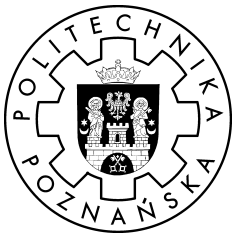
Poznań University of Technology
Faculty of Computing Science and Management
Institute of Computing Science



Wojciech Mruczkiewicz

Hyper-heuristics for Sequencing by Hybridisation Problem

Master thesis

Supervisor: Prof. dr hab. inż. Jacek Błażewicz

Poznań 2009

I would like to thank Professor Jacek Błażewicz, Professor Edmund Burke and Professor Graham Kendall for their support, fruitful discussions and my staying at the University of Nottingham.
I would also like to thank Ph.D. Aleksandra Świercz for her careful supervise, valuable remarks, all the time dedicated and Hanna Ćwiek for her editorial remarks.

Contents

Chapter 1
Introduction

Biologically inspired problems are becoming more and more common in the field of computing science. Bioinformatics has emerged over the years on the crossroad of molecular biology and information technology tools. A large part of research in this field is devoted to the analysis of DNA sequences. One of the problems inspired by analysing sequences of nucleotides in DNA strands is a sequencing by hybridisation problem (SBH). Although the SBH problem has been tackled many times with a lot of success this work gives a unique insight from the perspective of a rising direction in optimisation techniques — hyper-heuristic methods.

Hyper-heuristics are search methods that abstract from a given problem and manage a set of heuristics in order to solve the considered problem [27]. Similarly to metaheuristic techniques hyper-heuristics aim at solving hard computational problems in reasonable time with sub-optimal results. The main reason for appearance of hyper-heuristic methodology is to generalise and simplify the process of finding the solution. Many algorithms originating from metaheuristic methodology were adopted to the hyper-heuristic methods. Simulated annealing based approach can be found in [4, 5] and tabu search based methods in [16, 30]. The choice function based methodology [18, 20] is native for hyper-heuristic technique and is competitive to the other approaches [33]. However, this generalised architecture seems to impose a number of constraints on the design of optimisation techniques. This work shows how those constraints affect the process of designing algorithms that solve sequencing by hybridisation problem.

Sequencing by hybridisation method for reconstructing DNA fragments was described in [36] and [21] in the late 1980s. The method is based on hybridisation experiment in which a set of fragments of the examined DNA strand are determined. The DNA sequence is reconstructed by merging the matching fragments together. In order to find the original sequence the use of computational methods is required. Errors that can occur during the hybridisation process introduce ambiguities in the computational part of the experiment and the sequencing problem becomes difficult. It was proven to be strongly **NP**-hard in [13]. The problem is well-studied and many exact algorithms [7, 41] or heuristic algorithms [8, 14, 9, 10, 15] were proposed. The hyper-heuristic approach for solving SBH problem is analysed.

The objective of this work was twofold. First of all a hyper-heuristic approach to the SBH problem was to be applied. In order to solve the SBH problem with the use

of a general hyper-heuristic framework a number of software tools have been created. Design and development of the framework is based on the current research in the field of optimisation techniques. The second goal of this thesis was to check the behaviour and performance of the hyper-heuristic methods. Hyper-heuristics allow to change the set of moves that solver is capable of performing quickly and with a great flexibility. Sensitivity of modifications of this kind is examined in this work. In an ideal case hyper-heuristic method should work almost equally good with every reasonable set of allowed moves. The goal of hyper-heuristic algorithms is to learn and predict which moves are going to improve the solution and which are not. Thus, the aim of this work is to establish a verifiable use case for hyper-heuristic algorithms based on the solution to the SBH problem.

This thesis consists of five chapters and three appendixes. In Chapter 2 the sequencing by hybridisation problem is formulated. Sections 2.1 and 2.2 summarise the knowledge required to define the SBH problem formulated in Section 2.3. The remaining part of this chapter is devoted to the theoretical analysis of the SBH problem, i.e., a number of results collected from literature together with a new insight for the SBH problem are given. Chapter 3 concerns hyper-heuristic methods. After presenting the general concept of hyper-heuristic search in Section 3.1 the description of choice function based, tabu search based and simulated annealing based hyper-heuristics follows. Chapter 4 is concerned with various low-level heuristics that solve the SBH problem. Preliminary solution is presented in Section 4.1 and constitute a basis for more sophisticated and problem specific low-level heuristics which are the subject of Sections 4.2, 4.3 and 4.4. Implementation of the low-level heuristics is described in Section 4.5. Chapter 5 summarises experimental results of the hyper-heuristic methods for solving the SBH problem. Data sets used in tests are described in Sections 5.1 and 5.2. Experiment results are described in Sections 5.4 and 5.5. Appendix A describes a hyper-heuristics framework that was designed and developed in this research. Appendix B is a description of the software tools that were developed in order to solve the SBH problem. Appendix C lists the numerical results obtained from solving the data sets for the SBH problem.

Chapter 2

Sequencing Problem

Since the discovery of DNA double helix by James D. Watson and Francis Crick in 1953 scientists have been trying to learn DNA sequences of the huge variety of life forms known to humankind. This process is called genome sequencing. Genome sequencing consists of three main parts — DNA sequencing, sequence assembly and gene mapping. Those stages are distinguished by different biological experiments performed on DNA fragments of different scale. Various approaches to DNA sequencing have appeared up to date [23]. The problem analysed here is a fragment of one of them called standard sequencing by hybridization using DNA microarrays [36, 37].

This chapter introduces the DNA sequencing. Section 2.1 describes the biological issues related to the DNA sequencing. Section 2.2 contains a brief description of the concepts required to formulate the SBH problem which is done in Section 2.3. Section 2.4 is the analysis of a simpler variant of the SBH problem. Besides the summary of current knowledge it contains also new observations related to this problem. Section 2.5 is dedicated to study the SBH problem and contains summary of results collected from literature. The last Section 2.6 describes the algorithms and methods for solving the SBH problem found in literature.

2.1 DNA Sequencing

Deoxyribonucleic acid (DNA) is a compound chemical molecule that consists of two long polymers twisted together in a double helix. A single polymer is called a DNA strand and is a sequence of small units called nucleotides. Each nucleotide consists of three parts — a sugar, a phosphoric acid and a nitrogenous base. Nucleotides in one chain are joined together by asymmetric phosphodiester bonds acting between a sugar and a phosphoric acid of consecutive nucleotides. The sugar is a $2'$-deoxyribose i.e., a pentose that includes five carbon atoms labelled from $1'$ to $5'$. The phosphoric acid of a particular nucleotide is attached to the $3'$ carbon atom of the nucleotide's $2'$-deoxyribose. The same phosphoric acid attaches then to the $5'$ carbon atom of the $2'$-deoxyribose from the next nucleotide. This forms a phosphodiester bond between two nucleotides. The sequence of sugar and phosphoric acid forms the backbone of a DNA strand. The asymmetric bonds induces the direction of a DNA strand. In a double helix two strands have the opposite directions. Each strand can be read from $3'$ carbon atom (with a terminal hydroxyl group) to $5'$

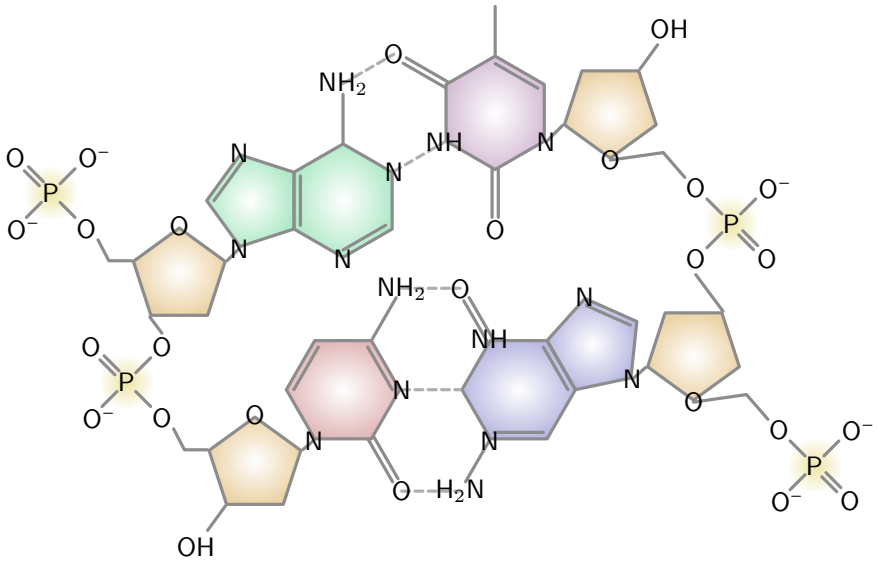


Fig. 2.1: Four different nucleotides forming two base pairs. Two DNA strands are connected by hydrogen bonds between adenine (green) and thymine (violet) or cytosine (red) and guanine (blue). The sugar 2′–deoxyribose (orange) and phosphoric acid (yellow) form a backbone of a DNA strand.

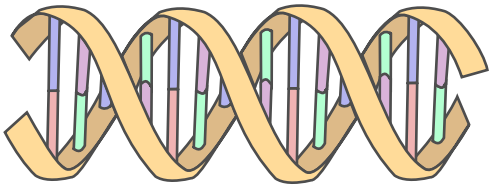


Fig. 2.2: The DNA forms a double helix. Colours correspond to the chemical compounds from Figure 2.1.

carbon atom (with a terminal phosphate group).

One of four possible nitrogenous bases is attached to $1'$ carbon atom of $2'$-deoxyribose. Those four nitrogenous bases are different chemical compounds and are called adenine (A), cytosine (C), guanine (G) and thymine (T). Adenine and guanine have a similar structure and are derivatives of purine. Cytosine and thymine are derivatives of pyrimidine and also mark a similar structure. The different nitrogenous bases introduce four different nucleotides depicted by letters A, C, G and T respectively. On Figure 2.1 the chemical structure of four nucleotides is presented.

Two DNA strands are connected together by hydrogen bonds of nitrogenous bases. Each of these bonds connects two nucleotides from different strands forming a base pair. Nucleotides from different strands are connect together according to the complementarity rule, A bonds only with T and C bonds only with G. A DNA sequence is a linear molecule i.e., it is a sequence of consecutive nucleotides that does not have any branches. It forms a double helix (Figure 2.2). The sequence of base pairs encodes genetic information that contains definition of life. A single DNA molecule is called a chromosome and can be composed of up to a few hundred million base pairs. More detailed description of DNA chemical structure can be found in [40, 28, 6].

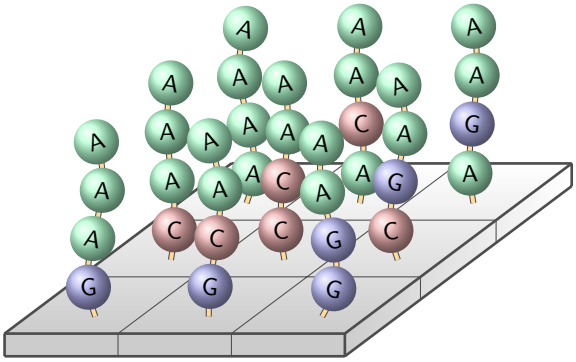


Fig. 2.3: DNA microarray is a square chip divided to large number of cells. Each cell contains an oligonucleotide that hybridizes with complementary DNA strand.

The standard sequencing by hybridization method [36, 21] is composed of two parts. A biochemical one is a hybridization experiment — the process of recognising the fragments that are contained in the examined sequence. A computational part is the process of merging the fragments together in order to reconstruct the sequence. At the beginning of the biochemical experiment a large number of copies of examined DNA strand is prepared and labelled radioactively or fluorescently in order to be recognized later. The solution with DNA strands is introduced then on a special DNA chip. A DNA chip is a square plate made of glass or silicon composed of many small cells. Each cell contains short sequences of nucleotides called oligonucleotides. In standard sequencing by hybridization every oligonucleotide is of length k. There are exactly 4^k cells on DNA chips i.e., every possible nucleotide sequence of length k is represented by a cell on the DNA chip. This is shown on Figure 2.3. If a fragment of the examined DNA strand is complementary to an oligonucleotide in a DNA cell then it hybridizes and its presence can be recognized. As a result an image of the DNA chip is obtained with cells of different intensities. The cells of large intensity inform that the oligonucleotides complementary to the ones contained in the given cells are a part of the examined DNA strand. For computational purposes the length n of the examined DNA strand is determined in the additional experiment — polyacrylamide gel electrophoresis.

As a result a set of oligonucleotides called spectrum is obtained from a hybridization experiment. All oligonucleotides from the spectrum have equal length k and are fragments (i.e., subsequences of the consecutive nucleotides) of the examined, longer sequence. The reconstruction of the original sequence from the oligonucleotides contained in the spectrum requires a combinatorial approach and is a computational part of the experiment. If the hybridization experiment is an ideal one then there is a one-to-one mapping between oligonucleotides in the spectrum and every fragment of length k in the original sequence i.e., each oligonucleotide in the spectrum is exactly one fragment of the original sequence and every possible fragment of length k in the original sequence has a representation in the spectrum. The spectrum size in that case is equal to $n - k + 1$.

However, the experiment is biased by two types of errors — positive and negative ones. A positive error appears when an oligonucleotide from the spectrum is not a fragment of the original sequence. A negative error appears when a fragment of the original sequence (of length k) does not have its representation in the spectrum. Two main reasons for

the occurrence of positive errors are when some not quite complementary oligonucleotide hybridizes partially with the DNA sequence and because sometimes the image of the chip is noisy. In the first case nucleotide sequences very similar to the ones in an ideal experiment are likely to occur (relatively small Hamming distance). In the second case completely random oligonucleotides can appear (relatively large Hamming distance). As for negative errors there are also two main causes. Firstly, they can appear due to unfavourable physical conditions e.g., wrong temperature or salt concentration. The hybridization process should be intensive but it is not. Another cause is that some oligonucleotides appear in the examined sequence more than once. Since the spectrum is a not a multiset they can be present only once and the information about number of their appearances is lost. Such errors are also called repetitions.

The length k of oligonucleotides used in hybridisation experiment is usually not larger than 10 because there are $4^{10} = 1048576$ different oligonucleotides needed to completely examine DNA strand. The largest DNA chips contain over 2 millions cells (NimbleGen ChIP-chip microarray). In a sequencing by hybridisation process relatively small DNA fragments are obtained with the size of at most a few thousand nucleotides. This is a much shorter sequence than the entire chromosome and additional stages of assembly and mapping are performed. Usually during the sequence assembly stage entire genes are reconstructed. Sequence assembly is a computational problem very similar to sequencing. The goal of a gene mapping is to determine the correct place of gene in the chromosome. Mapping is performed with the strong support of biological experiments. Sequence assembly and gene mapping are not of concern in this work.

2.2 Preliminaries

The sequencing problem refers to various mathematical objects and the notation used in following chapters is defined here. The notion of string is used very often. A *string* s over an alphabet Σ is a finite sequence of symbols from this alphabet. Empty string is denoted as ϵ. The string s can be written as a sequence of symbols $s = a_1 a_2 \dots a_n$ with $a_i \in \Sigma$, $1 \leq i \leq n$. The length of the string s is denoted as $|s|$ and in this case $|s| = n$. The notation $a_{i\dots j}$ denotes a substring composed of symbols a_i to a_j of string s, $a_{i\dots j} = a_i a_{i+1} \dots a_j$. Thus s can be also presented as $s = a_{1\dots n}$. If $i > j$ then $a_{i\dots j} = \epsilon$. Concatenation of strings u and v is denoted by $u\,v$. If $s = u\,v\,w$, where u and w are arbitrary strings, then string v is called a *substring* of s and written as $v \preceq s$. If $v \preceq s$ and $v \neq s$ then v is a *proper substring* of s, $v \prec s$. When v is a substring placed at the beginning of string s, $s = v\,w$ for arbitrary w, then v is a *prefix* of s. When v is a substring placed at the end of s, $s = w\,v$ for arbitrary w, then v is a *suffix* of s. Two strings are overlapping if a prefix of one of the strings is equal to a suffix of the other string. A predicate $\text{Overlaps}(u, v, k)$ is defined below. It is true if and only if last k symbols of string u are equal to the first k symbols of string v.

Definition 2.1. For strings u, v and integer $0 \leq k \wedge k \leq |u| \wedge k \leq |v|$ a predicate Overlaps is defined as:

$$\text{Overlaps}(u, v, k) \equiv u_{|u|-k+1\dots|u|} = v_{1\dots k}\,.$$

From the above definition the distance between strings is introduced. Even though u

and v can overlap on more than one position the distance between them is defined as a smallest possible shift such that u and v overlap.

Definition 2.2. For strings u and v the distance between them is given by function:

$$\text{distance}(u,v) \equiv \min\{d : m \leq d \leq |u| \wedge \text{Overlaps}(u,v,|u|-d)\}\ ,$$

where $m = \max\{1, |u| - |v| + 1\}$.

The concept of multiset i.e., a set with elements occurring more than once is also used. Summary of notations existing in literature is presented in [38]. If A is a set then *multiset* $\mathcal{A}$ over A is a pair $\mathcal{A} = (A, f)$ where function $f : A \to \mathbb{N}$ defines the number of occurrences of every element in A. A is called an underlying set of elements of multiset $\mathcal{A}$. A *support* S of multiset $\mathcal{A}$ is a set $S \subseteq A$ that contains every $x \in A$ such that $f(x) > 0$. If B is a set then intersection with multiset $\mathcal{A}$, $B \cap \mathcal{A}$ is a set $B \cap S$ where S is a support of $\mathcal{A}$. The list of elements of multiset is enclosed in square brackets instead of braces.

An introduction to the graph theory is presented in [26]. A *graph* G is a set of vertices V together with a set of edges E, $G = (V, E)$. Edges are pairs of vertices and represents the connection between two vertices from V. If directions of the edges are important then G is called a *directed graph* and a set of directed edges is called a set of *arcs* A. A *path* in a directed graph from vertex v_1 to v_i is a sequence $P = v_1, e_1, v_2, e_2, \ldots, e_{i-1}, v_i$ of alternating vertices from V and arcs from A such that $e_j = (v_j, v_{j+1})$ for $1 \leq j < i$. If every vertex appears exactly once in a path then the sequence is called a *simple path*. If $v_1 = v_i$ then P is a *cycle* in G. An *Eulerian path* is a path in G that visits each edge in A exactly once. If this path is a cycle then it is called an *Eulerian cycle*. The problem of finding Eulerian path in a directed graph G is called Eulerian Path, [26, chap. 6]. The path that visits each vertex in V exactly once is called a *Hamiltonian path* and respectively a *Hamiltonian cycle* if this path is also a cycle. The problem of finding the Hamiltonian path in G is called Hamiltonian Path and is computationally harder than Eulerian Path. An *adjoint* $G' = (V, X)$ of a graph $G = (U, V)$ is the graph with vertex set V and such that there is an arc from a vertex $x \in V$ to vertex $y \in V$ in G' if and only if the terminal endpoint of the arc x in G is the initial endpoint of arc y in G [12]. If graph does not contain multiple arcs between any two vertices then it is a 1-graph. A *directed line graph* is an adjoint of the 1-graph. A subgraph $G' = (V, S)$ of a directed graph $G = (V, A)$ is called *rooted spanning tree* if it connects every vertex in V without any cycle with exactly $n - 1$ arcs and every vertex in G' except the root has exactly one incoming arc.

An undirected graph is *connected* if for every pair of vertices in this graph there exists a path. Paths and cycles in undirected graphs are defined analogically to paths and cycles in directeds graph. If graph does not contain any cycle then it is *acyclic*. A *tree* is a connected and acyclic undirected graph. One of the vertices in a tree can be distinguished and called a *root* of the tree. Vertices of a rooted tree are called *nodes*. There exists exactly one simple path between every node in tree and the root node. For a node x a vertex y that is immediately after x on the path from x to the root node is called a *parent* of x, and x is a *child* of y. Vertices without children are called *leaves* or *external* nodes. Non-leaf nodes are called *internal* nodes.

It takes different amount of time for various algorithms to finish. The asymptotic running time of algorithms is analysed with use of $O(g(n))$, $\Theta(g(n))$ and $\Omega(g(n))$ notations.

A function $g(n)$ is any function of n and quoting the definition from [17, chap. 3]:

$$\Theta(g(n)) = \{f(n) : \exists c_1, c_2, n_0 > 0,\ \forall n \geq n_0,\ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}\ .$$

That is, $g(n)$ is an asymptotically tight bound for $f(n)$. $\Theta(g(n))$ is a set of functions that are bounded asymptotically tight between $c_1 g(n)$ and $c_2 g(n)$. Function $O(g(n))$ is an asymptotic upper bound for $f(n)$ and limits the $f(n)$ only from above:

$$O(g(n)) = \{f(n) : \exists c, n_0 > 0,\ \forall n \geq n_0,\ 0 \leq f(n) \leq cg(n)\}\ .$$

The set $\Omega(g(n))$ provides an asymptotic lower bound and is defined analogically to $O(g(n))$. As described in [17] set $O(g(n))$ is used to determine the worst case running time of the algorithms. Many times complexity results presented in this work are given by asymptotically tight approximation $\Theta(g(n))$ which is independent of particular input. Although $O(g(n))$ and $\Theta(g(n))$ are sets the notation is slightly abused and refers often to a function from those sets rather than the set of functions.

A number of terms from the computational complexity theory are used. Computational problems are generally divided to two categories [17, chap. 34]. The decision problems are problems for which only possible answer is *yes* or *no*. The optimisation problems are problems which distinguish the best solution among the set of all possible solutions. The goal of solving an optimisation problem is to find a solution that minimise or maximise the associated objective function. Decision problems are important for analysis because they can be easily solved by Turing machines. Turing machine is a mathematical object composed of a tape with a head, a set of states and a transition function that given symbol pointed by the current head position and the current state gives a symbol that should be written at the current head position, the direction where the head should be moved and the next state where Turing machine should put into. Transition function models algorithm that solves the given problem. Problem instance in order to be solved is written on the tape. The terminal state where Turing machine finishes gives an answer to the decision problem. If decision problem can be solved on Turing machine in a polynomial number of steps (polynomial time) then it is said that this problem belongs to class **P**. Non-deterministic Turing machine is a Turing machine with the transition function substituted by a relation i.e., for a given state and a symbol on the tape pointed be a head many possible transitions are possible. Problems solved by non-deterministic Turing machines in a polynomial time belong to class **NP**. In other words the class **NP** is a class of problems that are verifiable in polynomial time on a Turing machine. For details and definitions of Turing machines and complexity classes see also [34].

An important concept is a reducibility. Given two problems Π_1 and Π_2 if there exists a polynomial-time computable function that transforms instances from Π_1 to Π_2 and for every associated instance of Π_2 the decision is the same as for instance of Π_1 then this function is a polynomial-time reduction from Π_1 to Π_2. Thus if Π_1 belongs to class **P** or **NP** and a polynomial-time reduction from Π_1 to Π_2 exists then solving Π_2 implies solving Π_1 and Π_2 belongs to the same complexity class as Π_1. In this work every reduction is polynomial-time many to one reduction i.e., reduction in the Karp sense [34, chap. 8] or polynomial transformation. Problem Π belongs to **NP**-complete class if it belongs to **NP** and every other problem Π' from **NP** is polynomial-time reducible to Π. The

problem Π in the definition of **NP**-completeness must belong to the **NP** class and thus must be a decision problem. The class of **NP**-hard problems is introduced that takes into account the optimisation problems. If there exist a polynomial transformation from any **NP**-complete problem to the problem examined then this problem belongs to **NP**-hard complexity class. If for **NP**-complete or **NP**-hard problem the presence in those classes is not dependent on the numerical parameters (problem remains **NP**-complete or **NP**-hard even when numerical parameters are bounded by a polynomial in an instance size) then the problem is said to be a strongly **NP**-complete or a strongly **NP**-hard respectively [25, chap. 4].

Throughout this work the complexity classes of computational problems are written in bold e.g., **NP**-complete. Problem names are written in capital letters like Hamiltonian Path. Referenced problems are defined when needed. Most of the issues faced are of discrete nature so every number is assumed to be an integer unless stated otherwise. Presented algorithms are written in pseudo code using a style similar to [17]. Procedures and functions defined in algorithmic context are written in capital letters e.g., TabuSearch. Functions that are used as mathematical entities are defined outside the pseudo code and are written in small roman letters e.g., $\text{used}(\mathcal{S}_k, s)$. Predicates always start with a capital letter e.g., $\text{Overlaps}(u, v, k)$. Arguments of procedures can be assigned in the procedure body — in such a case the value assigned is expected to be seen outside i.e., in the procedure that called it. This allows passing values by a reference. It is clear from context which arguments are expected to aggregate some output and which not.

2.3 Problems Definitions

The definitions used here are based on the work [13] where detailed analysis of different variations of DNA sequencing by hybridisation problem is performed. Through this work the DNA sequencing by hybridisation problem is called also sequencing problem for short. The problem analysed is a standard sequencing problem with positive and negative errors (definition 2.6) but other variants of the problem are defined for the comparison and the analysis of problem complexity. The goal of this section is to define the computational part of this problem in an abstract, mathematical way.

The DNA sequence is a chain of nucleotides A, C, T or G. The nucleotides define alphabet $\Sigma = \{\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{T}\}$ and DNA sequence will be represented as a string over this alphabet. This string is depicted simply as a sequence of letters e.g., ACGTAT. The terms sequence or oligonucleotide will be used interchangeably with the term string depending on the context.

An ideal sequencing by hybridisation experiment without negative or positive errors results in a set of oligonucleotides. Each oligonucleotide is a unique fragment of the analysed sequence s and has a length k. This set is called an ideal spectrum and can be defined as follows.

Definition 2.3. The *ideal spectrum* $\mathcal{S}_k^*(s)$ of string $s = a_{1\ldots n}$, $1 \leq k \leq n$ is a set

$$\mathcal{S}_k^*(s) = \{a_{i\ldots i+k-1} : 1 \leq i \leq n-k+1\}$$

of size $n_s = n - k + 1$.

Now the sequencing problem without positive and negative errors can be formulated. The goal is to reconstruct the sequence s of known length n analysed in the hybridisation experiment.

Definition 2.4. The *standard sequencing without errors* problem (FREE SBH) is: given ideal spectrum $\mathcal{S}_k^*(s)$ find the sequence s of length $n = n_s + k - 1$ that contains all the strings from S where $n_s = |\mathcal{S}_k^*(s)|$.

As a result of this problem the sequence which the ideal spectrum is equal to this spectrum should be reconstructed. There can be many such solutions and this problem cannot be always solved uniquely. However, every such solution always leads to an ideal spectrum $\mathcal{S}_k^*(s)$ in an ideal hybridisation experiment. The condition $n_s = n - k + 1$ in definition 2.3 is important because it disallows repetitions. If a substring of length k appears in the sequence s more than once i.e., when $a_{i\ldots i+k-1} = a_{j\ldots j+k-1}$ for $i \neq j$, then $\mathcal{S}_k^*(s)$ cannot exist because its size is lower than $n-k+1$. This in fact makes FREE SBH less complex and allows for less sophisticated methods to solve it, see Section 2.4.

FREE SBH is designed to be used for an ideal but unrealistic experimental result. The ideal spectrum holds a strong connection with analysed sequence s. This is not a case in the real DNA sequencing by hybridisation experiment results. They contain errors and the resulting set is less constrained. Data obtained from the sequencing by hybridisation experiment is described by the notion of spectrum:

Definition 2.5. The *spectrum* $\mathcal{S}_k$, where k is a positive integer, is a set $\{s_1, s_2, \ldots, s_{n_s}\}$ of strings over alphabet Σ where $|s_i| = k$ for $1 \leq i \leq n_s$.

The family of all possible subsets of $\mathcal{S}_k$ is denoted as $2^{\mathcal{S}_k}$. The length k of the strings is also the length of oligonucleotides used in the DNA chip. The strings from the spectrum match directly those oligonucleotides i.e., they represent the experimental result. The string s does not appear explicitly in the above definition. This is because the spectrum obtained from the experiment contains positive and negative errors. Negative errors imply that some substrings of string s are not included in the spectrum. Positive errors imply that the spectrum contains strings that are not substrings of the string s. Thus theoretically it is possible to obtain an empty spectrum or spectrum with every string that is not a substring of examined sequence s i.e., taking into account every possible experiment result no relation can be made.

