm 4.6 clarifies that process.

This way, if no mutation occurs, each child has a 50% probability of taking one of the parents, and they always choose different parents. Also, the $favourableMutationProbability$ function chooses accordingly with the probability described in equation 4.7.

Optimization Runner

To elaborate the whole process of optimizing a gene, an entity called *optimizationRunner* was developed. It receives the redesign requirements chosen by the user, and also the target gene to redesign. When starting, it verifies the number of chosen redesign methods, and if there is only one, the single optimization method is fired. On the other hand, if the genetic

Algorithm 4.6 Reproduction process

```
for each Amino Acid position k in original sequence do
  if favourableMutationProbability() then
    children1[k] ← randomSynonymousCodon(aminoAcid(k))
    children2[k] ← randomSynonymousCodon(aminoAcid(k))
  else
    chosenParent1 ← randomParent()
    chosenParent2 ← otherParent(chosenParent1)
    children1[k] ← chosenParent1[k]
    children2[k] ← chosenParent2[k]
  end if
end for
```

algorithm is needed, this entity is responsible for conducting the evolution process. In figure 4.16 it is depicted an activity diagram to show the progression of this entity.

4.2.3 Verification

To test the influence of the genetic algorithm starting parameters in the final result, a study comprising seven different genes was made. To validate the convergence of the genetic algorithm, each of the genes was redesigned to enhance codon context and GC content simultaneously, repeating the process with several population sizes. From this study, it becomes clear that the resulting sequence AOF value greatly depends on the size of the population (as can be observed in figure 4.17). Also, beyond a certain size of population (that depends on each gene size) there is not more evolution on resulting sequence AOF, meaning that the size of the population has lost influence on the process, probably because the global maximum has been reached and thus cannot be exceeded.

Though it does not serve as a full proof, considering that the population is completely randomly generated, and there is a sharp mutation component that prevents the algorithm from converging to local maxima values, the fact that several runs of the algorithm always converges to the same result indicates that the resulting sequence is most likely the global maxima, or very close to it. Also, the same test was performed with higher mutation rates in order to induce even more randomness to the process, and the final AOF values were the same as the previous test.

4.3 File parsing and management

In order to redesign genes, they must first be read from some database, usually a genome file. Often these files represent a species that was sequenced[1] and therefore contain a representation of the genetic information held in its DNA. Reading and interpreting these files has several concerns and details that must be dealt with in order to correctly extrapolate their information. Also, keeping this information in memory is important to allow the quick opening of genes and specially for making calculations on the genome, like the codon usage

[1]Species genomes are sequenced in laboratory, meaning their DNA chain is read and stored in digital format. The attained result is held in databases.

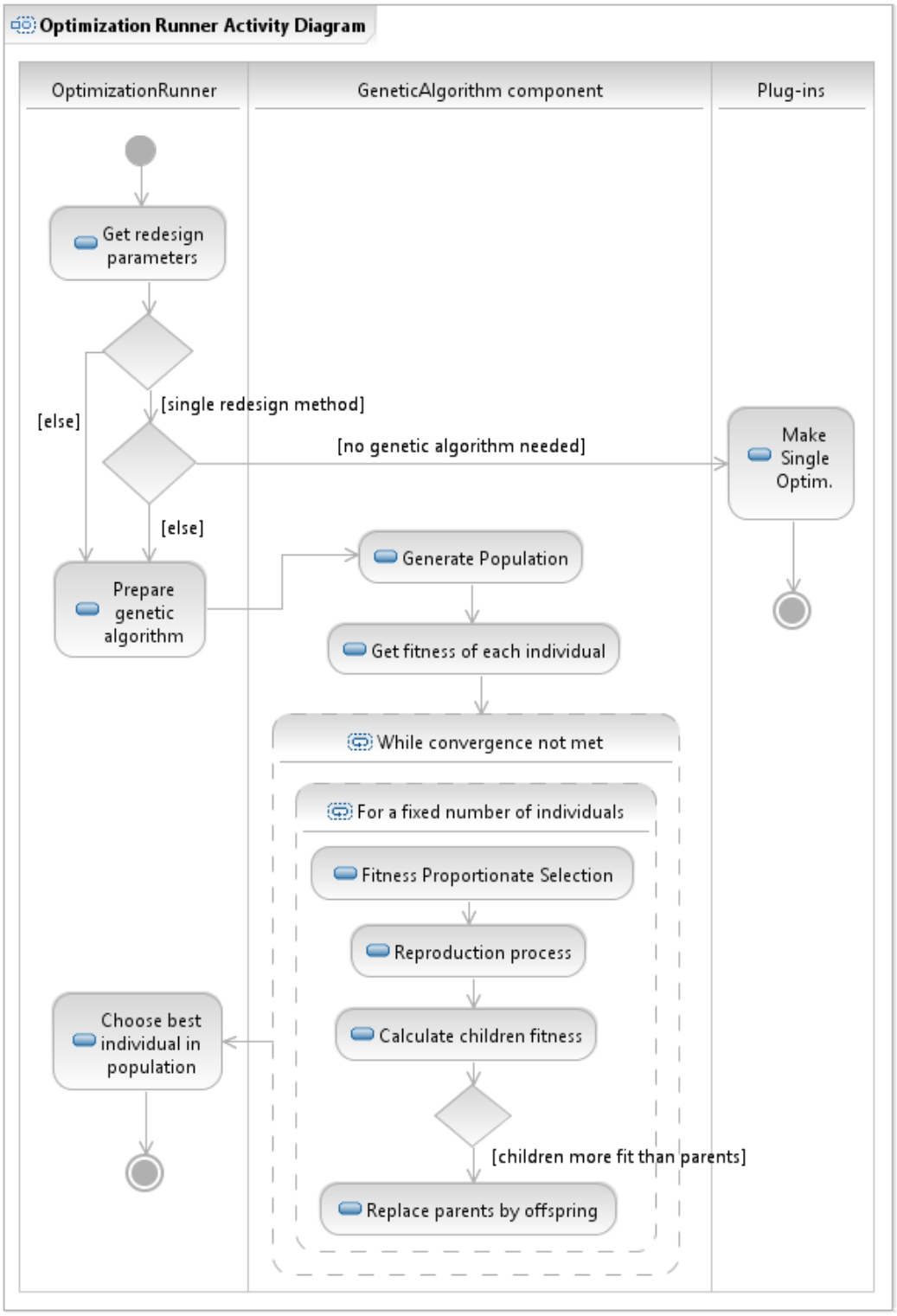

Figure 4.16: Optimization activity diagram.

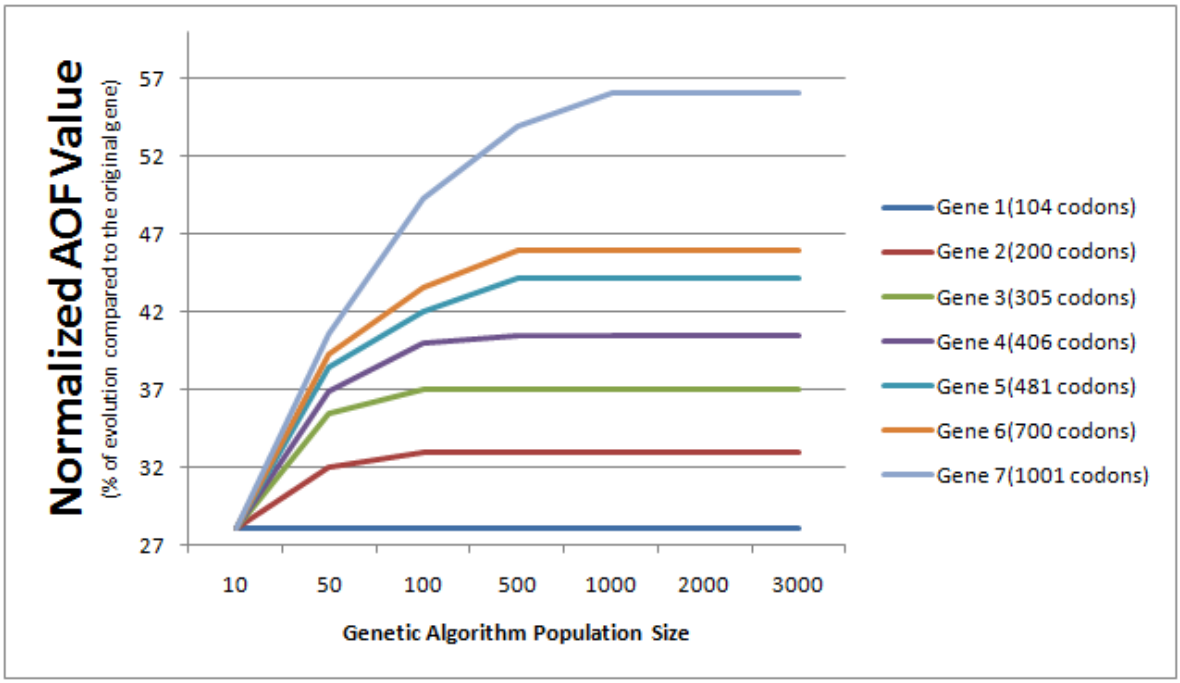

Figure 4.17: The test was made using seven genes from a single species, redesigning for codon context and GC content, considering convergence when no change in the best individual of the population occurs within 200 generations. The AOF values represent the percentage of increase in the punctuation comparing with the original gene. All genes AOF values were normalized to Gene 1, in order for them to start in the same point of the chart.

