D-Tailor: automated analysis and design of DNA sequences

Joao Guimarães, Miguel Rocha, Adam P Arkin, Guillaume Cambray

Bioinformatics Advance Access published January 6, 2014

Sequence analysis

D-Tailor: automated analysis and design of DNA sequences

Joao C. Guimaraes[1,2,3], Miguel Rocha[3], Adam P. Arkin[1,2,4,*] and Guillaume Cambray[2,*]

[1]Department of Bioengineering, University of California, Berkeley, CA, 94720, USA.

[2]California Institute for Quantitative Biosciences, University of California, Berkeley, CA, 94720, USA.

[3]Computer Science and Technology Center, School of Engineering, University of Minho, Campus de Gualtar, Braga, Portugal.

[4]Physical Biosciences Division, Lawrence Berkeley National Laboratory, Berkeley, CA, 94720, USA.

Associate Editor: Dr. John Hancock

ABSTRACT

Motivation: Current advances in DNA synthesis, cloning and sequencing technologies afford high throughput implementation of artificial sequences into living cells. However, flexible computational tools for multi-objective sequence design are lacking, limiting the potential of these technologies.

Results: We developed DNA-Tailor (D-Tailor), a fully extendable software framework, for property-based design of synthetic DNA sequences. D-Tailor permits the seamless integration of multiple sequence analysis tools into a generic Monte-Carlo simulation that evolves sequences toward any combination of rationally defined properties. As proof of principle, we show that D-Tailor is capable of designing sequence libraries comprising all possible combinations among three different sequence properties influencing translation efficiency in *E. coli*. The capacity to design artificial sequences that systematically sample any given parameter space should support the implementation of more rigorous experimental designs.

Availability: Source code is available for download at https://sourceforge.net/projects/dtailor/

Contact: aparkin@lbl.gov or cambray.guillaume@gmail.com

Supplementary information: Supplementary data are available at *Bioinformatics* online (D-Tailor Tutorial).

1 INTRODUCTION

The accumulation of genomic data has fueled the development of numerous computational tools that infer functional behavior from biological sequences. These algorithms essentially capture our understanding of how functional information is encoded in nucleic acid and protein sequences. As a result, molecular biologists can now access a plethora of sequence analysis tools to help them predict functional behaviors from plain sequences (Altschul, et al., 1997; Bailey, et al., 2009; Giardine, et al., 2005; Hofacker, 2003; Kingsford, et al., 2007; Markham and Zuker, 2008; Thomas-Chollier, et al., 2011). Common tasks comprise the identification of sequence motifs from nucleic acid (DNA/RNA) or protein sequences (e.g., promoter or termination activity, recombination or splicing sites), as well as the computation of sequence properties that are mechanistically linked to particular phenotypes (e.g., codon usage or propensity to form transmembrane protein domains).

Such sequence analysis tools are usually used to inform biological discovery in natural genomic sequences. However, considering recent advances in DNA technologies and the concomitant rise of synthetic biology applications (Cambray, et al., 2011; Carr and Church, 2009; Czar, et al., 2009; Endy, 2005; Ma, et al., 2012), these same tools may also be leveraged to guide the design of artificial sequences satisfying predefined functions of interest.

Ideally, elementary biological functions should be contained within well-defined sequence parts that could be re-used with acceptable reliability in different contexts (e.g., (Davis, et al., 2011; Mutalik, et al., 2013)). However, it is becoming increasingly clear that many molecular behaviors result from the combined influence of several sequence determinants that cannot be neatly encapsulated within the physical boundaries of a single part, but rather emerge at the interface between the different parts (Cambray, et al., 2013; Kosuri, et al., 2013; Mutalik, et al., 2012; Salis, et al., 2009). In this context, the multidimensional examination of DNA sequences becomes necessary to better capture the inherent complexity of biological behavior and further enable predictive design of synthetic sequence functions and activities (e.g. (Allert, et al., 2010; Dvir, et al., 2013; Kinney, et al., 2010; Na, et al., 2013; Rhodius and Mutalik, 2010; Rodrigo, et al., 2012; Salis, et al., 2009; Seelig, et al., 2006; Welch, et al., 2009)).

Valuable sequence design tools implementing heuristic searches have been successfully developed for multi-objective optimization within specific applications (e.g., protein synthesis optimization (Chung and Lee, 2012; Dana and Tuller, 2012; Gaspar, et al., 2013; Gaspar, et al., 2012; Raab, et al., 2010; Racle, et al., 2012; Salis, et al., 2009; Welch, et al., 2011)). However, application of such optimization procedures requires an objective function relating computed sequence properties to an expected performance score. Unfortunately, the data and models required to describe these relationships are generally not sufficient to support truly reliable functional design.

*To whom correspondence should be addressed.

Interestingly, sequence design tools can also be used upstream of the optimization process to produce libraries of sequences that are more suited for the development of predictive models. Although large-scale studies have mostly used random approaches to introduce variability in the synthetic sequences to be interrogated (Dvir, et al., 2013; Quan, et al., 2011), similar endeavours have greatly benefited from following well-established design of experiments (DoE) (Allert, et al., 2010; Antony, 2003; Kosuri, et al., 2013; Sharon, et al., 2012; Smith, et al., 2013).

DoE is a general framework that fully integrates planning and analysis phases, and comprises three major steps. The first one consists in identifying the factors of interest and defining the range of values for each factor. In the case of molecular sequences, factors are properties of the primary sequence itself and can be typically identified by reanalysing available functional genomic data and published mechanistic studies. The second step consists in implementing a particular experimental design wherein multiple combinations of factor levels are selected to create an experimental dataset providing maximal information to relate the design factors to the response variable(s). For example, one of the most informative DoE is the full-factorial design, where all possible combinations of factor levels across the different factors are performed. The resulting dataset not only permits to estimate the contribution of each factor to the measured response variable, but also robustly captures the interactions between the different factors (Antony, 2003; Mutalik, et al., 2013). Lastly, the third step includes the collection of experimental data and definition of a model relating the multiple factors to the response variable(s). Of note, this can be an iterative process wherein models derived from the third phase can inform the design of a new set of experiments.

Although implementation of experimental designs systematically varying easily manipulated factors can be straightforward (e.g., growth medium, pH, temperature or oxygen levels), the ability to design artificial sequences whose intrinsic properties can be systematically varied is not necessarily trivial (e.g., binding site affinity or the strength of an RNA secondary structure).

Here, we present D-Tailor, an extendable framework supporting integration of multiple sequence analysis tools to mine and design biological sequences. D-Tailor uses a heuristic search algorithm to enable flexible design of synthetic sequences varying multiple properties of interest so as to satisfy complex DoEs. We have validated our tool by successfully designing artificial sequence libraries conforming to full-factorial designs, which represent the upper bound of experimental design complexity. More specifically, we have designed libraries systematically varying multiple sequence properties known to impact translation efficiency in *E. coli*. To further demonstrate the versatility of the algorithm, we also used D-Tailor to design artificial bacterial promoter sequences varying multiple *cis*-regulatory properties (see Supplementary Information).

2 METHODS

D-Tailor essentially implements the two-step planning process outlined above (Fig. 1). The analysis mode computes property scores from plain biological sequences. Here, the user specifies input sequences and a predefined set of properties to be computed. The design mode integrates the analysis routines with a parameterizable Monte-Carlo algorithm that evolves an input sequence (seed) so as to match the specified combinations of property scores. In a typical workflow, users can use the analysis mode to identify sequence properties and operational ranges that seem worth exploring in design mode.

2.1 Sequence analyzer

D-Tailor provides a generic interface for multidimensional interrogation of DNA sequences. The software is designed with a modular architecture, so that users with basic programing skills can easily implement or extend

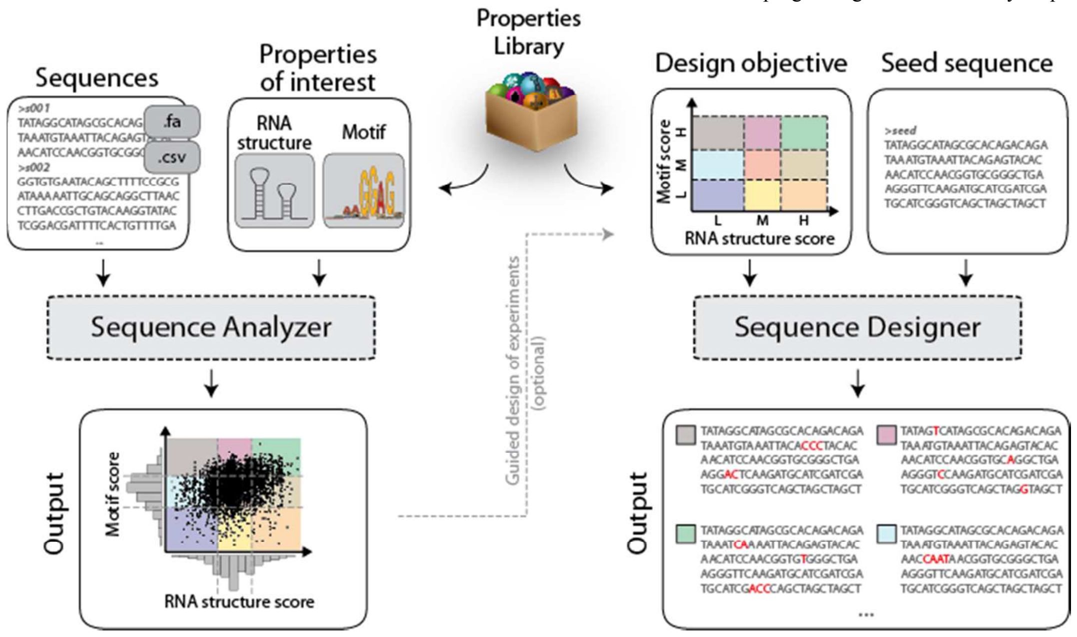


Fig. 1 D-Tailor enables multidimensional analysis and design of DNA sequences. D-Tailor provides a flexible and extendable architecture to interrogate different sequence properties (box in the middle). The left panel depicts an example of the retrieval process of two properties (RNA structure and motif prediction) from multiple input sequences that can come from either FASTA or CSV files. The resulting score profile can be used to identify general trends and further define ideal parameter ranges for the design objectives. The right panel shows the design mode of D-Tailor, wherein a seed sequence is evolved to meet a user-defined combination of sequence properties. The figure depicts a full-factorial design for two different properties of interest (RNA structure and motif scores) with three levels each (low, medium and high), which yields a total of nine different combinations (colored areas).

modules for handling any sequence property of interest. Such modules can be implemented using custom Python code or scripts connecting to third party software (see the Tutorial available in the Supplementary Information). In analysis mode, D-Tailor reads a set of sequences in either delimiter separated (e.g., CSV) or FASTA format files. A property profile is then computed for each of the input sequences by successively calling the analysis modules specified by the user. (Fig. 1, left panel).

D-Tailor currently comprises 14 different modules to compute various sequence properties involved in diverse mechanisms of gene regulation. This collection of sequence property evaluators includes algorithms to score promoter regions or transcription factor binding sites based on sequence logos (Thomas-Chollier, et al., 2011), estimate translation initiation rates based on the Shine-Dalgarno (SD) sequence (Shine and Dalgarno, 1975), predict propensity to form RNA structures, calculate nucleotide composition or compute the codon adaptation index (CAI) for a given gene sequence (Sharp and Li, 1987). Although the implementation of the different sequence property evaluators is usually self-contained within D-Tailor, the computation of specific properties may rely on third party softwares (e.g., UNAfold (Markham and Zuker, 2008) for the prediction of RNA secondary structure,). Together, these modules illustrate diverse implementation modalities and provide useful examples to guide future extensions (see Supplementary Information). The specification of adequate analysis routines is an essential prerequisite to running the design mode.

2.2 Sequence designer

As capacities for DNA synthesis increase exponentially, the ability to computationally design artificial sequences need to become more automated and transparent. The most innovative feature of D-Tailor is to provide a generic solution of designing synthetic sequences constrained by multiple properties of interest (Fig. 1, right panel).

The design process in D-Tailor starts with the specification of a seed sequence and the desired design objective (i.e. the DoE) (Fig. 1, right panel). Seed sequences serve as a template to bootstrap the evolutionary design process. Typically, users would use a particular sequence of interest from which they want to derive a mutational series. The DoE enumerates combinations of sequence properties that need to be generated, each of which constitutes a design target. D-Tailor provides a flexible scheme for the definition of DoE, which can vary from full-factorial to entirely customized designs.

The definition a finite number of targets requires the discretization of continuous property scores into a finite number of nominal or ordinal levels. For example, Fig. 1 shows the discretization of two sequence property scores into three ordinal levels (low, medium and high). This framework markedly differs from usual multi-objective optimization approaches (Chung and Lee, 2012; Raab, et al., 2010; Racle, et al., 2012), which operate to optimize a single continuous and integrated performance score rather than explicitly target different regions of the parameter space. As illustrated in the Results section, natural feature profiles extracted from available genomic sequences can be used to guide the discretization processes and ensure biological relevance of the sampled space. For each sequence property, users may define as many levels as necessary to attain the desired degree of resolution in the designed sequences. However, since the number of possible combinations increases geometrically with the number of properties/levels, their definition must be mindful of downstream experimental capacities.

Finding a sequence that conforms to an arbitrary combination of property levels is often computationally infeasible using a brute force approach. Indeed, the sequence space to be searched is gigantic (4^N where N is the number of nucleotides in the sequence to be designed, more if indels are allowed). To optimize the search process, D-Tailor uses a Monte-Carlo algorithm to evolve a given seed sequence towards the set of design targets (Fig. 2).

More specifically, the algorithm loops through cycles of evolution until all target combinations of property levels specified by the DoE are found. Each cycle consists in three consecutive steps: i) a target combination of property levels is randomly selected; ii) a template sequence is chosen from the repository of previously generated sequences using fitness proportionate selection (only seed sequences are available at the very first iteration); and iii) a predefined number of mutational iterations are performed until a sequence satisfying the target combination of the property level is found (Fig. 2, sequence evolver). We use the inverse of the cumulative Euclidean distance (D) between property levels, as a generic fitness measure of a sequence relative to a given design target (Eq. 1):

$$D = \sqrt{\sum (d_i - t_i)^2} \qquad \text{(Eq. 1)}$$

where n represents the number of sequence properties; d_i and t_i represent the levels of the ith sequence property in the designed sequence and the desired combination, respectively.

Each iteration of the sequence evolver also comprises three steps: i) the sequence being evolved is analyzed and a property requiring optimization (i.e., not within the target level) is randomly selected; ii) the template sequence is then mutated following user-specified mutational rules (see below); and, iii) the feature scores of the resulting sequence are analyzed and evaluated with respect to the current design target (Eq. 1). Every generated sequence is also screened for compliance to a user-defined set of rules meant to prevent the emergence of undesired properties in the final designed sequences (e.g., restriction sites, unexpected promoters or terminators). Only validated sequences are stored in the database.

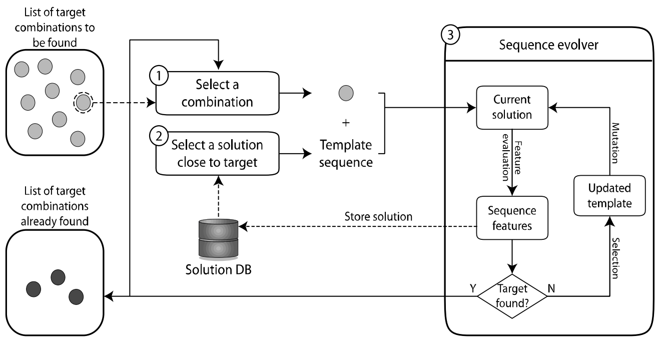


Fig. 2 Sequence designer algorithm comprised by three different steps described in the main text. Initially, a target combination of features is selected and then a sequence that is close (i.e., short Euclidean distance) to the desired target is chosen to serve as the template in the sequence evolution step. This last step applies successive mutations until it finds a sequence matching the target combination of features

Next, if the new sequence matches the target combination (D=0), then the target is marked as completed and the evolution cycle is terminated. Otherwise, the algorithm updates the template for the next mutational iteration, choosing between retaining the current template sequence or accepting the mutant just derived. At this point, we defined three different selective regimes: i) directional selection, where the sequence with the lower Euclidean distance to the target combination is chosen; ii) neutral selection, where any of the two sequences is selected with predefined probabilities; or iii) temperature selection, as inspired by simulated annealing optimization (Kirkpatrick, et al., 1983), where the sequence is selected based on a temperature schedule that allows worse sequences (longer distances) to also be selected with a probability that decreases with the number of iterations performed.