However, in order to reconstruct the sequence the assumption that most of the strings from the spectrum are the substrings of the examined sequence s is made. The DNA sequencing by hybridisation with positive and negative errors is defined as follows.

Definition 2.6. The *standard sequencing with positive and negative errors* problem (SBH) is: given spectrum $\mathcal{S}_k$ and integer $n > 0$ find the string s over Σ, $|s| \leq n$ that maximizes the number of strings from $\mathcal{S}_k$ that are the substrings of s.

SBH problem is a relaxed version of the errors free standard sequencing problem. It describes the real sequencing by hybridisation experiment more accurately. The spectrum given on input can contain both positive and negative errors. To reconstruct the sequence s not all of the oligonucleotides from spectrum have to be used. The optimisation criterion maximises the number of oligonucleotides used. It is designed to cope with the expected

experimental result. The formulated problem takes into account the experiment errors and assumes that they are only a fraction of the oligonucleotides from the spectrum and most of the oligonucleotides are correct.

When a solution to SBH problem is found then together with the spectrum $\mathcal{S}_k$ errors can be analysed more accurately. In order to do this a multispectrum i.e., an ideal spectrum that accounts for repetitions is defined.

Definition 2.7. The *multispectrum* $\mathcal{M}_k(s)$, $1 \leq k \leq n$ of string $s = a_{1\ldots n}$ is a multiset $\mathcal{M}_k(s) = (S, f)$ with

$$S = \{a_{i\ldots i+k} : 1 \leq i \leq n-k+1\}$$

and

$$f(u) = |\{i : 1 \leq i \leq n-k+1 \wedge u = a_{i\ldots i+k-1}\}|$$

for every $u \in S$.

Thus multispectrum is a multiset where each occurrence, $a_{i\ldots i+k-1}$ of a substring u in sequence s is remembered i.e., value of function $f(u)$ is equal to the number of occurrences of string u in s. Now more formal definition of positive and negative errors can be given.

Definition 2.8. Solution s for SBH problem with spectrum $\mathcal{S}_k$ contains a *positive error* if $\mathcal{S}_k \setminus S \neq \varnothing$ where S is a support of multispectrum $\mathcal{M}_k(s)$.

Definition 2.9. Solution s for SBH problem with spectrum $\mathcal{S}_k$ contains a *repetition* if there exists $m \in M$ such that $f(m) > 1$ for multispectrum $\mathcal{M}_k(s) = (M, f)$

Definition 2.10. Solution s for SBH problem with spectrum $\mathcal{S}_k$ contains a *negative error* if it contains a repetition or $S \setminus \mathcal{S}_k \neq \varnothing$ where S is a support of multispectrum $\mathcal{M}_k(s)$.

Positive errors are responsible for making the spectrum size larger and if the hybridisation experiment is free of positive errors then the maximal spectrum size is always lower or equal to $n-k+1$. The equality occurs only when every substring of length k of string s is represented in this spectrum uniquely. In such case the spectrum is an ideal spectrum and s is also free of negative errors.

The function used$(\mathcal{S}_k, s)$ is defined. It is used in algorithms throughout this work.

Definition 2.11. For spectrum $\mathcal{S}_k$ and string s a function $\text{used}(\mathcal{S}_k, s) \equiv |\mathcal{S}_k \cap \mathcal{M}_k(s)|$.

The value of used$(\mathcal{S}_k, s)$ gives a number of strings in $\mathcal{S}_k$ that are the substrings of s. In the following two sections the two problems defined above are studied in details. It is a summary of results found in literature together with a new insight.

2.4 Sequencing without Errors

The sequencing by hybridisation without errors problem (Definition 2.4) is a widely analysed theoretical problem. The polynomial time algorithm that solves this problem was presented in [35]. It is based on the transformation to the problem of finding Eulerian path in a directed graph $G = (V, A)$. The set of vertices V is fixed to be a set of prefixes and suffixes of length $k-1$ of every string in spectrum:

$$V = \bigcup_{a_{1\ldots k} \in \mathcal{S}_k(s)} \{v : v = a_{1\ldots k-1} \vee v = a_{2\ldots k}\}.$$

The arcs lead from a vertex represented by first $k-1$ symbols to the vertex represented by the last $k-1$ symbols of every string in spectrum:

$$A = \bigcup_{a_{1\ldots k} \in \mathcal{S}_k(s)} \{(v, w) : v = a_{1\ldots k-1} \wedge w = a_{2\ldots k}\}.$$

Any Eulerian path in such a graph represents a solution for FREE SBH and every possible solution has an associated Eulerian path. The first $k-1$ symbols from the first vertex on the Eulerian path are the first $k-1$ symbols of the solution. The last symbol of every remaining vertex on the Eulerian path contributes to the rest of the solution consecutively. The Eulerian path in the resulting directed graph can be found in $\Theta(|A|)$ time [26]. The graph G is referred to often as a Pevzner's graph. Pevzner's algorithm works for $k > 1$. For $k = 1$ any permutation of strings from an ideal spectrum can be concatenated to give the solution.

Theorem 2.1 (Pevzner [35])**.** FREE SBH *is solvable in polynomial time.*

Proof. If $k = 1$ then concatenate in any order strings from spectrum in $\Theta(n_s)$ time. For $k > 1$ use Pevzner's method as follows. When constructing the set V every possibly new vertex must be checked if it is not already used. Using a search tree every such a check requires $O(k)$ time. There are $\Theta(n_s)$ substrings to be checked so the total time for construction of set V is $O(k\,n_s)$. To construct the arc set every pair of vertices representing a substring of s must be identified. Again, using a search tree this can be done in total $O(k\,n_s)$ time. There are exactly n_s arcs and for such a graph EULERIAN PATH can be solved in $\Theta(n_s)$ time. It follows directly that building the solution takes $\Theta(k + n_s)$ time. Thus the total time required to solve the problem is $O(k\,n_s + k\,n_s + n_s + k + n_s) = O(k\,n_s)$ which is a polynomial in size of the input. □

Pevzner's graph does not contain any multiple edges because edges are directly represented by strings from a spectrum which is a set. The graph can have loops and they are present for arc (v, w) when $v = w$; $v, w \in V$ i.e., loops are generated from strings in spectrum that are composed of a single symbol: $a^k, a \in \Sigma$. There can be at most one loop attached to any vertex in V.

This problem always has a solution. The ideal spectrum set guarantees that. However, this does not imply that the solution is unique. There can be two different sequences s and s' such that $\mathcal{S}_k^*(s) = \mathcal{S}_k^*(s')$ i.e., there can by many different Eulerian paths in Pevzner's graph. For such instances from the theoretical point of view any solution is satisfactory. It is a greater problem for biologist. In the experiment the examined sequence is fixed and exactly this sequence is expected to be read. This ambiguity appears even in an ideal experiment and is a drawback of the sequencing by hybridisation method. However, it turns out that in case of the error free experiment increasing k just by one always leads to the only one solution s. To prove it the Lemma 2.2 is helpful. More details about number of solutions is stated in Theorem 2.3. Example 2.1 shows two graphs that have the properties mentioned in this theorem.

Lemma 2.2. *If for string s there exist an ideal spectrum $\mathcal{S}_k^*(s)$, $k > 1$ and the associated Pevzner's graph contains cycles or loops then there does not exist an ideal spectrum $\mathcal{S}_{k-1}^*(s)$.*

Proof. Every vertex v on the Eulerian path in G is a substring of length $k-1$ that appear at a different position in the reconstructed string s. Thus if a vertex v appears twice on the Eulerian path then the ideal spectrum $\mathcal{S}^*_{k-1}(s)$ does not exist because two appearances of substring v in s collapse to a single string of length $k-1$ and cannot be represented uniquely in an ideal spectrum.

If G contains a loop then a vertex v, to which the loop is attached, must follow itself somewhere on the Eulerian path. Thus v appears at least twice on the Eulerian path and $\mathcal{S}^*_{k-1}(s)$ does not exist.

On the Eulerian path every cycle in G must be traversed. From the definition of a cycle in a graph there exists a vertex v that is the beginning and the end of the cycle traversal. The vertex v appears twice in the Eulerian path and implies that $\mathcal{S}^*_{k-1}(s)$ does not exist. □

Theorem 2.3. *For every string s there exist such k that $\mathcal{S}^*_k(s)$ implies only one solution s for* Free SBH.

If there exist such k^* for which the ideal spectrum $\mathcal{S}^*_{k^*}(s)$ leads to many solutions of Free SBH *then for every $k < k^*$ an ideal spectrum $\mathcal{S}^*_k(s)$ does not exist and for every $k > k^*$ an ideal spectrum $\mathcal{S}^*_k(s)$ always leads to the only one solution s.*

Proof. The first part of the theorem is true for $k = n$. The set is composed then of a single string s which uniquely determines the solution.

In the second part of the theorem for k^* the Pevzner's graph must contain at least one cycle or a loop. Graph without any cycle or any loop forms a single path and the connected SBH problem has a single solution formed by this path. Thus the ideal spectrum $\mathcal{S}^*_{k^*}(s)$ must have a cycle or a loop in an associated Pevzner's graph. When $k^* > 1$ then $\mathcal{S}^*_{k^*-1}(s)$ does not exist from Lemma 2.2. It also follows for $1 \le k < k^* - 1$ because if sequence s contains two equal substrings at different positions of length $k^* - 1$ then also all the prefixes of length $k < k^* - 1$ of those substrings are equal and prevents from creating an ideal spectrum.

For $k > k^*$ if $\mathcal{S}^*_k(s)$ would not exist then there must have been some two substrings of length k in s that are equal. The prefixes of length k^* of those two substrings would also be equal and this implies that $\mathcal{S}^*_{k^*}(s)$ does not exist which is a contradiction. If $\mathcal{S}^*_k(s)$ would have many solutions then Lemma 2.2 and paragraph above would imply that $\mathcal{S}^*_{k^*}(s)$ would not exist which is also a contradiction. Thus for $k > k^*$ there must be only one solution equal to s. □

It is interesting to know if the number of solutions for $\mathcal{S}_k(s)$ can actually be counted. The number of different solutions is equal to the number of different Eulerian paths in a directed graph $G = (V, A)$. When every Eulerian path is not an Eulerian cycle then the starting vertex s and the ending vertex t are fixed. In order to calculate the number of different solutions in graphs with Eulerian paths a new graph $G' = (V' = V, A' = A \cup (t, s))$ is constructed from G with additional arc between vertices t and s. The number of non-equivalent Eulerian cycles $C_F(G')$ in graph $G' = (V', A')$ is given by the so-called BEST theorem [39]:

$$C_F(G') = M_T \prod_{v \in V'} (\deg^+(v) - 1)!$$

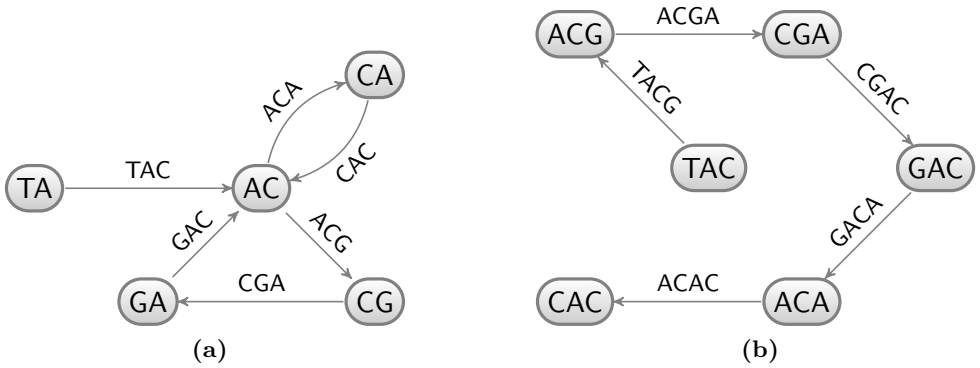


Fig. 2.4: Two Pevzner's graphs from example 2.1. The graph (a) is created from ideal spectrum $\mathcal{S}_3^*(s)$ and graph (b) from ideal spectrum $\mathcal{S}_4^*(s)$. Graph (b) does not contain any cycles and has an unique solution.

where $\deg^+(v)$ is the number of arcs leaving the vertex v. The M_T is the number of spanning trees in a graph and from Kirchhoff's matrix tree theorem it is equal to the absolute value of any cofactor of Laplacian matrix $L = (\ell_{i,j})_{|V'|\times|V'|}$ of directed graph G',

$$\ell(i,j) = \begin{cases} \deg^+(v_i) & \text{if } i = j, \\ -1 & \text{if } i \neq j \wedge (v_i, v_j) \in A', \\ 0 & \text{otherwise.} \end{cases}$$

Thus in this case $C_F(G')$ is equal to the number of Eulerian paths in G. When every Eulerian path in G is an Eulerian cycle then the number of different Eulerian paths is equal to $|V|C_F(G)$ because every vertex can be a starting point on every non-equivalent Eulerian cycle. This result however, is given for loop-less graphs and true if Pevzner's graph contains only cycles and no loops.

The Pevzner's solution was not the first for Free SBH. Lysov et al. [32] presented a transformation of Free SBH to the *Hamiltonian path* problem. The vertices of constructed graph G are strings from an ideal spectrum. The arcs are created between two vertices if the suffix of length $k-1$ of the first vertex is equal to the prefix of length $k-1$ of the second vertex in the arc. The solution of the sequencing by hybridisation problem without errors is any Hamiltonian path in graph G. However, finding a Hamiltonian path is a **NP**-hard problem and no polynomial time algorithm is known. It turns out that Lysov graphs can always be transformed to the Pevzner's graph. Such transformation is possible because Lysov's graphs belong to the class of *DNA graphs* [12] and are directed line graphs of Pevzner's graphs. The problems Hamiltonian Path and Eulerian Path for such a pair of graphs are equivalent and solvable in polynomial time.

Example 2.1. The sequence s = TACGACAC is of length $n = 8$. If $k = 3$ then an ideal spectrum is $\mathcal{S}_k^*(s) = \{\text{TAC}, \text{ACG}, \text{CGA}, \text{GAC}, \text{ACA}, \text{CAC}\}$ and has size $n_s = n - k + 1 = 6$. For this ideal spectrum the Pevzner's graph has five vertices v_1 = TA, v_2 = AC, v_3 = CG, v_4 = GA, v_5 = CA and contains two cycles (see Figure 2.4a). One of the Eulerian paths traverses vertices TA, AC, CG, GA, AC, CA, AC and the reconstructed path is TACGACAC. Theorem 2.3 dictates that $\mathcal{S}_2^*(s)$ does not exist and $\mathcal{S}_4^*(s)$ leads to one

solution. The set $\{\mathrm{TA}, \mathrm{AC}, \mathrm{CG}, \mathrm{GA}, \mathrm{CA}\}$ is of size 5 and cannot be an ideal spectrum $\mathcal{S}_2^*(s)$ of size $8 - 2 + 1 = 7$. Substring AC appears three times in s and is not represented uniquely. The ideal spectrum $\mathcal{S}_4^*(s) = \{\mathrm{TACG}, \mathrm{ACGA}, \mathrm{CGAC}, \mathrm{GACA}, \mathrm{ACAC}\}$ leads to the Pevzner's graph presented on Figure 2.4b with only one path and leading to the only one solution $s = \mathrm{TACGACAC}$. To count the number of Eulerian paths the BEST theorem is used. Because the Pevzner's graph from Figure 2.4a does not have an Eulerian cycle a new arc between the starting vertex v_1 and the ending vertex v_2 must be created. The Laplacian matrix L of Pevzner's graph for $\mathcal{S}_k^*(s)$ with additional arc (v_2, v_1) is

$$L = \begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ -1 & 3 & -1 & 0 & -1 \\ 0 & 0 & 1 & -1 & 0 \\ 0 & -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 0 & 1 \end{pmatrix}$$

and absolute value of every cofactor of this matrix is equal to 1. The number of Eulerian paths is $1 \cdot 1! \cdot 2! \cdot 0! \cdot 0! \cdot 0! = 2$. The other solution is TACACGAC.

2.5 Sequencing with Errors

However, the errors that can occur during the hybridisation experiment prevent from using the Pevzner method described in [35]. The sequencing by hybridisations with positive and negative errors problem is a more complex optimisation problem. The detailed analysis and proof of strong **NP**-hardness of this problem is given in Błażewicz and Kasprzak [13]. The proofs are included in this section in order to give more insight into SBH. The article defines two sub-problems of SBH — one for inputs that have only positive errors and one for inputs that only have negative errors. The decision version of those problems are formulated and proved to be strongly **NP**-complete separately. The slightly modified versions of decision problems are defined to be:

Definition 2.12. The *positive quasi-sequencing* problem (Positive SBH) is: given spectrum $\mathcal{S}_k$ and integer $n > 0$ is there a sequence s of length n that exactly $n - k + 1$ different strings from $\mathcal{S}_k$ are substrings of s?

Definition 2.13. The *negative quasi-sequencing* problem (Negative SBH) is: given spectrum $\mathcal{S}_k$ and integer $n > 0$ is there a sequence s of length n that every string from $\mathcal{S}_k$ is a substrings of s?

Positive quasi-sequencing problem is proved to be strongly **NP**-complete by polynomial transformation of the *directed Hamiltonian path between two vertices* problem [25] (a variant of the decision version of Hamiltonian Path in directed graphs) to the Positive SBH. The former is strongly **NP**-complete and so is the latter. Details of the transformation are explained in [13].

The complete proof of the strong **NP**-completeness of the negative quasi-sequencing problem is recalled here. To do this a *variant of the superstring* problem (Primitive Superstring) is used [24]: given set S' of primitive strings of length $k' \geq 3$ over an

unbounded alphabet Σ', does S' have a superstring of length n'? A *superstring* of a set of strings $S' = \{s_1, s_2, \ldots, s_{n_s}\}$ is a string s' containing each s_1, $1 \leq i \leq n_s$, as a substring. A string is *primitive* if no symbol appear more than once in it.

Lemma 2.4 (Błażewicz and Kasprzak [13])**.** Negative SBH *is strongly* **NP***-complete.*

Proof. The instance of Primitive Superstring is transformed in polynomial time to Negative SBH instance. Although unbounded, the alphabet Σ' in Primitive Superstring must have some length m. To encode every symbol in this alphabet $\lfloor \log_2 m + 1 \rfloor$ digits of symbols A and C from alphabet Σ are used i.e., symbols from Σ' are encoded in binary using symbols A and C. To separate each symbol from Primitive Superstring instance the third letter G is used. Thus every string from S' can be encoded using $k = k'(\lfloor \log_2 m + 1 \rfloor + 1)$ symbols from alphabet Σ. This is explained also in [24]. The set of such strings is a spectrum $\mathcal{S}_k$ and together with desired sequence of length $n = n'(\lfloor \log_2 m + 1 \rfloor + 1)$ forms an input for Negative SBH.

The answer for Negative SBH directly corresponds to the answer of Primitive Superstring. The separating digit G implies that every successive $\lfloor \log_2 m+1 \rfloor+1$ symbols from the sequence s corresponds to some symbol in Primitive Superstring and back. Thus sequence s represents a sequence s' of symbols from Σ'. Every string in $\mathcal{S}_k$ is a substring of s and corresponds to some substring of s' that is in S'. If an answer for Negative SBH is *yes* then also an answer for the Primitive Superstring is *yes*. If an answer for Negative SBH is *no* and s does not exist then also s' does not exist and answer for Primitive Superstring is also *no*. If s' would exists then it could be transformed directly to s which is a contradiction.

The number m is bounded by a polynomial of the number of symbols in an instance which is polynomial in instance size. Thus the transformation presented here is a polynomial transformation and Negative SBH is strongly **NP**-complete because Primitive Superstring also is [24] (actually the proof presented in [24] is for **NP**-completeness of this problem but relies on the polynomial transformation to the variant of Hamiltonian Path which is strongly **NP**-complete). □

Before the actual proof of strong **NP**-hardness of the sequencing by hybridisation with positive and negative errors problem the proof of the following variant of SBH is conducted.

Definition 2.14. The *general standard sequencing with positive and negative errors* problem (General SBH) is: given spectrum $\mathcal{S}_k$ and integers $0 < n_l \leq n_h$ find the string s over Σ, $n_l \leq |s| \leq n_h$ that maximizes the number of strings from $\mathcal{S}_k$ that are substrings of s.

The two sub-problems can be now used to prove the strongly **NP**-hardness of the general sequencing by hybridisation with positive and negative errors problem.

Lemma 2.5. General SBH *is strongly* **NP***-hard.*

Proof. With the above results this can be shown in two ways. An instance of Positive SBH is transformed to General SBH by using the same spectrum $\mathcal{S}_k$ and setting $n_l = n_h = n$. This transformation is trivially polynomial. If the solution for General SBH exist and $n - k + 1$ strings from $\mathcal{S}_k$ are used then answer for the original problem is *yes*.

If no solution exist or no $n - k + 1$ strings from $\mathcal{S}_k$ are used then answer is *no*. The $n - k + 1$ substrings is the maximum value of the search criterion and used in General SBH solution whenever possible. Thus solving General SBH solves Positive SBH i.e., General SBH is strongly **NP**-hard.

The same transformation is used for Negative SBH problem. In this case however, the answer is *yes* if every string from $\mathcal{S}_k$ is a substring of the solution of General SBH. The answer is *no* if not every string from $\mathcal{S}_k$ is used. Similarly, $|\mathcal{S}_k|$ is a maximum value for the search criterion and used in General SBH solution whenever possible. Solving General SBH solves Negative SBH and General SBH is strongly **NP**-hard. □

The proof for sequencing by hybridisation with positive and negative errors problem follows.

Theorem 2.6 (Błażewicz and Kasprzak [13])**.** SBH *is strongly* **NP***-hard.*

Proof. The polynomial transformation from the instance of General SBH to the instance of SBH can be constructed. The spectra are set equal and the parameter n in SBH is set equal to parameter n_h in General SBH, $n = n_h$. If the solution s to SBH has proper length i.e., $n_l \leq |s|$ then it is also the solution for General SBH. If the length is too short ($|s| < n_l$) then any $n_l - |s|$ symbols from Σ must be added to the end of s. This is also a correct result for General SBH. Thus solving SBH gives the answer for General SBH problem. The latter is strongly **NP**-hard and so is the former. □

The above theorem holds because of the presence of negative errors. However, they are not required for direct proof conducted by polynomial transformation from SBH to decision problems Negative SBH and Positive SBH.

As in sequencing by hybridisation without errors problem there can be many solutions to SBH. The theorem 2.3 cannot be extended directly to SBH because presence of errors makes the structure of spectrum $\mathcal{S}_k$ more general and incompatible with ideal spectra. The strong **NP**-hardness proofs rely on the transformation to the variant of Hamiltonian Path and presumably partially depends on number of solutions for this problem. However, this number of solutions is not preserved during the transformations because SBH is an optimization problem.

There always exist at least one, trivial solution — the empty string s of length 0. Thus the number of solutions is at least 1. The following lemma shows that number of solutions can be very large.

Lemma 2.7. *The number of solutions of* SBH *can be exponential in an instance size.*

Proof. The following instance can be prepared. The spectrum $\mathcal{S}_k$ is given by set of size n_s with strings of length $k = \lfloor \log_2 n_s + 1 \rfloor + 1$. Each string begins with a symbol G. The rest of the symbols are binary numbers $1, 2, \ldots, n_s$ encoded using alphabet $\{\mathrm{A}, \mathrm{C}\}$. The desired length n is set to $n = k\, n_s$. No two strings from this spectrum can overlap in the final sequence and optimal solutions uses every string in $\mathcal{S}_k$. Thus every optimal solution is a permutation of a strings in $\mathcal{S}_k$ i.e., the concatenation of every permutation of strings. Thus there are $n_s! = \Omega(2^{n_s}) = \Omega(2^{\frac{n}{k}})$ solutions which is exponential in problem size. □

In [13] SBH with a promise of solution uniqueness is considered i.e., the complexity of SBH for instances with a single solution. Promise problems [29] are problems with some predicate on the problem instances that always hold. For such problems this promise is always true and can be accounted for when designing the algorithms that solves given problem. The class **PROMISE-P** is a class of problems with a promise that are solvable in polynomial time. In [13] it is shown that the neither POSITIVE SBH nor NEGATIVE SBH with a promise of unique solution are in **PROMISE-P** unless **NP** = **RP**, where **RP** is a randomized polynomial time class. However, the conjecture **NP** ≠ **RP** is not as widely believed as **P** ≠ **NP** because **P** ⊆ **RP** ⊆ **NP** [29].

Example 2.2. In Example 2.1 the ideal spectrum $\mathcal{S}_3$(TACGACAC) is used. If for this ideal spectrum substrings ACG and ACA disappear because of negative errors and the substring ACT appear because of positive error then the instance of a SBH problem is spectrum $\mathcal{S}_3 = \{\text{TAC}, \text{CGA}, \text{GAC}, \text{CAC}, \text{ACT}\}$ together with desired length $n = 8$. There are nineteen optimal solutions in total and each solution uses four strings from $\mathcal{S}_3$. Fifteen solutions has length 8 and four solution has length 7. One of the solution is string TACGACAC. The multispectrum with $k = 2$ for this solution is $\mathcal{S}_2(TACGACAC) = [\text{TA}, \text{AC}, \text{CG}, \text{AC}, \text{AC}, \text{CA}, \text{AC}]$ and element AC appears three times in this multiset.

2.6 Existing Methods

The sequencing by hybridisation with positive and negative errors problem is widely analysed in literature. A number of exact algorithms and metaheuristics exist. The branch and bound method that finds an optimal solution is presented in Błażewicz et al. [7]. This method behaves well for instances with positive errors. In presence of repetitions or other negative errors the performance decreases. In article Zhang et al. [41] another exact method for instances with positive errors only are presented. In every possibly solution s found every substring of length k must be in $\mathcal{S}_k$. The solutions with missing substrings are not considered. This is also a branch and bound method that makes predictions about the number of oligonucleotides that can be possibly used and uses it to compute a bound. An interesting heuristic to include also negative errors is presented there. It constructs a new instance. Given the negative error level parameter a new spectrum is created. It consists of every string from $\mathcal{S}_k$ with addition of strings derived from every pair of strings in $\mathcal{S}_k$. Every overlapping of two strings that is not distant apart more than the value of negative error level is considered. Every intermediary substring of length k that results from this overlapping is added to $\mathcal{S}_k$. The new instance is solved using algorithm for positive errors only. Such a transformation generates a very large spectrum and new problems need considerably more time to be solved. However, the method from [41] does not solve the SBH problem presented here. It finds only solutions of exact length n. The Zhang et al. [41] method is implemented in this work for purpose of comparing solutions obtained from hyper-heuristic algorithms.

Many metaheuristics [27] are developed for SBH. The methods based on tabu search are presented in [8, 9] and based on evolutionary or genetic algorithms in [14, 10, 15]. They perform well and find satisfactory solutions in acceptable time. The representation of an instance and moves that appear in tabu search methods are partially adopted in this work.

The hyper-heuristics constructed here operate on a single solution and are conceptually closer to tabu search based methods than to genetic algorithm based methods.

A simple exhaustive search algorithm is written in the procedure BruteForce. It is used to analyse small instances and to check correctness of hyper-heuristics developed. The problem actually solved by this algorithm is a generalization of the SBH:

Definition 2.15. The *standard sequencing with negative error level* problem (Delta SBH) is: given spectrum $\mathcal{S}_k$, integer $n > 0$ and negative error level δ, $1 \leq \delta \leq k$ find the string s over Σ, $|s| \leq n$ which maximizes the number of strings from $\mathcal{S}_k$ that are the substrings of s and no more than δ consecutive substrings of s are missing from $\mathcal{S}_k$.

BruteForce$(\mathcal{S}_k, n, \delta, S, s)$

```
 1  if S = ∅ or used(S_k, s) > used(S_k, t) for any t ∈ S
 2     then S ← {s}
 3     else if used(S_k, s) = used(S_k, t) for any t ∈ S
 4             then S ← S ∪ {s}
 5  for u ∈ S_k
 6        do if s ≠ ε
 7             then for i ← 1 to δ
 8                     do if |s| + i ≤ n ∧ Overlaps(s, u, k − i)
 9                           then BruteForce(S_k, n, δ, S, s u_{k−i+1...k})
10             else BruteForce(S_k, n, δ, S, u)
```

When $\delta = k$ then Delta SBH is equivalent to SBH. The argument $\mathcal{S}_k$ for recursive procedure BruteForce is a spectrum for which sequence s of length n should be reconstructed. Parameter δ controls how many oligonucleotides can be consecutively missing in solution s. For $\delta = k$ the solutions found are solutions for a SBH problem. A set S stores currently optimal solutions found. It is updated during the algorithm run and contains all the best solutions after algorithm execution. The parameter s is also updated in such way and is a currently analysed solution. Initially $S = \varnothing$ and $s = \epsilon$. The function used$(\mathcal{S}_k, s)$ (Definition 2.11) and a predicate Overlaps(u, v, k) (Definition 2.1) are used.