---

and context tables assessment. Therefore, the correct and efficient maintenance of these files in memory is a task to take into consideration.

The following subsections describe the process of reading the genome files and interpreting its genes, keeping the parsed genome in memory through the use of appropriate structures, and making the necessary calculations on the read genomes to allow some redesign methods to work correctly.

4.3.1 Parsing and validating

Genomes files are usually available in text formats, like the FASTN format. This format is quite trivial to read and understand, even under the human comprehension. A file in the FASTN format is essentially a list of nucleotide sequence, each sequence starting with a line that begins with the character ">". The remaining of that line is considered a comment, and might be the description of the sequence, a database code referencing that sequence, or any other text, and should be ignored. The following lines represent the sequence itself, and contain characters describing its nucleotides (A, G, C and T). Those lines should not be longer than 80 characters. The following text is an example of a FASTN file:

Therefore, reading this format implicates searching for a line starting with a ">" and reading the subsequent lines until another ">" is found. Also, the described nucleotides represent parts of the DNA, but for the context of gene redesign only protein coding sequences (genes) are important. Thus, to assess if a nucleotide sequence is a valid gene sequence, several

```
> Gene1
ATGAAACGCATTAGCAACCATTACCACAGGTAACGGTGCGGGCTGA
> Gene2
ATGCGAGTGTTGAAGTTCGGCGGTTTTCTGCGTGTTGCCGATATTC
TGGAAAGCAATGCCAGGCAGTCCCCCGCCAAAATCACCAACCACCT
GTTAGGAGTCTGA
> Gene3
ATGGTTAAAGTTTATGCCCCGGCTTCCAGTGCCCTCGGGGCGGCGG
TGACACCTGTTGATGGTGCATTGCTCGGAGATACATTCAGTCTCAA
CAACCTCGGACGCTTTGCCGATAAGCTGCCGTATCAGTGCTGGGAG
GCGCACGAGTACTGGAAAACTAA
```

rules must be met. Given that the translation machinery responsible for decoding mRNA sequences into proteins reads three nucleotides (one codon) at a time, it is natural that the number of nucleotides in a gene sequence must be multiple of three. Another constraint is the start and stop codons: a gene sequence must always begin with a start codon and end with a stop codon. Those codons are signal that define the boundaries of each gene, and consequently they must be present in all cases. Furthermore, start codons can also be present in the middle of the gene sequence (since they also decode for an amino acid), however, stop codons cannot be in any other place rather than the end of the gene. Thus, summarizing, a correct gene nucleotide sequence must be a multiple of three, begin with a start codon, end with a stop codon and cannot have stop codons in the middle of the sequence (in frame). All sequences that do not respect these constraints should be ignored.

Moreover, different kinds of species translate codons into amino acids according to different rules. Thus, one species might interpret the AUG codon as the start codon and another species might interpret it as a normal codon, or a single codon might be decoded for different amino acids accordingly to their species. As a result, a table of translation exists (named genetic code table) to assist the decoding of codons in genes. This table indicates which codons decode to which amino acids and which ones are considered start and stop codons, and is divided in sections representing different codes (for instance, the Standard code, which is used to decode the majority of the genomes). The used table was supplied by National Center for Biotechnology Information (NCBI), and has its own format that must also be read and interpreted (see appendix A).

Implementation

Several objects participate in the process of opening and parsing a genome file, but the procedure starts when the user commands the opening of a genome in the *GenePoolGUI* entity. This object is responsible for maintaining the interaction with the user through a user interface. Once the process is fired, the *GenePoolGUI* launches the *GeneLoader* by calling *startGeneLoader* to open each of the users selected files and parse it using the $FastaParser$. Once a file is correctly parsed, it is added to the *GenePool*. The class diagram in figure 4.18 illustrates the associations.

To parse a file, the $FastaParser$ needs to load the file bytes into memory. This process should be efficient, since some genome files might be very big (gigabytes). There are several approaches to this subject, one being the loading of the entire file into memory, and then

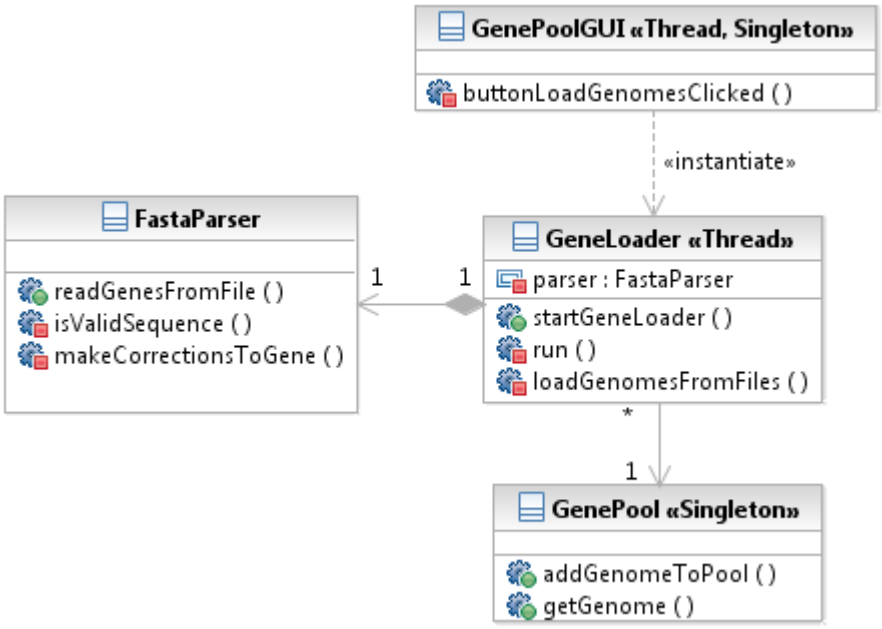

Figure 4.18: Genome Loading Class Diagram.

parsing it, and another being to read the file line by line and parsing it while reading. The first option has the disadvantage of being a serialized process separated in two steps: reading the entire file, and when done, parsing it. On the other hand, the second option has the advantage of being able to read and parse in a parallel fashion. However, reading and parsing simultaneously also has the great disadvantage of having to wait for a hard disk reading. This waiting can be very long compared to accessing the memory, and depends on how the bytes are scattered in the hard drive. Thus, a midterm between the two strategies was built to improve the reading/parsing efficiency: A buffered reader reads bytes from the file while buffering more bytes than currently needed. This was, each time another line of the file is needed, the line is already in memory, and thus quickening the reading process.

The process continues by reading the file until it has a complete sequence in memory (the nucleotides between two ">"). If that sequence is valid, it is saved in memory for further use. The whole process terminates when the file reaches the end. The activity diagram in figure 4.19 illustrates this process.

Also, this process is repeated for each file indicated by the user. The parsed genes that are valid are kept in a genome structure. Each genome structure is then added to the *GenePool* by invocating the *addGenomeToPool* method (more details in section 4.3.2). Furthermore, all this processing is made in a background process different from the user interface thread, meaning that all the user interfaces can be open/closed, or even other genomes can be loaded while the opening/parsing in the *GeneLoader* is running.