At each of the mutational iterations, new sequences can be generated through random mutation of the template sequence, as usual in many sequence optimization tools (Chung and Lee, 2012; Gaspar, et al., 2012;

Salis, et al., 2009). In addition, D-Tailor offers the possibility to implement specialized mutation operators that aim at improving the likelihood to generate desired property changes. Practically, a mutation operator randomly selects a property amongst those that are non-optimal in the current template ($d_i - t_i \neq 0$). We then distinguish between: i) targeted operators, which restrict the mutational space to specific regions of the sequence that are therefore more likely to affect the property that needs to be evolved; and ii) oriented operators, which further specify particular mutation patterns to bias the production of variants toward the current design target. For example, if the design goal specifies an increase in the CAI of a gene, the targeted mutation operator restricts the mutable region to the coding sequence and randomly replaces a codon by another one irrespective of its usage score. The oriented mutation operator further constrains the replacement of a randomly chosen codon with one associated with a higher usage score, thereby enforcing the required increase. For certain emergent features, the definition of oriented mutation might not be so straightforward. For example, we implemented oriented mutation operators for RNA secondary structure by specifically targeting mutations to bases that are predicted to be paired or unpaired, to respectively decrease or increase the strength of the mutated RNA structure. Importantly, any mutation operator targeting gene-coding sequences can be further constrained to only generate synonymous mutations, thereby preserving the encoded protein sequence while modifying the underlying DNA properties.

In some applications, it may be desirable to limit the overall divergence between sequences in the designed library, so that it provides small variations with respect to a particular reference sequence. Conversely, users might want to generate sequences that are as dissimilar as possible and, therefore, share as few confounding factors as possible. In D-Tailor, users can manipulate mutational patterns and the selective regime—two major parameters of the evolutionary design process—to indirectly control sequence diversity, and consequently impact the rate of sequence evolution, as well as the overall performance of the search algorithm (see below).

3 RESULTS AND VALIDATION

D-Tailor provides an integrated Python-scripting framework for multidimensional analysis of sequence properties and for the design of artificial sequences constrained by multiple sequence properties of interest.

As a case study, we have chosen three different previously reported sequence determinants of translation efficiency. In E. coli, two major factors have been shown to modulate the rate of translation initiation: i) the strength and position of a Shine-Dalgarno (SD) motif upstream of the start codon (Barrick, et al., 1994; Shine and Dalgarno, 1975); and ii) the propensity of these sequence signals to engage in mRNA secondary structures (de Smit and van Duin, 1994; Hall, et al., 1982; Kudla, et al., 2009). Subsequent to initiation, the rate of elongation may also affect the overall translation efficiency and is mainly determined by the codon usage of the gene (Gustafsson, et al., 2004; Ikemura, 1985; Kane, 1995; Sharp and Li, 1987; Welch, et al., 2009; Welch, et al., 2011). We first illustrate how D-Tailor analysis module can be used to examine such sequence properties in the natural genome of E. coli. Then, we demonstrate how to use D-Tailor design module to generate artificial sequence libraries systematically varying the three properties of interest according to a full-factorial design of experiments.

3.1 Using D-Tailor to interrogate sequences

We used D-Tailor to re-analyze three different sequence properties across the entire *E. coli* W3110 genome (Fig. 3). Mechanistically, the SD motif stabilizes the initial binding of the 30S subunit of the ribosome by establishing canonical base pairing with the 3' end of the 16S rRNA (anti-SD) (Shine and Dalgarno, 1975). We applied a sequence property evaluator that calculates the strength of the SD sequence by searching for a subsequence within the 25 nucleotides upstream of a start codon with highest affinity to the known anti-SD (Lithwick and Margalit, 2003). The presence of secondary structures in this region of the mRNA can hinder initiation by occluding the SD motif or the nearby start codon from recognition by the ribosomal subunits. For that purpose, we used an RNA structure evaluator to compute the minimum free energy of the 60 nucleotides subsequence centered on the start codon (Kudla, et al., 2009). Finally, we used a CAI calculator to score the codon usage of a gene sequence (Sharp and Li, 1987). Practically, the usage of these property evaluators and associated parameters requires a standard interface, which is provided by extending the abstract class Feature in D-Tailor (see Supplementary Information).

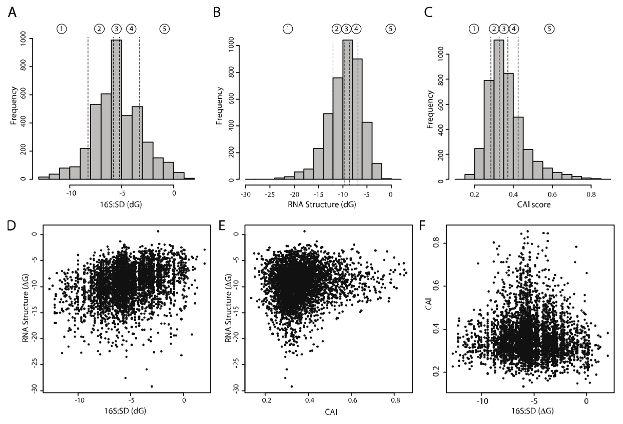


Fig. 3 (A-C) Distribution of the three different sequence properties (hybridization energy between the 16S rRNA and SD sequence (A), minimum folding energy of RNA structure in the translation initiation region (B) and codon adaptation index—CAI—of gene sequences (C)) influencing translation efficiency in *E. coli*. The dashed lines indicate the quintile boundaries for the scores of each property, which were later used in design mode to discretize the parameter space. (D-F) Scatter plots showing the cross-correlation between the three sequence properties of interest.

The sequence property profiles resulting from a genome analysis give a solid basis to identify trends in the properties of interest, and to further determine the relevant parameter space to explore during the design step (Fig. 3A-C). Correlations amongst property scores may also provide insights onto potential functional interactions although some may be purely incidental. For example, the modest correlation between RNA structure in the translation initiation region and the affinity between ribosomes and the SD sequence (Fig. 3D) might merely reflect the thermodynamic propensity of G-rich SD motifs to engage in secondary structures. Similarly, the peculiar shape of the relationship between CAI and RNA secondary structure (Fig. 3E) might stem from the joint contributions of independent evolutionary pressures for expression levels acting on these two properties to tune expression levels (highly expressed genes being both under selection for high CAI and for low structure (Gu, et al., 2010; Kudla, et al., 2009; Plotkin and Kudla, 2011; Tuller, et al., 2010)). It is then up to the user to define a DoE containing combinations of sequence property scores that are more adequate to test the research hypothesis to be investigated.

3.2 Using D-Tailor to implement experimental design on sequence properties

Although recent advances in DNA synthesis, cloning and sequencing make it possible to generate and experimentally probe thousands of custom DNA/RNA sequences (Dvir, et al., 2013; Kosuri, et al., 2013; Patwardhan, et al., 2012; Patwardhan, et al., 2009; Quan, et al., 2011; Sharon, et al., 2012; Smith, et al., 2013), the availability of computational tools to aid the rational design of large sequence libraries remains very limited.

The main purpose of D-Tailor is to provide a flexible computational tool to design custom sequences satisfying complex specifications. Such task can be extremely laborious when the properties of interest physically overlap in the sequence space. For instance, in our case study, the subsequence containing the SD motif influences the formation of RNA secondary structures in that same region. Likewise, the secondary structure can be affected when modifying codon usage at the beginning of the gene. Typically, such optimization problems are best solved using a trial-and-error approach wherein sequence variants are generated using random mutations until a desired combination of property scores is found (Allert, et al., 2010; Gaspar, et al., 2013; Raab, et al., 2010; Racle, et al., 2012; Salis, et al., 2009). To generalize this process, the design mode of D-Tailor provides a framework to integrate any sequence property evaluator into a parameterizable Monte-Carlo algorithm that iteratively evolves sequences toward a specific set of design targets (or combinations of property levels).

We used D-Tailor to design sequences that systematically vary the three properties of interest (or factors) defined above (Fig. 3). For each of these factors, we defined five contiguous ordinal levels on the basis of the quintiles observed in the natural genome (Fig. 3A-C, dashed lines). We then instructed D-Tailor to search for sequences conforming to a full-factorial DoE based on these levels. This DoE describes a total of 125 design targets corresponding to all combinations of five levels across the three different properties (5^3). To validate our approach, we compared the performance of four increasingly complex evolutionary strategies available in D-Tailor at deriving full-factorial libraries for 30 different genes randomly selected in *E. coli* (Fig. 4A-B). In these simulations, the algorithm was run for at most 3,000 generations—with a single mutational event per generation—allowing for unrestricted mutations in the 5’ UTR but only for synonymous mutations in the coding sequence.

We first explored the most rudimentary evolutionary strategy available in D-Tailor, random sampling, which does not implement any heuristic search and simply generates random sequences until all desired design objectives are completed. Every attempt to complete the full-factorial design before the threshold of 3,000 generated sequences failed (Fig. 4A-B, black line, 54.2 generated sequences per target found [gspt] on average). The second design strategy used D-Tailor’s generic heuristic algorithm (Fig. 2 and Methods) along with the simplest mutational method wherein new sequence variants are generated by random mutagenesis (Fig. 4A, yellow line). This strategy improved the efficiency of the search algorithm by a factor of 2 as compared to that of the random sampling method (24.8 vs 54.2 gspt on average, Mann-Whitney test p-value = 2.3×10^{-10}, Fig. 4B). Still, many sequences had to be generated to meet the various design objectives. The third mutational strategy employed spatially targeted mutation operators (see Methods) and improved the search algorithm efficiency by another factor of 2 (13.3 gspt on average, Fig. 4B). The fourth strategy used more ‘rational’ mutation operators that explicitly orient mutations toward the desired objective (see Methods) and provided slightly faster dynamics (Fig. 4A, orange line, 11.8 vs 13.3 gspt on average, Mann-Whitney test p-value = 0.129, Fig. 4B). Since the computational time necessary to achieve a given set of design targets is dependent on the number of generated sequences per target, these results illustrate the advantage of defining specific mutation operators whenever it is possible.

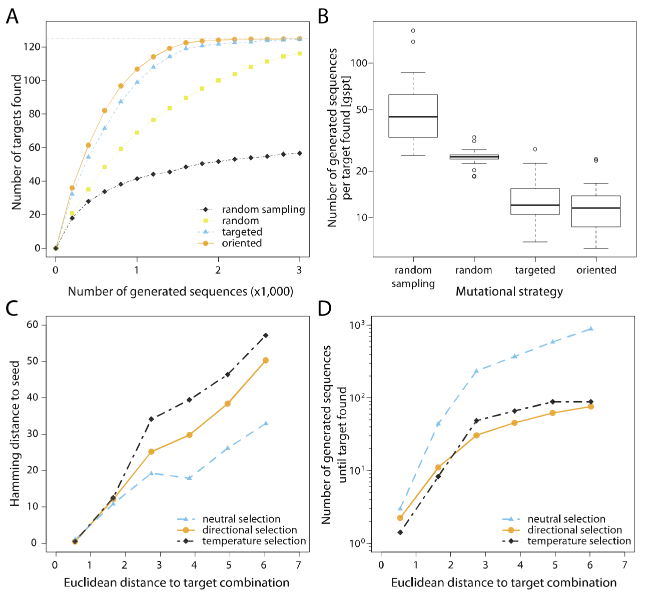


Fig. 4 D-Tailor design simulations. (A) We performed simulations of full-factorial design using 30 different initial sequences (seeds) and four different design strategies: random sampling (black) and heuristic search using random (yellow), targeted (blue) and oriented (orange) mutations. The different lines represent the average number (across 30 simulations) of target combinations found (out of 125) as a function of the number of generated sequences (up to 3,000) for the four different strategies. We observed sizeable variation between seeds (not shown for clarity, see Supplementary Information for details) (B) Number of generated sequences per target found (gspt) for the four different mutational strategies (n=30). (C) We used the same 30 different seeds to find six different target combinations at various Euclidean distances. The different lines show the average hamming distance between the seed and the sequence matching the target combination as a function of the Euclidean distance to the target combination using neutral (light blue), directional (orange) or temperature selection (black). (D) The number of generated sequences until the desired target is found as a function of the Euclidean distance to the target combination using either neutral (light blue), directional (orange) or temperature (black) selection.

When designing synthetic sequences, users may want to limit the divergence of the designed sequences with respect to the initial seed. To roughly control the spread of the generated sequences during the evolutionary process, users can manipulate the strength of selection toward the desired target(s). To better illustrate this point, we evolved each of the 30 seeds previously selected toward six different target combinations bearing different Euclidean distances from the seeds (Fig. 4C-D). We then examined the behavior and results of the algorithm in response to three contrasted selective regimes: neutral, directional and temperature selection (Methods).

As expected, we observed that a more relaxed selection process (neutral) is able to generate sequences matching the desired target that are more similar to the seed sequence than those resulting from the directional or temperature selection approach (average hamming distance of 21 vs 31.3 and 39.2, respectively; Mann-Whitney test p-value = 0.0005 and 1.03e-13; Fig. 4C). Nonetheless, the limitation of sequence diversity comes at the cost of longer computation time (Fig. 4D). In fact, for the 30 seed sequences, the neutral selection process requires the generation of 8 and 6 times more sequences per target than the directional and temperature selection approach, respectively and on average. For large designs, users may have to balance the desired divergence of the designed sequences with the available computational power. A hybrid approach, wherein the algorithm is initially set with weak selection and hard constraints to limit divergence, and then progressively configured with increased selection bias and/or relaxed mutational constraints (e.g., allow non-synonymous mutations in coding sequences if it is acceptable by the user) as the rate of target discovery slows down may then be recommended. The details of such procedure are likely specific for each application, and therefore we have not sought to implement an automatic schedule to control this behavior. Since the state of a D-Tailor design mode run is permanently stored in a database, we suggest users to manually experiment with adjusting these parameters.

4 CONCLUSION

Advances in DNA reading/writing technologies readily enable the construction and validation of complex genetic systems (Gibson, et al., 2010). However, rules to successfully design synthetic sequences to functional specifications have been limited by measurements from biased natural samples and/or small scale controlled synthetic samples comprising at most hundreds of sequences (e.g., (Allert, et al., 2010; Amit, et al., 2011; Barrick, et al., 1994; Garcia, et al., 2012; Mutalik, et al., 2012; Na, et al., 2013; Rhodius and Mutalik, 2010; Rodrigo, et al., 2012; Salis, et al., 2009)). This lack of knowledge strongly restrains the practical applications of *ab initio* design. Innovative experimental methodologies based on high-throughput technologies are scaling the characterization process up to tens of thousands of designed sequence variants, providing larger datasets to better understand sequence/activity relationships (Dvir, et al., 2013; Kinney, et al., 2010; Patwardhan, et al., 2012; Patwardhan, et al., 2009; Sharon, et al., 2012; Smith, et al., 2013). However dramatic, this increase in throughput remains limited in comparison to the sheer immensity of the sequence space. It is therefore crucial to reduce the dimensionality of the design space to a set of sequence properties of interest that can be independently varied to facilitate estimation of their individual contribution to the measured phenotype and further support predictable design of synthetic variants (Allert, et al., 2010; Sharon, et al., 2012; Smith, et al., 2013).

We developed D-Tailor as an extendable and flexible software platform for the multi-objective design of artificial sequences. It provides a generic interface to integrate multiple sequence analysis tools into a heuristic Monte-Carlo search procedure capable of evolving sequences towards pre-defined design targets (Fig. 1). D-Tailor presents significant differences to other multi-objective sequence optimization tools (Allert, et al., 2010; Chung and Lee, 2012; Dana and Tuller, 2012; Gaspar, et al., 2012; Raab, et al., 2010; Racle, et al., 2012; Salis, et al., 2009). First, it allows the definition of multiple design targets as combinations of sequence properties that embody particular design of experiments. A DoE can range anywhere from one specific combination of property levels to a full-factorial design, where the parameter space is fully explored. In contrast, traditional optimization tools describe design objectives in terms of desired response performances, which are linked to the sequence properties by a complex and pre-defined static objective function. Such formalization is suited for functional optimization, but do not explicitly support systematic exploration of the parameter space. Second, D-Tailor provides an evolutionary algorithm to optimize both coding and non-coding regions. Third, D-Tailor supports the implementation of advanced mutational strategies that can significantly enhance the heuristic search performance (Fig. 4B). Finally, our tool is not application-specific and provides an open source solution based on an extendable architecture, such that new sequence property evaluators can be easily implemented and integrated into the sequence design engine.

We demonstrate that D-Tailor can efficiently design artificial sequences to systematically vary any given set of properties of interest. To this end, we successfully derived full-factorial sequence libraries, starting from 30 different seed sequences, exploring the entire parameter space of three intertwined sequence properties affecting translation efficiency. Interestingly, we observed that the dynamics of target discovery varies slightly depending on the input seed (see Supplementary Information for details). This illustrates that different sequences may have distinct evolutionary landscapes; some being more amenable to generate widely variable profiles of property scores, with fewer mutational cycles than others (Cambray and Mazel, 2008; Wagner, 2008). For both targeted and oriented mutational methods, the average dynamics of target discovery revealed a relatively steady rate for the first ~80% of targets, followed by a sharp decrease in efficiency—presumably because the remaining targets specify combinations of property levels that are harder to attain (Fig. 4A, orange and light blue lines). We also confirmed that more simplistic design approaches—such as generation of random sequences—perform poorly in comparison to a heuristic search (Fig. 4A-B).

In addition to the case study detailed here, we have used D-Tailor to systematically design synthetic bacterial promoter sequences varying multiple *cis*-regulatory properties (see Tutorial for details), that way demonstrating the generality and flexibility of our methods and tool.