The assignment in line 2 is executed when currently analysed solution s is the best solution found or no other solutions are present. In the first case old solutions are rejected. If s is optimal then it is stored in line 4. In line 6 the condition for initial solution is checked. If $s = \epsilon$ then u is used as a starting oligonucleotide. The **for** loop in lines 7–9 iterates through all possible overlapping positions of strings s and u in order to construct a new solution. Condition in line 8 determines if length of a new solution is feasible and if first $k - i$ symbols of current string u are compatible with last $k - i$ symbols of string s.

The negative error level δ that appears in Delta SBH is not the same parameter as used in Zhang et al. [41]. In [41] this parameter is used to generate new oligonucleotides. If two different pairs generate a number of new oligonucleotides then some other pair can form an overlapping that has more than the value of negative error level consecutively missing oligonucleotides. Thus it is not a Delta SBH instance with parameter δ equal to the value of negative error level used in [41].

Chapter 3

Hyper-heuristics

The complexity of computationally hard problems leads to development of methods that do not solve those problems exactly but desirably close to optimality. The class of metaheuristics emerged over the years with methods like genetic algorithm, simulated annealing, tabu search, ant colony methods and many more. All of them focus on making problem specific decisions on current solution. The aim of hyper-heuristics is to abstract from problem specific moves and operate on a set of low-level heuristic. The hyper-heuristic algorithm does not use any problem specific information. All the problem dependent knowledge is contained in the set of low-level heuristics. An introduction to the hyper-heuristics idea can be found in [27, chap. 16].

In this chapter general concept of the hyper-heuristic is presented in Section 3.1. After this three hyper-heuristic methods follow: choice function in Section 3.2, tabu search in Section 3.3 and simulated annealing in Section 3.4.

3.1 Concept

The first ideas of hyper-heuristic approach can be traced to 1960s where in the work [22] probabilistic learning of combining many scheduling rules was experimentally verified to give better results than each schedule rule taken separately. Later, in 1990s the hyper-heuristic based approach was used a number of times in scheduling and evolutionary computation. The term hyper-heuristic was used for first time in [18].

All the hyper-heuristics designed in this work are local search methods. The notation used is similar to the one in [27, chap. 10]. Local search methods works in a number of turns or iterations. The current iteration number is labelled as t. It starts with 1 and advances with time. At every iteration t the solution from the last iteration ω_{t-1} is given. During this iteration the solution is modified (generally improved when the search performs intensification and worsened when the search performs diversification) and new solution ω_t obtained. The initial solution ω_0 is given on input. The search space is a set Ω of every possible solutions to the problem. The subset $\Omega^* \subseteq \Omega$ is a set of optimal solutions i.e., the solutions that are the most desirable. The set of iterations encountered during the search together with the initial iteration is labelled as $\mathcal{T}$. The initial iteration has number 0 and is used to obtain the initial values of search objects used.

Besides the initial solution ω_0 the set of low-level heuristics $\mathcal{H}$ is also given. Low-

level heuristic h can be applied to solution ω and new solution ω' is obtained, $\omega' = h(\omega)$. The hyper-heuristic algorithm works by selecting $h \in H$ and finding $\omega_t = h(\omega_{t-1})$ at each iteration t. This implies the ordering of low-level heuristics in time. The low-level heuristic used at iteration t is labelled as $\hat{h}_t$. In order to determine $\hat{h}_t$ the hyper-heuristic algorithm can check many low-level heuristics. The subset of low-level heuristics from $\mathcal{H}$ that are checked at iteration t is called $\hat{H}_t$. The computation of $h(\omega)$ can be very expensive and various techniques try to optimise the number of applied heuristics during each turn.

The choice of $\hat{h}_t$ and new value of ω_t is guided by the objective function. This function is problem specific and provided together with the problem definition. There are actually two objective functions — the main objective function $f_o : \Omega \to \mathbb{R}$ and an auxiliary objective function $f_a : \Omega \to \mathbb{R}$. This concept is described in [27, chap. 2]. The main objective function is the objective function that comes from the problem definition. However, it is used only to compare solutions during the selection of optimal ones. During the search process i.e., for selection of $\hat{h}_t$ and ω_t only the auxiliary search objective value is used. This function is the one that is optimised actually. The auxiliary function can behave differently than objective function in special cases but it should match the extrema of main objective search function in order to guarantee the search process to converge to optimal value. The problem solved by hyper-heuristics described in this work is assumed to be a maximisation problem and the value of f_a is maximised during the search. The set of solutions returned by the hyper-heuristics is a set of solutions composed of different ω_t such that $f_o(\omega_t) \geq f_o(\omega_i)$ for every $i \in \mathcal{T}$.

The reason for introducing this division is that the objective function can vary too little for different solutions. The objective function of SBH problem for example counts only the number of oligonucleotides used in the solution. And this number for solutions close to the optimum changes only by a very small and discrete number. Introducing an auxiliary search value with additional information of solutions capacity of including new oligonucleotides guides the search process much better.

In hyper-heuristic algorithms described in this chapter the function $\Delta : \Omega \times \mathcal{H} \to \mathbb{R}$ gives the change of auxiliary function after applying low-level heuristic h to the solution ω:

$$\Delta(\omega, h) = f_a(h(\omega)) - f_a(\omega).$$

The function $r : \Omega \times \mathcal{H} \to \mathbb{R}^+$ is a measure of resources used to apply the heuristic h to the solution ω. The resources can express any resources encountered in real life problems like computer memory, running time, etc. In this work only the running time of a low-level heuristic algorithm is taken into consideration. The function $\tau(t) : \mathcal{T} \to \mathbb{R}$ gives the time at the beginning of iteration t. To simplify the pseudo codes and functions defined it is assumed that variables t and ω_t are global i.e., they are accessible in every function or procedure called during the search.

The two phases of hyper-heuristic methods are generally distinguished i.e., the selection phase and acceptance phase. The purpose of selection phase is to select the low-level heuristic from $\mathcal{H}$ that is used to generate new ω_t at iteration t. The acceptance phase can reject or accept the heuristic proposed by the selection algorithm. If low-level heuristic h is accepted then it is applied to ω_{t-1} and new $\omega_t = h(\omega_{t-1})$ obtained. If h is rejected then ω_t remains unchanged i.e., $\omega_t = \omega_{t-1}$. For example the article [33] chooses from seven selection

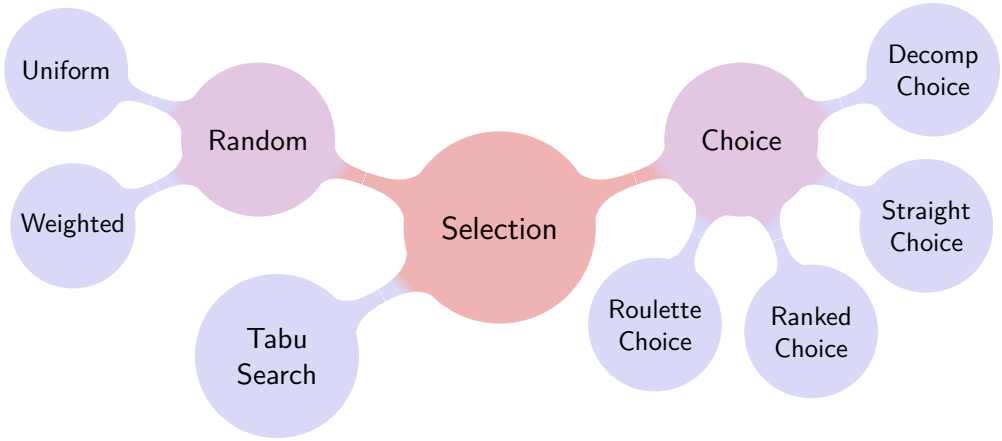


Fig. 3.1: The concept of select operation.

mechanisms and five acceptance mechanisms to build a hyper-heuristic algorithm. This makes a large number of different hyper-heuristics. In the above article four different hyper-heuristic schemes are presented. In this work the most common and accepted one is used. It is presented on procedure HyperHeuristicSearch.

HyperHeuristicSearch($\mathcal{H}, \omega_0$)

```
1  t ← 0
2  while termination conditions are not satisfied
3        do h ← SelectHeuristic(H)
4           if AcceptHeuristic(h) = true
5              then ω_t ← h(ω_{t−1})
6              else  ω_t ← ω_{t−1}
7           t ← t + 1
```

The set of low-level heuristics $\mathcal{H}$ and an initial solution ω_0 is given on input. During each turn the hyper-heuristic selects one low-level heuristic and uses it to obtain new ω_t when accepted. If h is rejected by the acceptance procedure then the current solution remains unchanged, $\omega_t = \omega_{t-1}$.

The diagram of selection mechanisms used in this work is presented on Figure 3.1. The two simplest selection mechanisms are based on randomly selecting a low-level heuristic. Those are the uniform and weighted probability random selections. The method based on tabu search metaheuristic is adapted to hyper-heuristic selection in articles [16] or [30]. The choice function heuristics appeared in [18] and they measure the past performance of heuristics in $\mathcal{H}$ to select a new one.

The acceptance algorithms are shown on the Figure 3.2. Those are monte carlo and all moves methods. The monte carlo method is used in a simulated annealing hyper-heuristic. The all moves acceptance algorithm is used in all the remaining hyper-heuristics. There are six different hyper-heuristics defined and analysed. Four of them uses four different choice functions, one of them uses tabu search as a selection criterion and the last one uses a weighted probability random selection together with a monte carlo acceptance criterion.

Fig. 3.2: The concept of accept operation.

3.2 Choice Function

The hyper-heuristic using a choice function method first appeared in [18]. It was used to solve an optimisation sales schedule problem i.e., a minimisation problem with large number of constrains. A number of small and relatively simple low-level heuristics were implemented and a choice function hyper-heuristic designed for selecting them in every turn. The results of this approach were satisfactory in comparison with the greedy method previously used.

Choice function method uses special measure to select a low-level heuristics that should be applied to current solution ω_{t-1}. The low-level heuristics are ranked using factors that are given by three functions c_s, c_p and c_t (in the article [18] the functions are labelled as f_1, f_2 and f_3 instead of c_s, c_p and c_t respectively). Functions c_p and c_s are designed to intensify search. Function c_t introduces an element of diversification. The purpose of function $c_s : \mathcal{T} \times \mathcal{H} \to \mathbb{R}$ is to give the information on the recent effectiveness of single low-level heuristic h. This function is defined as

$$c_s(t,h) = \begin{cases} \frac{\Delta(\omega_{t-1},h)}{r(\omega_{t-1},h)} + \alpha\, c_s(l_s(t,h),h), & \text{if } t > 0 \\ 0, & \text{otherwise,} \end{cases}$$

where function $l_s : \mathcal{T} \times \mathcal{H} \to \mathcal{T}$ is the number of the last iteration before the current iteration t that h was applied:

$$l_s(t,h) = \max_{i<t} \{ i : h \in \hat{H}_i \vee i = 0 \} .$$

The value of function c_s at that iteration is scaled by the factor α, $0 < \alpha < 1$ and added to the performance of current iteration. This performance is a fraction of the auxiliary objective value change $\Delta(\omega_{t-1}, h)$ and resources used $r(\omega_{t-1}, h)$ by an applied low-level heuristic h. The α factor is used to weight the past results. This formula is in fact a sum of all the past results with exponentially decreasing weights. At every iteration t the value of $c_s(t,h)$ for every $h \in \hat{H}_t$ is determined. The remaining values for $h \notin \hat{H}_t$ are left undefined.

The idea behind the c_s function is that if a low-level heuristic h recently improved the quality of solution then it is likely to continue to be effective. High value of this function causes the low-level heuristic h to be selected more likely in future iterations i.e., search is intensified and focused on the low-level heuristic h.

The function $c_p : \mathcal{T} \times \mathcal{H} \to \mathbb{R}$ holds an information about recent effectiveness of the pairs of low-level heuristics. At every iteration $t > 1$ two low-level heuristics are analysed,

h used in the current iteration and $g = \hat{h}_{t-1}$ used in the previous iteration.

$$c_p(t,h) = \begin{cases} \frac{\Delta(\omega_{t-1},h)}{r(\omega_{t-1},h)} + \beta\, c_p(l_p(t,g,h),h), & \text{if } t > 1 \\ 0, & \text{otherwise.} \end{cases}$$

Similarly as l_s the function $l_p(t,g,h) : \mathcal{T} \times \mathcal{H} \times \mathcal{H} \to \mathcal{T}$ gives the number of the last iteration that g following h was applied before the current iteration t:

$$l_p(t,g,h) = \max_{i<t}\{i : g \in \hat{H}_{i-1} \wedge h \in \hat{H}_i \vee i = 0\} .$$

This function looks for the last iteration t when the low-level heuristic h was applied and it was immediately preceded by the low-level heuristic g in the iteration $t-1$. This value is then scaled by a factor β, $0 < \beta < 1$ and added to the fraction of objective value change and resources used in the iteration t. This is a sum of all the results of heuristics g followed by h with exponentially decreasing weight. The goal of function c_p is to find and promote a co-operational behaviour of two low-level heuristics. If such a synergy exist then value c_p will be high and such a pair of low-level heuristics is more likely to be used.

The function $c_t : \mathcal{T} \times \mathcal{H} \to \mathbb{R}$ is the time span between the current iteration t and the last iteration $l_s(t,h)$ that the low-level heuristic h was used.

$$c_t(t,h) = \tau(t) - \tau(l_s(t,h))$$

If some low-level heuristic h has not been called for a long time then this value becomes high and despite low values of c_s and c_p the low-level heuristic h is accepted as $\hat{h}_t$. This behaviour diversifies the search and introduces element of randomness.

The choice factor functions c_s, c_p and c_t rank the low-level heuristics from $\mathcal{H}$ during the iteration t of hyper-heuristic execution. Now the choice function and its variation that can assume only positive values (positive choice function) can be defined. A positive choice function is used in one of the hyper-heuristics defined later.

Definition 3.1. The *choice function* $c_f(t,h) : \mathcal{T} \times \mathcal{H} \to \mathbb{R}$ that gives at the iteration t a rank of heuristic h is defined to be

$$c_f(t,h) = \alpha\, c_s(t,h) + \beta\, c_p(t,h) + \gamma\, c_t(t,h) .$$

Definition 3.2. The *positive choice function* $c_f^+(t,h) : \mathcal{T} \times \mathcal{H} \to \mathbb{R}^+$ that gives at the iteration t a positive rank of heuristic h is defined to be

$$c_f^+(t,h) = e^{\eta c_f(t,h)}$$

where $c_f(t,h)$ is a choice function and η is a scaling factor.

The choice function is used during the low-level heuristic selection phase and it can be implemented in various ways. In [18] four different selection heuristics are proposed: StraightChoice, RankedChoice, DecompChoice and RouletteChoice. At every iteration t they decide which low-level heuristic h should be applied to the solution ω_{t-1}.

The StraightChoice selection algorithm is the simplest one and requires the least number of resources to make the choice. It just chooses the low-level heuristic h that

maximizes $c_f(t,h)$. The selection can be performed in $\Theta(|\mathcal{H}|)$ time with the following algorithm.

StraightChoice($\mathcal{H}$)

```
1  h ← NULL
2  for l ∈ H
3        do if h = NULL or c_f(t,l) > c_f(t,h)
4              then h ← l
5  return h
```

The StraightChoice heuristic is abbreviated as Sc.

More computationally expensive selection heuristic is a RankedChoice method. It ranks low-level heuristics in $\mathcal{H}$ according to the value of $c_f(t,h)$ for every $h \in \mathcal{H}$ and applies the k low-level heuristics with the highest value of $c_f(t,h)$ to the solution ω_{t-1}. The low-level heuristic h that gives the best auxiliary objective value $f_a(h(\omega_{t-1}))$ is chosen.

RankedChoice($\mathcal{H}$)

```
1  h ← NULL
2  for g ∈ H
3        do HeapInsert(H, (c_f(t,g), g))
4  for i = 1 to k
5        do (c_f, g) ← HeapExtract(H)
6           if h = NULL or f_a(g(ω_{t-1})) > f_a(h(ω_{t-1}))
7              then h ← g
8  return h
```

This algorithm uses the heap operations (See [17, chap. 6]). The HeapInsert inserts a key–value $(key, value)$ pair on the heap H. The procedure HeapExtract(H) returns the key–value pair with the greatest *key* currently on the heap and removes it from the heap. In lines 2–3 the heap is being build. This can be done in $\Theta(|\mathcal{H}|)$ time but the implementation used in this work performs this step in $O(|\mathcal{H}| \log |\mathcal{H}|)$ time. The complexity of RankedChoice procedure in the first case is $O(|\mathcal{H}| + k \log |\mathcal{H}|)$. This is however of lesser importance when compared with a number of calls to low-level heuristics. There is $\Theta(k) = O(|\mathcal{H}|)$ calls of low-level heuristics during each selection process. This is quite expensive compared to the StraightChoice method. The value k can be adjusted to the problem solved by the hyper-heuristic framework. RankedChoice hyper-heuristic is abbreviated a Rc.

The DecompChoice is another selection method that evaluates a number of low-level heuristics. This time the number of calls is of $O(1)$. It takes into consideration at most four low-level heuristics: h_1 that has the greatest value of $c_s(t,h_1)$, h_2 that has the greatest value of $c_p(t,h_2)$, h_3 that has the greatest value of $c_t(t,h_3)$ and h_4 that has the greatest value of $c_f(t,h_4)$. It applies them to the solution ω_{t-1} and chooses the one that yields the best auxiliary objective value of obtained solution.

DecompChoice($\mathcal{H}$)

```
1  h ← NULL
2  for g ∈ H
3        do HeapInsert(H[1], (c_s(t, g), g))
4           HeapInsert(H[2], (c_p(t, g), g))
5           HeapInsert(H[3], (c_t(t, g), g))
6           HeapInsert(H[4], (c_f(t, g), g))
7  for i = 1 to 4
8        do (c_f, g) ← HeapExtract(H[i])
9           if h = NULL or f_a(g(ω_{t-1})) > f_a(h(ω_{t-1}))
10             then h ← g
11 return h
```

The above algorithm can be implemented to run in $\Theta(|\mathcal{H}|)$ time but the implementation used in this work takes $O(|\mathcal{H}| \log |\mathcal{H}|)$ time. The DecompChoice heuristic decomposes the choice function and evaluates the low-level heuristics according to the performance of its components. It is abbreviated as Dc.

The last choice function selection mechanism used is a RouletteChoice heuristic. The RouletteChoice algorithm chooses one low-level heuristic randomly. The probability of choosing each heuristic h is given by the probability function $p(h) : \mathcal{H} \to [0, 1]$,

$$p(h) = \frac{c_f^+(t, h)}{\sum_{g \in H} c_f^+(t, g)} .$$

This hyper-heuristic uses a non-negative version of choice function $c_f^+(n, h)$ that guarantees the function p to be defined correctly.

RouletteChoice($\mathcal{H}$)

```
1  h ← heuristic H chosen at random according to p.
2  return h
```

The selection of h in line 1 can be performed by the algorithm that divides the interval $[0, 1)$ to sections of length $p(h)$ for every $h \in \mathcal{H}$ and draws a real number in this range with uniform probability. The heuristic which is assigned to the drawn section is then returned. The RouletteChoice procedure must evaluate $c_f^+(t, h)$ for every $h \in \mathcal{H}$ and because of that this procedure takes $\Theta(|\mathcal{H}|)$ time. No low-level heuristic need to be executed during the RouletteChoice selection phase. This selection mechanism is abbreviated as Uc.

Besides the selection phase also an acceptance algorithm is required to construct a hyper-heuristic. All the choice function methods uses a trivial acceptance criterion that accept all the moves selected.

AllMoves(h)

```
1  return TRUE
```

This acceptance method is abbreviated as Am.

Thus there are four different choice function hyper-heuristics: ScAm, RcAm, DcAm and UcAm. All of them require to specify the parameters α, β and γ. Additionally,

the RcAm requires parameter k and UcAm requires the parameter η. Those parameters are problem specific and must be determined somehow with the use of problem specific knowledge. In this work they are chosen empirically for the SBH problem later. Another approach for setting those parameters is presented in [19] i.e., a mechanism to control them automatically is given. The parameters α and β can be controlled by awarding the good performance of applied heuristic and penalising for the bad performance of applied heuristic. The performance is measured by the change in the auxiliary objective function. The parameter γ can be increased to diversify the search when a large number of last iterations performed badly. It can be decreased when the search is to diverse and low-level heuristics are chosen uniformly at random.

The choice functions are used and analysed in a number of articles. In [31] choice function hyper-heuristic is compared with an intelligent random heuristic. This intelligent random heuristic weights the probabilities of heuristics chosen based on the number of times they were used by the hyper-heuristic. Hyper-heuristic performed noticeably better and it is capable of selecting intelligent combinations of low-level heuristics at hand and adapting those combinations to the problem solved and region of search space. The work [33] compares various combinations of hyper-heuristic selection and acceptance methods. According to it the choice function selections are competitive with the other approaches.

3.3 Tabu Search

Tabu search is a local search metaheuristic that at every iteration analyses a set of solutions called neighbourhood and chooses one of them as the next solution. Not all of the neighbouring solutions can be used. If moves that created them are placed on the tabu list then they can not be used. The moves are usually added to the tabu list after they are used in order to prevent from trapping the search in local minimum. The moves are removed from the tabu list after some arbitrary time. A good introduction to design of tabu search algorithms is presented in [27, chap. 2]. However, many techniques described there can not be copied to the hyper-heuristic approach because of the separation with specifics of the problem solved.

Two issues are of importance in designing the tabu search hyper-heuristic: the process of selecting the next low-level heuristic and handling the tabu list. In [16] the tabu search algorithm is described that solves the university timetabling problem. The selection of the next low-level heuristic used is based on this approach. Each low-level heuristics have assigned rank r_h, initially $r_h = r_m$. When the low-level heuristic improves the solution ω_t at iteration t then the rank is increased by one. In the other case the rank is decreased by one. Ranks are allowed to assume values in range $[r_m, r_M]$. During the selection of the next low-level heuristic the one with the greatest rank is chosen.

The construction of a tabu list is based on [30]. Every time when the used low-level heuristic does not improve the solution then this low-level heuristic is remembered on the tabu list for a duration of k iterations. During this time the low-level heuristic cannot be used. The parameter k called tabu iterations is adjusted for the problem. An algorithm described below expresses those ideas. The heap H of pairs (r_h, h) is initialised to contain all low-level heuristics $h \in \mathcal{H}$ with their initial values $r_h = r_m$.

TabuSearch($\mathcal{H}$)

```
1  E ← ∅
2  (r_h, h) ← HeapExtract(H)
3  while tabu[h] > t
4        do E ← E ∪ {(r_h, h)}
5           (r_h, h) ← HeapExtract(H)
6  if f_a(h(ω_{t-1})) − f_a(ω_{t-1}) > 0
7     then E ← E ∪ {(min(r_M, r_h + 1), h)}
8     else E ← E ∪ {(max(r_m, r_h − 1), h)}
9          tabu[h] ← t + k
10 for every (r_h, h) ∈ E
11       do HeapInsert(H, (r_h, h))
12 return h
```

In lines 2–5 the low-level heuristic h with the highest rank r_h that is not on the tabu list is selected. The important assumption for this algorithm is that $k < |\mathcal{H}|$. In other case the tabu list can cause to exist no low-level heuristic for selection. The set E of extracted pairs stores the pairs from heap H that were removed from H. In lines 6–9 the ranking of selected low-level heuristic h and tabu list are updated. At the end of algorithm the extracted pairs are inserted back on the heap for the next iteration to work. The time complexity of this algorithm is $O(|\mathcal{H}| \log |\mathcal{H}|)$ and exactly one evaluation of low-level heuristic is performed in each turn. The tabu search is parametrised by range $R = [r_m, r_M]$ and tabu iterations length k. This selection method is abbreviated as Ts.

3.4 Simulated Annealing

The simulated annealing [27, chap. 10] is a metaheuristic technique that at each iteration t generates a move that transforms the current solution ω to the new one ω'. If the new solution is better then it is used always. However, if the new solution is worse then the old one i.e., $f_a(\omega') < f_a(\omega)$ then it is accepted with probability

$$P\,(\text{Acceptance}) = \exp\left(\frac{f_a(\omega') - f_a(\omega)}{\delta(t)}\right) \tag{3.1}$$

The probability changes with time according to the temperature function $\delta : \mathcal{T} \to \mathbb{R}$. The temperature function $\delta(t)$ used in this work is

$$\delta(t) = \frac{\beta}{\log\left(\left\lfloor \frac{t}{m} \right\rfloor + 2\right)}\,. \tag{3.2}$$

The high values of this function cause the algorithm to accept the large number of solutions, even worse than the current one. Small values imply very low probability of accepting worse solutions. At the beginning of the search the value of this function is high and decreases with time. This causes the solution to become more stable with time. Parameter m is the number of iterations for which the temperature value holds. It controls the time characteristics of this function. Parameter β defines the initial temperature used and controls the initial acceptance probability.

Simulated annealing method transforms very well to the field of hyper-heuristics. Instead of designing moves that can transform the solution to the new one a set of low-level heuristics is used that do the same thing. In [3] simulated annealing hyper-heuristic was used to optimise assembling components on the printed circuit boards. The acceptance phase in this article used the randomised monte carlo acceptance criterion just like the simulated annealing metaheuristic uses. The temperature function used in this work is based on [5] but it has a slightly different characteristic. The procedure MonteCarlo is an acceptance phase of the simulated annealing approach.

MonteCarlo(h)
1 $\Delta \leftarrow f_a(h(\omega_{t-1})) - f_a(\omega_{t-1})$
2 **if** $\Delta > 0$
3 **then return** TRUE
4 **else if** $\exp\left(\frac{\Delta}{\delta(t)}\right) <$ random()
5 **then return** TRUE
6 **return** FALSE

This is just a pseudo code of the formula for acceptance probability defined above. Function random() returns a uniformly distributed random number in range $[0, 1)$. This aceptance phase is denoted as Mc.

The selection phase for simulated annealing approach is different from selection methods used for other hyper-heuristics. It is a simpler version of weighted probability selection defined in [4]. The low-level heuristic h is chosen randomly according to the probability function $p : \mathcal{H} \to [0, 1]$ defined as

$$p(h) = \frac{w_h}{\sum_{g \in H} w_g}.$$

The numbers w_h are the weights of low-level heuristic $h \in \mathcal{H}$. Each weight is updated after the selection similarly as a ranks in tabu search. If the change in auxiliary objective value is positive then w_h is increased by one. It is decreased by one in any other case. The bound for every w_h is the range $[w_m, w_M]$ and every $w_h = w_m$ initially. The condition $w_m > 0$ must hold in order to p be a probability function. The method Weighted presents this selection algorithm.

Weighted($\mathcal{H}$)
1 $h \leftarrow$ heuristic $\mathcal{H}$ chosen at random according to p.
2 **if** $f_a(h(\omega_{t-1})) - f_a(\omega_{t-1}) > 0$
3 **then** $w_h \leftarrow \min(w_M, w_h + 1)$
4 **else** $w_h \leftarrow \max(w_m, w_h - 1)$
5 **return** h

Evaluating the probability function and drawing a random low-level heuristic h takes $\Theta(|\mathcal{H}|)$ time. This algorithm is abbreviated as Wr and is parametrised by the allowed range $W = [w_m, w_M]$ of w_h for $h \in \mathcal{H}$.

Chapter 4

Application in Sequencing Problem

In this chapter the application of hyper-heuristic scheme to SBH problem is studied. The first Section 4.1 describes an introductory approach that is derived from a tabu search metaheuristic. Some of the moves used in this metaheuristic are adapted as the low-level heuristics. The extension of this approach is presented in Sections 4.2 through 4.5. It is derived mainly from the observations of the specifics of solution encoding. They allow to define a large variety of low-level heuristics used in hyper-heuristic search. The analysis on the particular instances of the SBH shows that those low-level heuristics inhibits various convergences to the optimum of the local search.