4.3.2 Storing in memory

Another issue that needs attention is the problem of storing the data in memory. After reading, parsing, and choosing only the valid gene sequences, they have to be kept in memory

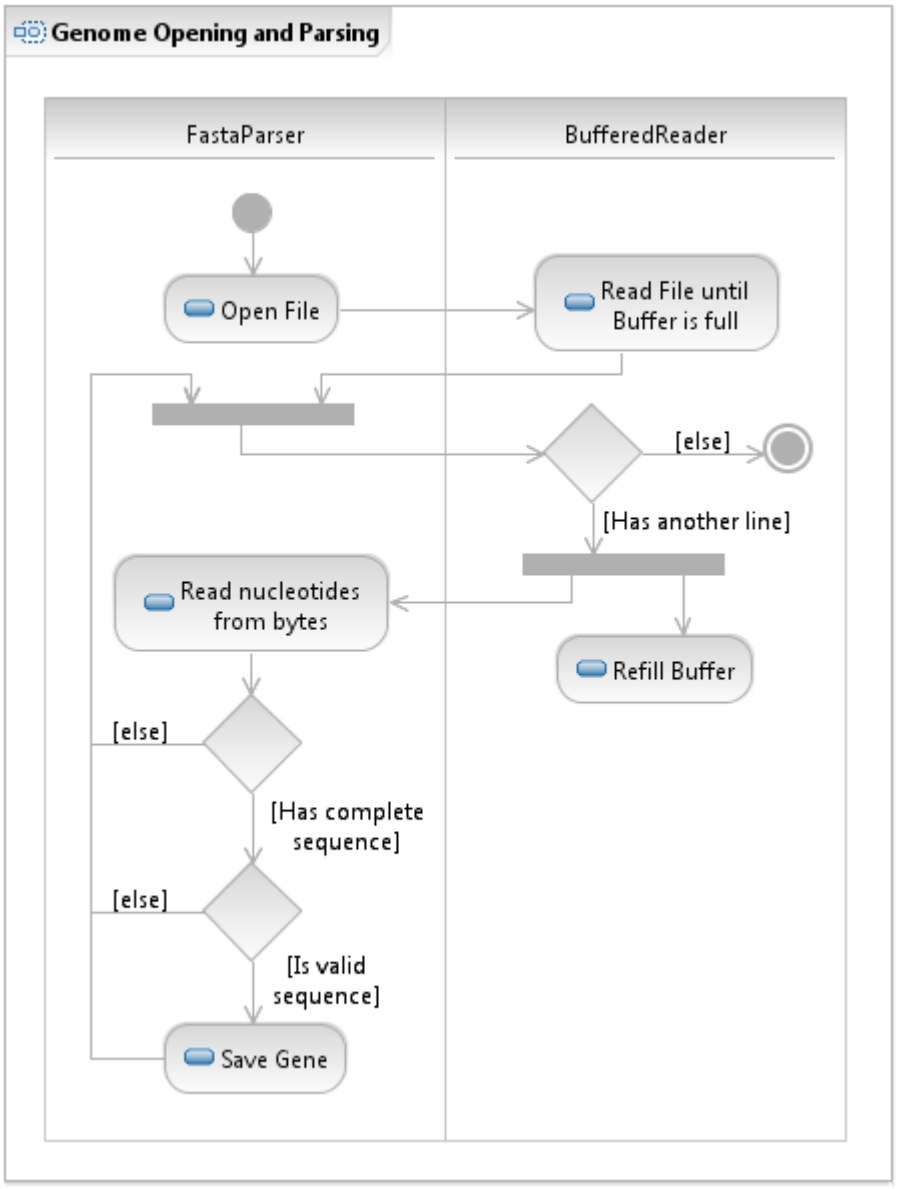

Figure 4.19: Reading Parsing Activity Diagram.

for making calculations on the genomes and for using genes in redesign studies. However, genome files can be very large, reaching hundreds or even thousands of megabytes. The files are usually plain text files encoded in ANSI[1] code or UTF-8, meaning each character of the text occupies 1 byte in disk. Nevertheless, using the Java programming language and environment, characters in the native data type occupy 2 bytes. This means using twice the capacity to keep the data on memory. This issue could become problematic when the user manipulates several large genomes simultaneously, and could easily lead to the lack of free

[1] Also known as Windows-1252, this is a character encoding of the Latin alphabet used in Microsoft Windows systems. It codes 255 different characters, using 1 byte.

memory to keep the whole data and the operating system would resort to swapping some of the data into hard disk. Also, another problem is the fact that the data type String in Java is immutable, meaning that once a String is created, it cannot be changed unless by creating another String object. Moreover, there is no process to directly indicate the Java Virtual Machine (JVM) to free up memory, but instead a garbage collector deals with unreferenced memory.

Thus, manipulating a genome in memory could be a heavy task both to the memory and processor, since data is double the original size and each modification means a copy without erasing the original data in memory. Therefore, the String object is not the most indicated to perform such task. To overcome this problem, a new data type called *ByteString* was created with the intention of allowing the direct manipulation of strings of text (in this context, strings of nucleotide letters) without the need to copy the object, as with String. Also, by using the Java native byte type to keep characters, each character occupies 1 byte in memory. This structure also holds several methods to manipulate the byte array that contains the characters as if it was a String, granting ease of manipulation.

Also, to correctly store the data, several other classes were created. The sequence bytes are not enough to define a gene, as it comprises other information, such as the gene name, secondary structure, and orthologs, among other. Thus, a structure named *Gene* was created to hold the gene structures (primary and secondary) and all the other information. Another structure called *Genome* gathers all the genes in an opened genome, and several other information regarding the genome species. Moreover, the *GenePool* structure holds all the open genomes. Therefore, the class diagram in figure 4.20 illustrates the associations between each structure.

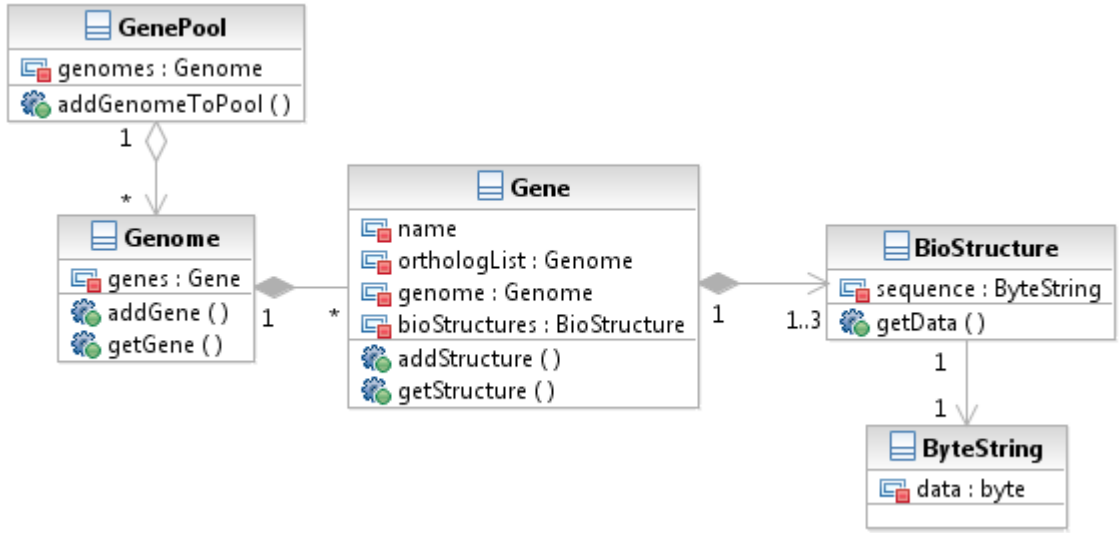

Figure 4.20: Internal database class diagram.

Furthermore, to hold information about the various structures that are related to a gene, a *BioStructure* class was integrated into the *Gene* class. This structure allows the gene to store an mRNA primary structure (codon sequence) and a resulting protein primary (amino acid sequence) and secondary structures. As a result, a gene can have several different structures

under the same data type (*BioStructure*), instead of having one class for each biological structure. Also, if more biological structures are needed, no more classes have to be created, but instead another entry added to the *StructureType* enumeration.

4.3.3 Post calculation

After opening, parsing and storing the files in memory, the genomes are immediately made available to the user. However, several redesign algorithms resort to calculations such as the codon usage or the codon context for its redesign. Thus, before the redesign methods can be used, these tables must be calculated. This process might take some time, as all the genes have to be taken into account to compute the codon usage and codon context.