D-Tailor permits the implementation of advanced experimental designs into artificial sequence samples that can serve as a basis to rigorously and consistently test sets of molecular hypothesis. We believe that comprehensive full-factorial libraries of sequences are needed to investigate complex biochemical activities and robustly dissect the contribution of individual factors as well as their interactions. Such libraries will aid characterizing complex multifactorial phenotypes and eventually derive quantitative relationships between sequence and activity.

Funding: This work was supported by Synthetic Biology Engineering Research Center under National Science Foundation [04-570/0540879]; Fundação para a Ciência e a Tecnologia (FCT) [SFRH/BD/47819/2008] to JCG; Human Frontiers Science Program [LT000873/2011-L] and the Bettencourt Schueller foundation to GC; European Regional Development Fund (ERDF) through the COMPETE Programme and by the FCT within projects COMPETE [FCOMP-01-0124-FEDER-015079, PEst-OE/EEI/UI0752/2011] to MR.

Conflict of interest: none declared.

REFERENCES

Allert, M., Cox, J.C. and Hellinga, H.W. (2010) Multifactorial determinants of protein expression in prokaryotic open reading frames, *J Mol Biol*, **402**, 905-918.

Altschul, S.F., *et al.* (1997) Gapped BLAST and PSI-BLAST: a new generation of protein database search programs, *Nucleic Acids Res*, **25**, 3389-3402.

Amit, R., *et al.* (2011) Building enhancers from the ground up: a synthetic biology approach, *Cell*, **146**, 105-118.

Antony, J. (2003) *Design of experiments for engineers and scientists*. Butterworth-Heinemann, Oxford ; Burlinton, MA.

Bailey, T.L., *et al.* (2009) MEME SUITE: tools for motif discovery and searching, *Nucleic Acids Res*, **37**, W202-208.

Barrick, D., *et al.* (1994) Quantitative analysis of ribosome binding sites in E.coli, *Nucleic Acids Res*, **22**, 1287-1295.

Cambray, G., *et al.* (2013) Measurement and modeling of intrinsic transcription terminators, *Nucleic Acids Res*, **41**, 5139-5148.

Cambray, G. and Mazel, D. (2008) Synonymous genes explore different evolutionary landscapes, *PLoS Genet*, **4**, e1000256.

Cambray, G., Mutalik, V.K. and Arkin, A.P. (2011) Toward rational design of bacterial genomes, *Curr Opin Microbiol*, **14**, 624-630.

Carr, P.A. and Church, G.M. (2009) Genome engineering, *Nat Biotechnol*, **27**, 1151-1162.

Chung, B.K. and Lee, D.Y. (2012) Computational codon optimization of synthetic gene for protein expression, *BMC Syst Biol*, **6**, 134.

Czar, M.J., *et al.* (2009) Gene synthesis demystified, *Trends Biotechnol*, **27**, 63-72.

Dana, A. and Tuller, T. (2012) Efficient manipulations of synonymous mutations for controlling translation rate: an analytical approach, *J Comput Biol*, **19**, 200-231.

Davis, J.H., Rubin, A.J. and Sauer, R.T. (2011) Design, construction and characterization of a set of insulated bacterial promoters, *Nucleic Acids Res*, **39**, 1131-1141.

de Smit, M.H. and van Duin, J. (1994) Control of translation by mRNA secondary structure in Escherichia coli. A quantitative analysis of literature data, *J Mol Biol*, **244**, 144-150.

Dvir, S., *et al.* (2013) Deciphering the rules by which 5'-UTR sequences affect protein expression in yeast, *Proc Natl Acad Sci U S A*, **110**, E2792-2801.

Endy, D. (2005) Foundations for engineering biology, *Nature*, **438**, 449-453.

Garcia, H.G., *et al.* (2012) Operator sequence alters gene expression independently of transcription factor occupancy in bacteria, *Cell Rep*, **2**, 150-161.

Gaspar, P., *et al.* (2013) mRNA secondary structure optimization using a correlated stem-loop prediction, *Nucleic Acids Res*.

Gaspar, P., *et al.* (2012) EuGene: maximizing synthetic gene design for heterologous expression, *Bioinformatics*, **28**, 2683-2684.

Giardine, B., *et al.* (2005) Galaxy: a platform for interactive large-scale genome analysis, *Genome Res*, **15**, 1451-1455.

Gibson, D.G., *et al.* (2010) Creation of a bacterial cell controlled by a chemically synthesized genome, *Science*, **329**, 52-56.

Gu, W., Zhou, T. and Wilke, C.O. (2010) A universal trend of reduced mRNA stability near the translation-initiation site in prokaryotes and eukaryotes, *PLoS Comput Biol*, **6**, e1000664.

Gustafsson, C., Govindarajan, S. and Minshull, J. (2004) Codon bias and heterologous protein expression, *Trends Biotechnol*, **22**, 346-353.

Hall, M.N., *et al.* (1982) A role for mRNA secondary structure in the control of translation initiation, *Nature*, **295**, 616-618.

Hofacker, I.L. (2003) Vienna RNA secondary structure server, *Nucleic Acids Res*, **31**, 3429-3431.

Ikemura, T. (1985) Codon usage and tRNA content in unicellular and multicellular organisms, *Mol Biol Evol*, **2**, 13-34.

Kane, J.F. (1995) Effects of rare codon clusters on high-level expression of heterologous proteins in Escherichia coli, *Curr Opin Biotechnol*, **6**, 494-500.

Kingsford, C.L., Ayanbule, K. and Salzberg, S.L. (2007) Rapid, accurate, computational discovery of Rho-independent transcription terminators illuminates their relationship to DNA uptake, *Genome Biol*, **8**, R22.

Kinney, J.B., *et al.* (2010) Using deep sequencing to characterize the biophysical mechanism of a transcriptional regulatory sequence, *Proc Natl Acad Sci U S A*, **107**, 9158-9163.

Kirkpatrick, S., Gelatt, C.D., Jr. and Vecchi, M.P. (1983) Optimization by simulated annealing, *Science*, **220**, 671-680.

Kosuri, S., *et al.* (2013) Composability of regulatory sequences controlling transcription and translation in Escherichia coli, *Proc Natl Acad Sci U S A*, **110**, 14024-14029.

Kudla, G., *et al.* (2009) Coding-sequence determinants of gene expression in Escherichia coli, *Science*, **324**, 255-258.

Lithwick, G. and Margalit, H. (2003) Hierarchy of sequence-dependent features associated with prokaryotic translation, *Genome Res*, **13**, 2665-2673.

Ma, S., Tang, N. and Tian, J. (2012) DNA synthesis, assembly and applications in synthetic biology, *Curr Opin Chem Biol*, **16**, 260-267.

Markham, N.R. and Zuker, M. (2008) UNAFold: software for nucleic acid folding and hybridization, *Methods Mol Biol*, **453**, 3-31.

Mutalik, V.K., *et al.* (2013) Quantitative estimation of activity and quality for collections of functional genetic elements, *Nature Methods*, **10**, 347-353.

Mutalik, V.K., *et al.* (2012) Rationally designed families of orthogonal RNA regulators of translation, *Nat Chem Biol*, **8**, 447-454.

Na, D., *et al.* (2013) Metabolic engineering of Escherichia coli using synthetic small regulatory RNAs, *Nat Biotechnol*, **31**, 170-174.

Patwardhan, R.P., *et al.* (2012) Massively parallel functional dissection of mammalian enhancers in vivo, *Nat Biotechnol*, **30**, 265-270.

Patwardhan, R.P., *et al.* (2009) High-resolution analysis of DNA regulatory elements by synthetic saturation mutagenesis, *Nat Biotechnol*, **27**, 1173-1175.

Plotkin, J.B. and Kudla, G. (2011) Synonymous but not the same: the causes and consequences of codon bias, *Nat Rev Genet*, **12**, 32-42.

Quan, J., *et al.* (2011) Parallel on-chip gene synthesis and application to optimization of protein expression, *Nat Biotechnol*, **29**, 449-452.

Raab, D., *et al.* (2010) The GeneOptimizer Algorithm: using a sliding window approach to cope with the vast sequence space in multiparameter DNA sequence optimization, *Syst Synth Biol*, **4**, 215-225.

Racle, J., Overney, J. and Hatzimanikatis, V. (2012) A computational framework for the design of optimal protein synthesis, *Biotechnol Bioeng*, **109**, 2127-2133.

Rhodius, V.A. and Mutalik, V.K. (2010) Predicting strength and function for promoters of the Escherichia coli alternative sigma factor, sigmaE, *Proc Natl Acad Sci U S A*, **107**, 2854-2859.

Rodrigo, G., Landrain, T.E. and Jaramillo, A. (2012) De novo automated design of small RNA circuits for engineering synthetic riboregulation in living cells, *Proc Natl Acad Sci U S A*, **109**, 15271-15276.

Salis, H.M., Mirsky, E.A. and Voigt, C.A. (2009) Automated design of synthetic ribosome binding sites to control protein expression, *Nat Biotechnol*, **27**, 946-950.

Seelig, G., *et al.* (2006) Enzyme-free nucleic acid logic circuits, *Science*, **314**, 1585-1588.

Sharon, E., *et al.* (2012) Inferring gene regulatory logic from high-throughput measurements of thousands of systematically designed promoters, *Nat Biotechnol*, **30**, 521-530.

Sharp, P.M. and Li, W.H. (1987) The codon Adaptation Index--a measure of directional synonymous codon usage bias, and its potential applications, *Nucleic Acids Res*, **15**, 1281-1295.

Shine, J. and Dalgarno, L. (1975) Determinant of cistron specificity in bacterial ribosomes, *Nature*, **254**, 34-38.

Smith, R.P., *et al.* (2013) Massively parallel decoding of mammalian regulatory sequences supports a flexible organizational model, *Nat Genet*.

Thomas-Chollier, M., *et al.* (2011) RSAT 2011: regulatory sequence analysis tools, *Nucleic Acids Res*, **39**, W86-91.

Tuller, T., *et al.* (2010) Translation efficiency is determined by both codon bias and folding energy, *Proc Natl Acad Sci U S A*, **107**, 3645-3650.

Wagner, A. (2008) Robustness and evolvability: a paradox resolved, *Proc Biol Sci*, **275**, 91-100.

Welch, M., *et al.* (2009) Design parameters to control synthetic gene expression in Escherichia coli, *PLoS One*, **4**, e7002.

Welch, M., *et al.* (2011) Designing genes for successful protein expression, *Methods Enzymol*, **498**, 43-66.

D-Tailor tutorial

Joao C Guimaraes, Miguel Rocha, Adam P Arkin and Guillaume Cambray

10/06/13

Index

1. Installing D-Tailor

1.1. Prerequisites

D-Tailor is implemented in Python. Python is an interpreted and interactive object-oriented programming language that is available for several platforms including Unix, Mac OSX and Microsoft Windows. Before starting to use D-Tailor, you need to install Python version 2. More information can be found at http://www.python.org.

D-Tailor uses a few command line utilities such as *cat or awk* that are commonly available for Unix or Unix-derived operating systems. When using Microsoft Windows it is necessary to run D-Tailor in a Unix-emulation environment such as Cygwin (http://www.cygwin.com/).

To have access to certain functionalities in D-Tailor, you will need to install third-party software to predict RNA structure (UNAFold v3.6 and RNAplfold v1.6) and transcription terminators (TransTermHP v2.08). The sources for these tools are located in the folder "3rdParty" and, after installation, the compiled binaries must be copied to each corresponding folder. For installation instructions of third party software, please refer to their respective websites:

- UNAFold—http://dinamelt.rit.albany.edu/download.php
- RNAplfold (Vienna RNA package)—http://www.tbi.univie.ac.at/~ivo/RNA/
- TransTermHP—http://transterm.cbcb.umd.edu/

All these tools are optional and hence only necessary if the user wants to use above-mentioned functionalities, namely predict RNA structure or transcription terminators.

1.2. Installation

D-Tailor is a Python project ready to be used. To start using D-Tailor, simply download it from https://sourceforge.net/projects/dtailor/ and copy the files to the destination folder.

1.3. License

D-Tailor is licensed under the BSD 2-Clause License.

2. D-Tailor Essentials

2.1. Scope and functionalities

D-Tailor is an extendable framework that automates the analysis and design of DNA sequences properties. To this end, it implements two distinct modules: Sequence Analyzer and Sequence Designer (Figure 2.1). In the Analyzer module, a predefined set of sequence properties of interest is automatically retrieved and evaluated from plain DNA sequences. Conversely, the Designer module evolves a DNA sequence to match specific combinations of sequence properties scores under given mutation constraints (e.g., sequences regions available for mutation or only synonymous mutations). In addition, it is possible to constraint diversity of the designed sequences and/or enforce validation tests to prevent final sequences from comprising undesired elements (e.g. restriction sites, unexpected promoters, terminators or internal ribosome binding sites).

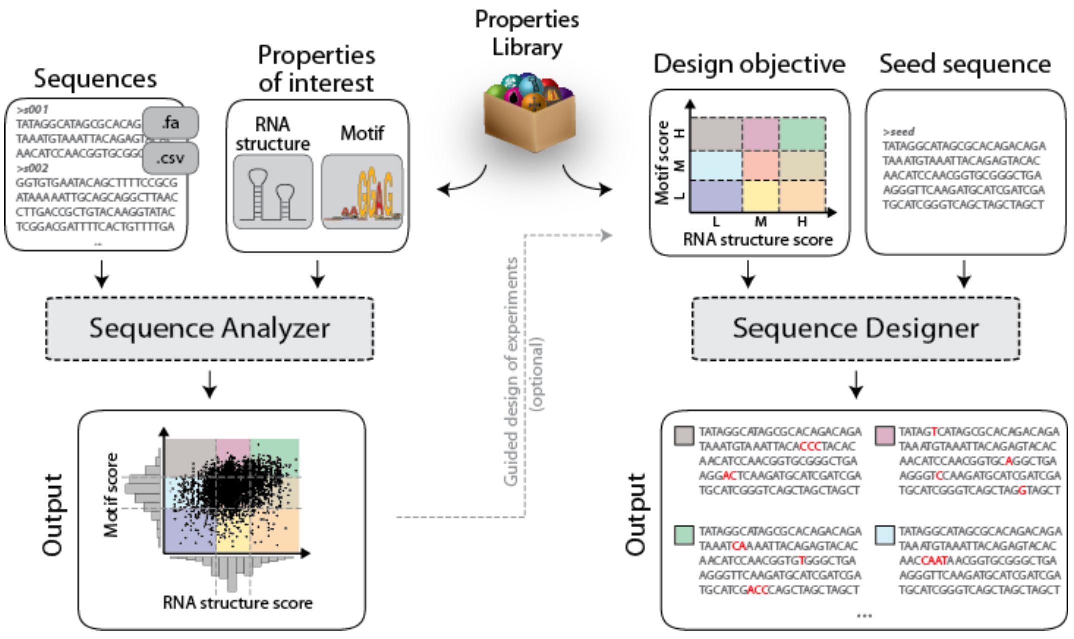


Figure 2.1 D-Tailor framework

D-Tailor provides automated analysis and design of DNA sequences. Because it is based on a modular architecture, it enables the independent development of sequence property evaluators that can be easily plugged-in to the software (Properties library). The left panel depicts an example of the evaluation of two properties from multiple DNA sequences. The right panel outlines the design mode of D-Tailor, where designed sequences are generated based on a design objective constraining two different sequence properties.

In summary, D-Tailor provides an integrated framework for the seamless extraction and evaluation of multiple properties of interest from plain DNA sequences. This analysis pipeline is also integrated in a Monte-Carlo algorithm that evolves input sequences under user-defined constraints toward a set of combinations of sequence properties scores, thereby enabling flexible multi-objective sequence design. D-Tailor is based on an extendable architecture to allow the independent development of new sequence property evaluators that can be easily plugged-in to the software (Figure 2.1).

2.2. Project Structure

D-Tailor uses object-oriented design and its core entities are:

- *Feature*—abstract object encapsulating all relevant attributes and methods to describe a particular sequence feature or property;
- *Solution*—concrete object containing all information for a particular sequence.

A *Solution* can have one or more *Feature* objects. *Solution* is a concrete class that stores a DNA/RNA sequence and corresponding properties of interest. In contrast, *Feature* is an abstract class that is extended by concrete property classes. D-Tailor comes packaged with many ready-to-use concrete properties that extend the abstract class *Feature*. We will use a detailed implementation of two of these properties to exemplify how users can easily implement their own sequence properties of interest (below).

The two main executable classes of are: *SequenceAnalyzer* and *SequenceDesigner*. These are abstract classes and must be extended by concrete classes implementing user-defined analyses and designs (e.g., which properties to compute or what are the mutation constraints). The design mode also requires the definition of a design objective (a class extending the abstract class *Design*). Several design methodologies are already implemented in D-Tailor (see section 6.2).

To provide a flexible storage environment and enable parallel computation, generated sequences are stored in a database. D-Tailor uses the abstract class *DBAbstract* to encapsulate a database management interface. We have extended this class to implement a storage environment based on SQLite (http://www.sqlite.org/). *DBSQLite* uses the built-in Python library *sqlite3* to implement a file-based SQL database engine to store information resiliently and in a structured way without the need to install additional software. Other database solutions can be implemented by extending *DBAbstract* to provide a storage environment compatible with other user preferences (e.g., SQLServer, MySQL, etc) without impacting the basic functionalities of D-Tailor.