4.1 Tentative Method

There is a number of local search methods in literature that were applied to the SBH problem. One of them is a tabu search meta-heuristic described in Błażewicz et al. [8]. The hyper-heuristic schemes analysed in this work also use a local search methodology. Thus it can be expected that this meta-heuristic is somehow compatible with the hyper-heuristic approach. The moves described in [8] are used in the first implementation of the low-level heuristics. The goal of this investigation is to have a reference point that can be extended further. Those raw low-level heuristics are also compared with more dedicated ones later.

In the mentioned tabu search algorithm solution is encoded using two collections. The first collection is an ordered set of oligonucleotides that occur in the final sequence. This ordered set is represented as list O and from this list a DNA sequence is reconstructed. The second collection is a set T called trash set. This pair (O,T) encodes solutions available to this method. The sequence is reconstructed with the help of a greedy algorithm that traverses a list O and tries to append every oligonucleotide o at the closest position possible to the end of constructed sequence ω. Sequence ω and new string o cannot overlap on more than $k-1$ symbols. The reconstructed sequence ω does not necessarily have a length lower or equal to n. Thus feasible solutions are only those for which $|\omega| \leq n$.

Quoted tabu search algorithm does not use the value of oligonucleotides used $f_o(\omega) = \text{used}(\mathcal{S}_k, \omega)$ in the current solution ω as a function that guides the search. The moves proposed there require a objective function that takes into account a sequence length. The measure called condensation is used. This is a ratio of the number of oligonucleotides

used to the length of solution ω, denoted as $f_c(\omega)$ and defined as

$$f_c(\omega) = \frac{\text{used}(\mathcal{S}_k, \omega)}{|\omega|}.$$

This measure tends to find the most dense solutions where there are no gaps between substrings of ω. However, the length of the constructed solution is ignored and this function does not solve the SBH problem. Tabu search method divides a search to two phases. During one of those phases the condensation is used only and solution becomes very dense because designed moves work well. During the second phase number of random moves is performed. They consist mostly of insertions and leads to the solution with large number of oligonucleotides. The hyper-heuristic schemes abstract from such a behaviour and no phases like that can be introduced. Instead of this a special auxiliary objective function f_a is designed.

$$f_a(\omega) = \alpha f_o(\omega) + \beta f_c(\omega) \tag{4.1}$$

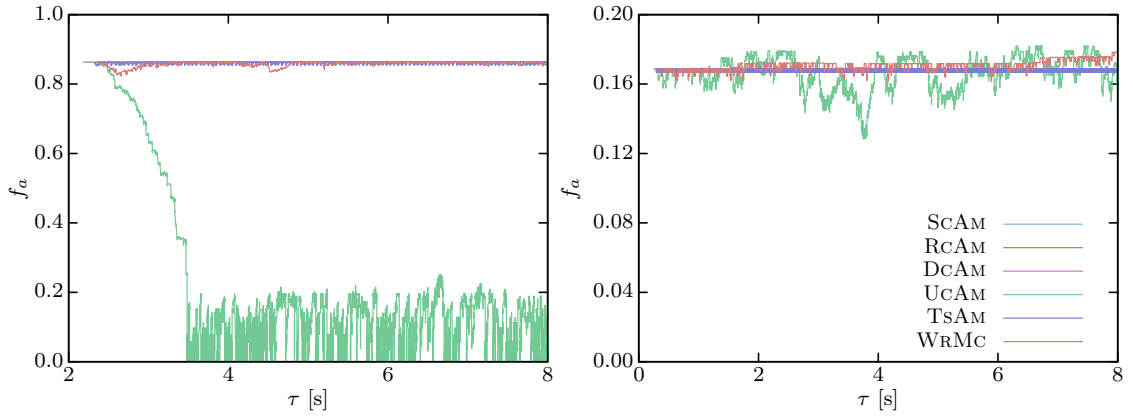
$$= \alpha \text{ used}(\mathcal{S}_k, \omega) + \beta \frac{\text{used}(\mathcal{S}_k, \omega)}{|\omega|} \tag{4.2}$$

This is a linear combination of objective function and condensation function. The factors $\alpha > 0$ and $\beta > 0$ are the weights of components and are determined empirically. Because $|\omega| = 0$ for empty solution this formula is only valid for $|\omega| > 0$ or $\text{used}(\mathcal{S}_k, \omega) > 0$. From Section 3.1 the value of $f_a(\omega)$ should match the extrema of $f_o(\omega)$ for search process to be convergent to the same optimal solutions. This formula does not satisfy this constraint for every α and β.

It should be noted that in the implementation of this method an approximation of the function $\text{used}(\mathcal{S}_k, \omega)$ is used. It is determined as $|\mathcal{S}_k| - |T|$ which not always gives the true value of $f_o(\omega)$ (the reason for that is explained in Section 4.2). However, this approximation is used only for auxiliary search value. The objective search value returned always gives a true value of $\text{used}(\mathcal{S}_k, \omega)$ and $f_o(\omega)$.

The concept of the cluster of oligonucleotides is introduced as a sequence of following oligonucleotides that are shifted by only one nucleotide relative to the previous oligonucleotide. Clusters are maximally condensed groups of oligonucleotides. They are very valuable because they give large increase in objective value and do not use much space. Some of the moves described in [8] operate on clusters and some on single oligonucleotides. Those moves are directly translated to the low-level heuristics. There are five different low-level heuristics.

- Insertion of the nucleotide — SI. A single oligonucleotide from the trash set is inserted into the solution. There are no constrains on the destination position where the oligonucleotide can be put. The low-level heuristic tries to insert every unused oligonucleotide on every possible position and chooses the best option i.e., the combination with the greatest $f_a(\omega)$.
- Shift of the oligonucleotide — SM. Moves one oligonucleotide from one position in the list O to another. During the shift no cluster can be destroyed. Only oligonucleotides outside the cluster may be shifted provided that they do not break any cluster. Every possibility is checked.



(a) BioServer [1], instance 15 without repetitions for $n = 200$ with 10% errors.

(b) Benchmark instance 11, without SDC low-level heuristic.

Fig. 4.1: Two executions of the set method demonstrating change of auxiliary objective value with the advance of run time. The maximum value of f_a is normalised to 1.

- Shift of the cluster to another position in the sequence — SMC. Cluster can be shifted only if it does not break another cluster. Again, every possible combination of shifted cluster and destination position is checked.
- Deletion of the nucleotide — SD. A single oligonucleotide from the solution is moved to the trash set. Only oligonucleotides outside the cluster or being one of the cluster ends may be deleted. The oligonucleotide that gives the greatest $f_a(\omega)$ is used.
- Deletion of the cluster — SDC. Every oligonucleotide from the cluster will be placed in the trash set. This low-level heuristic is quite invasive. It can break solution and change it widely.

Tabu search method remembers inserted or shifted oligonucleotides on the tabu list. The oligonucleotides on tabu list cannot be removed for a number of iterations. The purpose of those steps is to prevent reversal of the moves made. The methodology of hyper-heuristic search precludes copying the behaviour of tabu list. No substitution of tabu list is introduced to the low-level heuristics implemented.

This set of low-level heuristics was used to solve the instances of the SBH problem (detailed description of the instances sets is given later in Sections 5.1 and 5.2). Figure 4.1 shows two different executions of the hyper-heuristics using low-level heuristics from the set method. Differences between instances used in those executions are explained later but the benchmark instance on Figure 4.1b is much less correlated than the biological instance on Figure 4.1a and should give more diverse results. Two search results on Figure 4.1a are noticeable among the others. Those are the results for the simulated annealing WrMc and the roulette choice UcAm hyper-heuristics. The simulated annealing worsen the solution significantly two times. This is because Mc acceptance method allows for it probabilistically. The search process of UcAm is chaotic and approaches low values. This is because of the SDC low-level heuristic which causes to remove a very large part of the solution each time when it is called. In biological data instance the clusters often become large and removing one of them results in dramatic drop of the solution objective value.

The UC selection chooses SDC randomly. Instead of its bad performance the probability increases with time because of the choice function c_t. Figure 4.1b is an execution of the hyper-heuristics without the low-level heuristic SDC. The objective values during the search of roulette choice hyper-heuristic are much higher and even better than during the search processes of other hyper-heuristics.

This shows that hyper-heuristic search is not only dependent on the hyper-heuristic method used to guide the low-level heuristics. A set of low-level heuristics is also very important because it can influence the search process significantly. This tentative method has poor variety of low-level heuristics. There are only five low-level heuristics and at least insertion and deletion is required to construct the solutions. Thus in order to check the effect of low-level heuristics on the search process an extended, more diverse set of low-level heuristics is designed.

4.2 Solution Encoding

It is apparent from above results that large variety of low-level heuristics is required. The method used in this work is a generalisation and extension of moves and ideas presented in Błażewicz et al. [8]. The low-level heuristics are designed to be easly parametrised and manipulated without modyfing their source code. The change in parameters leads to various behaviour and gives a significantly different results. In the general case the hyper-heuristics are used mainly for very complex problems where even simple heuristics are difficult to design and implement. Different low-level heuristics for such problems lead to very different results and combining them lead to new solutions. SBH problems exhibits quite simple structure and low-level heuristics give predictable changes. Introducing large number of parameters leads to larger diversification of moves performed and presumably behaviour more compatible with the hyper-heuristic search.

To design new set of low-level heuristics a special attention is given to the solution encoding. Every possible solution to the SBH problem is considered; i.e., even purely theoretical solutions with completely random substrings not included in the spectrum are considered. Definition 2.6 of the SBH problems allows for such solutions. However, it turns out that those solutions can be omitted without loosing the generality.

The solution to the SBH problem is a string ω. The content of this string can be divided in two parts. First part consists of the substrings from spectrum $\mathcal{S}_k$ that were given on input. The rest of the solution are substring that do not overlap with any string from $\mathcal{S}_k$. This second part is not of much interest, it can be changed to any other sequence of symbols without influence on the objective function (provided that no string from $\mathcal{S}_k$ starts to overlap with the new sequence of symbols). Thus interesting solutions can be encoded as a sequence of oligonucleotides (list of oligonucleotides) from spectrum $\mathcal{S}_k$ with associated values of the shift between every pair of consecutive oligonucleotides. Feasible solutions must satisfy the predicate $\text{Overlaps}(u, v, k - x)$ for every oligonucleotide u followed by v on the list with shift $x \leq k$ between them in order to disallow random symbols in the solution. Shift is just a distance between the successive substrings in the final solution. Any oligonucleotide from spectrum can appear more than once in this list because every substring of ω that is in $\mathcal{S}_k$ is represented by an element on the list.

For two consecutive strings u and v on the oligonucleotides list with shift $x \leq k$ between

them there can exist smaller shift $x' < x$ such that predicate Overlaps$(u, v, k - x')$ is true. This means that u and v can be placed more closely to each other and this change never makes the solution infeasible: the distance between u and v is decreased and the total solution length is decreased by the same value, the predicates Overlaps between u and previous string on the list and v and the next string on the list remains unchanged (the distances between those strings are greater than 1 and Overlaps does not depend on any substrings following u and preceding v respectively). Thus solutions can be compressed to their shortest forms with no more compression possible. If solution is maximally compressed then information about shift is no longer required i.e., shift can be deduced directly from the pairs of oligonucleotides. This leads to the new list representation of the solution which is just a list of consecutive oligonucleotides. The solution can be build from such a list by appending consecutive strings as close to the sequence being build as possible. Going a step further and keeping the objective function value as a goal i.e., the number of used oligonucleotides, the list can be reduced to a permutation of the oligonucleotides used. The list can be traversed from the beginning to the end and only strings that did not appeared before added to the end of the permutation built. This operation can only reduce the length of the resulting sequence and thus keeps the solution feasible. This representation was used in [8] and is used in the preliminary method from Section 4.1. The methods described in this work require more information about the structure of the solution and thus a list representation is used.

With every current solution ω there is associated a list that encodes the solution. List is defined as a set L of elements e of the list together with functions $\text{key}(e) : L \to \mathcal{S}_k$, $\text{prev}(e) : L \to \mathscr{L}$, $\text{next}(e) : L \to \mathscr{L}$, where $\mathscr{L} = L \cup \{\epsilon\}$ is a list L with ϵ, an empty element. The function $\text{key}(e)$ associates an oligonucleotide (string) with the element e from the list L. The functions $\text{prev}(e)$ and $\text{next}(e)$ defines the order of elements. They assume value equal to e' when element e' precedes or succeeds element e respectively. If e is a first element on the list then $\text{prec}(e) \equiv \epsilon$ and $\text{head}(L) \equiv e$. Also when e is last element on the list then $\text{succ}(e) \equiv \epsilon$ and $\text{tail}(L) \equiv e$. If $L = \varnothing$ then $\text{head}(L) \equiv \text{tail}(L) \equiv \epsilon$. Although L is a set it is identified with the list of elements. The list L and defining functions are closely associated with the current solution ω. However, for simplicity no distinction of this fact in notation is made. It is assumed that all definitions are for current solutions ω.

The concept of range of elements is used often in many definitions and algorithms. A range of size n is a sequence $(e_1, e_2, \ldots e_n)$ of elements from L that appear consecutively on the list i.e., $\text{next}(e_i) = e_{i+1}$ for $1 \le i < n$. Set of all possible ranges of size n is denoted as $\mathcal{R}_n$. The set of all ranges is denoted as $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2, \ldots, \mathcal{R}_{|L|}$. It is apparent that $\mathcal{R}_1 = L$. The functions $\text{next}(e)$ and $\text{prev}(e)$ are easily extended to ranges, $\text{prev}(r) = \text{prev}(e_1)$ and $\text{next}(r) = \text{next}(e_n)$ for any $r = (e_1, e_2, \ldots, e_n) \in \mathcal{R}_n$. The set of oligonucleotides in range r is given by function $\text{spec}(r) : \mathcal{R} \to 2^{\mathcal{S}_k}$, $\text{spec}(r) \equiv \{\text{key}(e_1), \text{key}(e_2), \ldots, \text{key}(e_n)\}$.

The maximum value of the shift x^* between oligonucleotides from ω in the list representation is constrained by $x^* \le k$. This is because shifts larger than k can lead to appearance of new oligonucleotides. As described above the arbitrary string v can be placed after u with minimal shift $d = \text{distance}(u, v)$. When this happens a new combined string with prefix u and suffix v is created. One of the substrings s placed between u and v with shift d', $1 \le d' < d$ relative to u can be in $\mathcal{S}_k$. In that case this substring s is said to be forced by u and v. This is presented on Figure 4.2a and described by predicate Forced

(a) Forced string. **(b)** 2-cluster.

Fig. 4.2: Three oligonucleotides u = CGTAA, v = AATAC and s = GTAAT. On figure (a) placing u and v next to each other forces appearance of s in the combined sequence. Figure (b) shows a 2-cluster that is formed by those three oligonucleotides.

defined below.

Definition 4.1. String s is forced by strings u and v if the predicate

$$\text{Forced}(s,u,v) \equiv \exists d' < d \ \text{Overlaps}(u,s,k-d') \wedge \text{Overlaps}(s,v,k+d'-d)$$

where $d = \text{distance}(u,v)$ holds.

This is the reason why $x^* \leq k$. If shift is greater than k between u and v then compressing those oligonucleotide can lead to appearance of new, forced oligonucleotides and list that encodes the solution expands. The forced oligonucleotides are accounted in low-level heuristics created. In every algorithm through this work any change in the list representation of ω leads always to a new list with every forced oligonucleotide present in this list. Doing so many redundant moves are avoided e.g. inserting oligonucleotide that is already present in the solution. They are important for deletion operations i.e., it is meaningless to remove already forced oligonucleotide since it will be present in the solution anyway. Forced strings are also a main reason why list representation instead of set representation is used. In set representation not every forced oligonucleotide can be represented. The set $\mathcal{F}$ of all forced ranges in list representation of solution ω will be helpful.

$$\mathcal{F} = \{r \in \mathcal{R} : \text{prev}(r) \neq \epsilon \wedge \text{next}(r) \neq \epsilon \wedge \forall s \in \text{spec}(r) \ \text{Forced}(s, \text{key}(\text{prev}(r)), \text{key}(\text{next}(r)))\}$$

It follows from the above definition that forced ranges are only those that will reappear after deletion. No range that starts at the beginning or ends at the end of list is a forced range.

One type of parameters used in designed low-level heuristics are sets of allowed ranges. To describe them a notion of d-clusters is introduced. The definition of clusters from [8] as a maximal sequences of strings shifted by one symbol is extended in the following definition.

Definition 4.2. A d-cluster is a range $(e_1, e_2, \ldots, e_n) \in R$ that satisfies

$$\forall 1 \leq i < n \ \text{distance}(\text{key}(e_i), \text{key}(e_{i+1})) \leq d\,.$$

An example of 2-cluster is presented on Figure 4.2b. When range $r = (e_1, e_2, \ldots, e_n)$ and $r' = (e_i, e_{i+1}, \ldots, e_j)$ for $1 \leq i$ and $j \leq n$ and $r' \neq r$ then r' is part of r, $r' \prec r$. The d-cluster r' is maximal if there does not exist d-cluster r such that $r' \prec r$. The d-cluster is interesting if at least one of the distance inequalities in Definition 4.2 is an equality. The set of all maximal interesting d-clusters is denoted as $\mathcal{C}$. The solution ω inherits an

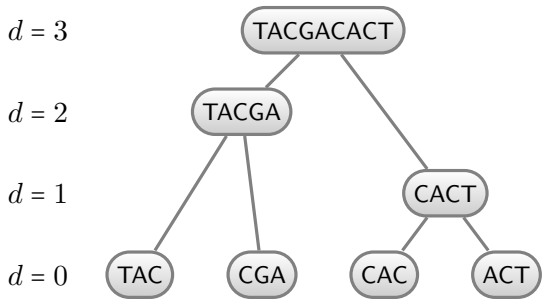


Fig. 4.3: The tree formed by d-clusters of solution TACGACACT for spectrum $\mathcal{S}_3 = \{\mathrm{ACT}, \mathrm{CAC}, \mathrm{CGA}, \mathrm{TAC}\}$. The root of a tree is a cluster that forms entire solution and leaves are strings from spectrum.

interesting structure from the point of view of d-clusters. The set $\mathcal{C}$ of d-clusters forms a tree. Leaves of this tree are 0-clusters i.e., the single substrings. The 1-clusters forms a first level of the tree and their children are 0-clusters. Higher levels of the tree are formed by the remaining clusters and their children can be nodes on the lower levels. A root of this tree is a k-cluster that extends on the entire list L. One example of such tree is shown on Figure 4.3. Although the tree is not used directly in the implemented algorithms it can lead to implementations with algorithms having better time complexity.

The following paragraphs define the sets that are used as a parameters for low-level heuristics in the extended method. One more function, the distances(r) function that uses Definition 2.2 is required to define allowed ranges. The function distances(r) $\equiv$ $\{\text{distance}(e_i, e_{i+1}) : 1 \le i < n\}$ of range $r = (e_1, e_2, \ldots, e_n)$ is a set of distances that appear in this range. Now, the clusters set $R_c^{A,D}$ is a set of clusters that satisfies following constraints:

$$R_c^{A,D} = \{c \in \mathcal{C} : \text{distances}(c) \subseteq D \wedge \text{spec}(c) \subseteq A \wedge c \notin \mathcal{F}\} .$$

The set $A \subseteq \mathcal{S}_k$ is a set of allowed oligonucleotides that can be present in the clusters. Set D is an interval of integer numbers that defines which distances can be present in the clusters of $R_c^{A,D}$. Sets A and D are used as a parameters for low-level heuristics. The $R_c^{A,D}$ does not allow to any of the clusters to be a forced range. The relaxed version of this set is forced clusters set $R_f^{A,D}$,

$$R_f^{A,D} = \{c \in \mathcal{C} : \text{distances}(c) \subseteq D \wedge \text{spec}(c) \subseteq A\} .$$

It differs from $R_c^{A,D}$ only in forced range constraint i.e., it includes ranges that are forced. In various removal operation one more set of ranges is used. It is not connected with clusters and is a minimal set of ranges that can be removed with reasonable effect. It uses function mincut(r) : $\mathcal{R} \to \mathbf{N}$ which is a minimal distance that is affected by removal of this range,

$$\text{mincut}(r) \equiv \min \bigcup_{x \in M} \text{distances}(x)$$

where $M = (p \times \{e_1\}) \cup (\{e_n\} \times n) \cup \{r\}$, $r = (e_1, e_2, \ldots, e_n)$, $p = \{\text{prev}(e_1)\}$ or $p = \varnothing$ if $\text{prev}(e_1) = \epsilon$ and $n = \{\text{next}(e_n)\}$ or $n = \varnothing$ if $\text{next}(e_n) = \epsilon$. Thus mincut($r$) is the set distances($r$) with two additional distances — of position before and position after range

r in the list L. This last set $R_m^{A,d}$ is called minimal ranges set,

$$R_m^{A,d} = \{r \in \mathcal{R} : d \leq \text{mincut}(r) \wedge \text{spec}(r) \subseteq A \wedge r \notin \mathcal{F} \wedge \forall r' \prec r : r' \in \mathcal{F}\} .$$

In this definition the natural number d instead of range D is used. This set is a set of ranges of minimal size that are not forced and that do not break any pair of oligonucleotides placed closer than d in the solution ω. Those three sets $R_c^{A,D}$, $R_f^{A,D}$ and $R_m^{A,d}$ are used to select ranges of strings for moves or deletion.

The low-level heuristics that are responsible for inserting strings or ranges into list L use a set of allowed positions P^d and set of unused oligonucleotides U^A. To define the former one more set $\mathcal{P}$ i.e., a set of all possible positions in list L is required.

$$\mathcal{P} = \mathcal{R}_2 \cup \{(\epsilon, \text{head}(L)), (\epsilon, \text{tail}(L))\} .$$

Now this set of positions can be specified by parameter d which prevents from insertions between pairs of strings that are distant closer than d,

$$P^d = \{(b, a) \in \mathcal{P} : b = \epsilon \vee a = \epsilon \vee d \leq \text{distance}(\text{key}(b), \text{key}(a))\} .$$

The set U^A of unused oligonucleotides is a set of substrings that can be inserted into list L with guaranteed increase of the objective function. It is defined as

$$U^A = \{s \in \mathcal{S}_k : s \notin \text{spec}(L) \wedge s \in A\} ,$$

where $\text{spec}(L)$ is a set of strings from spectrum used in L, defined similarly as $\text{spec}(r)$ for a ranges.

Example 4.1. This example shows how the sets defined above look like. The solution TGATACTGACTCG to the SBH problem with spectrum $\mathcal{S}_3 = \{\text{ACT}, \text{GAC}, \text{TCG}, \text{TGA}\}$ and $n = 13$ is presented on Figure 4.4. The list L that encodes the solution has seven elements, $L = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7\}$ with $\text{key}(e_1) = \text{key}(e_4) = \text{TGA}$, $\text{key}(e_2) = \text{TAC}$, $\text{key}(e_3) = \text{key}(e_6) = \text{ACT}$, $\text{key}(e_5) = \text{GAC}$ and $\text{key}(e_7) = \text{TCG}$.

The set $\mathcal{F}$ of forced ranges is equal to $\{e_3, e_4, e_5, e_6\}$. No ranges of length greater than 1 appears there. The set of maximal interesting clusters with distance $d = 0$ is $\{e_1, e_2, e_3, e_4, e_5, e_6, e_7\}$, $d = 1$ is $\{(e_2, e_3), (e_4, e_5, e_6)\}$, $d = 2$ is $\{(e_2, e_3, e_4, e_5, e_6, e_7)\}$ and $d = 3$ is $\{(e_1, e_2, e_3, e_4, e_5, e_6, e_7)\}$. The union of those sets is a set of clusters $\mathcal{C}$. If $A = \mathcal{S}_k \setminus \{\text{TCG}\}$ then set $R_f^{A,[0,1]}$ of forced clusters is

$$R_f^{A,[0,1]} = \{e_1, e_2, e_3, e_4, e_5, e_6, (e_2, e_3), (e_4, e_5, e_6)\} ,$$

and set $R_c^{A,[0,1]}$ of clusters that do not contain forced ranges is

$$R_c^{A,[0,1]} = \{e_1, e_2, (e_2, e_3), (e_4, e_5, e_6)\} .$$

The set of minimal clusters $R_m^{A,1}$ that allows to intersect distances greater or equal to $d = 1$ is equal to

$$R_m^{A,1} = \{e_1, e_2, (e_3, e_4), (e_4, e_5), (e_5, e_6), e_7\} .$$

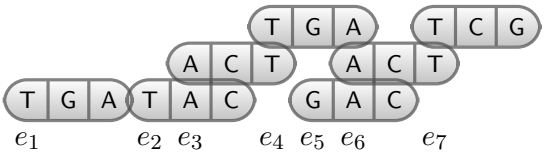


Fig. 4.4: The solution TGATACTGACTCG considered in example 4.1. Oligonucleotides ACT and TGA appear two times in the sequence. Labels are the elements on the list L connected with oligonucleotides.

Set P^2 of positions with distance greater or equal to $d = 2$ is

$$P^2 = \{(\epsilon, e_1), (e_1, e_2), (e_3, e_4), (e_6, e_7), (e_7, \epsilon)\} .$$

The set $U^A = \varnothing$ since all the oligonucleotides are used.

4.3 Low-level Heuristics

There are four main types of low-level heuristics used: insert, move, swap and delete. Each of those heuristics is specified by a number of parameters and leads to many different derived low-level heuristics. Before describing them a number of structures and operations on list L used to encode solution must be defined. However, not all of them will be presented in pseudo codes. Thus during the search besides the list of oligonucleotides three more structures are used: $count[s]$, $history[s]$ and *unused* for $s \in \mathcal{S}_k$. In actual implementation also length of the entire sequence is updated. Including this is straightforward with given pseudo codes. An array *count* holds an information how many times the string s appears on list L. The value of $history[s]$ is the value of last iteration that string s was used in some operation. The set *unused* is similar to the trash set used in [8], it holds a set of strings from $\mathcal{S}_k$ that are not already in the list L. Those structures are updated by two helper procedures LowerCount and RaiseCount.

LowerCount(s)
1 $count[s] \leftarrow count[s] - 1$
2 $history[s] \leftarrow t$
3 **if** $count[s] = 0$
4 **then** $unused \leftarrow unused \cup \{s\}$

This procedure is called every time a string is removed from L. A variable t used in line 2 gives the current iteration number. It is the same global variable as defined in Section 3.1. The *count* table is updated and string added to *unused* when it no longer appears on the list.

RaiseCount(s)
1 **if** $count[s] = 0$
2 **then** $unused \leftarrow unused \setminus \{s\}$
3 $count[s] \leftarrow count[s] + 1$
4 $history[s] \leftarrow t$

The RaiseCount works analogically to LowerCount and is called every time s is added to the list. Actually the structures *count*, *history* and *unused* are connected with the current solution ω and every change of those structures transforms it to another solution ω'. However, no distinction of this fact will be explicitly shown.

Functions used to define list in the last section are directly mapped to the structures used in pseudo codes. However the value of $prev[x]$ and $next[x]$ is a reference to the other element of the list or constant NULL when no such element exist. The value of $first[r]$ and $last[r]$ for range $r = (e_1, e_2, \ldots, e_n)$ are elements e_1 and e_n respectively. Every insertion of a new range r to the list that encodes the solution is performed with the help of following procedure.

```
InsertRange(r, (b, a))
1  ListInsert(r, (b, a))
2  InsertForced((key[first[r]], b))
3  InsertForced((key[last[r]], a))
4  x ← first[r]
5  while x ≠ next[last[r]]
6        do RaiseCount(key[x])
7           x ← next[x]
```

The pair (b, a) is a position where r should be inserted. Procedure ListInsert$(r, (b, a))$ is a list insertion procedure, it inserts r between b and a and takes care of updating list structures. After inserting the forced strings that can appear on the edges of range r in lines 2 and 3 the usage count is updated in lines 4–7. The forced strings are inserted using procedure presented below.

```
InsertForced((b, a))
1  if b ≠ NULL and a ≠ NULL
2     then for every s such that Forced(s, key[b], key[a]) in order of occurrence
3               do y ← ListNewElement(s)
4                  RaiseCount(s)
5                  ListInsert(y, (b, a))
6                  b ← y
```

It inserts every string s that is a forced by placing strings $key[b]$ and $key[a]$ next to each other to the list L. Procedure ListNewElement is responsible for allocating a memory to hold a list element x with $key[x] = s$.