One approach to this problem would be making the counting while reading the genes. However, this would induce an unnecessary overhead to the parsing process, causing it to take a longer time. The parsing process should be quick, in order to rapidly provide the genes to the user. Thus, another approach is to provide the genes/genomes to the user as soon as the parsing ends and start another process to perform the necessary calculations. As a result, the user could quickly load the genes, and in the mean while the codon usage and context tables are computed. By the time the tables would be needed, they are probably already available (taking into consideration the time it takes to calculate the tables, and the time the user takes to load a gene and start using a redesign method that needs the tables). If a process needs a codon context or usage table and it is not available yet, the process blocks until the usage/context calculation ends.

To implement that process, a new entity named *UsageAndContextTables* was designed (figure 4.21). This entity is started by the GeneLoader when the parsing process ends, and runs alone by analysing the genes on a single genome making the necessary calculations.

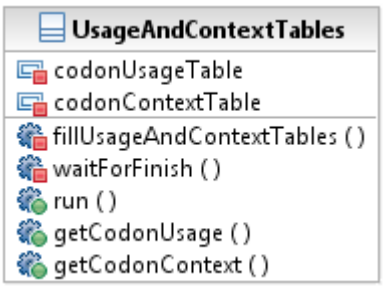

Figure 4.21: Entity responsible for making the calculations about codon usage and codon context.

Each time one of the tables is needed, this class is used by calling the *getCodonUsage* or the *getCodonContext* methods. However, when the tables are still not available for use, the call to these methods must be blocked until they are ready. Therefore, the method *waitForFinish* is called in the beginning of each of those methods to verify if the calculations are done (and if not, the call waits there until notified).

4.4 EuGene

The modular architecture, optimization system and the ability to efficiently parse and store genes, culminated into an application to meet most of the requirements. In this section, some details about the application structure, building, and resulting user interface, are discussed.

4.4.1 Overview

The application was divided into several packages for understandable separation of concerns. Each package represents an important sector of the application, and classes are built and distributed in order to maintain low coupling and high cohesion between packages. For instance, the entities responsible for the genome file opening and parsing are in the package *FileOpeningParsing*, while the parsed data is stored in the *GenePool* class inside the *GeneDB* package. Also, to assist the user in selecting the files to open and start the parsing process, the graphical user interface from the *GenePoolGUI* class - inside the *GUI* package - is used. This kind of separation resorts to the Model View Controller (MVC) architectural pattern, where the model is the GenePool, the viewer is the GenePoolGUI, and the controllers are the classes from FileOpeningParsing that handle the file parsing. The package diagram in figure 4.22 illustrates the arrangement and relations between each package in the application.

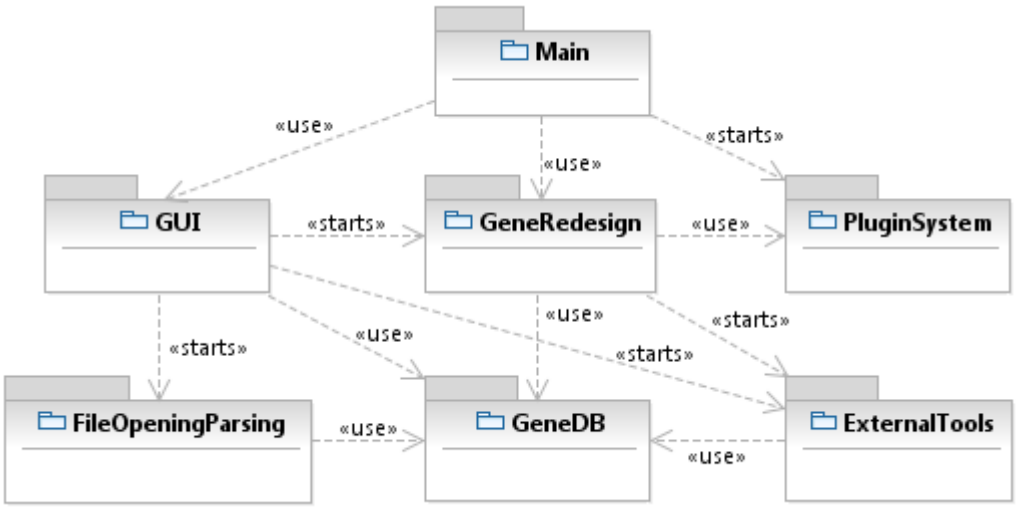

Figure 4.22: General package diagram. Most of the relations are common ≪use≫ associations, but some are only referred as ≪start≫ associations since they only represent the launching of an entity. For instance, as soon as the application starts (in the main package), the PluginLoader entity (PluginSystem package) is started as a thread to run alone, and there is no other direct communication between both packages.

These packages are responsible for several requirements related to their concerns. For instance, the *ExternalTools* package accounts for the use cases Align Orthologs, Fetch Orthologs, Calculate secondary Structure and Fetch Tertiary Structure, since these use cases need external tools to obtain their results. External tools might be servers that provide web

services like the Protein Data Bank (PDB)[1] and European Bioinformatics Institute (EBI)[2] servers, or local tools that perform calculations, like the secondary structure computation (achieved with the PsiPred tool [31]). The use case diagram in figure 4.23 illustrates these relations.

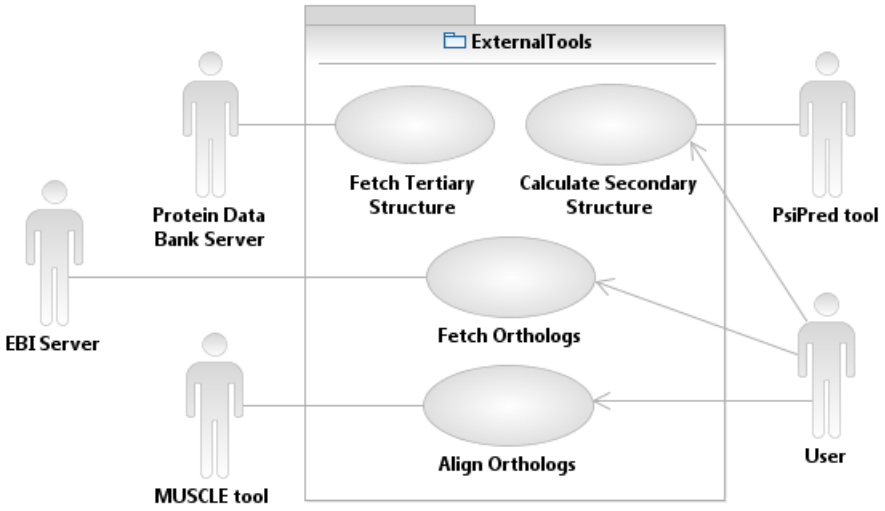

Figure 4.23: External Tools use cases. The external tools are explicit as actors, except for the user.

On the other hand, the gene redesign package makes use of external classes rather than tools, by reading class files from a file system folder. The user might use the application to redesign a gene, to analyse it, or to multi-redesign a gene, but in any way, these use cases will get their ultimate execution from the loaded plug-ins (even considering that a genetic algorithm will be used). The use case diagram in figure 4.24 illustrates the idea.

Also accessing the file system is the FileOpeningParsing package. By user command, the package classes start reading and interpreting genome files by communicating with the file system and using the genetic code table to assist in the parsing process. The main use case in this package is the "Open FASTN Files" which handles the opening/parsing procedure after the user having selected the files to open and the genetic code table to use. This use case also calls the "filter genes" and the "calculate codon usage and context" use cases to handle the requirements of selecting only valid genes and automatically compute the codon usage/context of opened genomes. An illustration of this usage can be seen in figure 4.25.

To guide the user into performing gene redesign, study and analysis, viewing gene secondary and tertiary structures, opening genomes, among other tasks, the GUI package supplies these use cases resolution. As the application is user event guided (all actions occur by user initiative), a strong user interaction support is of major concern, and therefore, this package cooperates with the main packages in order to provide easy access to complex tasks. To ease the understanding of the image, the diagram in figure 4.26 does not show the relations with other packages, but almost all use cases use the processing supplied by other packages.