Figure 2.2 depicts a unified model language (UML) class diagram that captures the multiple dependencies between classes implemented in D-Tailor.

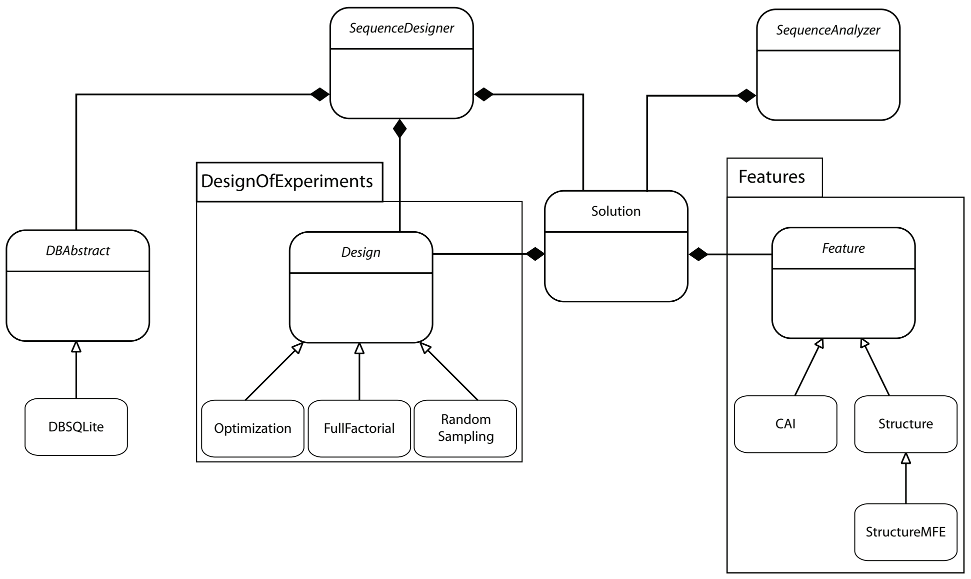


Figure 2.2 UML class diagram of D-Tailor

The two main executable classes (*SequenceAnalyzer* and *SequenceDesigner*) contain one or more instances of the class *Solution*, which contains a list of one or more instances of the class *Feature*. The *SequenceDesigner* requires the instantiation of the class *Design,* which provides basic information about the design target(s), and the class *DBAbstract*, where designed sequences are stored. The diagram shows examples of classes extending the abstract classes *Design* and *Feature*.

The project itself is organized in a series of folders and packages:

- Root directory: contains several core classes of the software (*SequenceAnalyzer*, *SequenceDesigner* and *Solution*). It also contains two auxiliary modules (*Data* and *Functions*) with relevant data structures and functions. *Data* contains all data variables/structures shared by the many classes and can be seen as a repository for global variables. *Functions* provides a repository for common functions that are used by different classes.
- Packages:
    - o *DBOperation*—contains the abstract and concrete classes implementing the storage management system;
    - o *DesignOfExperiments*—includes all classes defining design objectives;

    - *Features*—collection of sequence property evaluators implemented in D-Tailor;
    - *Running examples*—contains usage examples for the two different modes of D-Tailor (analysis and design);
    - *Utils*—a set of auxiliary tools (e.g., database statistics, below).
- Folders:
    - *3rdParty*—folder with 3rd party software that may be required to run certain sequence property evaluators (e.g., UNAFold to predict RNA structures);
    - *testFiles*—a collection of test files that are used by the running examples (e.g., sequences for all *E. coli* genes);
    - *tmp*—a folder where temporary outputs generated by the sequence property evaluators are saved (e.g., structure files produced by UNAFold);
    - *db*—where the databases generated in the design mode can be stored;

3. *Feature* class: handling sequence properties

The class *Feature* encapsulates the concept of a sequence property (i.e., a variable whose score can be inferred/calculated from the raw sequence). This class stores all relevant information about a particular property and contains all methods necessary to calculate its score(s). In D-Tailor, *Feature* is an abstract class that must be extended by classes implementing concrete properties. Figure 3.1 shows an example of a class CAI that calculates of the codon adaptation index. Basically, this class only needs to implement a constructor, which has to call the super class constructor from *Feature* and further define specific attributes, and the method *set_scores*, which computes property score(s). Importantly, the score of a given property needs to be stored in a dictionary called *scores* using the appropriate key, which must be a string resulting from concatenating the given label and the property's class name (Figure 3.1).

```
class CAI(Feature):

    def __init__(self, caiObject = None, solution = None, label="",
                     args = { 'cai_range' : (0,59), 'mutable_region' : None,
                              'cds_region' : None , 'keep_aa' : True }):

        if caiObject == None: #create new instance
            #General properties of feature
            Feature.__init__(self, solution=solution, label=label)
            #Specifics of this Feature
            self.cai_range  = args['cai_range']
            self.sequence   = solution.sequence[self.cai_range[0]:(self.cai_range[1]+1)]
            self.set_scores()
            self.set_level()
        else: #copy instance
            …

    def set_scores(self, scoring_function=Functions.analyze_cai):
        self.scores[self.label+"CAI"] = scoring_function(self.sequence)
```

Figure 3.1 Definition of a class implementing a feature (CAI)

The constructor receives three input parameters: a *Solution* (sequence to evaluate), a *string* (label) and a dictionary *args* with all the parameters necessary to configure the property. In this case, to calculate CAI score, we only need the region of the sequence where we want to compute it. This parameter comes in the dictionary *args* and is accessed via the key *cai_range* (a tuple with the starting and ending position). Next, to calculate CAI score, we implemented the method *set_scores* (Figure 3.1), which uses the routine *analyze_cai* to calculate the geometric mean of the weight associated with each codon within *sequence* (Figure 3.2). To enhance software reusability, we decided to implement functions like this in the *Functions* module.

```
def analyze_cai(seq):
    seq = seq.lower();
    score = 0
    len_sq = 0
    for i in range(0,len(seq),3):
        if cai_table.has_key(seq[i:i+3]):
            score += log(cai_table[seq[i:i+3]])
            len_sq += 1
    score /= len_sq
    return exp(score)
```

Figure 3.2 Calculation of CAI score

Given the simplicity of CAI score calculation, *analyze_cai* can be entirely implemented in Python. However, many complex properties require sophisticated algorithms that are already available in third party software. D-Tailor can also be used to provide a streamlined way to call such software. For example, we implemented another property to evaluate RNA secondary structures (Structure) that uses the external command line tool—UNAfold (Figure 3.3).

```
class Structure(Feature):

    def __init__(self, structureObject = None, solution = None, label="",
                        args = { 'structure_range' : (0,59), 'mutable_region' : None,
                                 'cds_region' : None,'keep_aa' : True }):

        if structureObject == None: #create new instance
            #General properties of feature
            Feature.__init__(self, solution=solution, label=label)
            #Specifics of this Feature
            self.structurefile      = solution.solid + label
            self.structure_range    = args['structure_range']
            self.sequence           =
                   solution.sequence[self.structure_range[0]:(self.structure_range[1]+1)]
            self.set_scores()
            self.set_level()
        else: #copy instance
            …

    def set_scores(self, scoring_function=Functions.analyze_structure):
        scoring_function(self.sequence, self.structurefile)
```

Figure 3.3 Definition of the class Structure

Similarly to CAI, this class only implements the constructor and the *set_scores* method. Here, the parameter specifying the region of the sequence where the structure should be predicted is given by *structure_range* (in *args*). Then, the function *analyze_structure* is used to call the external RNA structure prediction tool and process its output. In this case, the function saves the RNA structure predicted by UNAfold with a predefined name and moves it to 'tmp/structures/' (Figure 3.4).

```
def analyze_structure(seq,filename,ensemble=False):

    chdir(project_dir)
    system("echo '" + str(seq) + "' > " + filename + ".seq")
    fnull = open(devnull, 'w')   # omit output generated by UNAFOLD
    if ensemble:
        call("./3rdParty/unafold/UNAFold.pl -n RNA " + filename + ".seq", shell = True, stdout =
fnull, stderr = fnull)
    else:
        call("./3rdParty/unafold/hybrid-ss-min -n RNA " + filename + ".seq", shell = True, stdout =
fnull, stderr = fnull)
    system("mv %s*.ct tmp/structures/" % filename)
    # remove tmp files
    system("rm %s*" % filename)
    fnull.close()

    return 1
```

Figure 3.4 Prediction of RNA structure using external software

The class Structure does not compute any specific score *per se*. This design pattern is useful when different scores can be derived from the same object, as it avoids re-instantiating the parent object. For example, multiple scores can be inferred from the same RNA secondary structure (e.g., minimum free energy or paired/free nucleotides). In this case, sub-classes implementing the different score calculations should extend the parent class (Structure). Figure 3.5 shows the class StructureMFE, which computes the minimum folding energy (MFE) for an RNA structure predicted using the class Structure. In this particular example, the score is computed by calling another tool of the UNAFold package that calculates the MFE from a structure file. Finally, since this class computes a property score, it is required to update the dictionary *scores*.

```
class StructureMFE(Structure):
    """
    Manipulate the structure MFE
    """
    def __init__(self, structureObject, label = "", regionOfInterest= None):
        Structure.__init__(self,structureObject)
        self.label = self.label + label
        self.set_scores()
        self.set_level()

    def set_scores(self, scoring_function=Functions.analyze_structure_mfe):
        self.scores.update(Functions.appendLabelToDict(scoring_function(self.structurefile), self.label))

Functions.py:

def analyze_structure_mfe(filename,region = None):
   …
   if path.exists(project_dir+"/tmp/structures/"+filename+".ct"):
      output = check_output(["./3rdParty/unafold/ct-energy" , "tmp/structures/"+filename+".ct"]).rstrip()
      mfe_list = [float(a) for a in output.split('\n')]
      data['StructureMFE'] = mfe_list[0]
   else:
      data['StructureMFE'] = 0
   …
```

Figure 3.5 Definition of the class StructureMFE extending Structure

In summary, a class extending the abstract class *Feature* will have the following attributes:

- *label*—a user defined label for the property;
- *solution*—an object of class Solution where the property should be calculated;
- *subfeatures*—a dictionary with all sub-properties associated with this property;
- *scores*—a dictionary containing the score for the property and its sub-properties.

D-Tailor comes out-of-the-box with several properties implemented. Most of them are directly related to sequence properties impacting gene expression. As documented above, the software can easily be extended to implement any other property of interest. A list of the properties currently implemented in D-Tailor is detailed below. Users are encouraged to contact the authors if they need help implementing new properties and/or want to contribute with new ones to future releases of the tool.

Property class	Description	Parameters
CAI	Scores a gene sequence codon usage as compared to that of highly expressed genes. It computes a score between 0 and 1, where the higher the score, the closer is the overall codon usage to the reference set.	*cai_range* : a pair of integers with starting and ending nucleotide positions of the sub-sequence where the CAI should be calculated.
Structure	This property evaluator uses UNAFold to predict the MFE RNA secondary structure. It uses UNAfold and stores the generated structure-related files to the folder '*tmp/structures/*'. Structure class can then be accessed by inheriting sub-classes that compute specific feature scores (see below).	*structure_range* : a pair of integers with starting and ending nucleotide positions of the sub-sequence where the structure should be predicted
StructureMFE	Extends the class Structure to retrieve the MFE structure score, as defined by the Gibbs free energy (ΔG).	*None*
Structure SingleStranded	Extends the class Structure to compute a list and count the total number of single stranded bases (i.e., free) in the MFE structure.	*None*
Structure DoubleStranded	Extends the class Structure to compute a list and count the total number of double stranded bases (i.e., paired) in the MFE structure	*None*
StructureEnsemble	This property evaluator uses UNAFold to compute an ensemble of RNA structures. The predicted structures are stored to 'tmp/structures'.	*structure_range* : a pair of integers with starting and ending nucleotide positions of the sub-sequence where the structure should be predicted

StructureEnsemble Accessibility	Extends the class StructureEnsemble to calculate the average accessibility for each nucleotide (i.e., probability of a nucleotide being free across all structures in the ensemble) and overall average.	*None*
StructureProb	This class uses the software *RNAplfold* from the Vienna RNA package to calculate the average probability of unpaired bases across a sliding window of RNA structures.	*structure_range* : a pair of integers with starting and ending nucleotide positions of the sub-sequence where the structure should be predicted *acc_region* : a list with nucleotide positions if the average of a specific region is desired *window* : window size
HydropathyIndex	This class calculates the average hydropathy index of a peptide based on the properties of its amino acids. Larger scores indicate more hydrophobic properties.	*hi_range* : a pair of integers with starting and ending nucleotide positions of the amino acid subsequence
NucleotideContent	This property evaluator calculates the nucleotide content of a particular sequence (% of A, C, G, T, AT, GC)	*ntcontent_range* : a pair of integers with starting and ending nucleotide positions of the subsequence of interest.
RNADuplex	This class predicts the hybridization of any two RNA molecules. This structure is then saved to 'tmp/structures'.	*rnaMolecule1region* : a pair of integers with starting and ending nucleotide positions of the first RNA molecule *rnaMolecule2region* : as above but for second RNA molecule
RNADuplex Ribosome	Extends the class RNADuplex to implement the interaction between an RNA molecule and the 16S rRNA.	*rnaMolecule1region* : a pair of integers with starting and ending nucleotide positions of the RNA molecule
RNADuplexMFE	Extends the class RNADuplex to calculate the MFE of the duplex.	*None*
Motif	This class implements the search for a given motif (as defined by a position weight matrix (PWM)). D-Tailor comes with pre-configured PWMs for *E. coli*, namely for SD and promoter regions (see module *Data*).	*pwm* : a dictionary where the keys (1-letter conventional symbols for DNA, RNA or amino acids) are associated with a list of weights (one per position) *motif_range* : a pair of integers with starting and ending nucleotide positions of the sequence to be searched
MotifScore	Extends the class Motif to calculate motif score	*None*
MotifPosition	Extends the class Motif to calculate motif position	*None*

4. *Solution* class: handling sequences

The class *Solution* is the realization of a particular sequence along with all the properties of interest that are computed from it, and it has the following basic attributes:

- *solid*—unique solution identifier;
- *sequence*—the full sequence to be analyzed, subsequences being specified at feature level;
- *features*—a dictionary filled with pairs (Feature's label, Feature object);
- *scores*—a dictionary that aggregates all scores for the features of this *Solution* (keys are the labels defined for each feature concatenated with class name);

Some of these attributes are defined when the object is created, namely *solid* and *sequence*. Following the creation of a *Solution*, objects of type *Feature* can be added using the generic method *add_feature*. This method will automatically update *features* and *scores* dictionaries. This way *Solution* objects can be easily created and further populated with an arbitrary set of properties of interest (Figure 4.1).

```
>>> from Solution import Solution
>>> from Features.CAI import CAI
>>> from Features.Structure import Structure, StructureMFE, StructureDoubleStranded
>>> from Features.NucleotideContent import NucleotideContent

## Instantiate an object of type 'Solution'
>>> solution = Solution(sol_id = 'b0001', sequence = 'TATAGGCATAGCGCACAGACAGATAAAAATTACAGAGTACACAA
CATCCATGAAACGCATTAGCACCACCATTACCACCACCATCACCATTACCACAGGTAACGGTGCGGGCTGA')

## Instantiate Feature objects of interest
# Feature to calculates the codon adaptation index
>>> cai_obj = CAI(solution=solution,label="cds",args= {'cai_range':(49,115)})
#Feature to predicts RNA Structure
>>> st1_obj = Structure(solution=solution ,label="utrCds",args= { 'structure_range' : (19,78) } )
# Two sub-features inheriting from the class Structure
>>> st_mfe = StructureMFE(st1_obj)
>>> st_ss  = StructureDoubleStranded(st1_obj)
>>> st1_obj.add_subfeature(st_mfe)
>>> st1_obj.add_subfeature(st_ss)
# Feature to calculate nucleotide content
>>>nuc_obj = NucleotideContent(solution=solution ,label="utr",args= { 'ntcontent_range':(0,50) } )
## Add features to solution
>>> solution.add_feature(cai_obj)
>>> solution.add_feature(st1_obj)
>>> solution.add_feature(nuc_obj)

## Retrieve feature score
>>>  solution.scores
{'cdsCAI': 0.6136121593930156, 'utrCdsStructureDoubleStrandedList':[18, 19, 25, 26, 38, 39, 44, 45],
 'utrCdsStructureDoubleStranded': 8, 'utrCdsStructureMFE': -2.5,
 'utrNucleotideContentAT': 0.63,  'utrNucleotideContentG': 0.16,'utrNucleotideContentT' : 0.18,
 'utrNucleotideContentC': 0.22, 'utrNucleotideContentA' : 0.45, 'utrNucleotideContentGC': 0.37}
```

Figure 4.1 Definition of an object Solution with multiple features

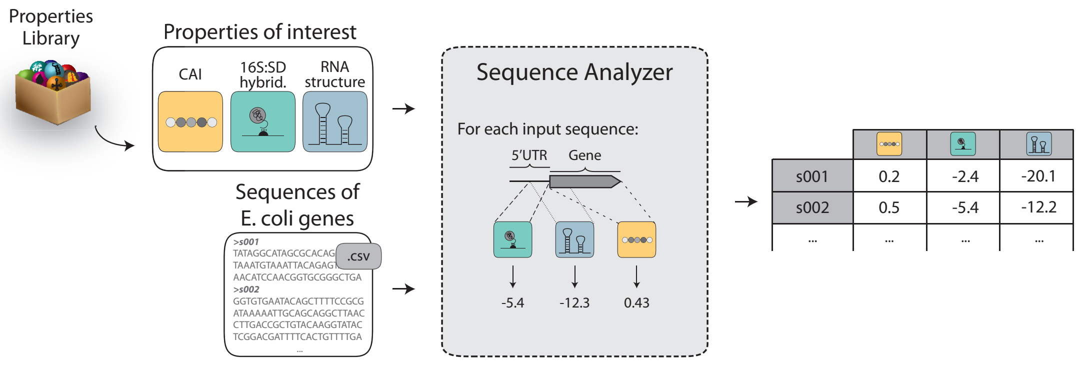
5. Sequence Analyzer

The sequence analyzer mode of D-Tailor provides an integrated solution for the multidimensional interrogation of sequences. Sequences to be analyzed can be read from files in CSV or FASTA format (CSV files must contain the headers 'name' and 'sequence'). Before starting the sequence analyzer, the user needs to extend the abstract class *SequenceAnalyzer* and implement the following methods:

- *configureSolution*—this method instantiates all properties to compute for a sequence (Solution). Its architecture is similar to the one shown in Figure 4.1;
- *outputStart*—called once at the beginning of the method to initialize the output (e.g., open a file and/or write a table header);
- *output*—called after each sequence is analyzed and can be used to perform operations on the retrieved features (e.g., print to the screen).