Removal of range r from a list is performed by a procedure DeleteRange. When the boolean argument *patch* is true then forced strings that can appear after removal of r are inserted into the list.

DeleteRange$(r, patch)$
1 $(b, a) \leftarrow (prev[first[r]], next[last[r]])$
2 $x \leftarrow first[r]$
3 **while** $x \neq next[last[r]]$
4 **do** LowerCount$(key[x])$
5 $x \leftarrow next[x]$
6 ListRemove(r)
7 **if** *patch* = TRUE
8 **then** InsertForced$((b, a))$

Procedure ListRemove removes a range r from list L and updates the list structures appropriately.

The purpose of insert heuristic is to select one string from $\mathcal{S}_k$ and introduce it on some position in the list that encodes the solution. Not all string from $\mathcal{S}_k$ can be selected for insertions. The low-level heuristics are controlled by two sets — set S and interval N. Set S is set of allowed strings. If $S = \mathcal{S}_k$ then all oligonucleotides are taken. If $S = \varnothing$ then all oligonucleotides are filtered and never any of them is selected for operation. Interval N is a subset of natural numbers and controls the frequency of using particular string. The string s can be included only if last iteration when it was changed is in N. The last iteration is determined using *history* structure. From those two sets the set A of allowed strings is determined.

$$A = \{s \in S : (t - history[s]) \in N\} .$$

The core of insert low-level heuristics is a procedure SbhInsert. It takes a string s and position (b, a) and inserts s between b and a on the list.

SbhInsert$(s, (b, a))$
1 InsertRange(ListNewElement$(s), (b, a)$)

This operation uses InsertRange procedure described earlier. Actually there is number of different insert heuristics. They differ in selection of inserted string and destination position. The first one is a $\diamond\mathrm{TI}_d^{N,S}$ low-level heuristic. The inserted string is selected using parameters N and S i.e., the set A is determined first and then set of unused strings U^A. From this set one of string is taken at random. The allowed position is determined using parameter d. The set of positions P^d is found and one of the position selected at random.

$\diamond\mathrm{TI}(A, d)$
1 $s \leftarrow$ any string from U^A chosen at random
2 $(b, a) \leftarrow$ any position from P^d chosen at random
3 SbhInsert$(s, (b, a))$

The second insert heuristic is $\triangle\mathrm{TI}_d^{N,S}$ and differs from the previous one in selection of the destination position. Now all positions are tried and on every possible position the randomly selected string s is inserted. The position which gives the greatest increase in $f_a(\omega)$ is selected.

$\triangle\mathrm{TI}(A, d)$

1 $s \leftarrow$ any string from U^A chosen at random
2 **for** every $(b, a) \in P^d$
3 **do** execute the SbhInsert$(s, (b, a))$ that results in greatest $f_a(\omega)$.

The third insert heuristic chooses destination position at random and tries to insert every available string there.

$\triangledown\mathrm{TI}(A, d)$

1 $(b, a) \leftarrow$ any position from P^d chosen at random
2 **for** every $s \in U^A$
3 **do** execute the SbhInsert$(s, (b, a))$ that results in greatest $f_a(\omega)$.

Finally, the $\mathrm{TI}_d^{N,S}$ insert heuristic checks every possible combination of inserted string and destination position and uses the best one.

$\mathrm{TI}(A, d)$

1 **for** every $(s, (b, a)) \in U^A \times P^d$
2 **do** execute the SbhInsert$(s, (b, a))$ that results in greatest $f_a(\omega)$.

Each of those insert heuristics has quite different effect. The N and S parameters controls the domain of the search for this particular heuristic. The parameter d controls the span of changes to the existing solution. Low values imply widespread changes and new solution can differ much. High values destroys only weak overlaps. Random versions of the insert heuristic have the advantage that they can generate every possible solution. However, the search process can take significantly longer. The $\mathrm{TI}_d^{N,S}$ heuristic can find the good solutions very fast but it is not guaranteed that this solution is optimal one.

In the implementation of above algorithms besides the actual insertion methods also the simulation of insertion is implemented. The simulation methods does not change the solution and only compute the change in auxiliary and objective search function that would occur after actual insertion. This way the actual move in above algorithms can be found. The other alternative would be to copy the entire solution, perform the insertion and compare the objective values. The former method is more complicated but much faster.

The purpose of the delete heuristic is to remove some range r from the list. The core method that do this is SbhDelete. Deleted ranges are selected in two ways. They can be selected using set of clusters $R_c^{A,D}$ or set of minimal ranges $R_m^{A,d}$ defined in the Section 4.2. There are four different delete heuristics: $\mathrm{TD}^{N,S,d}$, $\mathrm{TD}^{N,S,D}$, $\diamond\mathrm{TD}^{N,S,d}$ and $\diamond\mathrm{TD}^{N,S,D}$. The first two tries to remove every possible range and selects the move which gives the greatest $f_a(\omega)$. They differ only by type of parameters in upper index. This is enough to determine which set is used i.e., $R_c^{A,D}$ in the first case and $R_m^{A,d}$ in the second. The last two delete heuristics choose the removed range at random.

SbhDelete(r)

1 DeleteRange(r, TRUE)

This procedure calls helper method DeleteRange with parameter *patch* = TRUE that causes to generate forced strings after removal and insert it to the list. The parameter d in delete heuristic is important. It determines how much the search should be diversified. The heuristic $\mathrm{TD}^{\mathbb{N},\mathcal{S}_k,[1,1]}$ is similar to the delete clusters heuristics from [8].

The move low-level heuristic selects some range $r = (e_1, e_2, \ldots, e_n)$, removes it from the old position in the list and inserts it on the new position on the list. The possible position are determined by set P^d with the threshold d. However, not all of them can be always used. Only the positions that are not coincident with r can be used i.e., for position (b, a) the condition $e_i \neq b \wedge e_i \neq a$ must hold, $1 \leq i \leq n$. The possible ranges can be selected in two ways. Either using set $R_c^{A,D}$ or set $R_m^{A,d}$. In the first case the move heuristic is denoted as $\mathrm{TM}_d^{N,S,D}$ and in the second case as $\mathrm{TM}_d^{N,S,d}$. The $\mathrm{TM}_d^{N,S,D}$ or $\mathrm{TM}_d^{N,S,d}$ low-level heuristics tries every combination of moved range and destination position and chooses the one that gives the greatest increase in $f_a(\omega)$. Similarly as in insert heuristic the randomised versions are implemented also e.g., ${}^{\triangle}\mathrm{TM}_d^{N,S,D}$ that selects range at random, ${}^{\triangledown}\mathrm{TM}_d^{N,S,D}$ that selects destination position at random and ${}^{\diamond}\mathrm{TM}_d^{N,S,D}$ that selects both moved range and destination position at random. They all use a common core procedure SbhMove.

SbhMove$(r, (b, a))$
1 DeleteRange$(r,$ TRUE$)$
2 InsertRange$(r, (b, a))$

It uses procedure DeleteRange with parameter *patch* = TRUE to remove the range from the list. After that this range is inserted on the destination position.

Swap heuristic is a combination of delete and insert heuristics. It chooses one of the ranges and exchanges it with some unused string. This is a combination of insert and delete but without the temporary decrease in auxiliary and objective search value. This can be important for hyper-heuristics because they might not foresee such moves. The removed range can be taken from three sets: cluster ranges set $R_c^{A,D}$, forced ranges set $R_f^{A,D}$ or minimal ranges set $R_m^{A,d}$. The forced ranges set can be used here because after the removal of selected range immediately follows insertion, no forced strings are generated. The inserted string is chosen from unused set $U^{A'}$ with allowed set A' constructed in the same way as A but from the interval of iterations N' and the set of strings S'. The swap heuristic that checks every combination of elements from $R_c^{A,D}$ and $U^{A'}$ is denoted as $\mathrm{TS}_{N',S'}^{N,S,D}$. Using the set $R_f^{A,D}$ results in swap heuristic $\mathrm{TS}_{N',S'}^{\dagger N,S,D}$ and using $R_m^{A,d}$ in $\mathrm{TS}_{N',S'}^{N,S,d}$. All those swap heuristics uses the procedure SbhSwap as a core algorithm.

SbhSwap(r, s)
1 $(b, a) \leftarrow (prev[first[r]], next[last[r]])$
2 DeleteRange$(r,$ FALSE$)$
3 InsertRange$($ListNewElement$(s), (b, a))$

In this case the DeleteRange procedure is called with parameter *patch* = FALSE because the inconsistency that appears is suppressed by the inserted string. Similarly to insert and move heuristic, the randomized variants are also present e.g., ${}^{\triangle}\mathrm{TS}_{N',S'}^{N,S,D}$ with randomly chosen removed range, ${}^{\triangledown}\mathrm{TS}_{N',S'}^{N,S,D}$ with randomly chosen inserted string and ${}^{\diamond}\mathrm{TS}_{N',S'}^{N,S,D}$ with both elements chosen at random.

Heuristic	Best Move	Random Source	Random Destination	Completely Random
Insert	TI	$\triangle$TI	$\triangledown$TI	$\diamond$TI
Delete	TD	–	–	$\diamond$TD
Move	TM	$\triangle$TM	$\triangledown$TM	$\diamond$TM
Swap	TS	$\triangle$TS	$\triangledown$TS	$\diamond$TS

Tab. 4.1: Summary of low-level heuristics. Each of those can be parametrised further by sets of allowed strings and modified distances.

The summary of low-level heuristics is presented in Table 4.1. For clarity the parameters that characterise those heuristics are not shown. Every delete and move low-level heuristics can use set $R_c^{A,D}$ or $R_m^{A,d}$ so there is one additional alternative for those heuristics. The swap heuristics can use $R_c^{A,D}$, $R_f^{A,D}$ or $R_m^{A,d}$ sets so there are three alternatives for every swap low-level heuristic.

4.4 Obtaining Parameters

This section describes how the sets that are parameters for low-level heuristics can be obtained. Those sets are determined on demand when needed. This is required for exhaustive variants of low-level heuristics. They obtain the list of all possible moves and tries every combination. The random variants also uses the lists of possible moves to allow the probability distribution of choosing the next move to be uniform.

The DISTANCE procedure is a helper function. It gives the distance determined by the function distance(u, v) between the strings that are the keys of elements x and y placed on the list. On the edges of the list the distance is defined to be ∞.

DISTANCE(x, y)
1 **if** x = NULL or y = NULL
2 **then return** ∞
3 **return** distance$(key[x], key[y])$

In this section the ranges are often identified as two elements x and y on the list. They denote the range constructed from elements $(x, next(x), \ldots, y)$.

The procedure FORCED shown below checks the predicate Forced(s, u, v) for every string s that is a key of any element in the range x–y. The u in Forced(s, u, v) is a key of element preceding this range and v in Forced(s, u, v) is a key of element succeeding this range.

```
FORCED(x, y)
 1  if prev[x] = NULL or next[y] = NULL
 2     then return TRUE
 3  d ← DISTANCE(prev[x], x) + DISTANCE(y, next[y])
 4  z ← x
 5  while z ≠ y and next[z] ≠ y
 6        do d ← d + DISTANCE(z, next[z])
 7           z ← next[z]
 8  if d = DISTANCE(prev[x], next[y])
 9     then return TRUE
10  return FALSE
```

The fact that successive strings u and v on the list that encodes the solution are placed as close as possible implies that every pair of elements of the list induces some unique string of length at most $2k$. Placing the third string s between them certainly cannot make the combined string shorter. If it expands then predicate Forced(s, u, v) does not hold. This predicate only holds in the case that the length of the combined string is not changed and this is the main idea of above algorithm. The condition in line 8 is only true if range x–y does not expand the combined string and predicate Forced(s, u, v) holds for every string s in this range.

The pseudo code of algorithm for determining the cluster ranges set $R_c^{A,D}$ is given below. It takes as a parameters sets A and D and returns a set of ranges C that are not forced d-clusters for $d \in D$. As before, range is identified as a pair of elements of the list.

```
CLUSTERS(A, D)
 1  C ← ∅
 2  for every d ∈ D
 3        do x ← head
 4           c ← (x, x)
 5           append ← TRUE
 6           match ← FALSE
 7           while x ≠ NULL
 8                 do last[c] ← x
 9                    if key[x] ∉ A or DISTANCE(x, next[x]) < min D
10                       then append ← FALSE
11                    if next[x] = NULL or d < DISTANCE(x, next[x])
12                       then if append = TRUE and (match = TRUE or d = 0) and
                                        ¬FORCED(first[c], last[c])
13                               then C ← C ∪ {c}
14                            c ← (next[x], next[x])
15                            append ← TRUE
16                            match ← FALSE
17                    if d = DISTANCE(x, next[x])
18                       then match ← TRUE
19                    x ← next[x]
20  return C
```

The main **for** loop in lines 2–19 traverses every d that is of interest. In each iteration every allowed d-cluster is added to C. Range c holds a currently constructed d-cluster. The boolean variable *allowed* is true if every element of the current cluster c belongs to the set A and every distance in the current cluster c belongs to D. Those are constraints imposed by sets A and D on $R_c^{A,D}$. The boolean variable *match* is true if any of the distances in the current range c is equal to d. This is a condition for cluster to be interesting (see Section 4.2). The cluster is added to C in line 13 but before that the conditions for cluster feasibility are checked in line 12. Time complexity of this procedure is $\Theta(|D||L|)$. Because $D = O(k)$ and $|L| = O(n-k) = O(n)$ this can be expressed as $O(k\,n)$. The equality $O(n-k) = O(n)$ holds because $k = O(n)$ and $O(n-n) = O(n)$.

The check for cluster c for not being forced in line 12 is required for cluster ranges set $R_c^{A,D}$. However, if it is dropped then this algorithm generates forced cluster ranges set $R_f^{A,D}$. Time complexity of determining $R_f^{A,D}$ is the same as for $R_c^{A,D}$ and equal to $O(k\,n)$.

The minimal ranges set $R_m^{A,d}$ inherits quite different structure than two sets analysed previously. The separate procedure MINIMAL is used to obtain it. The actual procedure is initialised by the following pseudo-code.

```
MinimalInitialise(x, A)
1  cuts ← 0
2  disallowed ← 0
3  if x ≠ NULL and key[x] ∉ A
4      then disallowed ← disallowed +1
5  if x ≠ NULL and d > Distance(x, next[x])
6      then cuts ← cuts +1
7  return (x, cuts, disallowed)
```

During traversal of the list the quadruple $(x, y, \mathit{cuts}, \mathit{disallowed})$ is updated. The idea of algorithm is to move the range x–y from the beginning to the end of the list and stop every time the minimal not forced set appears. The first variable x in the quadruple is the beginning of this moved range and second variable y is the end of the moved range. The counter *cuts* determines how many times the current range window conflicts with the minimal distance condition $d \le \text{mincut}(r)$ in definition of $R_m^{A,d}$. The second counter *disallowed* determines how many strings in current range are not in set of allowed strings A. Procedure MINIMALINITIALIZE takes care of appropriate initialisation of those variables. After initialisation the actual procedure for determining $R_m^{A,d}$ is executed.

```
MINIMAL(A, d)
 1  R ← ∅
 2  x ← head
 3  (y, cuts, disallowed) ← MINIMALINITIALISE(x, A)
 4  while x ≠ NULL
 5        do while FORCED(x, y)
 6                do (y, cuts, disallowed) ← MOVELAST(y, cuts, disallowed, A)
 7           while x ≠ y and ¬FORCED(next[x], y)
 8                do (x, cuts, disallowed) ← MOVEFIRST(x, cuts, disallowed, A)
 9           if cuts = 0 and disallowed = 0 and ¬FORCED(x, y)
10             then R ← R ∪ {(x, y)}
11           if next[y] = NULL
12             then return R
13           (y, cuts, disallowed) ← MOVELAST(y, cuts, disallowed, A)
14           (x, cuts, disallowed) ← MOVEFIRST(x, cuts, disallowed, A)
15  return R
```

In lines 5–6 the right end of the range is expanded until the range stops being forced. When that starts to be true then for condition of range being minimal the left end must be moved until this range would become forced again. It is performed in lines 7–8. If encountered range satisfies all the condition then it is added to the returned set R in line 10. The assignments in lines 13 and 14 guarantees the method progress. This algorithm works because of fact that when some range is forced then contracting it does not change the predicate Forced for this range. Similarly, expanding not forced range never can make this range to become forced. The procedure uses two helper procedures. First one moves the left end of the range window.

```
MOVEFIRST(x, cuts, disallowed, A)
1  if key[x] ∉ A
2     then disallowed ← disallowed −1
3  if d > DISTANCE(prev[x], x)
4     then cuts ← cuts −1
5  x ← next[x]
6  return (x, cuts, distance)
```

The main goal if this procedure is to appropriately update variables *cuts* and *disallowed*. The procedure MOVELAST is analogical and moves the right end of the range window.

```
MOVELAST(y, cuts, disallowed, A)
1  y ← next[y]
2  if key[y] ∉ A
3     then disallowed ← disallowed +1
4  if d > DISTANCE(y, next[y])
5     then cuts ← cuts +1
6  return (y, cuts, distance)
```

The procedures MinimalInitialise, MoveFirst, MoveLast perform the steps in $\Theta(1)$ time. The total time complexity of finding the minimal cluster sets $R_m^{A,d}$ is $\Theta(|L|) = O(n)$.

The two more sets are needed for low-level heuristics. The set of allowed positions P^d is determined with the help of following procedure.

```
Positions(d)
 1  x ← NULL
 2  y ← head
 3  P ← ∅
 4  while y ≠ NULL
 5        do if d ≤ Distance(x, y)
 6              then P ← P ∪ {(x, y)}
 7                   x ← y
 8                   y ← next[y]
 9  P ← P ∪ {(x, y)}
10  return P
```

List that encodes the solution is traversed from the beginning to the end. Every position with appropriate distance between two strings on this position is added to the returned set P. The time complexity of this procedure is $\Theta(|L|) = O(n)$. The set of unused strings U^A can be determined by this straightforward algorithm.

```
Unused(A)
1  U ← ∅
2  for every s ∈ unused
3        do if s ∈ A
4              then U ← U ∪ {s}
5  return U
```

The time complexity of Unused procedure is bounded by size of the list: $\Theta(|L|) = O(n)$.

4.5 Implementation

The algorithms described in previous sections are implemented as a software applications. All the applications developed are written in Java language. The reason for using this programming language are stated in the description of hyper-heuristic framework in Appendix A and the applications for solving the SBH are clients of this hyper-heuristic library.

There are two different heuristic solvers for the SBH problem implemented. First of them is a program for the tentative method described in Section 4.1. The other is an extended method described in the rest of this chapter, which is also the main part of this work. Tentative method uses the acronym *set* and the extended method uses the acronym *tree*. Java package where the first method is placed is called `wojtek.sbh.hyper.set` and package where the second method is placed is called `wojtek.sbh.hyper.tree`. The *set* acronyms inherits its name from the trash set described in [8]. The acronym *tree* comes from interesting structure of tree formed by the d-clusters as described in Section 4.2.

The implementation of the *set* method is oriented around a set of the unused oligonucleotides which is a collection of `java.lang.SortedSet` type and a set of oligonucleotides in the solution which is a collection of `wojtek.collections.FlexList` type. The latter is a list implementation that provides more flexibility of additions and removals than `java.util` package collections. There is a number of low-level heuristics that implements the `LowLevel` interface from package `wojtek.hyper`. Every heuristic implements methods to simulate the move and to accept the move. During the simulation every possibility is checked and everything prepared for accepting the changes. To give the time complexity of the implemented algorithms two variables are introduced — the size n_s of the spectrum $\mathcal{S}_k$ in solved problem $n_s = |\mathcal{S}_k|$, and the maximal number of unique oligonucleotides that can be present in the solution $n_o = \min\{n-k+1, n_s\}$. The insertion heuristic SI is handled by the class `InsertHeuristic`. During the trial phase every combination is checked in $O(1)$ time and an overall time complexity is $O(n_s\, n_o)$. When solution is accepted by the `commit()` method a list of clusters must be updated and the time complexity of this phase is $O(n_o)$. The move SM and move cluster SMC low-level heuristics are implemented in the `MoveHeuristic` and `MoveClusterHeuristic` classes. Both of them have time complexity $O(n_o^2)$ of the simulation phase and $O(n_o)$ time complexity of the acceptance phase. Delete SD and delete clusters SDC low-level heuristics are handled by the `DeleteHeuristic` and `DeleteClusterHeuristic`. In this case every phase takes $O(n_o)$ time i.e., every possible oligonucleotide and every possible cluster must be checked. The low-level heuristics do not change the solution during the simulation phase but only evaluates the auxiliary and objective value changes. Solution is changed only in acceptance phase on `commit()` method call.

The structure of the more advanced *tree* method is quite different than in the *set* method. The low-level heuristics in *set* method make a list of every possibility on their own i.e., iteration over set of possible moves is implemented in the low-level heuristics. Low-level heuristics in *tree* method obtains a list of every possible cluster, range, position or oligonucleotide before actual simulation phase. The set of unused oligonucleotides U^A is provided by the `Solution` class which is a class that holds a solution. Other mentioned sets are accessible through the special `Clusters` class. This class is at the same time a special list implementation that holds a list L of the oligonucleotides that forms a solution. This class implements the algorithms described in Section 4.4 with the same time complexity as defined there. In this representation a number of list elements is of $O(n)$ and does not depend on $\mathcal{S}_k$. The `Solution` class holds a cache of the length of the solution and usage of the every oligonucleotide in order to improve efficiency of the implemented algorithms.

Low-level heuristics used in *tree* method implements interface `LowLevel` to be used in hyper-heuristic search. Similarly as in *set* method two phases are distinguished i.e., simulation and acceptance phase. The insertion low-level heuristic is implemented in class `InsertHeuristic` from package `wojetk.sbh.hyper.tree`. During the simulation phase every version of this heuristic i.e., TI, $\triangle$TI, $\triangledown$TI, $\diamond$TI obtains the sets P^d and U^a so their time complexity is at least $O(n)$ in this case. Every simulation of the insertion or the actual insertion must update the forced oligonucleotides. It follows from the Definition 4.1 of forced ranges that at most $k-1$ strings can be induced that way (for a single position in L). Thus the time complexity of this update is $O(k)$. Because $k = O(n)$ the time complexity of simulation phase of $\diamond$TI is $O(n)$. Heuristics $\triangle$TI and $\triangledown$TI performs

$O(n)$ trials and their time complexity is $O(n\,k)$. The TI low-level heuristics additionally tries every combination and its time complexity is of $O(n^2\,k)$. The acceptance phase of those low-level heuristics is $O(k)$ in every case.

The delete low-level heuristic is implemented in the `DeleteHeuristic` class. However, two more derived classes are used depending on the set of ranges used. If minimal set of ranges $R_m^{A,d}$ is used then class `DeleteMinimalHeuristic` is instantiated. If set of clusters $R_c^{A,D}$ is used then class `DeleteClusterHeuristic` is instantiated. Both delete heuristics TD and $\diamond$TD make a list of those sets. This step is performed in $O(n)$ time when using $R_m^{A,d}$ and in $O(k\,n)$ time when using $R_c^{A,D}$ as described in Section 4.4. Randomised version $\diamond$TD makes one trial that can take $O(n)$ time. So the time complexity of this step is dictated by the choice of ranges set. The exhaustive version of the delete heuristic TD performs $O(n)$ or $O(k\,n)$ trials depending on the ranges set used. However, it turns out that the amortised time complexity is $O(k\,n)$ and $O(k^2\,n)$ respectively. This follows from the total number of oligonucleotides in $R_m^{A,d}$ and $R_c^{A,D}$ sets i.e., every oligonucleotide in this set can be removed exactly once from L. The factor k comes from induction of forced oligonucleotides. Acceptance phase takes $O(n)$ time in every case because size of the range is $O(n)$ and every string from this range must be removed from L.

Move low-level heuristics uses sets $R_m^{A,d}$, $R_c^{A,D}$ or $R_f^{A,D}$ and P^d. Size and operations complexity on sets $R_c^{A,D}$ and $R_f^{A,D}$ are exactly the same, $O(k\,n)$. Obtaining set P^d takes $O(n)$ time. Thus the preparation for simulation phase is done in $O(n)$ time in case of using $R_m^{A,d}$ and $O(k\,n)$ time when using $R_c^{A,D}$ or $R_f^{A,D}$. The class structure is similar to the delete low-level heuristic i.e., the class `MoveHeuristic` has two derivative classes: `MoveMinimalHeuristic` and `MoveClusterHeuristic`. Only one of them is instantiated depending on the ranges set used. The simulation of a move operation is implemented to be done in $O(k)$ time. When using the full search of destination position i.e., in case of TM or $\triangledown$TM heuristic this complexity must be multiplied by $O(n)$, a number of positions available. When using the full search of range moved this i.e., in case of TM and $\triangle$TM, this must be multiplied further by the size of ranges set — $O(n)$ or $O(k\,n)$. Acceptance phase is implemented using the algorithm that takes $O(n)$ time in every case.

Swap low-level heuristics removes one of the ranges from $R_m^{A,d}$, $R_c^{A,D}$ and inserts some oligonucleotide from U^A. The former sets are of size $O(n)$ or $O(k\,n)$. The later set is of size $O(n)$. Similarly to delete and move low-level heuristic there is general class `SwapHeuristic` with two derivative classes `SwapMinimalHeuristic` and `SwapClusterHeuristic` instantiated depending on the ranges set used. If only one inserted oligonucleotide is checked (heuristics $\triangledown$TS or $\diamond$TS) then amortised time complexity for set $R_c^{A,D}$ and heuristic $\triangledown$TS is $O(k^2\,n)$. For every other case the time complexity is $O(k\,n)$. When every possible inserted oligonucleotide is checked then those complexities are $O(k^2\,n^2)$ and $O(k\,n^2)$ respectively with TS instead of $\triangledown$TS in the first case. Acceptance phase takes $O(n)$ time in every case.

This time complexity analysis shows that extended *tree* method is slightly slower than the *set* method. However, the choice of parameters drastically changes the size of ranges, positions and unused sets. Thus the above upper bounds are very rough. Additionally, some of the above algorithms can be improved for some subsets of specified low-level heuristics. Once the desired set of low-level heuristic is formulated then many optimisations are possible e.g., checking for contention in $R_c^{A,D}$, $R_f^{A,D}$ $R_m^{A,d}$, P^d, U^A instead explicitly listing them or better management of solution length. For purposes of the anal-

ysis of hyper-heuristic search methods the current implementation is satisfactory.

The implemented software can be executed in two ways. For both methods there exists a batch command line utility and a visual analysis tool. The examples of usage and demo of the visual tool are presented in Appendix B. The *tree* method requires a sophisticated low-level heuristics scheme and the one based on the configuration files is implemented.

Chapter 5

Experimental Results

A series of computational experiments was conducted. The results are summarised in this chapter. In the first two Sections 5.1 and 5.2 the data sets that were used for tests are described in details. Next Section 5.3 describes the process of setting parameters for low-level heuristic. Results of the experiments from the point of view of low-level heuristics are presented in Section 5.4. They conclude how different low-level heuristics work. The analysis of results in Section 5.5 is focused on hyper-heuristics and presents the differences between them. Section 5.6 describes results for the biological data sets and they relevance.

5.1 Biological Data Sets

The first two data sets come from the real biological data. The first one was used to test methods in [11] and is publicly available on web site [1]. This data set is divided to three categories. Two of them are relevant for this work and the third contains instances to a different problem. The first relevant category contains instances with positive and negative errors percentage of 5%, 10% and 20%. The length n of examined sequences varies between 200 and 600 and oligonucleotides length is always equal to $k = 10$. The instances are generated from real DNA sequences that code human proteins coming from GenBank database. Spectra from fragments of length n of those sequences are created and experimental errors are artificially introduced. Errors are added by removing some of the oligonucleotides from the spectra and by adding some new random oligonucleotides to the spectra. The number of removed or added oligonucleotides is dictated by the error percentage. DNA sequences from the first category are specially selected not to contain any repetitions. The sequences from the second category contains repetitions in the original sequences.