[1] http://www.pdb.org/
[2] http://www.ebi.ac.uk/

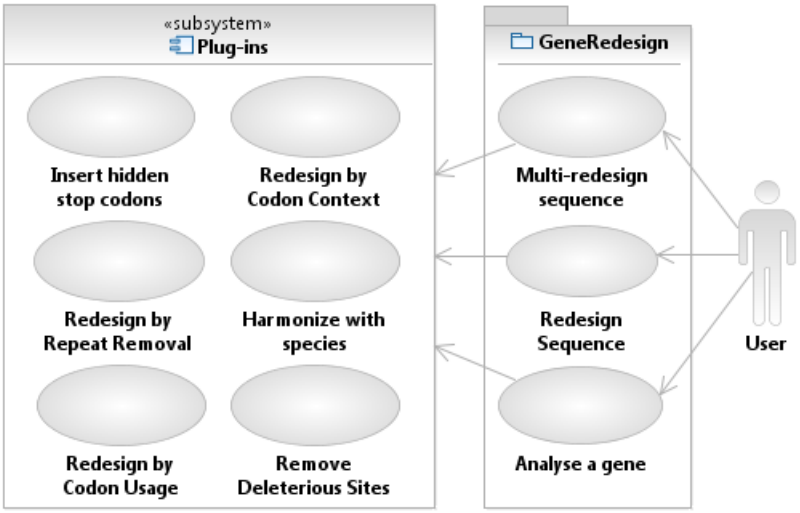

Figure 4.24: Gene Redesign use cases diagram.

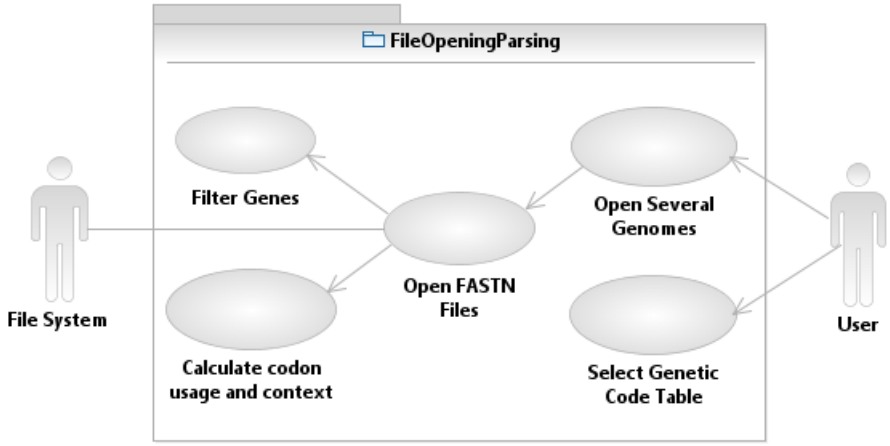

Figure 4.25: File Opening and Parsing use cases diagram.

4.4.2 Parallelization

In the scope of an application, several tasks can run concurrently by dividing the process into several different running entities. These entities (named threads) exist in the context of a process, and share several resources, one of them being the memory. The concurrent characteristic allows different threads to perform separate tasks at the same time[1]. Also, threads are lighter entities than processes and the communication between them is facilitated

[1]This characteristic varies depending on the computational system on which the application lies on: in a single processor/core, the actual parallelization is only perceived by the user, as the time dedicated by the processor is divided by the threads in a frequent manner that seems to be simultaneous (time-division multiplexing). However, in multi processor/core systems, different threads usually run in different cores, achieving true parallelization.

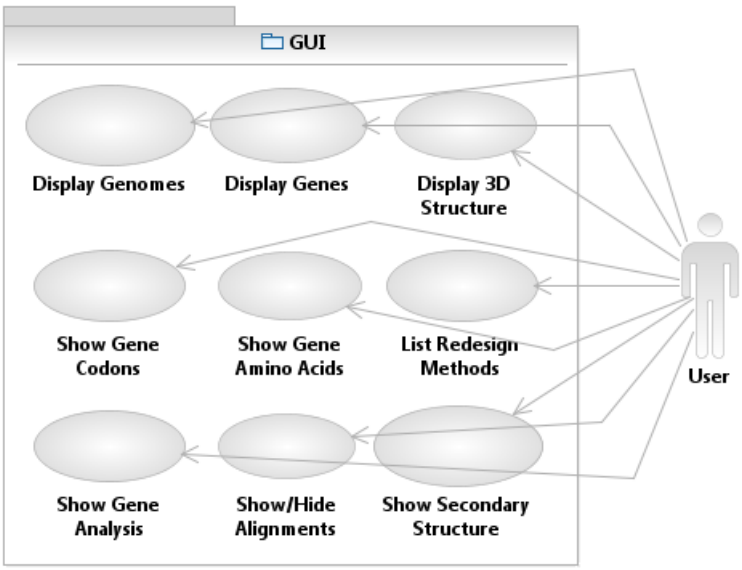

Figure 4.26: Graphical User Interface use cases diagram.

as a result of the memory sharing.

Various factors contribute to the positive use of threading in the application. One of them is that the majority of new computers comes with multi-core or multi-processing systems, thus allowing the use of threads in a true parallel fashion. The use of threads may supply performance enhancement in many occasions, especially when more than one calculation or procedure can be achieved simultaneously. Moreover, when communicating with each other, threads must take synchronism issues into account to guarantee the correct use of the shared memory, but the overhead given by synchronization is derisive because in this application the threaded procedures are mainly to process information or communicate with exterior entities, thus having few concurrent situations where synchronism is essential to guarantee performance. Using this approach, several instances of computation in the application can be made in a parallel thread, thus allowing the user to perform other tasks without waiting for the first to finish. One clear usage of this strategy in the application is executing a gene multi-redesign (a task that may take some time to finish) in an independent thread. As a result, the user is able to proceed with other uses of the application (even other gene redesigns) until the redesign procedure is over and the result is shown. The sequence diagram in figure 4.27 illustrates this idea.

So the application launches threads and continues its work until some thread signals that results are available, thus having essentially an asynchronous relation with threads. Also, the more threaded procedures there are in the application, the more flexibility the user has to perform multiple simultaneous tasks. Following that strategy, several actions in the application were made independent through the use of threads. These threads act alone until the processing is done and some other thread is signalled that the result is available[1]. The

[1]Threads do not always signal the main application when results are available, but this is the general case. Some threads send the results directly into some container, like the *GeneLoader* puts the loaded genomes into the *GenePool* after opening/parsing a file.

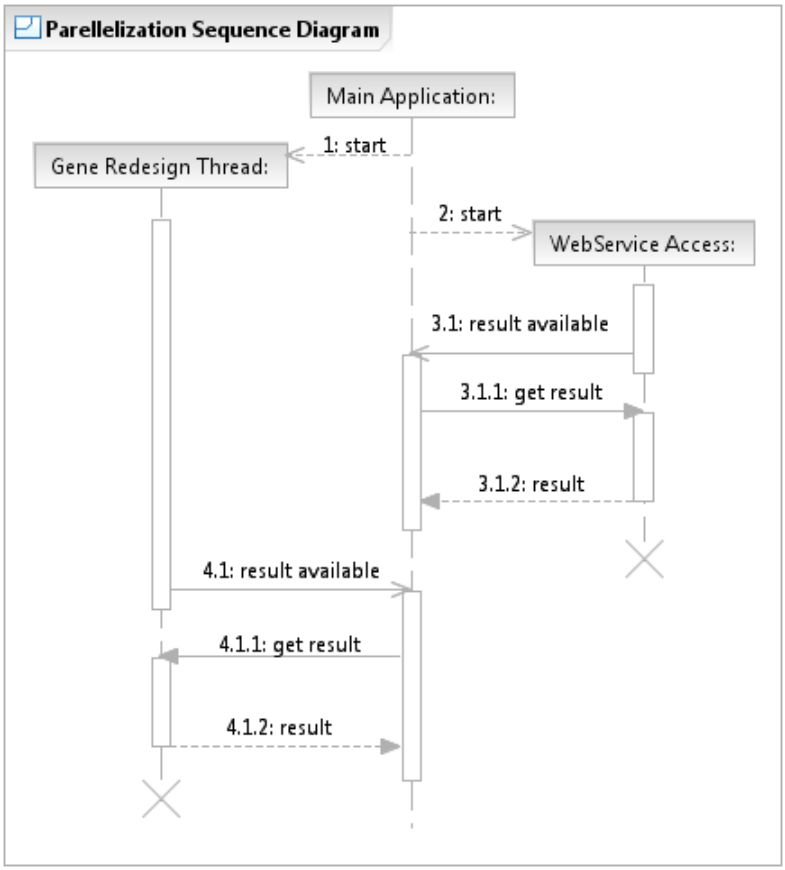

Figure 4.27: Parallelization Sequence Diagram. Three threads are presented: the middle one, Main Application represents the main thread, and launches two other threads to perform different tasks. The Main Application continues available to perform other tasks. When the threads are finished, they signal the main thread to fetch the result.