Figure 5.1 shows a schematic workflow for the analysis of three different features influencing translation efficiency in *E. coli*:

- CAI, a proxy for the translation elongation rate along the gene;
- Hybridization energy between Shine-Dalgarno (SD) region and 16S rRNA;
- RNA Structure around translation initiation region.



```
>s001
TATAGGCATAGCGCACAG
TAAATGTAAATTACAGAGT
AACATCCAACGGTGCGGGCTGA
>s002
GGTGTGAATACAGCTTTTCCGCG
ATAAAAATTGCAGCAGGCTTAAC
CTTGACCGCTGTACAAGGTATAC
TCGGACGATTTTCACTGTTTTGA
...
```

s001	0.2	-2.4	-20.1
s002	0.5	-5.4	-12.2
...	...	...	...

Figure 5.1 Sequence analyzer workflow

The user can read the sequences to analyze from a CSV file. Then, the user selects the sequences properties to evaluate (three in this case: CAI, 16S:SD hybridization energy and RNA structure) from a large library of available properties (box with balls). The sequence analyzer module will then extract and evaluate the sequence property scores for each of the input sequences. The output is a standard table where each row contains property scores for a given input sequence.

Figure 5.2 depicts the corresponding implementation in D-Tailor—class *TranslationFeaturesEcoli* (located in *RunningExamples/Analyzer*).

```
from SequenceAnalyzer import SequenceAnalyzer
from Features import CAI,Structure,RNADuplex
from Functions import validateCDS

class TranslationFeaturesEcoliAnalyzer(SequenceAnalyzer):

    '''
    Class to analyze CAI, SD strength and structure in E. coli
    '''

    def __init__(self, input_file, input_type):
        SequenceAnalyzer.__init__(self,input_file,input_type)

    def configureSolution(self, solution):
        solution.valid = validateCDS(solution.sequence[49:])

        if solution.valid:
            #CAI
            cai_obj = CAI.CAI(solution=solution,label="cds",args= {  'cai_range' :
                                                                      (49,len(solution.sequence)) } )

            #Look for RBS
            dup_obj1 = RNADuplex.RNADuplexRibosome(solution1=solution, label="sd16s",
                                                      args = { 'rnaMolecule1region' : (25,48) })
            dup_mfe = RNADuplex.RNADuplexMFE(dup_obj1)
            dup_obj1.add_subfeature(dup_mfe)

            #MFE [-30,30]
            st1_obj = Structure.Structure(solution=solution,label="utr",
                                                    args= { 'structure_range' : (49-30,49+30) } )
            st_mfe = Structure.StructureMFE(st1_obj)
            st1_obj.add_subfeature(st_mfe)

            solution.add_feature(cai_obj)
            solution.add_feature(dup_obj1)
            solution.add_feature(st1_obj)

    def outputStart(self):
        print "gene_name,sd_hyb_energy,mfe_structure,cai"

    def output(self, solution):
        if solution.valid:
            print solution.solid,",",
                  solution.scores['sd16sRNADuplexMFE'],",",
                  solution.scores['utrStructureMFE'],",",
                  solution.scores['cdsCAI']

if __name__ == '__main__':
    seqAnalyzerTest = \
             TranslationFeaturesEcoli("../../testFiles/genomes/partial_ecoli_genome.csv","CSV")
    seqAnalyzerTest.run()
```

Figure 5.2 Class *TranslationFeaturesEcoliAnalyzer* calculates multiple features for all E. coli genes

This class loads all *E. coli* gene sequences along with the 49 nucleotides preceding them (this table can be found at *testFiles/genomes/ecoli_genome.csv*) into the sequence analyzer module. The three properties of interest are configured in the *configureSolution* method, which also checks if the provided coding sequences are valid (i.e., have start codon and no in-frame stop codons). The user can further define output options using the methods *outputStart* and *output*. In this example, we simply print the computed property scores to the screen (note that the key for each score is the label of the feature concatenated with the property class name, e.g., 'utrStructureMFE'). The program above will print to the screen a table-like output that is partially shown in Figure 5.3.

```
macbook:D-Tailor jcg$ PYTHONPATH=. python RunningExamples/Analyzer/TranslationFeaturesEcoliAnalyzer.py
gene_name,sd_hyb_energy,mfe_structure,cai
b0001,-1.8,-1.925,0.613612159393
b0002,-7,-9.16,0.34043688741
b0003,-5.7,-13.2,0.341658034933
b0004,-3.2,-5.5,0.385891327353
b0005,-7.3,-6.76,0.377281853234
b0006,-6.1,-14.75,0.342733396212
b0007,-5.6,-8.4,0.319183029826
b0008,-2.7,-8.1,0.604195702312
b0009,-3.5,-7.5,0.396623675448
b0010,-6.2,-9.6,0.574062247682
b0011,-0.3,-4.5,0.286738246339
b0013,-2.3,-7.8,0.362374253526
b0014,-5.4,-7.08333333333,0.723381361599
b0015,-4.8,-6.45,0.525547136369
…
```

Figure 5.3 Partial output of *TranslationFeaturesEcoliAnalyzer*

This output can then be easily imported into statistical tools such as SciPy or R for posterior analysis. For instance, Figure 5.4 shows how three different features impacting translation efficiency (hybridization energy between 16S and SD sequence, RNA structure with MFE and CAI score) are distributed and related in *E. coli*.

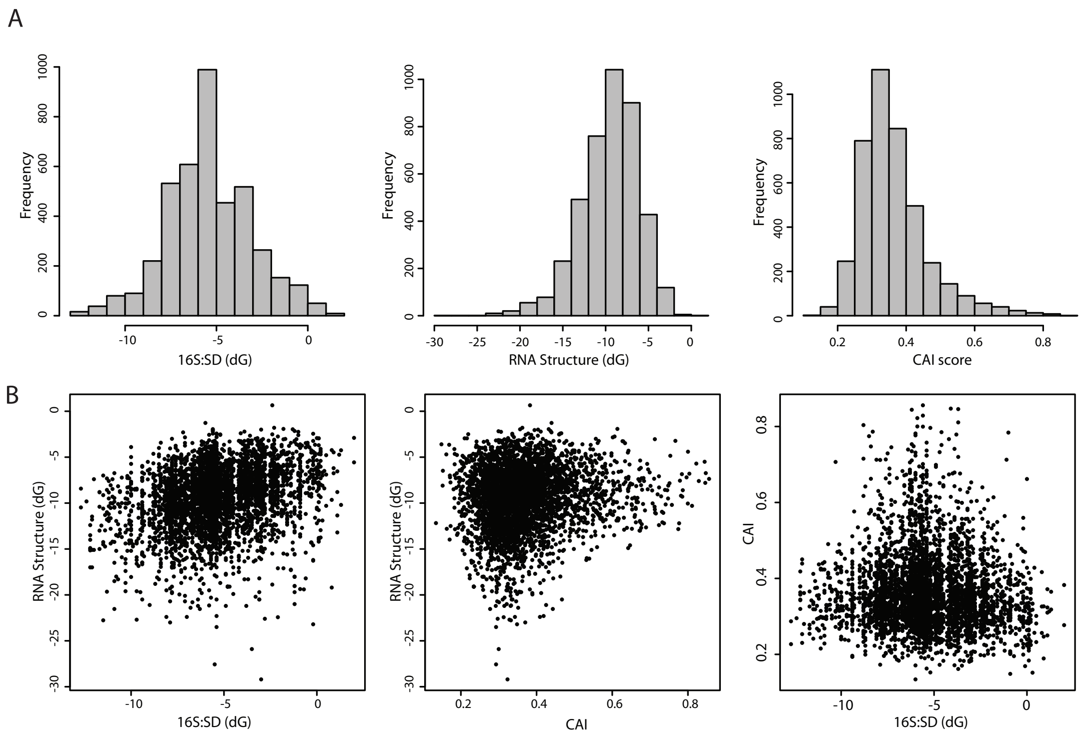


Figure 5.4 Distribution (A) and relationship (B) of three different translation features in *E. coli*

6. Sequence Designer

The most innovative functionality of D-Tailor is the ability to design sequences that meet user-defined goals (Figure 6.1). This section provides a detailed description of how to define a class extending *SequenceDesigner* to use this functionality. Briefly, the user needs to provide a starting seed sequence (from which the designed sequences will be derived), the properties to design for, a design objective (one or more target combinations of feature scores) and a database filename (where generated sequences will be stored). Additionally, we describe multiple parameters by which users can constrain the way sequences are mutated and selected.

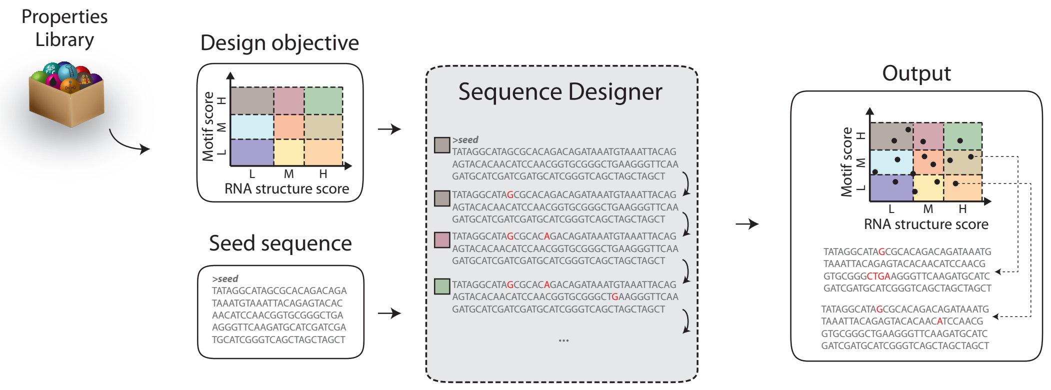

Properties Library
Design objective
Motif score
L
M
H
RNA structure score
Seed sequence
>seed
TATAGGCATAGCGCACAGACAGA
TAAATGTAAATTACAGAGTACAC
AACATCCAACGGTGCGGGCTGA
AGGGTTCAAGATGCATCGATCGA
TGCATCGGGTCAGCTAGCTAGCT
Sequence Designer
>seed
TATAGGCATAGCGCACAGACAGATAAATGTAAATTACAG
AGTACACAACATCCAACGGTGCGGGCTGAAGGGTTCAA
GATGCATCGATCGATGCATCGGGTCAGCTAGCTAGCT
TATAGGCATAGCGCACAGACAGATAAATGTAAATTACAG
AGTACACAACATCCAACGGTGCGGGCTGAAGGGTTCAA
GATGCATCGATCGATGCATCGGGTCAGCTAGCTAGCT
TATAGGCATAGCGCACAGACAGATAAATGTAAATTACAG
AGTACACAACATCCAACGGTGCGGGCTGAAGGGTTCAA
GATGCATCGATCGATGCATCGGGTCAGCTAGCTAGCT
TATAGGCATAGCGCACAGACAGATAAATGTAAATTACAG
AGTACACAACATCCAACGGTGCGGGCTGAAGGGTTCAA
GATGCATCGATCGATGCATCGGGTCAGCTAGCTAGCT
...
Output
Motif score
L
M
H
RNA structure score
TATAGGCATAGCGCACAGACAGATAAATG
TAAATTACAGAGTACACAACATCCAACG
GTGCGGGCTGAAGGGTTCAAGATGCATC
GATCGATGCATCGGGTCAGCTAGCTAGCT
TATAGGCATAGCGCACAGACAGATAAATG
TAAATTACAGAGTACACAACATCCAACG
GTGCGGGCTGAAGGGTTCAAGATGCATC
GATCGATGCATCGGGTCAGCTAGCTAGCT

Figure 6.1 Sequence designer workflow

The user provides a design objective (indicating the sequence properties and levels he/she wants to design for—in this example there are 2 properties with 3 levels each [colored rectangles]) and a seed sequence, which will then be evolved until all combinations of sequence properties levels defined by the user are found.

This chapter provides detailed information about the multiple entities and concepts of the sequence designer module of D-Tailor. It also contains two case studies for the design module: 1) design of sequences using the three features analyzed in the previous chapter; 2) designing artificial bacterial promoter sequences varying multiple *cis*-regulatory properties.

6.1. Definition of features

Users first need to create a class extending *SequenceDesigner.* Similarly to the extension of the *SequenceAnalyzer* class (above), the concrete class has to implement the method *configureSolution* where all the features classes are instantiated and associated with a given *Solution* (Figure 6.2). After that, *SequenceDesigner* requires the configuration of three additional parameters to guide the design process:

- *mutational_region*—a list of all the positions that can be mutated;
- *cds_region*—a pair of integers defining the starting and ending positions of the coding sequence, if any;
- *keep_aa*—a Boolean indicating if only synonymous mutations can be performed.

```
    def configureSolution(self, solution):
        '''
        Solution configuration
        '''
        if solution.sequence == None:
            return 0

        ## Designer specific

        solution.mutable_region=range(0,len(solution.sequence)) # whole region
        solution.cds_region = (49,len(solution.sequence))
        solution.keep_aa = True

        ## Populate solution with desired features

        # CAI
        cai_obj = CAI.CAI(solution = solution,label="cds",
                          args = {  'cai_range' : (49,len(solution.sequence)),
                                    'mutable_region' : range(49,len(solution.sequence)) } )

        # Search SD
        dup_obj1 = RNADuplex.RNADuplexRibosome(solution1=solution, label="sd16s",
                                args = { 'rnaMolecule1region' : (25,48),
                                         'mutable_region' : range(25,48) })
        dup_mfe = RNADuplex.RNADuplexMFE(dup_obj1)
        dup_obj1.add_subfeature(dup_mfe)

        # MFE [-30,30]
        st1_obj = Structure.Structure(solution=solution,label="utr",
                                args = { 'structure_range' : (49-30,49+30)
                                         'mutable_region' : range(49-30,49+30)} )
        st_mfe = Structure.StructureMFE(st1_obj)
        st1_obj.add_subfeature(st_mfe)

        solution.add_feature(cai_obj)
        solution.add_feature(dup_obj1)
        solution.add_feature(st1_obj)
```

Figure 6.2 Definition of the method *configureSolution* in *TranslationFeaturesEcoliDesigner*

This code shows the definition of three sequence features to be computed for a *Solution.* In design mode, some constraints have to be defined to guide mutation process. These can be defined in the *Solution* object or at the *Feature* level (e.g., CAI). In this example, we are declaring that the entire solution region can be mutated, i.e., from position 0 to the length of the entire sequence. Additionally, we define that the gene starts at position 49 and ends at the end of sequence region (attribute *cds_region*), and that we only want to perform synonymous mutations (*keep_aa* = *True*). Since CAI is only affected by mutations within the coding region, we override the mutation constraints for this particular feature and indicate that, to alter this feature score, we should perform mutations only within the gene sequence by configuring the parameter *mutable_region* in the instantiation of the class CAI.

6.2. Definition of a design objective

After the user defines the properties of interest, it is necessary to define a design objective for D-Tailor. A design objective can be one or more target combinations of property scores. Alternatively, when using random sampling, the design objective is simply the number of sequences to be generated. In D-Tailor, a class defining a design objective extends the abstract class *Design* and there are already four predefined methods:

- Optimization—only one specific combination of property scores is desired. For example, to increase the expression of a given gene, we may want to design a sequence with high CAI, strong binding between SD and the 16S rRNA and weak mRNA secondary structure around the initiation region.
- FullFactorial—all possible combinations between the levels of the different properties are generated. This methodology is appropriate to systematically vary the multiple properties and quantify their effect the observed phenotype.
- CustomDesign—this is a more flexible design where the user can indicate each combination of property scores that he/she wants to design for.
- RandomSampling—this method does not enforce any particular combination of properties *a priori*. It can be used to generate a predetermined number of new sequence variants and observe how they scatter across the property space.