The data set used in articles [14, 9, 10] is available at web site [2]. The two categories of sets from this data set are used. The first one is derived from real DNA sequences with artificial errors introduced. Errors in this data set are created in the same way as in [1] i.e., by removing existing and adding new oligonucleotides. Every instance contains 20% of positive and 20% of negative errors. The length of the sequences n varies from 109 to 509. Oligonucleotides length is always $k = 10$. Second category of tests in this data set differs in the origin of generating sequences. In the first category of sets sequences come from real DNA sequences coding human proteins from GenBank database. Instances from

the second category of tests use randomly generated sequences. In either case negative errors coming from repetitions are not present. There exist a third category with such instances but it is not used in this work.

5.2 Benchmark Data Set

Biological data sets [1] and [2] are important because of the biological nature of SBH problem, the main reason why this problem was formulated. However, from results in Section 5.6 it is apparent that those data sets are highly correlated and are not good benchmark for testing different low-level heuristics and hyper-heuristics. The solutions for those data sets are very dense and homogeneous. Thus a data set with more random instances is prepared specially for purpose of testing hyper-heuristics. The execution of instances from this data set gives more random results that can be verified statistically.

Two random instances generator are implemented in order to generate new instances. The first random generator is designed to simulate errors that can appear during the hybridisation experiment (class `ExperimentGenerator`). It is capable of generating instances very close to the multispectrum of some randomly generated sequence s. The other random instances generator (class `RandomGenerator`) is designed to abstract from biological nature of the problem and generates totally random instances without any correlation between oligonucleotides.

The first generator is helpful in comparing exact methods and testing correctness of the hyper-heuristic algorithms. Small data instances that can be verified by hand are generated for these purposes. The errors in random generator that simulates the hybridisation experiment are controlled by two parameters, p and q that are used to determine the amount of positive and negative errors respectively. The randomly generated sequence s of length n is decomposed to $n - k + 1$ substrings of length k, i.e., the multispectrum $\mathcal{M}_k(s)$ is determined. For every such substring negative and positive errors are generated independently. Negative errors are generated by omitting every oligonucleotide with probability q. However, $\mathcal{M}_k(s)$ is a multiset and substring can belong to this multiset more than once. Removed oligonucleotide can appear in final set despite the omission. Moreover, omitted oligonucleotide can be generated from a positive error and appear in the final set again. The positive error probability p used to generate positive errors is a probability of a single nucleotide flip. From every string of length k there are generated $\binom{k}{e}$ strings with e errors for $e = 1, 2, \ldots k$. Every such string is included in the final set with probability equal to p^e. One string can lead to $2^k - 1$ new strings but this is very unlikely.

Benchmark data set is created with the help of a second generator, random instances generator implemented in class `RandomGenerator`. This generator is specified by three arguments — length of the sequence n, length of the oligonucleotides k and number of oligonucleotides in the spectrum $|\mathcal{S}_k|$. Every instance is created by generating $|\mathcal{S}_k|$ random oligonucleotides of length k. As a result twenty one tests were generated. The parameters used to generate them are given in table 5.1. This data set was generated iteratively. A large number of instances with parameter n in the range 50–800, k in the range 4–12 and $|\mathcal{S}_k|$ in the range 50–4000 were generated. When created they were solved with the preliminary set of low-level heuristics and every possible hyper-heuristic. Fraction of

Test	n	k	$\lvert\mathcal{S}_k\rvert$	Test	n	k	$\lvert\mathcal{S}_k\rvert$	Test	n	k	$\lvert\mathcal{S}_k\rvert$
1	60	5	93	8	300	6	432	15	400	10	1500
2	60	6	148	9	300	6	556	16	500	8	991
3	100	5	143	10	300	8	400	17	500	9	500
4	100	6	246	11	300	10	400	18	500	10	500
5	100	7	297	12	400	8	499	19	500	10	1997
6	200	6	291	13	400	8	1962	20	500	12	500
7	200	8	398	14	400	10	500	21	800	8	991

Tab. 5.1: Parameters of the benchmark data set. There are 21 instances, each one of desired sequence length n with $|\mathcal{S}_k|$ oligonucleotides of length k.

generated instances that gave very homogeneous solutions for different configurations was discarded and new instances with parameters similar to the tests with varying results were generated. This procedure was repeated twice. At the end twenty one most diverse tests among every test generated of different sizes and solutions characteristic were selected to compose the benchmark data set.

Large difference between this data set and the biological data sets is the size of the spectrum. In biological data sets every spectrum has fixed size of length $n - k + 1$. To diversify the set of possible optimal or sub-optimal solutions the size of the spectra have to be greater than this value. Small sizes of spectra give very uniform results.

5.3 Tuning Parameters

Hyper-heuristics developed in this work require a few parameters to set. Those parameters are problem specific. They depend mainly on the change of the auxiliary objective value at each iteration, running time of the low-level heuristics and solutions neighbourhood. The parameters for acceptance and selection methods are determined empirically with four different sets of preliminary low-level heuristics.

The simulated annealing hyper-heuristic WrMc cooling schedule described by Equation 3.2 is controlled by the initial temperature factor β and number of iterations for which every temperature holds m. For the desired number of iterations and the average run time of the algorithms the best results are obtained for the initial temperature between 0.01 and 0.02. For smaller values of this factor the search is too steady and solutions obtained not diverse enough. For larger values the solutions are worsened with the increasing iteration number i.e., too many worsening solutions are accepted. This value was fixed to 0.015. The value of parameter m gives the most expected results for values between 10 and 13. This allows for significance drop in simulated annealing temperature during the expected run time of the search. This value was fixed to 13.

Tabu search selection method described in Section 3.3 is controlled by one parameter, i.e., number of tabu iterations k for which the tabu active low-level heuristics cannot be used. The preliminary results shows that this parameter is highly dependent on the set of low-level heuristics. However, the best results for which the search does not fall into local minima and obtained solutions are diverse enough are obtained for relatively small

values between 3 and 5. The value for tabu iterations k is set to $\min\{4, |\mathcal{H}|\}$ for each configuration of low-level heuristics.

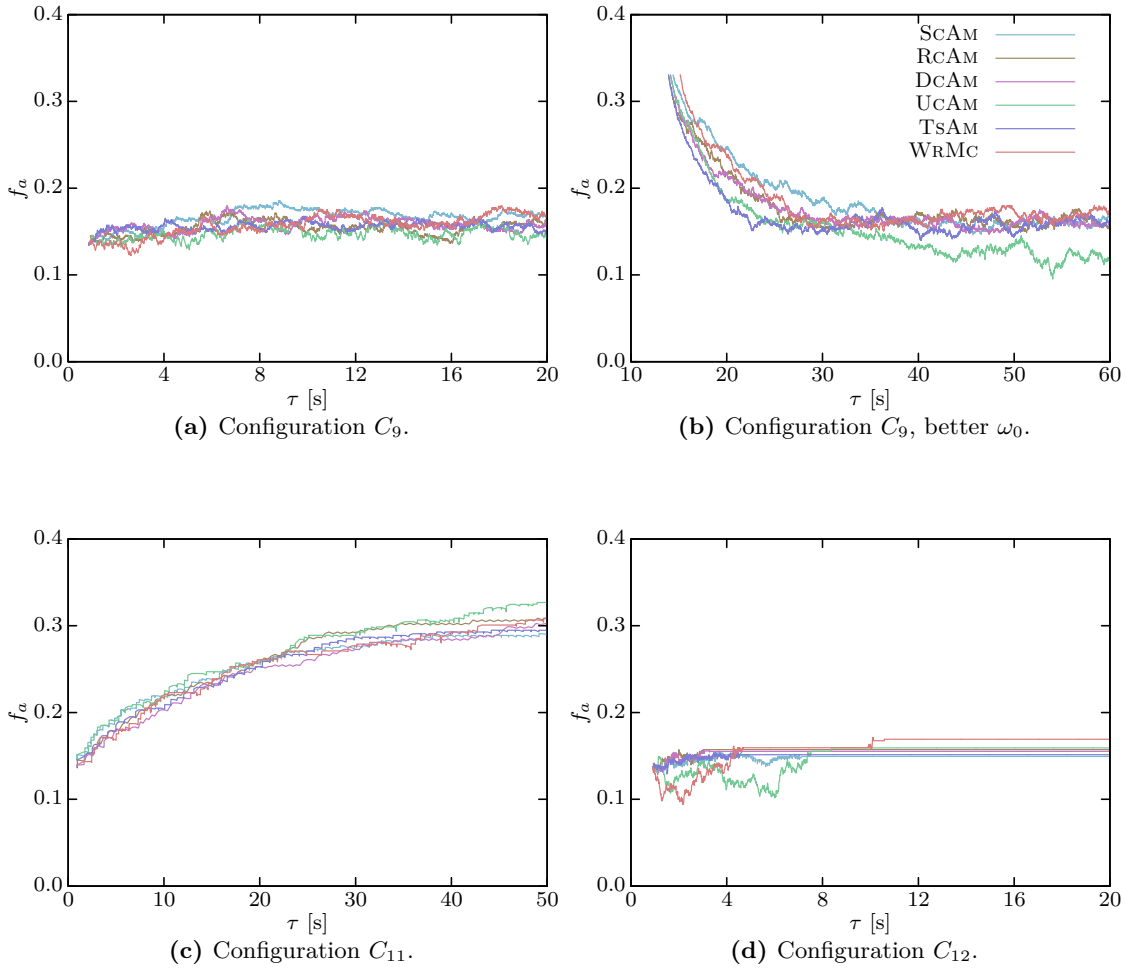
Every choice function selection method described in Section 3.2 uses at least three parameters α, β and γ that are coefficients for choice function c_f in Definition 3.1. They were analysed using DC selection method that is not dependent on any other factor. The γ coefficient is responsible for low-level heuristics rotation in time. Low values leads to great influence of the timer factor and diversification of the search. High values leads to low rotation of the low-level heuristics. Values between 0.1 and 50 were analysed and it turns out that for preliminary sets of low-level heuristics this parameter does not have large influence on the solutions obtained during the search. It was set to 10. For the same selection method DC parameters α and β responsible for rewarding a good performance of single low-level heuristic and pairs of low-level heuristics were checked. Values near the center of allowed range $(0, 1)$ gave a slightly better results than values close to the boundaries. Parameters α and β were set to 0.5.

Two hyper-heuristics in the choice function family requires specifying additional variable parameters to work. Ranked choice RC selection uses parameter k that determines a number of low-level heuristics to be checked in every iterations. It turned out that best results are obtained for k between 2 and 3. Value of k lower than 2 is meaningless because selection method becomes then SC selection. Values greater than 3 lead to very homogeneous solutions. With many sets of low-level heuristics too many checks leads quickly to the local minima which is not overcomed because required moves are discarded. Value of this parameter was fixed to 3. The roulette choice UC selection uses a positive choice function from Definition 3.2. This choice function uses factor η that scales the choice function to the appropriate range. Preliminary analysis shows that the reasonable magnitudes of c_f^+ are obtained for η between 0.001 and 1. Lower values results in c_f^+ very close to 1 for possible values of c_f and values larger than 1 causes c_f^+ to be very sensitive in the change in c_f. This value was fixed to 0.1.

Setting the above parameters can be avoided with the adaptive control mechanisms. However, such mechanisms are not the subject for this work and with the parameters of selection and acceptance methods tuned for the SBH problem the analysis of low-level heuristics is performed.

5.4 Low-level Heuristics

An extensive analysis of different configurations of the low-level heuristics was performed. This section summarises the most significant conclusions. Over 70 low-level configurations with 6 hyper-heuristics was checked against 21 test cases from the benchmark data set. All the tests were conducted on IBM PC with Intel Core2 Duo 3.16GHz processor and 2048MB of physical memory. Machine was running Windows XP operating system and the version of Java runtime environment used was Sun's JRE 6 Update 13. In order to shorten the notation some parameters in low-level heuristics are not written explicitly. For example low-level heuristic $\mathrm{TI}_d^{N,S,D}$ written as $\mathrm{TI}_d^{S,D}$ is assumed to have fixed set $N = (0, \infty)$. Writing it as TI_d^{D} assumes further that $S = \mathcal{S}_k$, i.e., the entire spectrum for particular problem is used. Also for every instance the value $f_a^\dagger$ denotes the maximum auxiliary objective value that was found among set of all tests.



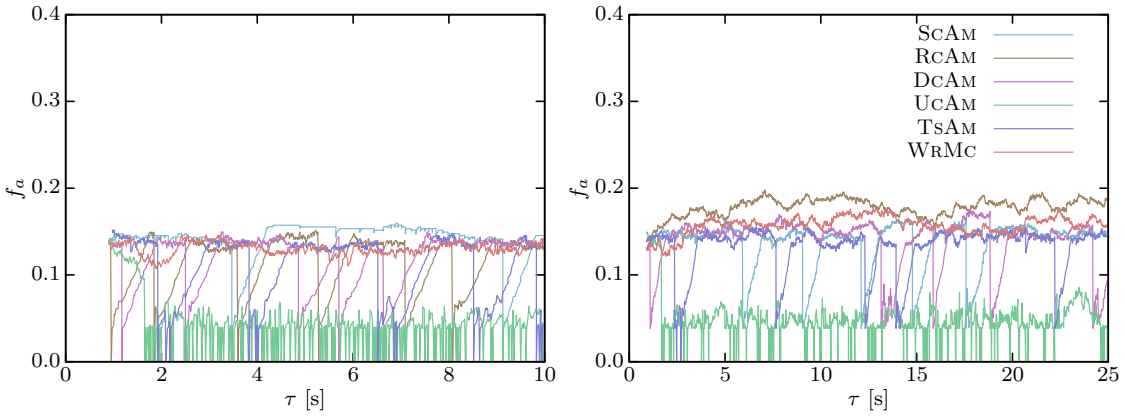
(a) Configuration C_9. **(b)** Configuration C_9, better ω_0.

(c) Configuration C_{11}. **(d)** Configuration C_{12}.

Fig. 5.1: Four different sets of primitive low-level heuristics and their different behaviour. Results are obtained for instance 16 from benchmark data set.

The most basic configuration that is theoretically required in order to reach every possible solution is composed of two random low-level heuristics: insertion of a single oligonucleotide and deletion of a single oligonucleotide. This configuration can be written as $C_9 = \{\diamond\mathrm{TI}_1, \diamond\mathrm{TD}^{[0,0]}\}$. Configurations are identified with capital letter C with the associated subscript. The plot on Figure 5.1a shows the first 20 seconds of the change of auxiliary objective value during the search process for this configuration. Each of the six hyper-heuristics considered behaves almost in the same way. The auxiliary objective value during the search stabilises at approximately $0.4 f_o^\dagger$. Configuration C_9 of low-level heuristics does not lead to good results.

The configuration C_9 on Figure 5.1a starts with an initial solutions constructed by the maximal number of random insertions performed at the beginning of the search. Figure 5.1b shows the same configuration C_9 but with an initial solution that was generated with maximal number of insertions that checked every possibility. Although better values are present at the beginning at the search they eventually drop to the same auxiliary objective value as in random initial solution. The search is biased at the same low value regardless



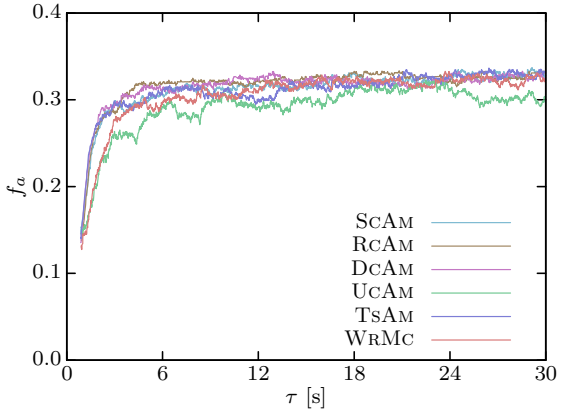
(a) Configuration C_{13} **(b)** Configuration C_{16}

Fig. 5.2: Two hyper-heuristic search results for instance 16 in benchmark data set that gives a very diverse auxiliary objective values.

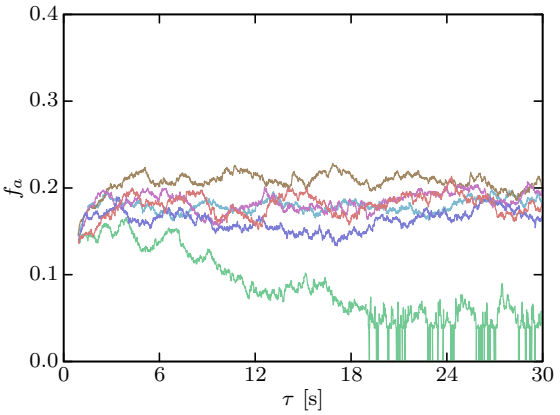
of the initial solution.

Figure 5.1c shows a search result for a little modified C_9. New configuration $C_{11} = \{\diamond\mathrm{TI}_1, \mathrm{TI}_1, \diamond\mathrm{TD}^{[0,0]}\}$ includes an additional insert low-level heuristic that checks every possible combination of inserted oligonucleotide and destination position. This small addition results in great improvement. Hyper-heuristics start to converge to $\approx 0.9 f_o^\dagger$ and it also can be noticed that UcAm is slightly better than others. On Figure 5.1d a configuration $C_{12} = \{\diamond\mathrm{TI}_1, \diamond\mathrm{TD}^{[3,k]}\}$ with modified delete low-level heuristic in C_9 was used to guide the search. This time it is evident that search is blocked at $\approx 0.4 f_o^\dagger$. There is no change in auxiliary objective value. The $\diamond\mathrm{TD}^{[3,k]}$ selects small number of ranges that span on large number of oligonucleotides with large gaps possible. This results in smaller number of choices and in fact solution becomes trapped. Using $\mathrm{TD}^{[3,k]}$ gave very similar results but search stabilised at $\approx 0.7 f_o^\dagger$ for this particular test. However, modifying C_9 to $\{\diamond\mathrm{TI}_1, \diamond\mathrm{TD}^{[1,k]}\}$ leads to very dramatic changes shown on Figure 5.2a. Distance set equal to $[1, k]$ can remove almost every part of this solution. Some hyper-heuristics must accept every choice made even if almost entire solution is destroyed. Hyper-heuristics UcAm, TcAm or ScAm restarts the search periodically. Definitely the worst results gives UcAm in this case.

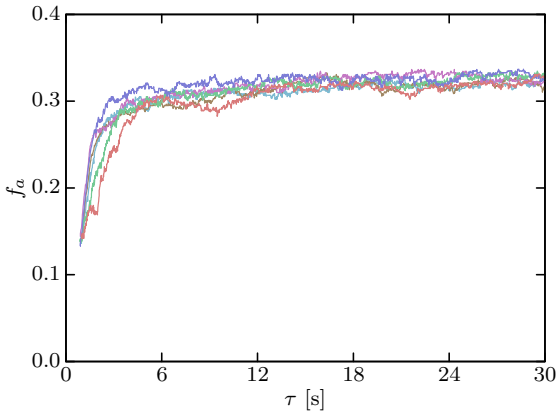
Also different settings for delete low-level heuristics was checked. Configurations $C_{14} = \{\diamond\mathrm{TI}_1, \diamond\mathrm{TD}^1\}$, $\{\diamond\mathrm{TI}_1, \diamond\mathrm{TD}^{[4,\infty],\mathcal{S}_k,1}\}$ or $\{\diamond\mathrm{TI}_1, \diamond\mathrm{TD}^{[14,\infty],\mathcal{S}_k,1}\}$ are not significantly different than results for C_9, i.e., the difference for analysed random instance can not be noticed. However, in some cases limiting deleted oligonucleotides stabilises WrMc slightly. Similarly, dividing low-level heuristics to separate subsets of oligonucleotides from spectrum did not make any changes. Used configuration was $\{\diamond\mathrm{TI}_1^{S_1}, \diamond\mathrm{TI}_1^{S_2}, \diamond\mathrm{TD}^{S_1,[0,0]}, \diamond\mathrm{TD}^{S_2,[0,0]}\}$ where set $S_1 = \{o_1, o_2, \ldots o_{30}\}$ and $S_2 = \{o_{31}, o_{32}, \ldots o_{|\mathcal{S}_k|}\}$ for $o_1, o_2, \ldots, o_{|\mathcal{S}_k|}$ being successive oligonucleotides from spectrum $\mathcal{S}_k$ of analysed instance. Changing C_{14} to $C_{15} = \{\diamond\mathrm{TI}_1, \diamond\mathrm{TD}^2\}$ makes a big difference. The results are becoming significantly different for different types of low-level heuristics. Hyper-heuristics ScAm and WrMc are the best for this configuration and UcAm or TsAm performs very badly. Generally the search is biased at the same auxiliary objective value as C_{14}, $\approx 0.4 f_o^\dagger$. However, rejecting the



(a) Configuration C_{54}



(b) Configuration C_{55}



(c) Configuration C_{57}

Fig. 5.3: Configurations composed of insert, swap and delete low-level heuristics for different ranges sets. Search results are obtained for instance 16 from the benchmark data set.

random behaviour in delete heuristic and using configuration $\{\diamond\mathrm{TI}_1, \mathrm{TD}^2\}$ leads to very good results at value of $\approx 0.8 f_o^\dagger$.

Introducing $\diamond\mathrm{TS}^{[0,k]}$ to configuration C_9 results in periodical decrease of auxiliary search value in almost every hyper-heuristic search. The results of this configuration $C_{16} = \{\diamond\mathrm{TI}_1, \diamond\mathrm{TS}^{[0,k]}, \diamond\mathrm{TD}^{[0,0]}\}$ are presented on Figure 5.2b. It is very similar to configuration C_{13} where distance interval $[0, k]$ was used in delete low-level heuristic instead of swap low-level heuristic. This behaviour is strongly connected with the removed ranged clusters set, i.e., using $\triangledown\mathrm{TS}^{[0,k]}$ instead of $\diamond\mathrm{TS}^{[0,k]}$ leads to significant improvement of every hyper-heuristic search that are biased and stable at the value $\approx 0.8 f_o^\dagger$. Using $\triangle\mathrm{TS}^{[0,k]}$ however, improves only WrAm and RcAm hyper-heuristics. Using $\mathrm{TS}^{[0,k]}$ results in great improvement leading to $\approx 0.9 f_o^\dagger$ for all the hyper-heuristics. Quite similar behaviour is observed for move low-level heuristic. Introducing $\diamond\mathrm{TM}_2^{[0,k]}$ to C_9 causes quite dramatic drop in auxiliary objective value for hyper-heuristics ScAm, TsAm and DcAm. This change does not influence remaining hyper-heuristics much. Using at leas one not random choice, i.e., $\triangle\mathrm{TM}_2^{[0,k]}$, $\triangledown\mathrm{TM}_2^{[0,k]}$ or $\mathrm{TM}_2^{[0,k]}$ makes the solution stable again.

Among the basic configurations two of them give one of the best results for every hyper-

heuristic used. Those are configuration $C_{19} = \{\diamond\mathrm{TI}_1, \mathrm{TS}^{[0,k]}, \diamond\mathrm{TD}^{[0,0]}\}$ and configuration $C_{30} = \{\diamond\mathrm{TI}_1, \mathrm{TI}_1, \mathrm{TS}^{[0,k]}, \diamond\mathrm{TD}^{[0,0]}\}$. They use swap low-level heuristic $\mathrm{TS}^{[0,k]}$ that tries every combination of the choices available. The best result here is the highest auxiliary value of any solution that was reached during the search process. The worst auxiliary values were obtained for configurations C_{12}, C_{14} or C_{15} regardless of the hyper-heuristics used. The worst results are when the highest auxiliary value of any solution reached during the search was one of the lowest for a particular instance. Every hyper-heuristic that was used gave similarly low results.

Described configurations led to formulation of new sets of low-level heuristics. The best results obtained previously always used the swap low-level heuristic. Figure 5.3 shows three plots dedicated to analysis of this low-level heuristic. The first search progress on Figure 5.3a uses configuration $C_{54} = \{\diamond\mathrm{TI}_1, \triangledown\mathrm{TS}^{[0,k]}, \diamond\mathrm{TD}^{[0,0]}\}$ to guide the search. All hyper-heuristics are quite similar and stabilises at $\approx 0.8 f_o^\dagger$ with the exception of DcAm that manifests a slight drop in auxiliary objective value from time to time. Changing swap low-level heuristic slightly to $C_{55} = \{\diamond\mathrm{TI}_1, \triangledown\mathrm{TS}^{[2,k]}, \diamond\mathrm{TD}^{[0,0]}\}$ results in observable drop in auxiliary objective value for hyper-heuristic UcAm. Other hyper-heuristics also perform slightly worse with a drop to $\approx 0.5 f_o^\dagger$. This ranges set in swap low-level heuristic is limited in this configuration and only large ranges are left. Selecting swap low-level heuristic results in removal of larger ranges. However, changing C_{54} to $C_{57} = \{\diamond\mathrm{TI}_1, \triangledown\mathrm{TS}^{[0,k]}, \diamond\mathrm{TD}^{[3,k]}\}$ surprisingly results in improvement for all the hyper-heuristics. The other instances from benchmark data set give for configuration C_{57} similar results as for C_{55}.

A number of more complex, combined configurations of low-level heuristics was also created. The results of the experiment shows that these large sets of mixed low-level heuristics give the best results for almost every instance and hyper-heuristic used. Configuration

$$C_{42} = \{\diamond\mathrm{TI}_2, \mathrm{TI}_1, \mathrm{TM}_2^{[0,k]}, \triangledown\mathrm{TM}_3^{[0,k]}, \diamond\mathrm{TS}^{[3,k]}, \mathrm{TS}^{[0,k]}, \triangledown\mathrm{TS}^{[0,k]}, \diamond\mathrm{TD}^{[0,0]}, \diamond\mathrm{TD}^3, \mathrm{TD}^0\}$$

is the best configuration obtained. The search progress for this set of low-level heuristic is shown on Figure 5.4a. It contains two insert, two move, three swaps and three delete low-level heuristics. Swap and delete low-level heuristics used are not the most possibly invasive low-level heuristics. Two other configuration

$$C_{44} = \{\diamond\mathrm{TI}_2, \mathrm{TI}_1, \mathrm{TM}_2^{[0,k]}, \triangledown\mathrm{TM}_3^{[0,k]}, \mathrm{TS}^{[0,k]}, \diamond\mathrm{TD}^{[0,0]}\}$$

that is C_{42} without number of swap and delete low-level heuristics and configuration

$$C_{45} = \{\diamond\mathrm{TI}_2, \mathrm{TI}_1, \mathrm{TM}_2^{[0,k]}, \diamond\mathrm{TS}^{[4,k]}, \mathrm{TS}^{[0,k]}, \triangledown\mathrm{TS}^{[0,k]}, \diamond\mathrm{TD}^{[0,0]}\}$$

that is C_{42} without a number of move and delete low-level heuristics give also one of the best results obtained. However, they are slightly more stable than C_{42}. Depending on test they give just a little better or worse solutions. Search process for every hyper-heuristic using these configuration is stable and slowly increasing (although hyper-heuristic DcAm for C_{42} traps in local minima). It is interesting that introducing $\triangle\mathrm{TS}^{[0,k]}$ to C_{45} results in huge drop of auxiliary objective value and some of the hyper-heuristics restarts the search periodically.