---

table 4.1 lists the threaded procedures in the application.

4.4.3 User Interface

From a practical perspective, the application had to meet the visual and user interaction requirements explicit in chapter 3. In order to fulfil those requirements, the presented visual sketch served as a guide to the final appearance. The screenshot presented in figure 4.28 demonstrates the final application façade.

Hence, the layout was achieved in a general matter: the available gene redesign algorithms on the left (each containing its own parameters), the loaded genes in the centre (in the mockup only one gene per study was considered, but having more genes turned out to be more flexible to the user to compare them), and the three dimensional conformation of the resulting protein on the right side of the window. In addition, some other panels were created to increase the available information and interaction with the user: the progress panel (below the redesign algorithms) shows the progress of currently working threads, so the user can have feedback on the state of tasks like gene redesign or orthologs alignment, that can take some time to

Table 4.1: Threaded procedures in the application.

File Opening/Parsing
File Buffering
Codon usage/context calculation
Gene Redesign
Gene Analysis
Secondary Structure Calculation
Tertiary Structure Fetching
Ortholog Fetching
Ortholog Alignment
GUI Display

complete; Also, the gene information panel (below the three dimensional structure viewer) lists details on the selected gene, like its GC content, number of codons it has, gene name, genome name, etc.

4.5 Summary

The proposal of an architectural plug-in model for the support of extensibility by the application is presented in this chapter. Since extensibility is crucial, and a major part of the application (the redesign methods) relies on different algorithms, the best suited approach in this context is a modular architecture, based on plug-ins. This concept allows the application to be independent from the algorithms it uses, reducing the coupling between functionalities while maintaining high cohesion of the different parts. Using a contract to encapsulate redesign methods also allows the access to completely different modules with the same language (the contract itself), and modules can be created outside of the application without the need to reconstruct it, just by respecting the interface.

As the main purpose of the application is the redesign of sequences and one of the requirements is the multi-objective redesign, optimization is presented as a mathematical problem with several distinct solutions that were explored. It was exposed that searching through the whole solution space (one by one) is an unfeasible task, given that the search space grows exponentially with the number of codons in the sequence. Some algorithms try to navigate through the solution space selecting only solutions that might lead to the global best solution. These algorithms, such as simulated annealing, offer a trade off between the quality of the final solution, and the time searching for it. Other methods like iterating the redesign methods or iterating the original sequence often do not return the global optimum, or not even a good result, since they do not have a general overview of the optimization process but instead optimize locally or in a sequential manner. A genetic algorithm was presented as the best approach to this problem, by simultaneously optimizing several synonymous sequences and combining them in a natural evolution and selection based process. Also, genetic algorithms allow high scalability in the process, by using an AOF as the fitness function, and thus making it possible to have a very flexible customization of parameters.

Also, the strategy used to open, parse, validate and store genes in memory was presented. Genome files can be very big, and consequently its management has to be carefully planned. Several entities and structures were built to hold the genetic information and perform parsing

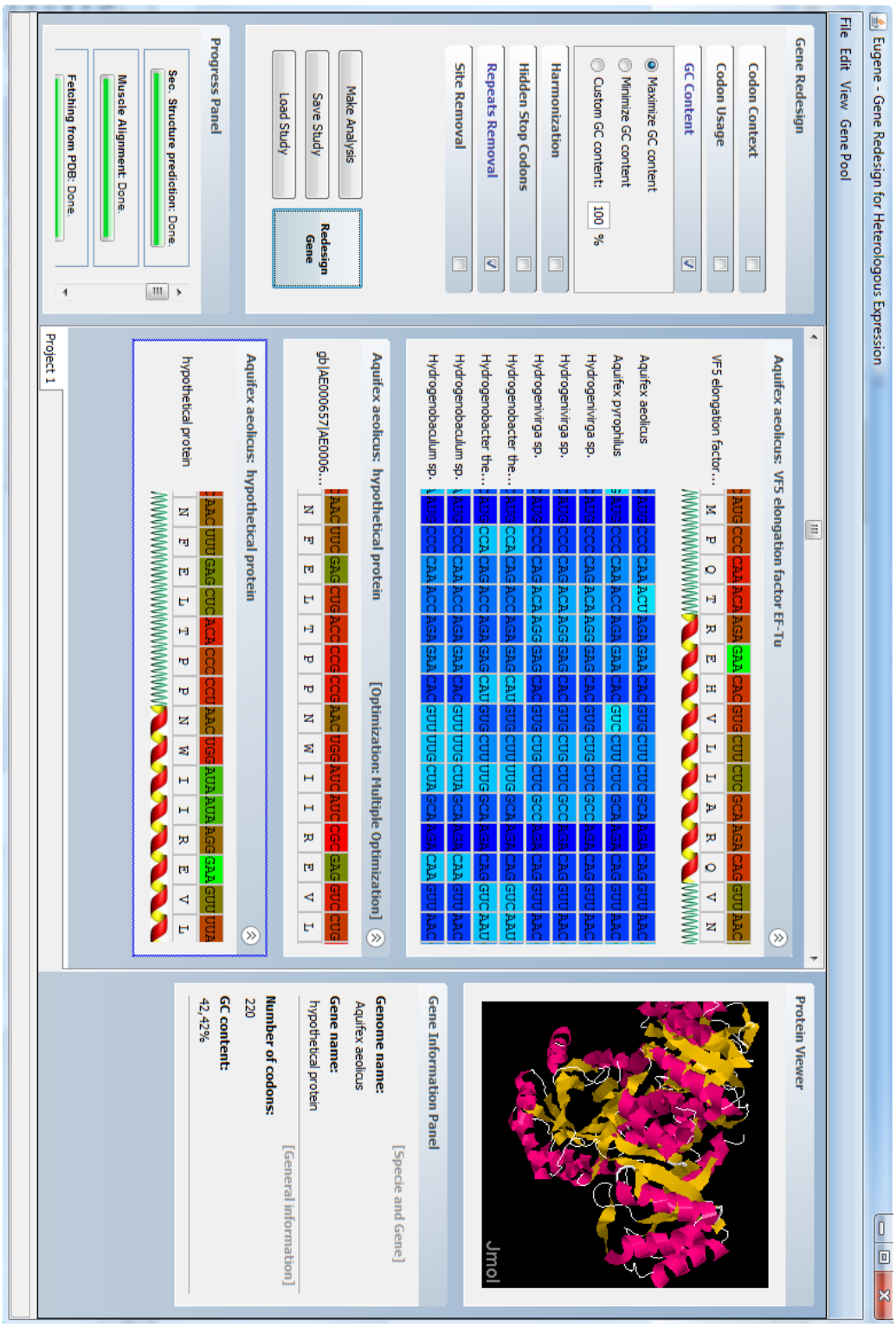

Figure 4.28: EuGene application final appearance.

and validating operations. To provide the opened data to the user the quickest possible, several approaches were made, like the use of buffered file readers, and make usage and context calculations in background after making the genome available to the user. Many other procedures were made concurrently to take advantage of multi-core and multi-processor systems and enhance the performance of the application.

Chapter 5

Conclusions

The main targets of research in this thesis were the gathering of already available algorithms to perform gene redesign by codon synonymous replacement, the development of a supportive platform to hold the heterogeneity in redesign algorithms, and the study of optimization processes that could help in achieving the best results when redesigning genes for one or more redesign constraints.