Design methods are based on the concept of property levels, which are obtained by discretization of the scores (if necessary), allowing users to define design targets in a more coarse-grained fashion and to yield finite full-factorial designs. The user freely defines the number of levels for each property. Here, the more levels are defined, the higher the resolution (and the smaller the predicted functional difference between levels).

A set of properties and their respective levels need to be inputted to instantiate a sub-class of *Design*. This is given in the form of a dictionary, where for each property it is necessary to define a type (REAL, INTEGER or TEXT) and a list of levels containing the respective lower and upper bounds (Figure 6.3). As mentioned before, the user can freely decide how to discretize the levels for each property, but to ensure biological relevance it may be useful to perform that based on the analysis of natural genomes. For example, let's use the sequence properties influencing translation efficiency analyzed in the previous chapter to discretize the scores space and define a design objective (i.e., one or more combinations of property levels). First, we need to decide in how many levels we want to split each of the properties of interest. Here, we chose to divide each property into 5 different categorical levels (very low, low, medium, high and very high) and used the quintiles identified in the genomic analysis to define their boundaries (Table 1).

Table 1 Definition of feature levels

	Very low (0-20%)	**Low (20-40%)**	**Medium (40-60%)**	**High (60-80%)**	**Very high (80-100%)**
16S:SD	[-12.7, -7.3[	[-7.3, -5.8[	[-5.8, -5.2[	[-5.2, -3.3[	[-3.3,2.0]
RNA structure	[-29.2, -12.2[	[-12.2, -9.95[	[-9.95, -8.4[	[-8.4, -6.73[	[-6.73, 0.65]
CAI	[0.13, 0.29[	[0.29, 0.33[	[0.33, 0.37[	[0.37, 0.42[	[0.42, 0.86]

Second, we need to define a design objective by instantiating one class of type *Design*. The design methods implemented in D-Tailor are located in the package *DesignOfExperiments*. These classes have an attribute (*listOfDesigns*), which is a vector of strings containing the multiple target combinations of property levels (e.g., ‘1.1.1’ indicates a combination where all scores are within level 1—or very low).

Figure 6.3 also shows the definition of a full-factorial design by instantiating the class *FullFactorial*, which only needs to be parameterized with the three sequence properties and the respective level thresholds. Of note, level identifiers must be ordered. To perform a full-factorial design it is necessary to generate all combinations between the 5 levels for each of the 3 properties (i.e., 5*5*5 = 5^3 = 125 combinations).

```
>>> from DesignOfExperiments.Design import FullFactorial

#Design Methodology and thresholds
>>> design_param = {
    "sd16sRNADuplexMFE": { 'type' : 'REAL' ,
                           'thresholds' : { '1': (-12.7,-7.3), '2': (-7.3,-5.8),
                                            '3': (-5.8,-5.2),  '4': (-5.2,-3.3), '5': (-3.3, 2.0) } },
    "utrStructureMFE": { 'type' : 'REAL' ,
                         'thresholds' : { '1': (-29.2,-12.2), '2': (-12.2,-9.95),
                                          '3': (-9.95,-8.4), '4': (-8.4,-6.73), '5': (-6.73,0.65) } },
    "cdsCAI"         : { 'type' : 'REAL' ,
                         'thresholds' : { '1': (0.13,0.29), '2': (0.29,0.33),
                                          '3': (0.33,0.37), '4': (0.37,0.42), '5': (0.42,0.86) } } }

>>> design = FullFactorial(["sd16sRNADuplexMFE","utrStructureMFE","cdsCAI"],design_param)
>>> design.listDesigns
['1.1.1','1.1.3','1.1.2','1.1.5','1.1.4','1.3.1','1.3.3','1.3.2','1.3.5','1.3.4','1.2.1','1.2.3',
'1.2.2','1.2.5','1.2.4','1.5.1','1.5.3','1.5.2','1.5.5','1.5.4','1.4.1','1.4.3','1.4.2','1.4.5',
'1.4.4','3.1.1','3.1.3','3.1.2','3.1.5','3.1.4','3.3.1','3.3.3','3.3.2','3.3.5','3.3.4','3.2.1',
'3.2.3','3.2.2','3.2.5','3.2.4','3.5.1','3.5.3','3.5.2','3.5.5','3.5.4','3.4.1','3.4.3','3.4.2',
'3.4.5','3.4.4','2.1.1','2.1.3','2.1.2','2.1.5','2.1.4','2.3.1','2.3.3','2.3.2','2.3.5','2.3.4',
'2.2.1','2.2.3','2.2.2','2.2.5','2.2.4','2.5.1','2.5.3','2.5.2','2.5.5','2.5.4','2.4.1','2.4.3',
'2.4.2','2.4.5','2.4.4','5.1.1','5.1.3','5.1.2','5.1.5','5.1.4','5.3.1','5.3.3','5.3.2','5.3.5',
'5.3.4','5.2.1','5.2.3','5.2.2','5.2.5','5.2.4','5.5.1','5.5.3','5.5.2','5.5.5','5.5.4','5.4.1',
'5.4.3','5.4.2','5.4.5','5.4.4','4.1.1','4.1.3','4.1.2','4.1.5','4.1.4','4.3.1','4.3.3','4.3.2',
'4.3.5','4.3.4','4.2.1','4.2.3','4.2.2','4.2.5','4.2.4','4.5.1','4.5.3','4.5.2','4.5.5','4.5.4',
'4.4.1', '4.4.3','4.4.2','4.4.5','4.4.4']
```

Figure 6.3 Defining a class of type Design (Full-Factorial)

6.3. Mutational strategies

During the design process, our algorithm applies mutations to generate sequence variants that match desired combination of property levels. Commonly, a random mutation approach is used to generate the multiple variants. However, this can be inefficient because properties are usually located in different regions of the sequence. To optimize this mutational process, different properties can be configured with different mutational regions, that way targeting mutations toward regions of the sequence that are more susceptible to affect the feature score. For example, we can define the *mutable_region* for the 16S:SD hybridization energy feature to comprise the region between [25,48] nucleotides, i.e., where the SD sequence is located. We call this guided procedure: <u>targeted mutagenesis</u>.

In some cases, a good knowledge of the relationship between sequence and property score might allow to devise smart operators that 'rationally' guide the mutation process and increase the likelihood of producing new sequences with the desired score using fewer mutational steps. For example, if meeting the design goal requires CAI to increase, a smart mutation operator can readily replace a poor codon by an alternative one with a higher CAI score. We call this guided procedure: <u>oriented mutagenesis</u>.

Oriented mutational strategies provide some improvements over random and target mutation operators, and therefore should be implemented whenever possible.

The default mutation operator defined in the abstract class *Feature* implements the 'targeted' mutation operator with equiprobable mutation at all predefined mutable positions. When developing a new property, users can override this operator with an oriented one by implementing the method *mutate* in the respective feature (Figure 6.4). Specific instruction regarding the direction the target score can be defined in the method *defineTarget* and stored in the class variable *targetInstructions* (note that abstract class *Feature* implements a minimal version of this method, where the direction is set to '+' if increasing the feature score is needed, or '-' otherwise) (Figure 6.4).

The code implementing oriented RNA structure mutations is depicted in Figure 6.4. Here, we mutate either paired or unpaired bases if we want to decrease or increase structure strength, respectively.

```
Feature.py:

    def defineTarget(self,desiredSolution):
        '''
        Function that determines if a target wasn't hit and, if not, updates target instructions
        '''
        if desiredSolution == None:
            return True

        #check if there is a target
        if not desiredSolution.has_key(self.label+self.__class__.__name__+"Level"):
            return False
        else:
            target_level = desiredSolution[self.label+self.__class__.__name__+"Level"]

            if target_level == 0:
                return False

            if  target_level != self.level:
                level_info =
            self.solution.designMethod.thresholds[self.label+self.__class__.__name__][target_level]

                if isinstance(level_info, tuple): #Then it's a numeric range
                    if  level_info[0]-self.scores[self.label+self.__class__.__name__] > 0:
                        self.targetInstructions['direction'] = '+' #increase
                    elif level_info[0]-self.scores[self.label+self.__class__.__name__] < 0:
                        self.targetInstructions['direction'] = '-' #decrease
                else:
                    self.targetInstructions['direction'] = 'NA' #not applicable

                return True

            return False

Structure.py:

    #Overriding the mutation method to implement oriented mutation
    def mutate(self, operator=Functions.SimpleStructureOperator):
        if not self.targetInstructions:
            return None
        ss_bases = None if not self.scores.has_key(self.label+'StructureSingleStrandedBasesList') else
self.scores[self.label+'StructureSingleStrandedBasesList']
        ds_bases = None if not self.scores.has_key(self.label+'StructureDoubleStrandedBasesList') else
self.scores[self.label+'StructureDoubleStrandedBasesList']
        new_seq = operator(self.solution.sequence, self.structurefile,  self.structure_range,
self.mutable_region, self.cds_region, self.targetInstructions['direction'], ss_bases=ss_bases,
ds_bases=ds_bases)
        if not new_seq:
            return None
        return Solution.Solution(sol_id=str(uuid4().int), sequence=new_seq, cds_region =
self.cds_region, mutable_region = self.mutable_region, parent=self.solution,
design=self.solution.designMethod)

Functions.py:

def SimpleStructureOperator(sequence, structurefile, structure_range, mutable_region, cds_region,
direction, keep_aa = True, ss_bases=None, ds_bases=None):

    if not mutable_region: #it's not possible to mutate
        return None

    # get single stranded bases
    if ss_bases == None:
        ss_bases = structureAnalysis(structurefile, "ss")
    # get double stranded bases
    if ds_bases == None:
        ds_bases = structureAnalysis(structurefile, "ds")

    # for structure, increasing structure score (MFE) (+) means that we want to produce weaker
structures, so we will mutate double stranded bases
```

```
    if direction == '+':
        #get double stranded bases
        baseToMutate = [(b+structure_range[0]-1) for b in ds_bases \
                                                  if (b+structure_range[0]-1) in mutable_region]
    # conversely to increase structure we mutate single stranded bases
    elif direction == '-':
        #get single stranded bases
        baseToMutate = [(b+structure_range[0]-1) for b in ss_bases \
                                                  if (b+structure_range[0]-1) in mutable_region]
    else:
        sys.stderr.write("Direction Unknown")

    mutated = False
    iteration = 0

    #try to mutate up to 100 different times
    while not mutated and iteration <= 100:
        #select a position to mutate at random
        index_to_mutate = baseToMutate.pop(randint(0,len(baseToMutate)-1)) if len(baseToMutate) != 0
else mutable_region.pop(randint(0,len(mutable_region)-1))

        #mutate base keeping amino acids (omitted)
        if keep_aa == True and index_to_mutate >= cds_region[0] and index_to_mutate <= cds_region[1]:
            …
        #mutate without keeping amino acids
        else:
            mutated = True
            new_seq = list(sequence)
            if direction == '+':
                comp = complementary(sequence[index_to_mutate])
            else:
                comp = randomMutation(sequence[index_to_mutate])
            new_seq[index_to_mutate] = comp
            #print sequence
            #print "".join(new_seq)

        iteration+=1

    return "".join(new_seq)
```

Figure 6.4 Definition of the method *mutate* in *Structure*

6.4. Designer algorithm

The algorithm that generates desired sequences is implemented by the method *run* in *SequenceDesigner*. Briefly, the algorithm loops through an evolution cycle until it finds all the user-defined combinations of property scores. The pseudocode and a schematic of the algorithm are presented in Figure 6.5 and Figure 6.6, respectively.

Each evolution cycle consist of three steps: i) a particular target that is yet to be found is selected (step 1); ii) the repository of sequences previously generated (including the seed sequence) is searched to select a template sequence based on a fitness proportionate method, where the fitter the sequence, the shorter the Euclidean distance between its feature scores and the target combination (step 2); iii) a defined number of mutational iterations starting with the selected template sequence is performed (step 3).

In each mutational iteration, the current sequence is evaluated and a property, whose score does not match the desired combination, is randomly selected and a mutation is applied. As mentioned before, two types of mutations can be applied: 1) targeted mutation, and 2) oriented mutation. Briefly, the former specifically targets sequence regions that are more likely to alter the score, whereas the latter applies mutations that will specifically move a property score toward the desired level. Next, the scores for the new sequence variant are evaluated and if the new sequence matches the target combination, then the sequence is validated and the evolution cycle is terminated. Otherwise, one of the two sequences (the template or the mutated one) is chosen as the template for the next iteration of the evolution cycle depending on the selection option:

- *neutral*—one of the sequences is randomly selected;
- *directional*—the sequence with shorter Euclidean distance between the feature scores and the desired combination of feature levels is selected;
- *temperature*—the sequence is selected based on a temperature schedule.

```
while combinations_to_find != []:
     desired_combination = getElement(combinations_to_find)
     #get a sequence already generated that is close to the desired combination in the feature space.
     solution = getSolutionFromDataBaseCloserTo(desired_combination)

     #Evolution cycle
     while solution != desired_combination or iteration != MAX_ITERATIONS:
          old_solution = solution
          solution = solution.mutate() #mutate current solution to get a new sequence
          DataBase.store(solution)
          If solution.combination in combinations_to_find:
               Combinations_to_find.remove(solution.combination)
          #select sequence for next iteration
          #The “selectionMethod” can be either directional or neutral or based on temperature schedule
          solution = selection(old_solution,solution,”selectionMethod”)
```

Figure 6.5 Pseudocode of *SequenceDesigner* algorithm

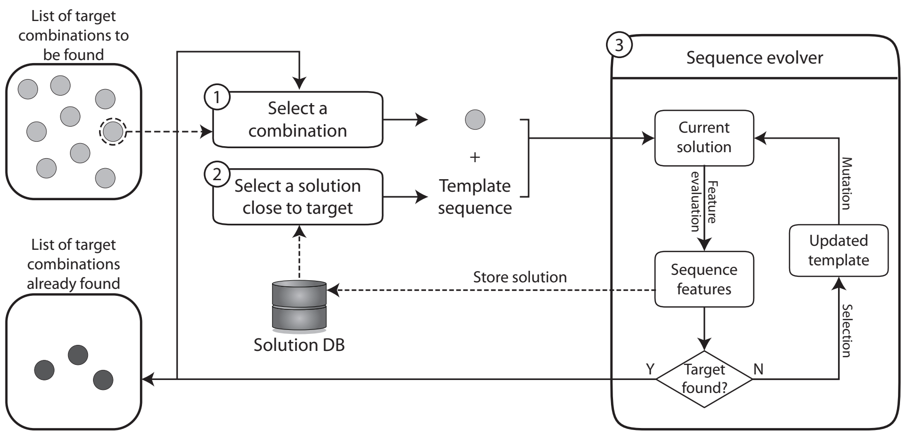


Figure 6.6 Schematic of the *SequenceDesigner* algorithm

6.5. Database of designed sequences

D-Tailor uses an SQLite database engine to store solutions generated during the design process. This database contains three different tables:

- desired_solution—dynamic table created on-the-fly during initialization of the database containing user-defined target combinations of feature levels;
- generated_solution—dynamic table where all generated solutions are stored;
- worker—table stores the SequenceDesigner programs that already ran.

D-Tailor can be easily extended to other database engines. For that, it is only necessary to extend abstract class *DBAbstract* and implement the required methods.