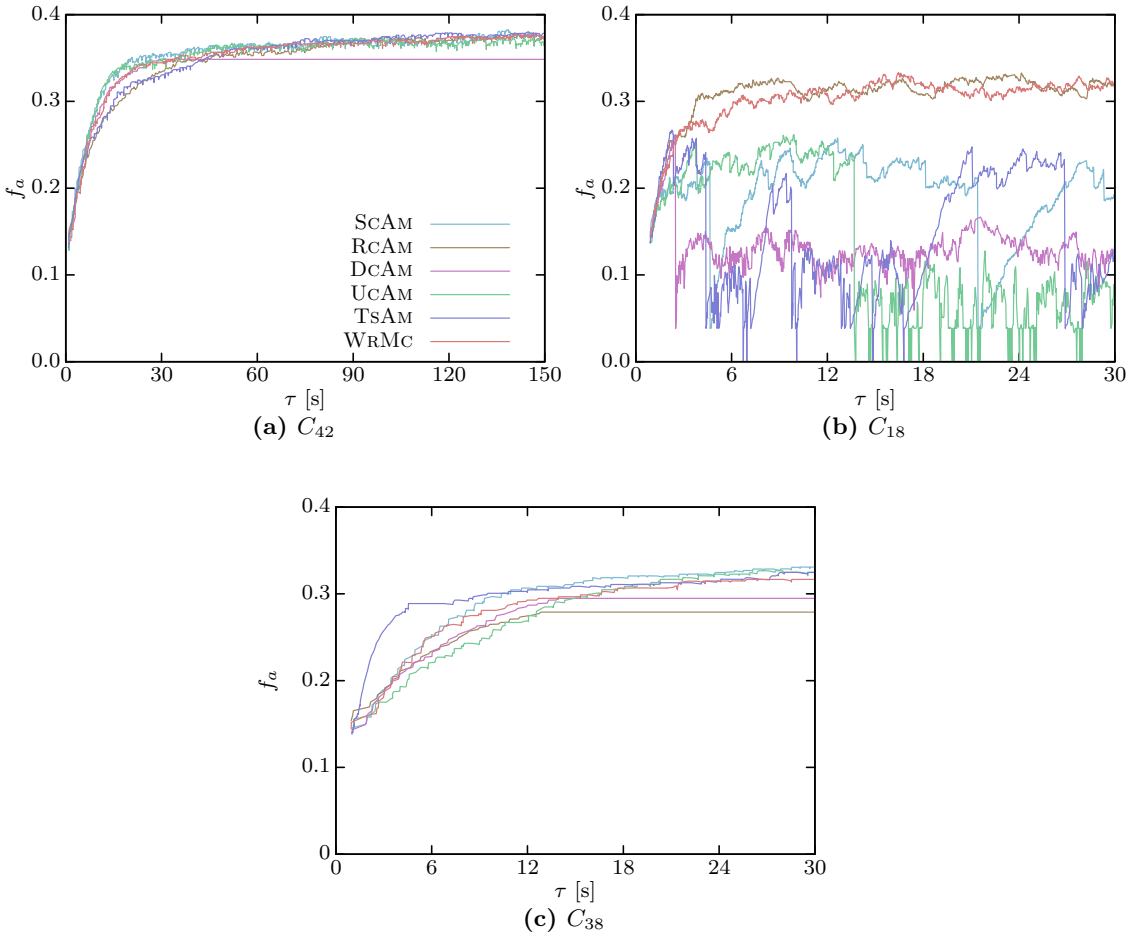


Fig. 5.4: Three different search progress plots that shows differences between hyper-heuristic methods. All of them are results for instance 16 in benchmark data set but use different sets of low-level heuristics.

5.5 Hyper-heuristics

The results of a computational experiment shows that in most cases the six different hyper-heuristics lead to the similar results. However, some of them are more uniform than others especially in the special cases, i.e., when the set of low-level heuristics leads to very stable or unstable results. Figure 5.4 shows three different plots of the change in auxiliary objective value in time for the instance 16 in benchmark data set. Three different configurations are used that leads to more or less stable results.

Figure 5.4a uses configuration C_{42} defined in Section 5.4. This configuration contains large variety of low-level heuristics and given enough time leads to very good results. Every hyper-heuristic used except DcAm gives very similar results. Hyper-heuristic DcAm stops the search in local minimum and does not improve the solution after about 40s of the search. Other hyper-heuristics performed very similarly and found their best values between 0.3808 and 0.3826 using this configuration.

The search progress on Figure 5.4b is the most unsteady among these three cases.

Configuration $C_{18} = \{\diamond\mathrm{TI}_1, \triangle\mathrm{TS}^{[0,k]}, \diamond\mathrm{TD}^{[0,0]}\}$ used contains the swap low-level heuristic that many hyper-heuristics are unable to cope with. Only for simulated annealing WrMc and ranked choice RcAm stable results are obtained. Simulated annealing hyper-heuristic prevents from using $\triangle\mathrm{TS}^{[0,k]}$ that leads to very dramatic change of auxiliary objective. Similarly RcAm evaluates all three low-level heuristics each time and prevents from using $\triangle\mathrm{TS}^{[0,k]}$. Remaining hyper-heuristics are forced to use this swap low-level heuristic periodically and when they call it then a dramatic drop in auxiliary objective value is observed. The worst results are obtained for UcAm.

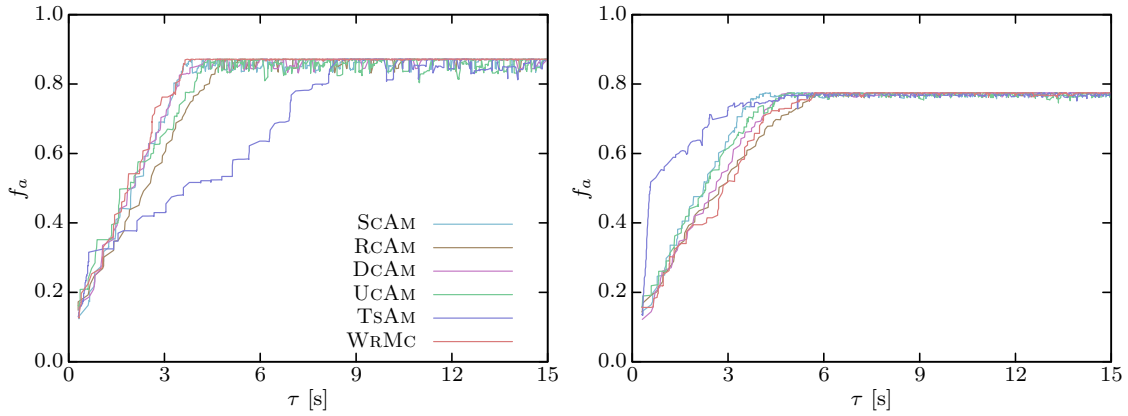
The third plot on Figure 5.4c is the other extreme of the search process. This time the configuration $C_{38} = \{\diamond\mathrm{TI}_1, \mathrm{TI}_1, \mathrm{TM}_2^{[0,k]}, \mathrm{TD}^{[0,0]}\}$ is used. This is a set of low-level heuristics that checks every possible move and chooses the best one. Delete operation is very gentle and removes only single oligonucleotides. This results in very stable search progress for every hyper-heuristic. Hyper-heuristics RcAm, DcAm and WrMc stabilises after about 20s of the search. The rest of hyper-heuristics improves their values slightly but then they also stabilise after about 50s of the search. None of them approaches best result for this test case $f_a^\dagger$.

Thus a difference between the hyper-heuristics is significant. Hyper-heuristic UcAm is the most sensitive one. If one of the low-level heuristic present in the configuration changes the solution significantly then the search process becomes very diverse and often the auxiliary search value drops to the lowest possible value with the empty solution. Similarly TsAm or ScAm can lead to chaotic results. All those hyper-heuristics evaluate only one low-level heuristic and use them regardless of the change in the solution. Hyper-heuristic DcAm also gives similar results but it happens rarely. The TsAm quite often falls into cycles where similar operations are applied periodically and they do not improve the solution on overall. The hyper-heuristics DcAm and RcAm are very vulnerable to the low-level heuristics that check every possible move and give very predictable outcome. For such cases they are often trapped in the local minima of the search process. Hyper-heuristic WrMc is not sensitive for low-level heuristics that change the solution greatly. This leads to quite stable search progress. Additionally, the results for this hyper-heuristic do not tend to be trapped in local minima. Thus WrMc leads to very stable search process and relatively high value of optimised auxiliary objective value. For good configurations of low-level heuristics hyper-heuristic UcAm also gives very good results, i.e., the search process is more diverse than WrMc. Generally good configurations are not dependent on the hyper-heuristic used. The differences between hyper-heuristics are very significant for bad sets of low-level heuristics.

5.6 Biological Data Results

The biological instances from data sets [1] and [2] were first analysed using sets of every possible low-level heuristics. One of the best results were obtained for configuration C_{42}. Two instances with a different error rate are presented on Figure 5.5. It can be noticed that 10% of errors results in more cyclic behaviour of some hyper-heuristics than for 20% of errors. This is repeatable among the other instances from this data set.

Among the all analysed configurations seven best promising were selected and used to solve every instance from biological data set. Fraction of those configurations were versions



(a) 10% error rate. **(b)** 20% error rate

Fig. 5.5: Progress of the hyper-heuristics search using tree method for instance 15 from biological data set [1] for sequences without repetitions with $n = 400$.

that started with a initial solutions generated using maximal possible number of best insert moves instead of maximal possible number of random insert moves. Obtained values shows that no hyper-heuristic gives the best values for every instance. One configuration can give the best results for one instance and the worst for other. However, the hyper-heurisic UcAm was selected because it gives the pseudo optimal value in the greatest number of runs. Numerical results to the biological data sets for hyper-heuristic UcAm with configuration C_{42} are listed in Appendix C.

In most of the cases obtained objective values are equal to the pseudo optimal values determined by sequences used to generate test instances. However, sometimes the obtained values are lower than optimum, i.e., the set method finds better solutions. This happen especially for large instances. Tree methods require a little bit more time to find better solutions and this is also evident from the tables in Appendix C. The time given to tree method for solving large instances was not large enough.

Chapter 6
Conclusions

In this work sequencing by hybridisation problem was studied and analysed in details. Despite the fact that only a few constraints are defined the problem is computationally hard and obtaining good results for hard instances requires sophisticated and elaborate techniques. A hyper-heuristic framework was developed for the purpose of solving SBH problem. Although designed specially for this sole purpose thanks to its generality the framework allows to be reused in other optimisation problems.

In this research two hyper-heuristic approches to solving SBH problem have been analysed. The first approach which uses a set of low-level heuristics adopted from the previously proposed tabu search method gives unsatisfactory results. The second approach with widely extended set of low-level heuristics allows for great flexibility and manipulation of the SBH solution search process. Properly configured it gives much better results than the preliminary approach. Performed analysis shows that the extensions introduced in this approach are required, especially for a benchmark data set that consists of hard and very random instances. However, not all of the extensions turned out to be quite relevant, i.e., dividing low-level heuristics to operate on different subsets of oligonucleotides from spectrum has not as much influence as controlling the distance between the oligonucleotides in the solution where various moves are possible. Introducing various types of swap moves where insert and delete heuristics are combined in a single move gives very good results. This is because the used hyper-heuristics are local search methods and they cannot predict the number of moves in advance. The swap low-level heuristics partially overcomes this disadvantage.

Results presented in Chapter 5 show that the most important issue for the entire hyper-heuristic procedure is the configuration of low-level heuristics. The set of low-level heuristics biases the objective value obtained during the search. This set can lead to great chaos or to local minima where the search process traps. However, if a good configuration of low-level heuristics is used then the objective function values can be greatly improved. Thus the first approach based on five tabu search moves did not provide enough variety to the search process. The second approach with more sophisticated low-level heuristics introduces a large number of possible search algorithms.

Another important factor for search effectiveness was the type of hyper-heuristic. Some of the hyper-heuristics are more stable (WrMc, RcAm) and some of them more chaotic (UcAm, TsAm) but with a good configuration each of them performs quite well. However,

the results in Appendix C show that hyper-heuristic UcAm gives the best results for almost every tested instance. This is the most random hyper-heuristic and presumably for the SBH problem which has very strict solutions and inflexible structure more randomness is needed. The SBH problem itself is quite different from the problems that hyper-heuristics are designed to deal with, having a complex structure and large numbers of different low-level heuristics possible. This work concludes that hyper-heuristics can also be applied for problems with a simple structure like SBH.

As a result of this work a number of software tools were developed including a visual analysis tool for observing the solutions for the instances of SBH problem and a large number of batch utilities for performing the computational experiments. All developed hyper-heuristic solvers use a general and reusable hyper-heuristic framework.

Analysis conducted in this work is a general one. The sets of low-level heuristics were optimised based on an overall data set performance or hyper-heuristic method performance. Further improvements in objective function value can be achieved by an instance specific optimisation. The extended method allows for a large variety of changes and the collected results indicate that there are cases where different sets of low-level heuristics work better on some instances and worse on the others. A very relevant issue is also choosing the appropriate parameters for the auxiliary objective value formula (Equation 4.1). The values of these parameters used in this work are chosen empirically and are not very accurate. This results in inconsistencies in Table C.1 where for some cases the best objective value does not correspond to the best auxiliary value. Manipulating with these parameters should give considerable improvements.

Bibliography

[1] Spectra derived from real DNA seqeunces for standard and isothermic sequencing. http://bio.cs.put.poznan.pl/index.php/research/36-current-research/65-standard-and-isothermic-spectra, 2004.

[2] Spectra for standard DNA sequencing derived from real and random DNA sequences. http://bio.cs.put.poznan.pl/index.php/research/36-current-research/66-standard-spectra, 2004.

[3] Masri Ayob and Graham Kendall. A Monte Carlo Hyper-Heuristic To Optimise Component Placement Sequencing For Multi Head Placement Machine. In *Placement Machine, InTech'03 Thailand*, pages 132–141, 2003.

[4] Ruibin Bai, Edmund K. Burke, Graham Kendall, and Barry Mccollum. A Simulated Annealing Hyper-heuristic for University Course Timetabling, 2006.

[5] Ruibin Bai, Jacek łażewicz, Edmund K. Burke, Graham Kendall, and Barry Mccollum. A simulated annealing hyper-heuristic methodology for flexible decision support. Technical report, School of CSiT, University of Nottingham, UK, 2007.

[6] Jeremy M. Berg, John L. Tymoczko, and Lubert Stryer. *Biochemistry, Fifth Edition : International Version.* W. H. Freeman, 2002.

[7] Jacek Błażewicz, Piotr Formanowicz, Marta Kasprzak, Wojciech T. Markiewicz, and J. Węglarz. DNA sequencing with positive and negative errors. *J. Comput. Biol.*, 6: 113–123, 1999.

[8] Jacek Błażewicz, Piotr Formanowicz, Marta Kasprzak, Wojciech T. Markiewicz, and Jan Węglarz. Tabu search for DNA sequencing with false negatives and false positives. *European Journal of Operational Research*, 125(2):257–265, 2000.

[9] Jacek Błażewicz, Fred Glover, and Marta Kasprzak. DNA Sequencing–Tabu and Scatter Search Combined. *INFORMS J. on Computing*, 16(3):232–240, 2004.

[10] Jacek Błażewicz, Fred Glover, and Marta Kasprzak. Evolutionary Approaches to DNA Sequencing with Errors. *Annals OR*, 138(1):67–78, 2005.

[11] Jacek Błażewicz, Fred Glover, Marta Kasprzak, Wojciech T. Markiewicz, Ceyda OGuz, Dietrich Rebholz-Schuhmann, and Aleksandra Świercz. Dealing with repetitions in sequencing by hybridization. *Computational Biology and Chemistry*, 30(5): 313 – 320, 2006.

[12] Jacek Błażewicz, Alain Hertz, Daniel Kobler, and Dominique de Werra. On some properties of DNA graphs. *Discrete Applied Mathematics*, 98(1-2):1 – 19, 1999.

[13] Jacek Błażewicz and Marta Kasprzak. Complexity of DNA sequencing by hybridization. *Theor. Comput. Sci.*, 290(3):1459–1473, 2003.

[14] Jacek Błażewicz, Marta Kasprzak, and Wojciech Kuroczycki. Hybrid Genetic Algorithm for DNA Sequencing with Errors. *Journal of Heuristics*, 8(5):495–502, 2002.

[15] Jacek Błażewicz, Ceyda Oguz, Aleksandra Świercz, and Jan Węglarz. DNA Sequencing by Hybridization via Genetic Search. *Oper. Res.*, 54(6):1185–1192, 2006.

[16] Edmund K. Burke, Graham Kendall, and Eric Soubeiga. A Tabu-Search Hyperheuristic for Timetabling and Rostering. *Journal of Heuristics*, 9(6):451–470, 2003.

[17] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition.* The MIT Press, 2001.

[18] Peter Cowling, Graham Kendall, and Eric Soubeiga. A Hyperheuristic Approach to Scheduling a Sales Summit. In *PATAT '00: Selected papers from the Third International Conference on Practice and Theory of Automated Timetabling III*, pages 176–190, London, UK, 2001. Springer-Verlag.

[19] Peter Cowling, Graham Kendall, and Eric Soubeiga. A parameter-free hyperheuristic for scheduling a sales summit. In *Proceedings of the 4th Metaheuristic International Conference, MIC 2001*, pages 127–131, 2001.

[20] Peter Cowling, Graham Kendall, and Eric Soubeiga. Hyperheuristics: A Tool for Rapid Prototyping in Scheduling and Optimisation. In *EvoWorkShops. Lecture Notes in Computer Science*, pages 1–10. Springer, 2002.

[21] Radoje Dramanac, Ivan Labat, Ivan Brukner, and Radomir Crkvenjakov. Sequencing of megabase plus DNA by hybridization: Theory of the method. *Genomics*, 4(2):114 – 128, 1989.

[22] Thompson G.L. Fisher M.L. Probabilistic learning combinations of local job-shop scheduling rules. *Industrial Scheduling*, pages 225–251, 1963.

[23] Lilian T. C. França, Emanuel Carrilho, and Tarso B. L. Kist. A review of DNA sequencing techniques. *Quarterly Reviews of Biophysics*, 35(02):169–200, 2002.

[24] John Gallant, David Maier, and James A. Storer. On Finding Minimal Length Superstrings. *Journal of Computer and System Sciences*, 20(1):50–58, 1980.

[25] Michael R. Garey and David S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences).* W. H. Freeman, 1979.

[26] Alan Gibbons. *Algorithmic Graph Theory.* Cambridge University Press, 1985.

[27] Fred W. Glover and Gary A. Kochenberger. *Handbook of Metaheuristics (International Series in Operations Research & Management Science).* Springer, 2003.

[28] Anthony J. F. Griffith, Jeffrey H. Miller, David T. Suzuki, Richard C. Lewontin, and William M. Gelbart. *An introduction to genetic analysis.* Freeman and Company, 1996. GRI a 96:1 1.Ex.

[29] David S. Johnson. The NP-Completeness Column: An Ongoing Guide. *J. Algorithms*, 6(2):291–305, 1985.

[30] Graham Kendall and Naimah Mohd Hussin. An Investigation of a Tabu-Search-Based Hyper-Heuristic for Examination Timetabling. In *Multidisciplinary Scheduling: Theory and Applications*, pages 309–328. Springer US, 2005.

[31] Graham Kendall, Eric Soubeiga, and Peter Cowling. Choice Function and Random Hyperheuristics. In *Proceedings of the fourth Asia-Pacific Conference on Simulated Evolution And Learning, SEAL*, pages 667–671. Springer, 2002.

[32] I. u. P. Lysov, V. L. Florent'ev, A. A. Khorlin, K. R. Khrapko, and V. V. Shik. [Determination of the nucleotide sequence of DNA using hybridization with oligonucleotides. A new method]. *Dokl. Akad. Nauk SSSR*, 303:1508–1511, 1988.

[33] Ender Özcan, Burak Bilgin, and Emin Erkan Korkmaz. A comprehensive analysis of hyper-heuristics. *Intelligent Data Analysis*, 12(1):3 – 23, 2008.

[34] Christos M. Papadimitriou. *Computational complexity.* Addison-Wesley, Reading, Massachusetts, 1994.

[35] Pavel A. Pevzner. 1-Tuple DNA sequencing: computer analysis. *J. Biomol. Struct. Dyn.*, 7:63–73, 1989.

[36] Edwin M. Southern. United Kingdom Patent Application GB8810400, 1988.

[37] Edwin M. Southern. DNA chips: analysing sequence by hybridization to oligonucleotides on a large scale. *Trends in Genetics*, 12(3):110 – 115, 1996.

[38] Apostolos Syropoulos. Mathematics of Multisets. In *WMP '00: Proceedings of the Workshop on Multiset Processing*, pages 347–358, London, UK, 2001. Springer-Verlag.

[39] William T. Tutte. *Graph Theory (Encyclopedia of Mathematics and Its Applications, Volume 21).* Longman Higher Education, 1984.

[40] James D. Watson and Francis H. Crick. Molecular structure of nucleic acids: A structure for deoxyribose nucleic acid. *Nature*, 171:737–738, 1953.

[41] Ji-Hong Zhang, Ling-Yun Wu, and Xiang-Sun Zhang. Reconstruction of DNA sequencing by hybridization. *Bioinformatics*, 19(1):14–21, 2003.

Appendix A

Hyper-heuristics Framework

This appendix describes general technical issues relevant for the implementation of hyper-heuristics framework. It is a developer guide that introduces basic concepts required to use this Framework.

Hyper-heuristics aims to abstract from problem-specific knowledge. This approach allows to design a modular and problem independent framework for solving any optimisation problem. Such a framework is created in this work. This hyper-heuristic framework is written in Java language. Java is an object oriented programming language that supports the modular model used in the framework. Methods and algorithms from the hyper-heuristic framework compose to an independent Java class library that can be used in other projects. Programs written in Java are platform independent i.e., this library can be used on any platform equipped with Java Virtual Machine. Java programming language is popular in scientific community and there are many tools both free and commercial that aid development and testing process. Access to this framework should be easy for a large number of programmers and researchers.

The goal of this section is to introduce the overall construction of the hyper-heuristic framework. The UML class diagram of the framework with the relevant classes included is presented on Figure A.1. Hyper-heuristic framework is enclosed in the `wojtek.hyper` package. Java class with the main loop of the search process is a `HyperHeuristic` class. The diagram shows only a fraction of the methods in this class. There are other methods used for execution control and resources measurement during the search process. They are not described here. Complete reference and source code documentation is included in the source code release. The main purpose of method `solve()` is to implement the procedure HyperHeuristicSearch from Section 3.1. One of the arguments for this procedure is a set of low-level heuristics. Low-level heuristics are defined by the interface `LowLevel`. Creation of the problem specific low-level heuristics is done by implementing this interface in problem specific classes. Communication with the framework is performed by four methods. Low-level heuristics can be used at most once at each iteration. First, the method `apply(Solution)` is called and it initialises the use of low-level heuristic. It should perform a simulation of application of the low-level heuristic on a given solution but the solution itself should not be modified. When this method finishes an internal state should be set and the low-level heuristic methods `getAuxiliaryValue()` and `getObjectiveValue()` should return the search auxiliary value and objective value after

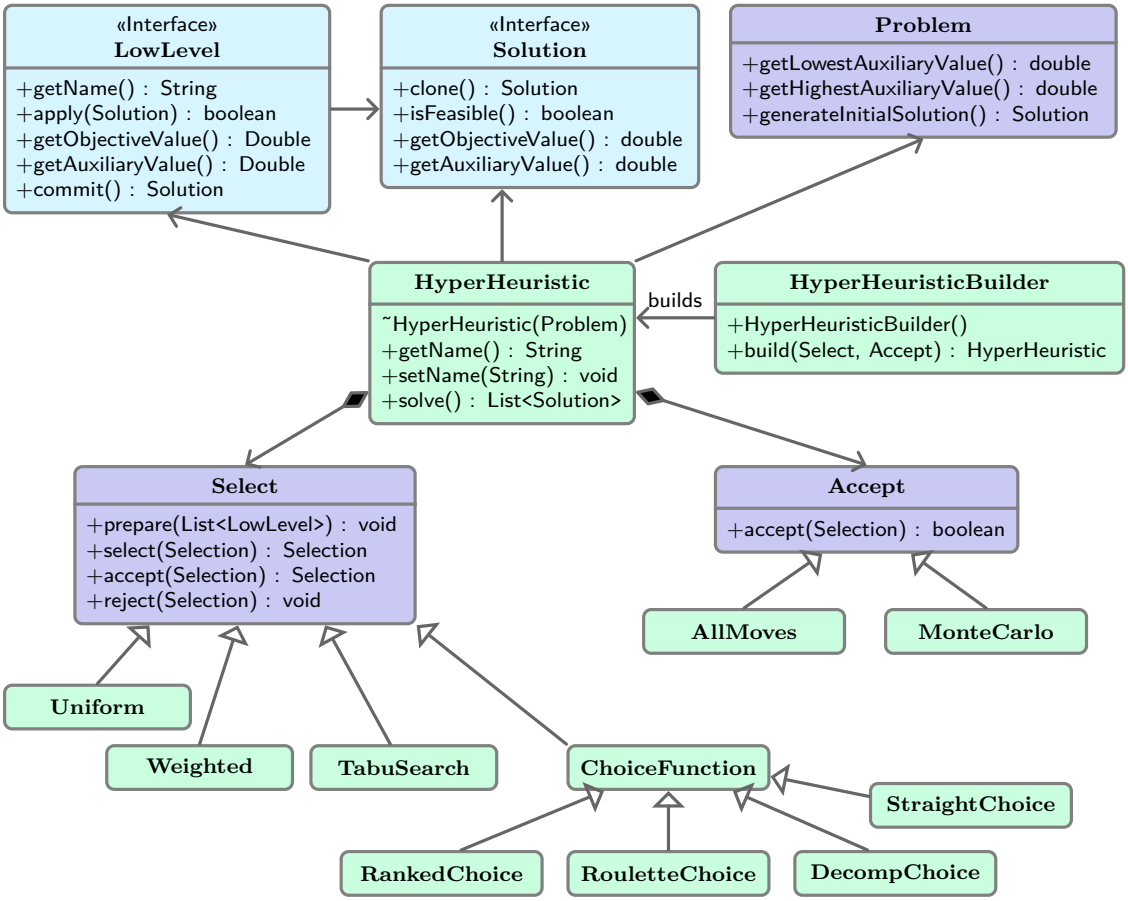


Fig. A.1: UML diagrams of the Hyper-heuristic framework. Figure represents the main classes used during the hyper-heuristic search and building pattern for constructing the solver.

the low-level heuristic would be applied to given solution. The hyper-heuristic method can decide to modify the solution by calling method `commit()` and discard the changes by calling method `apply(Solution)` again at the next iteration. This cycle can be executed many times, hyper-heuristic search process applies some low-level heuristic at each iteration but approves the changes only when decides to do so. Such a system allows for using many different programming techniques in the implementation of problem specific low-level heuristics.

Besides the low-level heuristic two more abstractions must be specified for each problem — an interface `Solution` and an abstract class `Problem`. The interface `Solution` represents solutions that can appear during the search. Solutions are modified by the low-level heuristics as described above. During the search process the best solutions are stored and to do this they must implement the `clone()` method. The hyper-heuristic search procedure modifies a single instance of the solution and stores a set of copies of them from different iterations. The copies are used later to return a list of best solutions found. The class `Problem` provides a general configuration for the specific problem solved. Methods for retrieving lowest and highest auxiliary value from this class are used during the search for normalisation purposes. These values are important for simulated annealing heuristic because the probability in equation 3.1 is dependent of the magnitude of change of the auxiliary value. Normalisation is also relevant for choice function heuristics because of values α and β being constants. Method `generateInitialSolution()` is used to generate the solution ω_0 for the search process.

The low-level heuristic selection and acceptance methodology outlined on Figures 3.1 and 3.2 is directly reflected in the object oriented approach. The UML diagram A.1 contains abstract class `Select` which is responsible for selecting an appropriate low-level heuristic. The actual selection is done by method `select()` which returns the selection information in object of type `Selection`. This class is not included on the diagram for clarity. The selection can be accepted or rejected. In the first case hyper-heuristic search calls method `accept(Selection)` and in the second case the method `reject(Selection)`. This allows for selecting heuristic in order to update its internal state and to prepare for the next iteration. Various classes: `Uniform`, `Weighted`, `TabuSearch` and a family of `ChoiceFunction` derivatives implement those methods and represent the specific selection algorithms described in Sections 3.2 and 3.3. Similarly an abstract class `Accept` with two implementations — `AllMoves` and `MonteCarlo` represents the acceptance criterion. Method `accept(Selection)` decides to accept or reject the current solution candidate.

The correct construction of `HyperHeuristic` instance requires a number of steps. In order to make this easier a builder design pattern is used. Access modifier of the `HyperHeuristic` class is set to default, package access. To create instance of this class and start the search an intermediary class `HyperHeuristicBuilder` is introduced as shown in the UML diagram on Figure A.1. The method `build(Select, Accept)` takes an instance of `Accept` and `Select` classes and constructs the instance of `HyperHeuristic` class.