To achieve these goals, an application that allows researchers to successful use several algorithms to redesign genes was developed. This application uses a constructed plug-in architecture to support the different algorithms by loading them from separate modules, thus opening space for any kind of redesign method given that it respects a specified interface. This too allows that redesign algorithms can be easily integrated into the application, without having to rebuild it, in a plug-and-play manner. Many redesign algorithms were studied and implemented as plug-ins for the application, such as codon usage, codon context, repetition removal, introduction/removal of out-of-frame stop codons, among others. Finally, the most important piece of the application stands in the optimization elements. To maximize the gain of a set of chosen gene redesign algorithms, several optimization processes were studied, and using genetic algorithms turned out to be the most advantageous way of calculating the best synonymous codon sequence that meets all redesign requirements. It is not a trivial task to find the best sequence for any redesign parameters, given the quantity of possible synonymous sequences and the variety of algorithms. The majority of the redesign methods cause their choices to override the choices of other algorithms.

The majority of the requirements listed in chapter 3 was met by the final application, by achieving the discussed goals and also by integrating useful tools into the platform, such as gene alignment, secondary structure prediction, 3D protein visualization and accessing important on-line genetic databases web services to fetch useful information.

5.1 Result

The outcome of this research will likely have some impact on gene redesign and study processes, considering the available tools and their capacities. Moreover, with the constructed application, dissimilar redesign algorithms can be easily integrated to work together, and thus granting the means to redesign genes according to many specific parameters, yielding more flexibility and control to researchers when designing genes for protein expression. This integration allows future construction and insertion of algorithms, accounting for the constant

evolutionary characteristic of the biology field. Also, the use of genetic algorithms combined with simulated annealing can be advantageous to optimize gene codon sequences, as it returns good results in linear time, allows the combination of several redesign methods simultaneously, and offers a strong capacity to escape from local maxima results.

To conclude, the diagram in figure 5.1 illustrates a SWOT analysis.

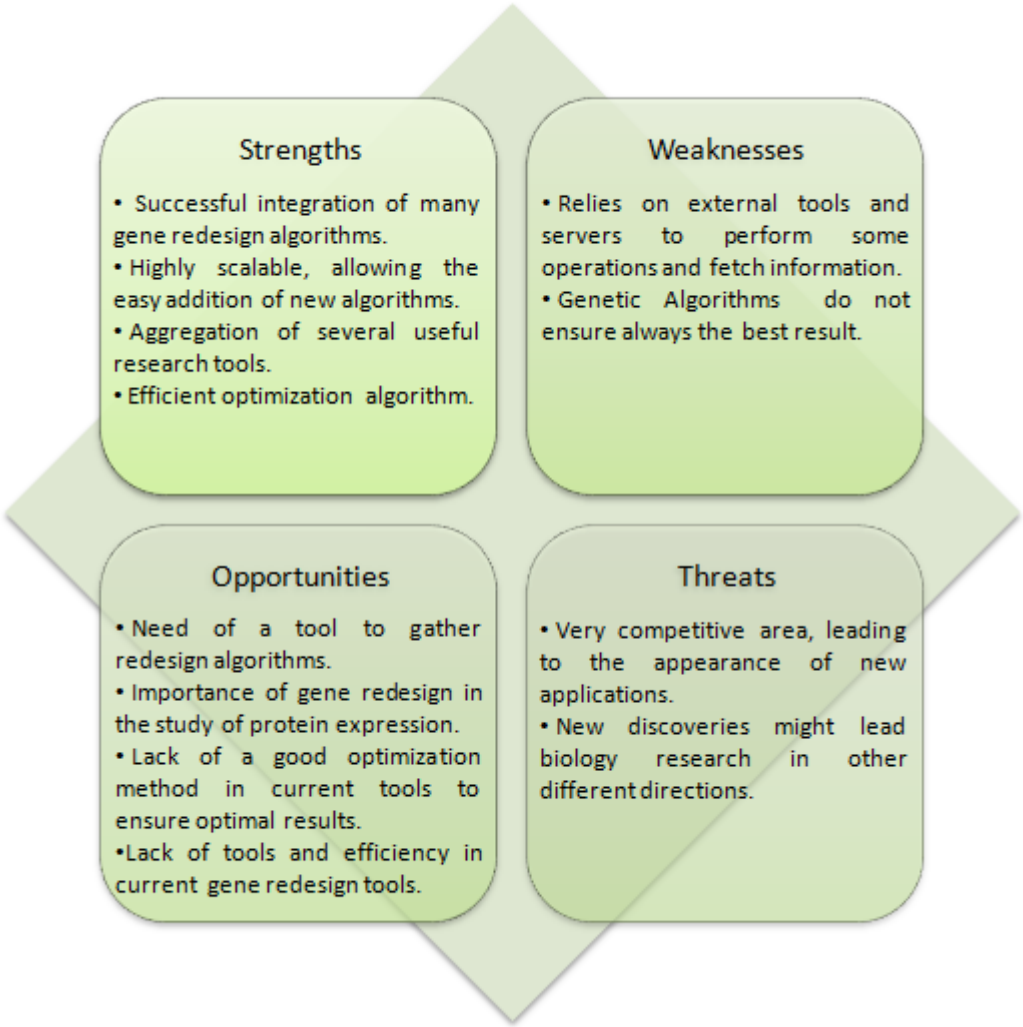

Figure 5.1: SWOT Analysis of the application.

5.2 Future Work

Some requirements were not met during the development of the application, and should be the main target of attention when continuing its construction. The main requirement that lacks in the final result, is the ability to make comparisons between several genes using a chart, by displaying values of different genes (like the codon usage) in order to compare them. This would allow researchers to compare genes before and after redesign, for example.

Some other features could be even more explored, like the optimization process. In the genetic algorithm, the percentage of population that will be chosen to reproduce is fixed, but it should instead be dynamic and dependent on the size of the input gene sequence. Also, as discussed before, the resulting sequence value varies with the number of individuals in the population, but it always converges to the same sequence value beyond a certain size of the population, meaning it got to the global maximum. This population size also depends on the gene codon sequence size, needing bigger populations with longer input sequences. Finding the relation between sequence size and population size could be used to calculate the necessary population that guarantees correct convergence. Many other parameters in the gene redesign machinery could be optimized to increase the speed and accuracy of redesign processes.

Furthermore, a major improvement of the application is the compression of data, both in memory and disk, by developing algorithms to compress sequences, since one nucleotide letter occupies one byte (using the ByteString object), but can only take four values (A, C, G and T). To represent these values, only two bits are necessary, and thereby, at least a fourfold compression is possible. This could lead to faster opening of genes, and less occupied space in the main memory.

Also, only the main part of the application was built (the opening/parsing, redesigning, and plug-in platform). Many other features were still not developed, like saving redesigned genes to file, saving projects or workspaces, implementing many other redesign methods, adding preferences to the application, try to parse gene headers from FASTN files, improving the user interface in several ways (adding information to the status bar, etc.), try to obtain the codon usage table from servers instead of calculating it, etc.

Appendix A

Genetic Code Table

The used table was supplied by NCBI, and has the following format:

```
Genetic-code-table ::= {
 {
  name "Standard" ,
  name "SGC0" ,
  id 1 ,
  ncbieaa  "FFLLSSSSYY**CC*WLLLLPPPPHHQQRRRRIIIMTTTTNNKKSSRRVVVVAAAADDEEGGGG"
  sncbieaa "---M---------------M---------------M----------------------------"
  -- Base1  TTTTTTTTTTTTTTTTCCCCCCCCCCCCCCCCAAAAAAAAAAAAAAAAGGGGGGGGGGGGGGGG
  -- Base2  TTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGG
  -- Base3  TCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAG
 },
 {
  name "Vertebrate Mitochondrial" ,
  name "SGC1" ,
  id 2 ,
  ncbieaa  "FFLLSSSSYY**CCWWLLLLPPPPHHQQRRRRIIMMTTTTNNKKSS**VVVVAAAADDEEGGGG"
  sncbieaa "--------------------------------MMMM---------------M------------"
  -- Base1  TTTTTTTTTTTTTTTTCCCCCCCCCCCCCCCCAAAAAAAAAAAAAAAAGGGGGGGGGGGGGGGG
  -- Base2  TTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGGTTTTCCCCAAAAGGGG
  -- Base3  TCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAGTCAG
 },
 (...)
 }
```

The lines that being with *ncbieaa* indicate the resulting amino acid from the codons indicated by the nucleotides in the lines that begin from *Base1* to *Base3*. The lines that begin with *sncbieaa* indicate alternative start codons. This table was also parsed from an input file in order to correctly decode nucleotide sequences into amino acid sequences.