6.6. Configuring and running the designer

To start running the algorithm we need a *seed* sequence from which the designed sequences will be generated, a design objective (from the class *Design*), a file path to create a database containing all the generated sequences (Figure 6.7). In the example below, we used a *seed* sequence that contains a 5'UTR of 49 nucleotides including a SD region, followed by the gene sequence encoding human insulin (NCBI: NM_000207.2).

```
>>> from RunningExamples.Designer.TranslationFeaturesEcoliDesigner import
TranslationFeaturesEcoliDesigner
>>> from DesignOfExperiments.Design import FullFactorial

#Seed sequence from which mutants will be derived
>>> seed='ttattaccggacaataatatttcaattcattaaagaggagaaaggtaccatggccctgtggatgcgcctcctgcccctgctggcgctgctgg
ccctctggggacctgacccagccgcagcctttgtgaaccaacacctgtgcggctcacacctggtggaagctctctacctagtgtgcggggaacgaggcttct
tctacacacccaagacccgccgggaggcagaggacctgcaggtggggcaggtggagctgggcgggggccctggtgcaggcagcctgcagcccttggccctgg
aggggtccctgcagaagcgtggcattgtggaacaatgctgtaccagcatctgctccctctaccagctggagaactactgcaactag'

#Design Methodology and thresholds
>>> design_param = {  "sd16sRNADuplexMFE": { 'type' : 'REAL' ,
                                            'thresholds' : { '1': (-12.7,-7.3), '2': (-7.3,-5.8),
                                                             '3': (-5.8,-5.2),  '4': (-5.2,-3.3),
                                                             '5': (-3.3, 2.0) } },
                     "utrStructureMFE"  : { 'type' : 'REAL' ,
                                            'thresholds' : { '1': (-29.2,-12.2), '2': (-12.2,-9.95),
                                                             '3': (-9.95,-8.4),  '4': (-8.4,-6.73),
                                                             '5': (-6.73,0.65) } },
                     "cdsCAI"           : { 'type' : 'REAL' ,
                                            'thresholds' : { '1': (0.13,0.29), '2': (0.29,0.33),
                                                             '3': (0.33,0.37), '4': (0.37,0.42),
                                                             '5': (0.42,0.86) } }
                   }

>>> design = FullFactorial(["sd16sRNADuplexMFE","utrStructureMFE","cdsCAI"],design_param)

>>> tirap_designer = TranslationFeaturesEcoliDesigner("tfec", seed, design,
"/Users/jcg/Documents/workspace/D-Tailor/testFiles/outputFiles/tfec_1", createDB=True)
>>> tirap_designer.run()
```

Figure 6.7 Running the *SequenceDesigner*

Please note that the regions of the seed sequence that can be mutated were already defined in the class method *configureSolution* (Figure 6.2). Additionally, the user may also want to implement the method *validateSolution*, which is called every time a new sequence is generated. This validation step is fundamental to avoid undesired properties in new sequence variants (e.g., a spurious restriction site). Only validated sequences will be stored in the database. Our exemplary class *TranslationFeaturesEcoliDesigner* implements a series of validation tests that one may want/need (Figure 6.8). Specifically, it checks if the new sequence does not include internal promoters, terminators and undesirable restriction enzymes sites (in this case BsaI sites).

```
def validateSolution(self, solution):
    '''
    Solution validation tests
    '''
    if solution.sequence == None or ('?' in solution.levels.values()):
        solution.valid = False
        return 0

    #check if solution is valid
    valid = True
    designed_region = solution.sequence

    #No internal Promoters
    (score, position, spacer) = Functions.look_for_promoters(designed_region)
    if score >= 15.3990166: #~0.95 percentile for Promoter PWM scores
        valid = False
        sys.stderr.write("SolutionValidator: High Promoter score\n")

    #No internal Terminator
    score = Functions.look_for_terminators(designed_region)
    if score >= 90: #90% confidence from transtermHP
        valid = False
        sys.stderr.write("SolutionValidator: High Terminator score\n")

    #No BsaI sites
    if 'ggtctc' in designed_region or 'gagacc' in designed_region:
        sys.stderr.write("SolutionValidator: Restriction enzyme found\n")
        valid = False

    solution.valid = valid

    return valid
```

Figure 6.8 Definition of the method *validateSolution*

The parameter *createDB* in the *SequenceDesigner* constructor (Figure 6.7) should be set to 'True' when a new empty database is desired. Otherwise, if the database is already created and we want to resume the designer algorithm or start multiple concurrent algorithms, we must to set this parameter to 'False'.

When the method *run()* is invoked in *SequenceDesigner*, the program will only stop when the design objective is achieved (i.e., all combinations are found). While running, the program will output the particular target combination it is looking for and statistics about generated sequence (Figure 6.9). When Optimization design is selected, the program will additionally print the final designed sequence (Figure 6.10).

```
macbook:D-Tailor jcg$ PYTHONPATH=. python RunningExamples/Designer/TranslationFeaturesEcoliDesigner.py
fullfactorial
looking for combination:  2.3.5
SolutionValidator: Restriction enzyme found
No solution could be found...
looking for combination:  3.2.5
No solution could be found...
looking for combination:  4.4.3
No solution could be found...
time elapsed: 76.58 (s)    solutions generated: 385          rate (last min.): 5.03 sol/s      rate
(overall): 5.03 sol/s
looking for combination:  3.5.3
…
Program finished...
```

Figure 6.9 Running *TranslationFeatureEcoliDesigner* (FullFactorial design)

```
macbook:D-Tailor jcg$ PYTHONPATH=. python RunningExamples/Designer/TranslationFeaturesEcoliDesigner.py
optimization 1.2.3
looking for combination:  1.2.3
Solution found... inserting into DB...

###########################
# Optimized solution:
# ID:  46124799975394009622803191427036818508
# Sequence:
ttattaccggacaataatatttcaattcattaaagaggagaaaggtaccatggcactttggatgcgcctcctgcccttactggcattactggcgctgtgggg
ccctgacccggccgccgccttcgtgaatcaacatctgtgcggatcacacttggttgaggctctttacttagtgtgcggggaacgcggtttttctacacacc
aaaaacgcgccgggaagcagaagacctgcaggttgggcaggtagaattaggtgggggccctggtgctggcagcctgcagcccctggccctggaaggatccct
gcagaaacgtggaattgttgaacaatgctgcaccagcatctgttcgttataccagttagagaactactgcaactag
# Scores:  ['sd16sRNADuplexMFE: -8.4', 'utrStructureMFE: -10.4', 'cdsCAI: 0.346776020332']
# Levels:  ['sd16sRNADuplexMFELevel: 1', 'utrStructureMFELevel: 2', 'cdsCAILevel: 3']
# Number of generated solutions:  66
# Distance to seed:  49
###########################

Program finished...
```

Figure 6.10 Running TranslationFeatureEcoliDesigner (Optimization design)

Lastly, when using D-Tailor to randomly sample the sequence space, users must indicate a number of sequences variants to generate (Figure 6.11).

```
macbook:D-Tailor jcg$ PYTHONPATH=. python RunningExamples/Designer/TranslationFeaturesEcoliDesigner.py
randomsampling 1000
time elapsed: 61.98(s)  solutions generated: 297 rate(last min.): 4.79 sol/s rate(overall): 4.79 sol/s
time elapsed: 134.74(s) solutions generated: 665 rate(last min.): 5.06 sol/s rate(overall): 4.94 sol/s
time elapsed: 199.14(s) solutions generated: 962 rate(last min.): 4.61 sol/s rate(overall): 4.83 sol/s
RandomSampling: 1000 solutions generated.
Program finished...
```

Figure 6.11 Running *TranslationFeaturesEcoliDesigner* (RandomSampling design)

6.7. Running examples

6.7.1. Designing sequences systematically varying sequence properties impacting translation efficiency

The code used to run this design example can be found at
'RunningExamples/Designer/TranslationFeaturesEcoliDesigner.py'

We have already shown how to use D-Tailor to analyze three different sequence properties impacting translation efficiency across the entire *E. coli* genome (Figure 5.4). We then discretized property scores into five different levels based on their respective quintiles (Table 1). A full-factorial design based on such configuration yields a total of 125 (5^3) different target combinations of all property levels across the three variables.

To demonstrate flexibility of the software to use different seed sequences, we randomly selected 30 gene sequences along with their 5'UTR from *E. coli* and compared four different strategies within D-Tailor to design a set of sequences conforming to a full-factorial design for each of the seeds. We also defined that designed sequences could be generated by unrestricted mutations in the entire 5'UTR region, composed by 49 nucleotides, but only synonymous mutations were allowed in the gene coding sequence.

We first used the most rudimentary design strategy available in D-Tailor, random sampling, to generate random sequences until the 125 different targets were found. Every attempt to complete this design goal using this purely random procedure was aborted after 3,000 generated sequences due to its obvious inefficiency (Figure 6.12A-B, black solid and faded lines). The second design strategy included the canonical heuristic algorithm implemented by D-Tailor (Figure 6.6) and used the simplest mutation method, wherein new sequences are consecutively generated by random mutation (Figure 6.12B, yellow line). This strategy significantly improved the efficiency of the search algorithm as compared to that of the random sampling method. Nonetheless, the overall performance of the algorithm was still modest since many sequences had to be generated to find the required targets. The third mutational strategy remarkably improved the search algorithm efficiency by employing spatially targeted mutations that more rapidly evolve a sequence towards some desired feature scores target (Figure 6.12B, light blue line). Lastly, a fourth strategy using more 'rational' mutation operators that orient mutations toward the desired target provided slightly faster dynamics (Figure 6.12A-B, orange solid and lines).

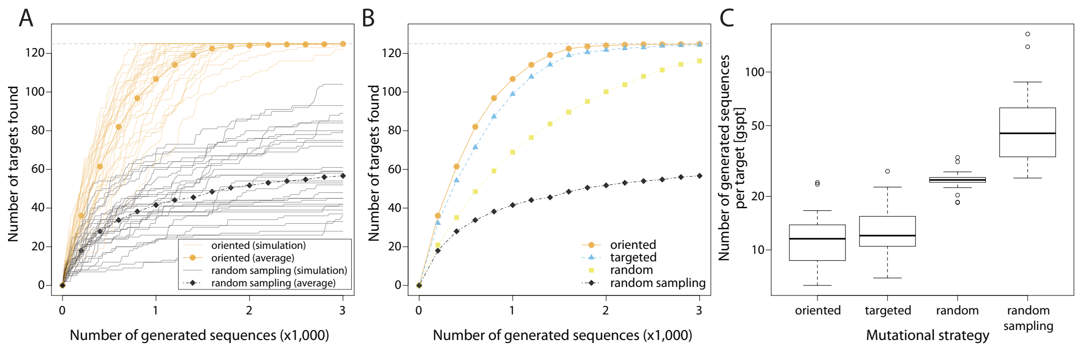


Figure 6.12 Mutational strategies performances

(A) Number of combinations found (out of 125) as a function of the number of generated sequences. Two different mutational strategies are depicted: oriented (orange) and random sampling (black). For each strategy, we performed 30 different simulation of a full-factorial design (faded lines) using different seed sequences. The solid lines represent the average number of target combinations found (across 30 replicates) as the number of generated sequences increases. (B) The average performance of the four different mutational strategies. (C) The number of generated sequences per target combination found using the different mutational strategies.

One other important functionality available to the user is the option to define the selection bias in the heuristic algorithm. The option is configured using the parameter *selection* in the method *run()*, and there are three different options available:

- *neutral*—the sequence for the next mutational iteration is randomly selected between the template sequence and newly sequence variant.
- *directional*—the sequence with shorter Euclidean distance between the feature scores and the desired combination of feature levels is selected.
- *temperature*—the sequence is selected based on a temperature schedule that allows worse sequence (longer distances) to also be selected.

To test these different selections, we evolved the previously selected 30 seed sequences toward six different target combinations bearing different Euclidean distances to the seeds (Figure 6.13). Then, we examined the behavior of the algorithm in response to the three contrasted selective regimes: neutral, directional and temperature selection. As expected, when using D-Tailor with a less constrained selection (i.e., neutral), it was necessary to generate more sequences to find the target combination(s). This relaxed selection does not select fitter strains and, hence, takes more evolution cycles to find the desired target (Figure 6.13B and D). However, the final designed sequences using the neutral selection option will be more similar to the initial seed sequence (measured using the hamming distance to the seed sequence) than when using the option directional selection (Figure 6.13A and C).

Conversely, using more biased selection procedures (i.e., directional or temperature) affords the design of sequences bearing the desired combination of feature levels using much fewer evolution cycles (Figure 6.13B and D) at the expense of generating sequences less similar to the original seed (Figure 6.13A and C). Of note, the directional selection shows slightly better performance than the temperature selection once it requires fewer sequences to be generated and obtains shorter hamming distances.

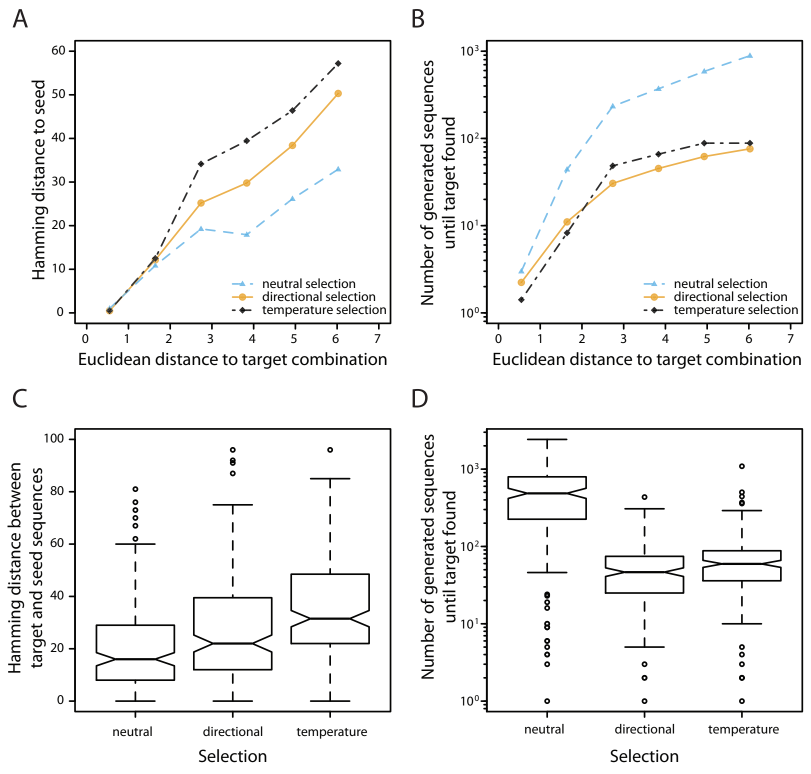


Figure 6.13 Selection options in *SequenceDesigner*

(A) The different lines show the average hamming distance between the seed and the sequence matching the target combination as a function of the Euclidean distance to the target combination using neutral (light blue), directional (orange) or temperature (black) selection. (B) The number of generated sequences until the desired target is found as a function of the Euclidean distance to the target combination using either neutral (light blue), directional (orange) or temperature (black) selection. (C and D) The hamming distance (C) and number of generated sequences until target is found (D) for the 30 different simulations using the three different selections.

6.7.2. Designing bacterial promoter sequences systematically varying *cis*-regulatory properties

The code used to run this design example can be found at ‘RunningExamples/Designer/BacterialPromotersDesigner.py’

In this section, we used D-Tailor to design bacterial promoter sequences varying multiple properties. Promoter strength is not only determined by the affinity between the sigma factor and binding motifs (-35 and -10 boxes), but also by the presence of an UP-element (that can bind the RNA polymerase holoenzyme α subunit carboxy-terminal domain) and transcription factor binding sites. To demonstrate its versatility, we used D-Tailor to design artificial bacterial promoter sequences varying five different regulatory properties (Figure 6.14):

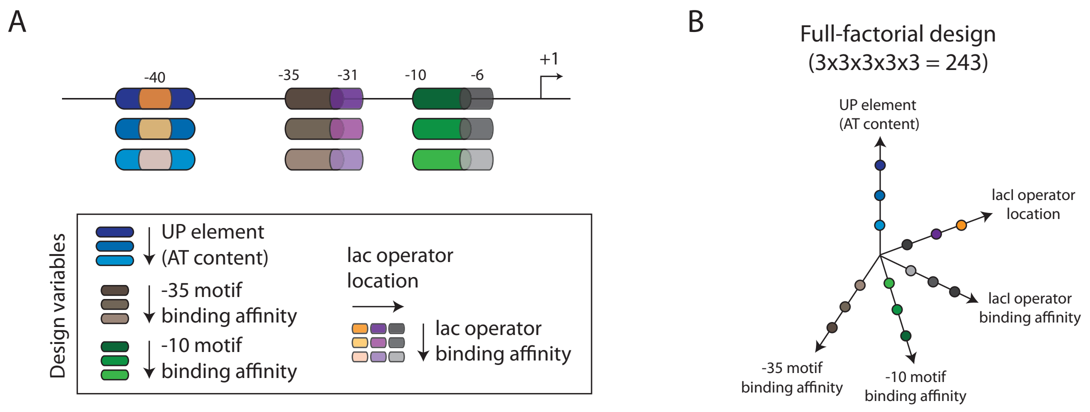

A
-40
-35
-31
-10
-6
+1
Design variables
UP element (AT content)
-35 motif binding affinity
-10 motif binding affinity
lac operator location
lac operator binding affinity
B
Full-factorial design (3x3x3x3x3 = 243)
UP element (AT content)
lacI operator location
lacI operator binding affinity
-35 motif binding affinity
-10 motif binding affinity

Figure 6.14 Design of artificial bacterial promoter sequences

(A) Five different regulatory properties can be changed to create promoter diversity: UP-element, -35 motif, -10 motif, and the lacI operator motif and its location. (B) The different five design axes (one per property), each containing three different levels. A full-factorial design systematically varying these five properties with 3 levels each yields a total of 243 different combinations.

Similarly to the previous example, we need to start by implementing the class *BacterialPromotersDesigner* defining the multiple sequence properties of interest and their location in the DNA segment (Figure 6.15). Then, we discretized the range of sequence property scores into three different levels as detailed in Table 2. Next, we configured the designer algorithm with a randomly generated seed sequence (with no specific biological function) and the design objective of constructing a full-factorial library where all combinations of levels across the five different variables are produced (3x3x3x3x3 = 243) (Figure 6.14B).

```
class BacterialPromotersDesigner(SequenceDesigner):

    def __init__(self, name, seed, design, dbfile, createDB=True):
        SequenceDesigner.__init__(self, name, seed, design, dbfile, createDB)
        self.max_sol_counter = 1000000
        self.max_iterations  = 1000

    def configureSolution(self, solution):
        '''
        Solution configuration
        '''
        #Populate solution with desired features
        solution.mutable_region=range(0,len(solution.sequence)) # whole region
        solution.cds_region = None
        solution.keep_aa = False

        # The entire promoter + 5' utr will have 75 nucleotides (and TSS will be at nucleotide 50)
        #   UP element     -35         spacer       -10                5' utr
        # UUUUUUUUUUUUUUU MMMMMM SSSSSSSSSSSSSSSSSS DDDDDD AAAA OOOOOOOOOOOOOOOOOOOOOOOOO

        #UP element (-50,-36)
        up_obj = NucleotideContent.NucleotideContent(solution=solution,label="up",
                               args= {  'ntcontent_range' : (0,14), 'mutable_region' : range(0,15) } )
        upat_obj = NucleotideContent.NucleotideContentAT(up_obj)
        #-35 motif (-35,-30)
        m35_obj = Motif.Motif(solution=solution,label="m35",
            args= {  'motif_range' : (15,20), 'pwm' : Data.pwm_35, 'mutable_region' : range(15,21) } )
        m35score_obj = Motif.MotifScore(m35_obj)
        #-10 motif (-10,-5)
        m10_obj = Motif.Motif(solution=solution,label="m10",
            args= {  'motif_range' : (40,45), 'pwm' : Data.pwm_10, 'mutable_region' : range(40,46) } )
        m10score_obj = Motif.MotifScore(m10_obj)
        #lacI operator (-6, +25)
        mlacI_obj = Motif.Motif(solution=solution,label="mlacI",
            args= { 'motif_range' : (0,74), 'pwm' : Data.pwm_lacI,'mutable_region' : range(0,74) } )
        mlacIscore_obj = Motif.MotifScore(mlacI_obj)
        mlacIpos_obj = Motif.MotifPosition(mlacI_obj)

        solution.add_feature(upat_obj)
        solution.add_feature(m35score_obj)
        solution.add_feature(m10score_obj)
        solution.add_feature(mlacIscore_obj)
        solution.add_feature(mlacIpos_obj)
```

Figure 6.15 Configuration of class BacterialPromoterDesigner

Table 2 Discretization of sequence properties

	Level 1	**Level 2**	**Level 3**
UP-element (%AT content)	[0 , 0.25]	[0.25 , 0.75]	[0.75 , 1]
-35 motif (binding affinity)	[-12.0 , -6.81]	[-6.81 , 0.63]	[0.63 , 11.0]
-10 motif (binding affinity)	[-12.0 , -8.19]	[-8.19 , 0.32]	[0.32 , 11.0]
lacI operator (binding affinity)	[0 , 4]	[4 , 8]	[8 , 12]
lacI operator location	-40	-31	-6

We have repeated the full-factorial design simulation ten different times and the number of combinations found as a function of generated sequences is depicted in Figure 6.16. We saw that D-Tailor took an average of ~8,000 generated sequence variants to find the 243 desired combinations across the ten different simulations. We also observed that D-Tailor could find the multiple target combinations at relatively steady rate up to

~80% of the total 243 targets (similar target discovery performance rate was observed when designing sequences varying translation-related features (Figure 6.12)). This may have to do with the increase in difficulty to attain certain combination of features.

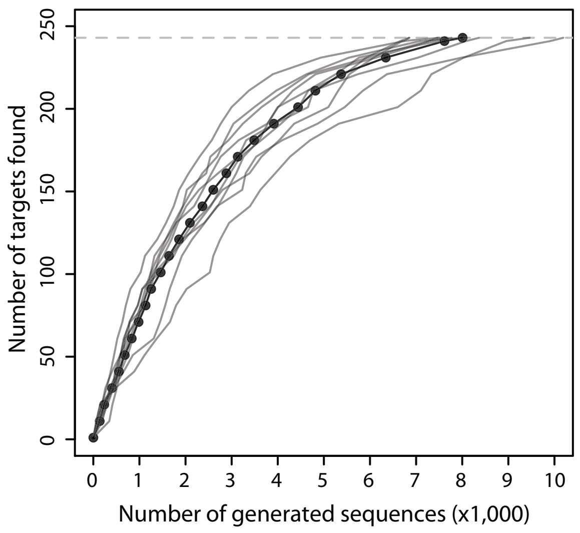


Figure 6.16 Full-factorial design of bacterial promoter sequences

The number of generated solutions as a function of the number of target combinations found for the ten simulations (grey lines). The dotted black line shows the average number of generated solutions across the different simulations as a function of the number of targets found. Grey dashed line indicates the total number of combinations to be found in the full factorial design (243).

To evaluate the convergence of the algorithm, we selected three different target combinations of sequence properties bearing different Euclidean distances from the starting seed. Then, we ran five different simulations for each of the three targets. Figure 6.17 shows the convergence of the distance between generated sequence variants and target combination as measured by the Euclidean distance (or the objective function).

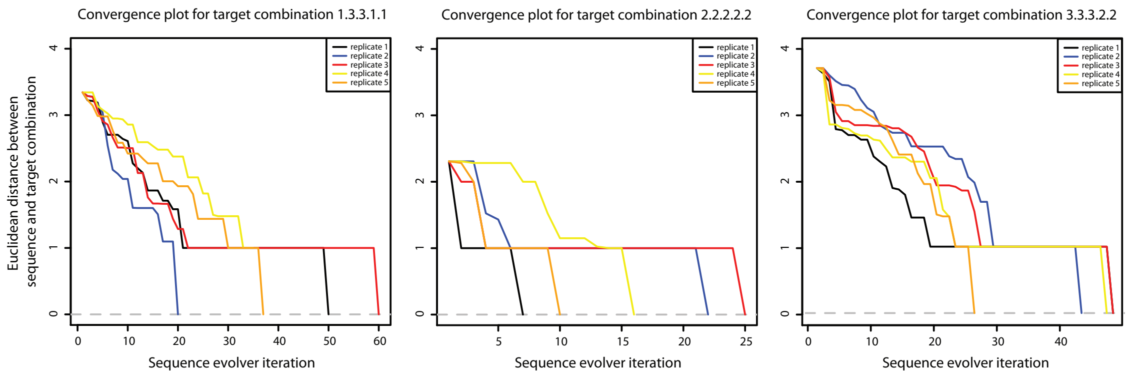


Figure 6.17 Convergence of the search algorithm for three different target combinations

The Euclidean distance to target combination is depicted for the sequence selected after each iteration.

We can see that, for each of the targets, the number of iterations necessary to find the desired combination varies across the different simulations (or replicates). This is expected given the stochastic nature of the Monte-Carlo algorithm being used. We can also see that the average number of iterations (across the five simulations) varies from target to target. As expected, targets that are further away from the seed sequence require a greater number of iterations than the ones that are closer (e.g., compare combination 2.2.2.2.2 with 3.3.3.2.2).

We further looked at the ruggedness of the landscape to evaluate how the different generated sequence variants populate the fitness landscape and get a hint about the difficulty to achieve each target combination. Figure 6.18 depicts the fitness landscape, as defined by the Euclidean distance between the properties of a generated sequence and the desired combination of property scores, for the three different targets already explored above. We see that depending on the target combination, the fitness landscapes can vary widely. Of course the surface of these landscapes is unpredictable and it will vary depending on the sequence properties as well as the target combinations being explored.

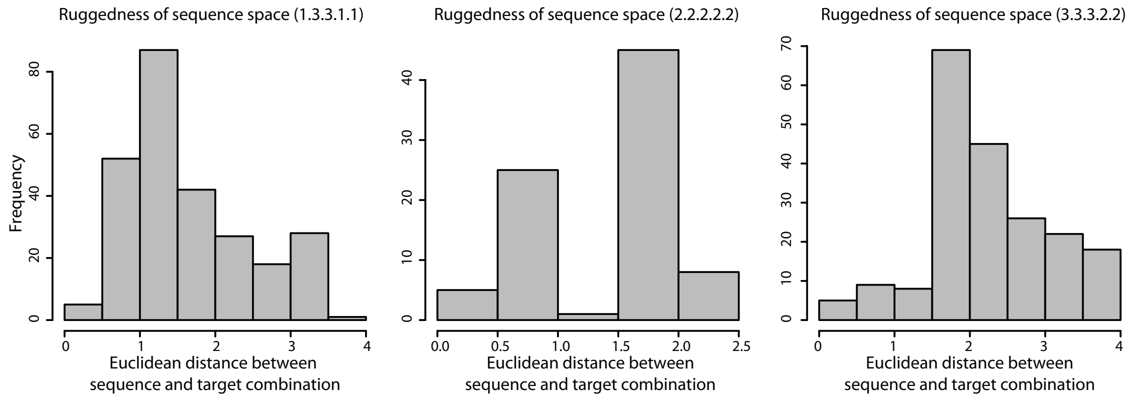


Figure 6.18 Ruggedness of sequence space for three different targets

Histograms depict the Euclidean distances between the properties of the generated sequence variants and the target combination. To generate each histogram, five different simulations were performed for each target.

6.8. Utilities

D-Tailor uses an SQLite database to store all the generated sequences and their features. That way generated solutions can be accessed using a standard SQLite client (e.g., SQLite Manager add-on for Firefox) or use some D-Tailor utilities to export the generated sequences and retrieve statistics (Figure 6.19). The following utilities are available in the package *Utils*:

- DB2CSV—exports the tables containing all the generated solutions and desired combination to a CSV file specified by the user;
- DB2FASTA—exports all the generated sequences to FASTA format;
- DBStatistics—a script that can be used to query a SQLite instance and print the number of sequences generated and different combinations found;
- DBKinetics—prints a time series of defined size (default 50) with the number of generated solutions and combinations found over time.

D-Tailor provides one more tool that will be essential for full-factorial designs. Because the number of combinations required by these designs can be extremely hard to achieve, many solutions may be generated during the design process. For example, to generate a full factorial design for the three features affecting translation with 5 levels each (a total of 125 combinations) it was necessary to generate an average of approximately 1,500 solutions across 30 different seeds (Figure 6.12). The generation of many sequences will pose an *a posteriori* challenge, which is the selection of only one sequence per combination when one has many to choose from. D-Tailor includes one utility called *ComputeMinimalSet* that precisely addresses this problem. This tool encodes a Monte-Carlo method to select exactly one sequence for each desired combination, such that the total hamming distance between all the sequences is minimized (Figure 6.20).

```
######
# DB2CSV
macbook:D-Tailor jcg$ PYTHONPATH=. python Utils/DB2CSV.py testFiles/outputFiles/tfec_1.sqlite
Generating CSV files for testFiles/outputFiles/tfec_1.sqlite ... Done!

macbook:D-Tailor jcg$ head -n 3 testFiles/outputFiles/tfec_1.sqlite.generated_solutions.csv
generated_solution_id,des_solution_id,sequence,sd16sRNADuplexMFE,utrStructureMFE,cdsCAI,sd16sRNADuplex
MFELevel,utrStructureMFELevel,cdsCAILevel,sd16sRNADuplexMFEPosition,utrStructureMFEPosition,cdsCAIPosi
tion,worker_id
100470516384773921758647475759448978081,1.3.2,ttattaccggacaataatatttcaattcattaaagaggagaaaggtaccatggccc
tgtggatgcgcctcctgcccctgctggcgctgctggccctctggggacctgacccagccgcagcctttgtgaaccaacacctgtgcggctcacacctggtgg
aagctctctacctagtgtgcggggaacgaggcttcttctacacacccaagacccgccgggaggcagaggacctgcaggtggggcaggtggagctgggcgggg
gccctggtgcaggcagcctgcagcccttggccctggaggggtccctgcagaagcgtggcattgtggaacaatgctgtaccagcatctgctccctctaccagc
tggagaactactgcaactag,-8.4,-9.9,0.32,1,3,2,0.59,-0.94, 0.72, 249172630681921635831887521585739395265
285501686618022385962284569925274555241,1.2.2,ttattaccggacaataatatttcaattcattaaagaggagaaaggtaccatggccc
tgtggatgcgcctcttacccctgctggcgctgctggccctctggggacctgacccagccgcagcctttgtgaaccaacacctgtgcggctcacacctggtgg
aagctctctacctagtgtgcggggaacgaggcttcttctacacacccaagacccgccgggaggcagaggacctgcaggtggggcaggtggagctgggcgggg
gccctggtgcaggcagcctgcagcccttggccctggaggggtccctgcagaagcgtggcattgtggaacaatgctgtaccagcatctgctccctctaccagc
tggagaactactgcaactag,-8.4,-10.8,0.32,1,2,2,0.59,0.24,0.48, 249172630681921635831887521585739395265

macbook:D-Tailor jcg$ head -n 3 testFiles/outputFiles/tfec_1.sqlite.design_list.csv
des_solution_id,sd16sRNADuplexMFELevel,utrStructureMFELevel,cdsCAILevel,status,worker_id,start_time
1.1.1,1,1,1,DONE,249172630681921635831887521585739395265,None
1.1.3,1,1,3,DONE,249172630681921635831887521585739395265,None

######
# DB2FASTA
macbook:D-Tailor jcg$ PYTHONPATH=. python Utils/DB2FASTA.py testFiles/outputFiles/tfec_1.sqlite
Generating FASTA file(s) for testFiles/outputFiles/tfec_1.sqlite ... Done!

macbook:D-Tailor jcg$ head -n 4 testFiles/outputFiles/tfec_1.sqlite.generated_solutions.fa
>100470516384773921758647475759448978081 | 1.3.2
TTATTACCGGACAATAATATTTCAATTCATTAAAGAGGAGAAAGGTACCATGGCCCTGTGGATGCGCCTCCTGCCCCTGCTGGCGCTGCTGGCCCTCTGGGG
ACCTGACCCAGCCGCAGCCTTTGTGAACCAACACCTGTGCGGCTCACACCTGGTGGAAGCTCTCTACCTAGTGTGCGGGGAACGAGGCTTCTTCTACACACC
CAAGACCCGCCGGGAGGCAGAGGACCTGCAGGTGGGGCAGGTGGAGCTGGGCGGGGGCCCTGGTGCAGGCAGCCTGCAGCCCTTGGCCCTGGAGGGGTCCCT
GCAGAAGCGTGGCATTGTGGAACAATGCTGTACCAGCATCTGCTCCCTCTACCAGCTGGAGAACTACTGCAACTAG
>285501686618022385962284569925274555241 | 1.2.2
TTATTACCGGACAATAATATTTCAATTCATTAAAGAGGAGAAAGGTACCATGGCCCTGTGGATGCGCCTCTTACCCCTGCTGGCGCTGCTGGCCCTCTGGGG
ACCTGACCCAGCCGCAGCCTTTGTGAACCAACACCTGTGCGGCTCACACCTGGTGGAAGCTCTCTACCTAGTGTGCGGGGAACGAGGCTTCTTCTACACACC
CAAGACCCGCCGGGAGGCAGAGGACCTGCAGGTGGGGCAGGTGGAGCTGGGCGGGGGCCCTGGTGCAGGCAGCCTGCAGCCCTTGGCCCTGGAGGGGTCCCT
GCAGAAGCGTGGCATTGTGGAACAATGCTGTACCAGCATCTGCTCCCTCTACCAGCTGGAGAACTACTGCAACTAG

######
# DBKinetics
macbook:D-Tailor jcg$ PYTHONPATH=. python Utils/DBKinetics.py testFiles/outputFiles/tfec_1.sqlite
Generated Solutions       Desired Solutions Found
0       0
148     14
296     23
444     32
…
7400    124
7418    125

######
# DBStatistics

macbook:D-Tailor jcg$ PYTHONPATH=. python Utils/DBStatistics.py testFiles/outputFiles/tfec_1.sqlite
testFiles/outputFiles/tfec_1.sqlite       7418     125      0.0168509032084
```

Figure 6.19 D-Tailor utilities

```
macbook:D-Tailor jcg$ PYTHONPATH=. python Utils/ComputeMinimalSet.py
./testFiles/outputFiles/tfec_2.sqlite.generated_solutions.csv
stop random
1 (1):   550742   -1273
3 (2):   549469   -156
5 (2):   549313   -524
6 (1):   548789   -92
…
################## Summary ##################
number of combinations: 125
average distance nt: 38.21 +/- 20.25
##############################################

macbook:D-Tailor jcg$ ls testFiles/outputFiles/
tfec_2.sqlite
tfec_2.sqlite.design_list.csv
tfec_2.sqlite.generated_solutions.csv
tfec_2.sqlite.generated_solutions.pkl0
tfec_2.sqlite.generated_solutions.pkl1
tfec_2.sqlite.generated_solutions_min_set.fas
tfec_2.sqlite.generated_solutions_min_set_distance_matrix_nt.csv
tfec_2.sqlite.generated_solutions_min_set_feats.csv
```

Figure 6.20 Computing the minimal set

The tool receives a CSV file with all the sequences generated by D-Tailor (obtained using the DB2CSV script) and will generate the following files, where X is the name of the CSV file:

- X.generated_solutions_min_set_feats.csv—CSV file with the final set of sequences selected;
- X .generated_solutions_min_set.fas—FASTA file with final set of sequences selected;
- X.generated_solutions_min_set_distance_matrix_nt.csv—file containing the nucleotide distance matrix between all the selected features.