Bibliography

[1] "Mephitis european project," Jun 2010. [Online]. Available: http://www.mephitis.eu/

[2] J. Venter, K. Remington, J. Heidelberg, A. Halpern, D. Rusch, J. Eisen, D. Wu, I. Paulsen, K. Nelson, W. Nelson *et al.*, "Environmental genome shotgun sequencing of the Sargasso Sea," *Science*, vol. 304, no. 5667, p. 66, 2004.

[3] A. Lehninger, D. Nelson, and M. Cox, *Lehninger principles of biochemistry*. Wh Freeman, 2005.

[4] M. Welch, A. Villalobos, C. Gustafsson, and J. Minshull, "You're one in a googol: optimizing genes for protein expression," *Journal of The Royal Society Interface*, vol. 6, no. Suppl 4, p. S467, 2009.

[5] E. Angov, C. Hillier, R. Kincaid, and J. Lyon, "Heterologous protein expression is enhanced by harmonizing the codon usage frequencies of the target gene with those of the expression host," *PLoS One*, vol. 3, no. 5, 2008.

[6] A. Fedorov and T. Baldwin, "Cotranslational protein folding," *Journal of Biological Chemistry*, vol. 272, no. 52, p. 32715, 1997.

[7] M. Gouy and C. Gautier, "Codon usage in bacteria: correlation with gene expressivity," *Nucleic Acids Research*, vol. 10, no. 22, p. 7055, 1982.

[8] J. Parker, "Errors and alternatives in reading the universal genetic code." *Microbiology and Molecular Biology Reviews*, vol. 53, no. 3, p. 273, 1989.

[9] H. Salim and A. Cavalcanti, "Factors influencing codon usage bias in genomes," *Journal of the Brazilian Chemical Society*, vol. 19, pp. 257–262, 2008.

[10] G. Lithwick and H. Margalit, "Hierarchy of sequence-dependent features associated with prokaryotic translation," *Genome research*, vol. 13, no. 12, p. 2665, 2003.

[11] G. Moura, M. Pinheiro, J. Arrais, A. Gomes, L. Carreto, A. Freitas, J. Oliveira, and M. Santos, "Large scale comparative codon-pair context analysis unveils general rules that fine-tune evolution of mRNA primary structure," *PLoS One*, vol. 2, no. 9, 2007.

[12] R. Buckingham, "Codon context and protein synthesis: enhancements of the genetic code," *Biochimie*, vol. 76, no. 5, pp. 351–354, 1994.

[13] G. Moura, M. Pinheiro, R. Silva, I. Miranda, V. Afreixo, G. Dias, A. Freitas, J. Oliveira, and M. Santos, "Comparative context analysis of codon pairs on an ORFeome scale," *Genome Biology*, vol. 6, no. 3, p. R28, 2005.

[14] M. Pinheiro, V. Afreixo, G. Moura, A. Freitas, M. Santos, and J. Oliveira, "Statistical, Computational and Visualization Methodologies to Unveil Gene Primary Structure," *Methods Inf Med*, vol. 45, pp. 163–8, 2006.

[15] P. Yakovchuk, E. Protozanova, and M. Frank-Kamenetskii, "Base-stacking and base-pairing contributions into thermal stability of the DNA double helix," *Nucleic acids research*, vol. 34, no. 2, p. 564, 2006.

[16] R. Levin and C. Sickle, "Autolysis of high-GC isolates of Pseudomonas putrefaciens," *Antonie van Leeuwenhoek*, vol. 42, no. 1, pp. 145–155, 1976.

[17] H. Seligmann and D. Pollock, "The ambush hypothesis: hidden stop codons prevent off-frame gene reading," *DNA and cell biology*, vol. 23, no. 10, pp. 701–705, 2004.

[18] V. Phan, S. Saha, A. Pandey, and W. Tit-Yee, "Synthetic Gene Design with a Large Number of Hidden Stop Codons," in *IEEE International Conference on Bioinformatics and Biomedicine, 2008. BIBM'08*, 2008, pp. 141–146.

[19] D. Brégeon, V. Colot, M. Radman, and F. Taddei, "Translational misreading: a tRNA modification counteracts a+ 2 ribosomal frameshift," *Genes & Development*, vol. 15, no. 17, p. 2295, 2001.

[20] J. Shine and L. Dalgarno, "The 3'-terminal sequence of Escherichia coli 16S ribosomal RNA: complementarity to nonsense triplets and ribosome binding sites," *Proceedings of the National Academy of Sciences*, vol. 71, no. 4, p. 1342, 1974.

[21] H. Jin, Q. Zhao, E. de Valdivia, D. Ardell, M. Stenstr
"om, and L. Isaksson, "Influences on gene expression in vivo by a Shine–Dalgarno sequence," *Molecular microbiology*, vol. 60, no. 2, pp. 480–492, 2006.

[22] M. Welch, S. Govindarajan, J. Ness, A. Villalobos, A. Gurney, J. Minshull, and C. Gustafsson, "Design parameters to control synthetic gene expression in Escherichia coli," 2009.

[23] A. Williams, L. O'Brien, R. Phillpotts, and S. Perkins, "Improved efficacy of a gene optimised adenovirus-based vaccine for Venezuelan equine encephalitis virus," *Virology Journal*, vol. 6, no. 1, p. 118, 2009.

[24] J. Newcomb, R. Carlson, and S. Aldrich, "Genome synthesis and design futures: implications for the US economy," *Bio economic research associates*, 2007.

[25] M. Widmann, M. Clairo, J. Dippon, and J. Pleiss, "Analysis of the distribution of functionally relevant rare codons," *BMC genomics*, vol. 9, no. 1, p. 207, 2008.

[26] D. Lorimer, A. Raymond, J. Walchli, M. Mixon, A. Barrow, E. Wallace, R. Grice, A. Burgin, and L. Stewart, "Gene Composer: database software for protein construct design, codon engineering, and gene synthesis," *BMC biotechnology*, vol. 9, no. 1, p. 36, 2009.

[27] A. Villalobos, J. Ness, C. Gustafsson, J. Minshull, and S. Govindarajan, "Gene Designer: a synthetic biology tool for constructing artificial DNA segments," *BMC bioinformatics*, vol. 7, no. 1, p. 285, 2006.

[28] C. Kessler and V. Manta, "Specificity of restriction endonucleases and DNA modification methyltransferases a review (Edition 3)." *Gene*, vol. 92, no. 1-2, p. 1, 1990.

[29] S. Richardson, P. Nunley, R. Yarrington, J. Boeke, and J. Bader, "GeneDesign 3.0 is an updated synthetic biology toolkit," *Nucleic Acids Research*, 2010.

[30] R. Edgar and S. Batzoglou, "Multiple sequence alignment," *Current Opinion in Structural Biology*, vol. 16, no. 3, pp. 368–373, 2006.

[31] L. McGuffin, K. Bryson, and D. Jones, "The PSIPRED protein structure prediction server," *Bioinformatics*, vol. 16, no. 4, p. 404, 2000.

[32] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman, "Basic local alignment search tool," *Journal of molecular biology*, vol. 215, no. 3, pp. 403–410, 1990.

[33] R. Mitchell, J. McKim, and B. Meyer, *Design by Contract, by example.* Addison Wesley Longman Publishing Co., Inc. Redwood City, CA, USA, 2001.

[34] J. Jahn, *Introduction to the theory of nonlinear optimization.* Springer Verlag, 2007.

[35] G. Ausiello, *Complexity and approximation: Combinatorial optimization problems and their approximability properties.* Springer Verlag, 1999.

[36] R. Steuer, *Multiple criteria optimization: Theory, computation, and application.* & , 1986.

[37] P. Laarhoven and E. Aarts, *Simulated annealing: theory and applications.* Springer, 1987.

[38] S. Dasgupta, C. Papadimitriou, and U. Vazirani, *Algorithms.* McGraw-Hill, Inc. New York, NY, USA, 2006.

[39] Y. Xiang and B. Chaib-draa, Eds., *Advances in Artificial Intelligence, 16th Conference of the Canadian Society for Computational Studies of Intelligence, AI 2003, Halifax, Canada, June 11-13, 2003, Proceedings*, ser. Lecture Notes in Computer Science, vol. 2671. Springer, 2003.

[40] T. Back, *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms.* Oxford University Press, USA, 1996.

[41] T. Weise, "Global Optimization Algorithms–Theory and Application," *URL: http://www. it-weise. de, Abrufdatum*, vol. 1, 2008.