Two mechanisms for controlling and monitoring the search process by the client of the hyper-heuristic library are introduced. The first one is used to determine when the search should stop. Figure A.2 introduces the `Termination` interface with a single method `isTerminationConditionSatisfied()`. This method is called during the hyper-heuristic search to check for current status of termination conditions. If they are satisfied

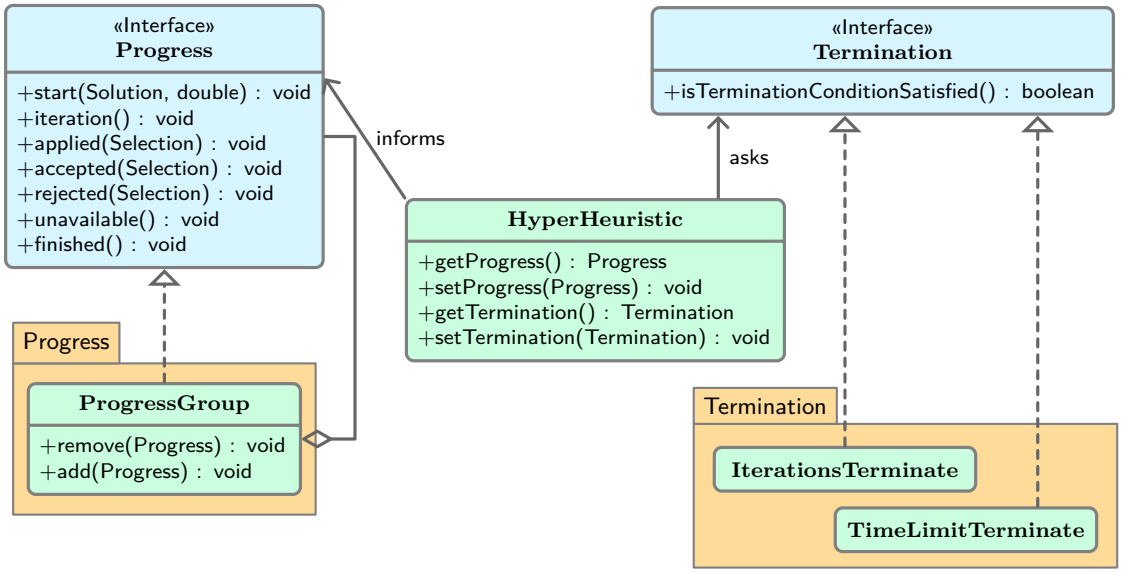


Fig. A.2: Notification process and termination conditions query.

then search is stopped. Two implementations of this interface are available in package `wojtek.hyper.termination`. The class `IterationsTerminate` terminates the search after a fixed number of iterations. Class `TimeLimitTerminate` terminates the search after a fixed time i.e., a fixed number of seconds since the beginning of the search. Other more elaborate termination conditions can be implemented e.g., termination after a number of iterations without improvement of the solutions or termination after failure of statistical test. However, for the purpose of analysis made in this work only the two basic termination conditions are required — fixed time limit and fixed number of iterations limit.

The second mechanism presented on Figure A.2 is a client notification system. The interface `Progress` contains a set of methods that are called by the hyper-heuristic framework during the search. The method `start(Solution, double)` is called only once at the beginning of the search when initial solution is found. Method `iteration()` is called at the beginning of each iteration. Methods `finished()` and optionally `unavailable()` are called once after the search finishes. Each time the solution is applied the method `applied(Selection)` is called and either `accepted(Selection)` or `rejected(Selection)` are called after this. Only one `Progress` object can be used with a `HyperHeuristic` instance. However, the class `ProgressGroup` in package `wojtek.hyper.progress` allows using a set of other `Progress` implementations. Every notification from the hyper-heuristic framework is propagated on those objects.

Appendix B

User Guide

There has been a number of command-line tools developed for the purpose of SBH problem analysis and solving its instances. In this appendix a brief usage guide of this tools is presented. In Section B.1.2 two random SBH instances generator and their usage is presented. In Section B.1.3 two methods for finding exact solutions to the SBH problem are demonstrated. Later sections presents two hyper-heuristic solvers for the SBH problem, i.e., *set* method approach and *tree* method approach. They differ in solution encoding and set of low-level heuristics. In Section B.2 a visual analysis tool that integrates each solver described previously is presented.

B.1 Batch Utilities

Each example in this section uses the `sbh.jar` class library that can be found in binary distribution of this software. In order to execute it in the most convenient way the Java class path should be configured as follows:

```
$ export CLASSPATH="sbh.jar"
```

Afterwards, the classes from the library can be used directly. The goal of the following sections is to give a brief usage examples of software tool. A comprehensive documentation can be found in the embedded help of command-line tools that is obtained by executing the tools without any parameters.

B.1.1 Instances Files

The instances for the SBH problem are stored in the text files. The exemplary content of an instance data stored in file named `designed_4.in` is shown on Figure B.1. Every file that describes the instance starts with a number n which denotes a desired length of the nucleotides sequence. In the next line a number n_s of oligonucleotides in the spectrum follows. After this line there are exactly n_s oligonucleotides, each on separate line and of length k. Oligonucleotides are encoded as a sequence of letters A, C, G or T in lower or upper case. The instance file can have an arbitrary number of comments that are lines starting with a `#` character.

```
#atgcgtgcgca
11
7
atgc
gcgt
cgtg
gtgc
tgcg
gcgc
cgca
```

Fig. B.1: An example of the instance file to the SBH problem.

B.1.2 Random Generators

There are two different random SBH instances generators as described in Section 5.2. The first one is used to simulate the result of sequencing by hybridisation experiment. Executing it without any parameters:

```
$ java wojtek.sbh.generator.ExperimentGenerator
```

prints a brief help with the list of possible options. Two mandatory options must be set. The `-n` options expects an integer number that determines the desired size of the nucleotides sequence n and the option `-k` that determines the length k of every oligonucleotide in the generated spectrum. Calling it with $n = 5$ and $k = 3$ results in generation of the SBH problem instance on the standard output.

```
$ java wojtek.sbh.generator.ExperimentGenerator -n 5 -k 3 -s 14 -w
# Generated with wojtek.sbh.generator.ExperimentGenerator
# s = 14, p = 0.0 (errors = 0), q = 0.0 (errors = 0)
# ggtgc
5
3
ggt
tgc
gtg
```

The additional switch `-w` prints the first three lines of comment that contains details about generated instance and the nucleotides sequence that was used to create the spectrum. Option `-s` sets the seed for a random number generators. There are two options that controls how much positive or negative errors should be introduced to the spectrum. Parameter `-p` determines the probability of a positive error and parameter `-q` determines the probability of a negative error. When set to non-zero values the errors can appear:

```
$ java wojtek.sbh.generator.ExperimentGenerator \
  -n 5 -k 3 -s 14 -p 0.1 -q 0.1
5
3
ctg
ggt
gtg
```

In this case from sequence GGTGC a positive error appeared as an additional oligonucleotide CTG and negative error as missing oligonucleotide TGC.

The second random SBH instances generator generates completely random instance. Executing it without any parameters shows a brief usage information. The two required parameters are exactly the same as in the experiment instances generator, the length of sequence n determined by option `-n` and the length of the oligonucleotides k determined by option `-k`:

```
$ java wojtek.sbh.generator.RandomGenerator -n 5 -k 3
5
2
agc
tct
```

This generator just prints a number of oligonucleotides of length k. The number of oligonucleotides is by default determined by a random variable of the Gaussian distribution $N(\mu, \sigma^2)$ with $\mu = n - k + 1$ and $\sigma = \frac{2}{5}\mu$. However, this value can be overridden using the option `-z`:

```
$ java wojtek.sbh.generator.RandomGenerator -n 5 -k 3 -z 4
5
4
acc
ttt
gaa
cct
```

B.1.3 Exact Solvers

There are two solvers that can give the optimal solution to SBH. The brute force method implementation can solve every instance of the SBH problem optimally. The implementation of the Zhang et al. [41] method solves only instances without negative errors optimally.

An example of solving the instance `designed_4.in` to the SBH problem is given by the command shown below.

```
$ java wojtek.sbh.exact.BruteForce designed_4.in -u
atgcgtgcgca: 3
```

The parameter `-u` causes the program to print a list of unique solutions for this particular instance. Number following the solution sequence is number of different possibilities of creating this sequence. For this particular instance the exhaustive search method found three possibilities of constructing sequence ATGCGTGCGCA. To solve the same instance using method described by Zhang et al. [41] the following command can be used.

```
$ java wojtek.sbh.zhang.Zhang designed_4.in -q
atgcgtgcgca
```

It assumes that only positive errors are present in instance from file `designed_4.in`. Parameter `-q` causes to show a list of solutions found.

Both above solvers have an optional parameter `-d` that is a negative error level. However, in case of brute force algorithm this parameter determines the maximum number of consecutively missing oligonucleotides in any of the solutions found. The default value is ∞ for this method. For the Zhang et al. [41] method this parameter is used to generate a new spectrum with additional oligonucleotides generated from overlaps of distance $1, 2, \ldots d$ between each pair of oligonucleotides from the original spectrum. The default value is 0 in this case.

B.1.4 Set Solver

The *set* method solver is a hyper-heuristic approach based on tabu search metaheuristic by Błażewicz et al. [8] as described in Section 4.1. Low-level heuristics used in set method correspond to the moves defined in this article. There are six solvers defined: `SimulatedAnnealing`, `TabuSearch`, `StraightChoice`, `RankedChoice`, `RouletteChoice` and `DecompChoice`. They are launched by selecting a different main class when launching the solver. Their general usage is quite similar and a few examples will be shown using the `SimulatedAnnealing` class. Executing it without arguments shows a list of possible options.

```
$ java wojtek.sbh.hyper.set.run.SimulatedAnnealing
```

The `SimulatedAnnealing` string can be substituted by any of the mentioned hyper-heuristic solvers. In order to start the search a list of low-level heuristics must be defined using the option `-h` and a file with the SBH problem instance to solve given. The following command starts the search using low-level heuristics SI, SM, SMC and SD.

```
$ java wojtek.sbh.hyper.set.run.SimulatedAnnealing \
  -q -h:SI:SM:SMC:SD designed_4.in
# length = 11, probe length = 4, used = 5, available = 7, repetitions = 0,
    missing = 3, consecutively missing = 3
atgcgcacgtg
# length = 11, probe length = 4, used = 5, available = 7, repetitions = 0,
    missing = 3, consecutively missing = 3
cgtgatgcgca
```

The `-q` option is responsible for printing a list of solutions found together with some statistics about each solutions. The above command found two solutions with objective value equal to 5 (i.e., a number of used oligonucleotides is equal to 5). There are many switches that control the way of presenting results of the search. Option `-u` prints a single floating point number that is a best auxiliary search value obtained during the search. Options `-p` and `-c` take a file name where the search progress and compact progress respectively should be reported. Option `-s` saves a list of solutions found to the file that can be later opened in the visual analysis tool.

The default settings causes the search to finish after 100 iterations. This can be changed with the help of `--time-limit` and `--iterations-limit` options:

```
$ java wojtek.sbh.hyper.set.run.SimulatedAnnealing \
  -q -h:SI:SM:SMC:SD --time-limit 60 designed_4.in
```

In this example instead of the iterations limit a time limit is used. It is set to 60 seconds and after this time the search will stop. The set method has also an option for determining the type of initial solution that is generated. This is determined by `-i` option and possible values are an empty initial solution or an initial solution created using the maximal number of insertions.

B.1.5 Tree Solver

The *tree* method is an extended version of the set method and its detailed description is subject of the Chapter 4. As previously, there are six solvers defined: `SimulatedAnnealing`, `TabuSearch`, `StraightChoice`, `RankedChoice`, `RouletteChoice` and `DecompChoice`. Executing for example the `SimulatedAnnealing` solver without any parameters reveals a list of possible options.

```
$ java wojtek.sbh.hyper.tree.run.SimulatedAnnealing
```

The mandatory option `-h` required for the search to start is a file with the low-level heuristics configuration. The format of this configuration is described in Section B.1.6. Besides the configuration the file with an instance data for SBH problem must be given.

```
$ java wojtek.sbh.hyper.tree.run.SimulatedAnnealing \
  -q -h heuristics.yml designed_4.in
# length = 11, probe length = 4, used = 7, available = 7, repetitions = 1,
    missing = 0, consecutively missing = 0
atgcgtgcgca
```

Every possible presentation option described in Section B.1.4 is also available for tree method and option `-q` has a similar behaviour. The above execution found an optimal value for this instance with a number of oligonucleotides used equal to 7.

The tree method solvers have options to control and tune the hyper-heuristics parameters. The initial temperature and the temperature iterations parameters for simulated annealing hyper-heuristic can be controlled. For the tabu search hyper-heuristic a tabu iterations parameters can be set and in the choice function hyper-heuiristics the coefficients α, β and γ can be changed. The roulette choice hyper-heuristic solver has an option to change the value of η and the ranked choice hyper-heuristic solver can manipulate the value of a number of ranked low-level heuristics.

B.1.6 Configuration for Tree Method

The configuration files for tree method solver contain a list of low-level heuristics that should be used during the search. It also contains the configuration of a method for generating initial solution and the configuration of the logging level. An example of this configuration file is presented on Figure B.2. There are three meaningful possible logging levels: **off**, **info** and **debug**. The default logging level is set to **off**. Using the **info** logging level causes to log a basic information about the search, i.e., what kind of parameters are used and when the search started or finished. The **debug** logging level causes to print verbose information about the low-level heuristic selection and acceptance phases.

```
---
logging: { level: info }
initial: random
heuristics:
  - !insert
    positions: { distance: 1 }
  - !swap
    random: { inserted: false, removed: false }
    removed: !clusters { distance: "<0, k>" }
  - !delete
    removed: !clusters { distance: "<0, 0>" }
```

Fig. B.2: An example of the low-level heuristics configuration for the tree method.

There are three possible values for initial solution generation method. Those are **empty**, **random** and **full**. The default one is **empty** and causes to generate an empty solution. The other two methods performs a maximal number of insertions possible. The first one uses random version of the insert low-level heuristic and the second performs a complete search among every possible insertion. Those two methods are greedy algorithms.

The configuration files are written in YAML language and their syntax follows its specification. Every low-level heuristic on the list of low-level heuristics starts with a „-" character followed by a type of low-level heuristic. There are four possible types: `insert`, `delete`, `move` and `swap` and they correspond to TI, TD, TM and TS low-level heuristics. An example with almost every option of the insert low-level heuristic is shown below.

```
- !insert
  random: { inserted: false, position: false }
  inserted: { allowed: "{1-30}", iterations: "<4, inf>" }
  positions: { distance: 1 }
```

The random property controls whether inserted oligonucleotide should be selected at random or to be chosen among every possibility. Similarly, the destination position can be selected at random or not. The possible values are **true** or **false**. The inserted oligonucleotide is specified by a set of allowed oligonucleotides (numbers represents oligonucleotides from spectrum as they appeared in the instance file) and interval of allowed iterations (oligonucleotide is candidate for insertion if the iteration when it was used last belongs to this interval). The default value for allowed oligonucleotides is an entire spectrum and default value for possible iterations is a range of all iterations. The positions parameter determines the possible positions on the list of oligonucleotides that encodes the solution where the inserted oligonucleotide can be placed. Distance is a minimum number of nucleotides between two successive oligonucleotides. The above configuration corresponds to the $\mathrm{TI}_1^{[4,\infty),\{o_1,o_2,\ldots,o_{30}\}}$ low-level heuristic where $o_1, o_2, \ldots, o_{30}$ are successive oligonucleotides from $\mathcal{S}_k$. Another insert low-level heuristic configuration can be written as follows:

```
- !insert
  random: { inserted: false, position: false }
  inserted: { allowed: "{1,3,8-12,13-s}", iterations: "<4, inf>" }
  positions: { distance: 1 }
```

Delete low-level heuristic selects a range from the oligonucleotides list that should be removed. Thus randomness configuration is just a boolean value. However, there are two different types of ranges that can be used for deletion. The first one is a cluster range set denoted by the type `clusters` and the second one is a minimal range set denoted by type `minimal`. The example below show those two possibilities.

```
- !delete
  random: true
  removed: !clusters { allowed: "{3-s}", distance: "<0, 0>" }
- !delete
  random: false
  removed: !minimal { iterations: "<6, inf>", distance: 0 }
```

Both sets are specified by set of allowed oligonucleotides and possible iterations just like the insert low-level heuristic. The possible iterations interval is omitted in cluster ranges set in the example and the set of allowed oligonucleotides is omitted in minimal ranges set in the example. Those two sets differ in the distance parameter that determines the allowed distance between oligonucleotides in resulting ranges. For the cluster ranges set the distance is an interval. For minimal ranges set the distance is an integer value that denotes the minimal possible distance. The distance parameter is required for those sets. Those two delete low-level heuristics are description of the $\diamond\mathrm{TD}^{(0,\infty),\{o_3,o_4,\dots\},[0,0]}$ and $\mathrm{TD}^{[6,\infty),\mathcal{S}_k,0}$ low-level heuristics respectively, where $o_3, o_4, \dots$ are succesive oligonucleotides from $\mathcal{S}_k$.

Move low-level heuristics combines the parameters presented above. The moved range can be either cluster ranges set or minimal ranges set. For this low-level heuristic also a set of possible positions is required. Randomness settings determines if choice should be made randomly or exhaustive search of every possibility should be used. An example of $\mathrm{TM}_0^{(0,\infty),\mathcal{S}_k,[3,k]}$ move low-level heuristic configuration is shown below.

```
- !move
  random: { moved: false, position: false }
  moved: !clusters { distance: "<3, k>" }
  positions: { distance: 0 }
```

Very similar configuration concerns also the swap low-level heuristic. The exemplary configuration for the $\triangle\mathrm{TS}^{\dagger(0,\infty),\mathcal{S}_k,[3,k]}_{\{o_5,o_6,\dots\},[5,\infty)}$ low-level heuristic where $o_5, o_6, \dots$ are successive oligonucleotides from spectrum can be presented like this:

```
- !swap
  random: { inserted: false, removed: true }
  removed: !clusters { distance: "<3, k>", forced: true }
  inserted: { allowed: "{5-s}", iterations: "<5, inf>" }
```

There is one additional parameter for the removed ranges set if a cluster ranges type is used. This parameter is called `forced` and determines if the ranges in the removed set can be forced or not.

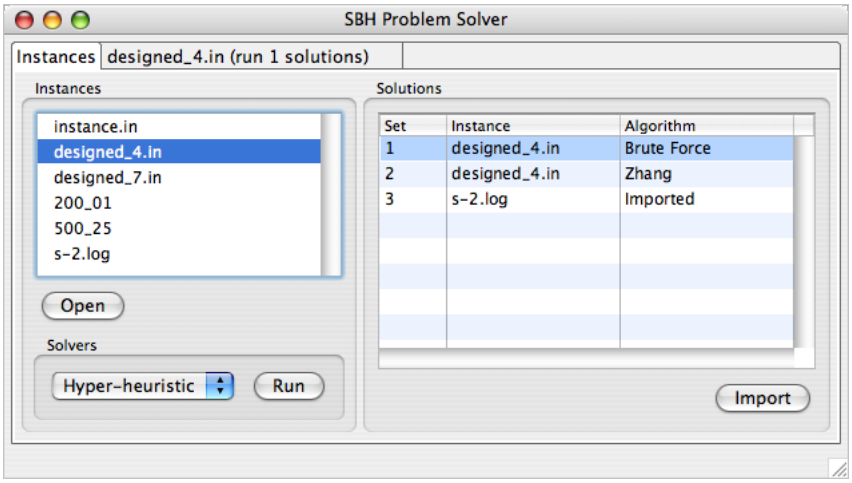


Fig. B.3: Visual analysis tool main window. On the left there is a list of loaded SBH problem instances with the possible solvers. On the right there is a list with completed execution results or imported solutions.

B.2 Visual Tool

To visually observe the solutions to the SBH problem and to ease the procedure of executing the solvers a visual tool was developed. This tool is implemented using Java SWT technology. This tool can be executed using the following command.

```
$ java -Dwojtek.unicode=true -jar sbh-visual.jar
```

The `wojtek.unicode` property allows for using special unicode characters in names of the low-level heuristics and parameters. Not every platform has a complete support of unicode characters that is requirement for this option.

Figure B.3 presents the main window that appears just after execution of the mentioned command. There are two sections: the first one shows a list of opened instances to the SBH problem and allows to open a new instances. It also contains an option to run different kinds of solvers (brute force method, Zhang et al. [41] method or hyper-heuristic search). Second one is the list of solutions sets and is shown on the right side of the main window. Double mouse click on any element of those lists opens a detailed view of instance or solutions. An example of detailed view for solutions of brute force method execution on `designed_4.in` instance file is presented on the Figure B.4. The overview of every solution found during the execution of this solver is listed there. Selecting one of them reveals a details for a solution. This is shown on Figure B.5a. In the top section of this view a visual representation of the solution to the SBH problem is placed. Colours of every nucleotide corresponds to the degree of coverage of this particular nucleotide by oligonucleotides from spectrum. The nucleotides that are covered by relatively small number of oligonucleotides have green or blue colours. The nucleotides with high coverage have red and orange colours. In this view also more detailed statistics like number of unused and used oligonucleotides from spectrum or number of repetitions are shown.

Selecting the hyper-heuristic solver in the main window results in appearance of hyper-heuristic start options presented on Figure B.5b. The set or tree method for solving the

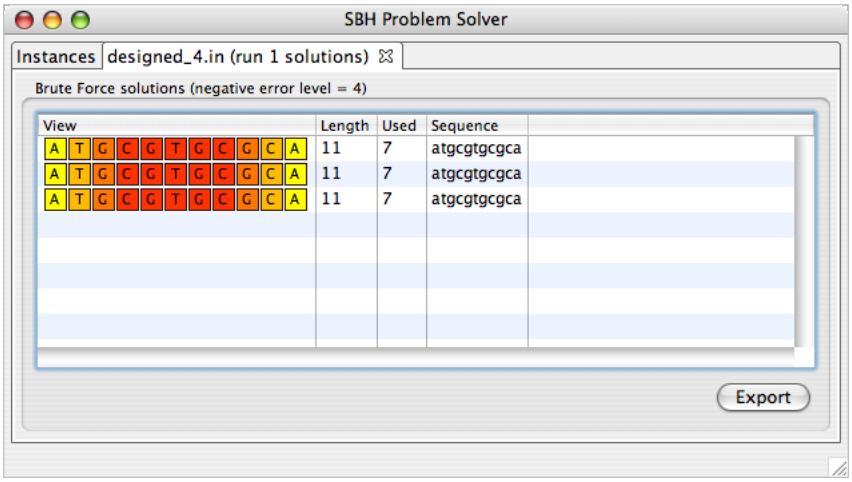
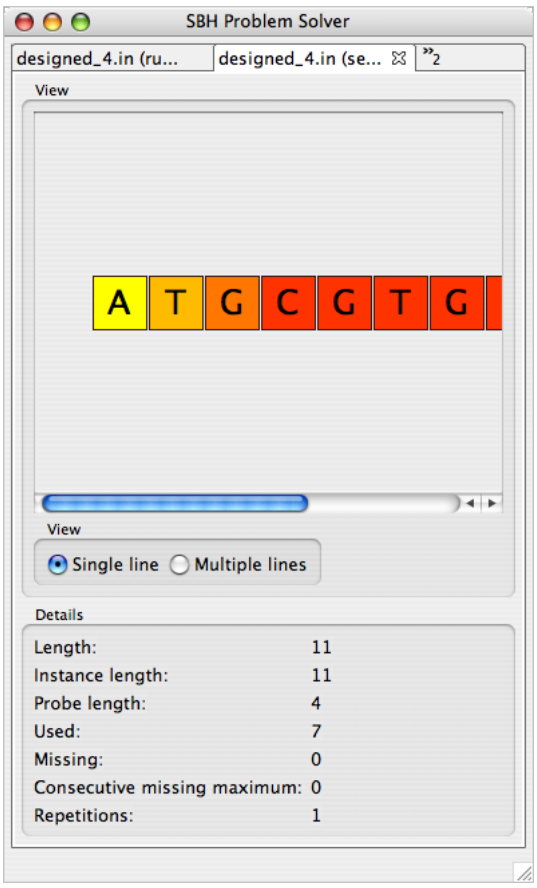
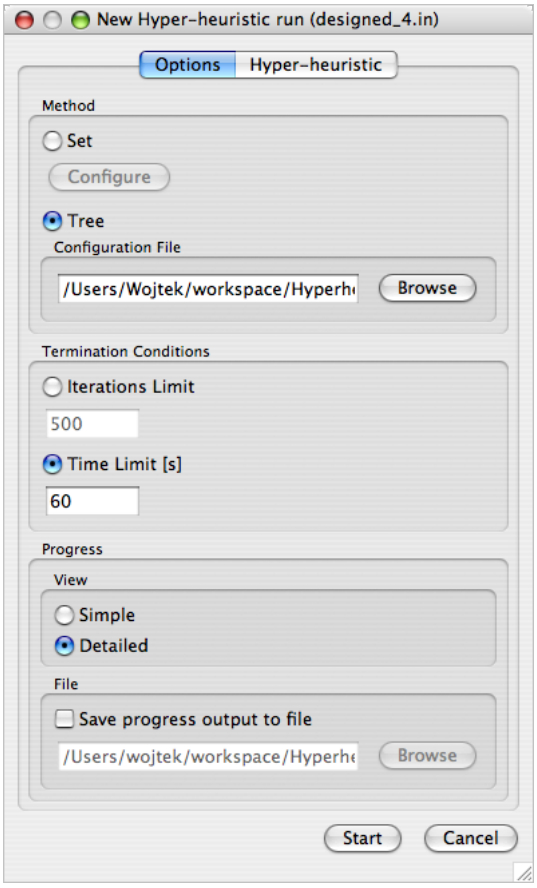


Fig. B.4: List of solutions. Detailed view of the solutions to the SBH problem.



(a) Details view for the particular solution for the SBH problem. Allows to visually analyse the solution and provides a number of solution's statistics.



(b) Hyper-heurisic search start view. Allows to choose the type of solver (*set* or *tree* method), termination conditions and additional search options.

Fig. B.5: Hyper-heuristic solution's details view (a) and start view (b).

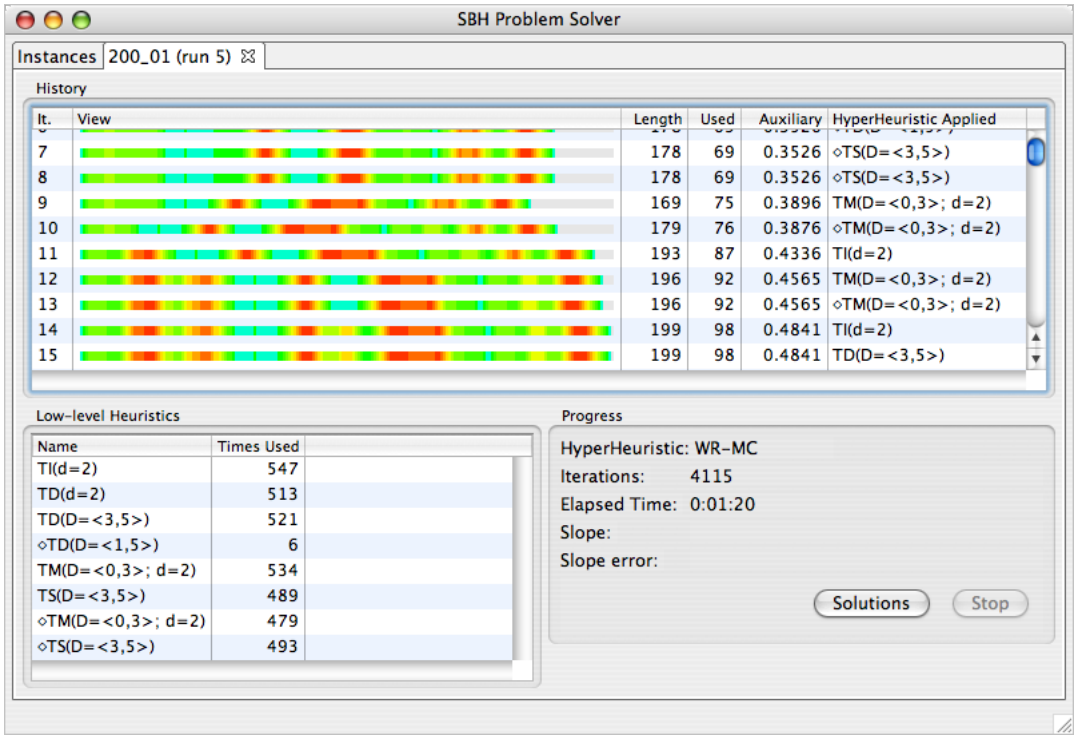


Fig. B.6: Detailed progress view of the hyper-heuristic search. During the search every accepted solution is added to the history of obtained solutions. Also the usage history of low-level heuristics and search progress status is shown.

SBH problem can be used. Tree method is configured using external configuration files that must be selected. Set method is configured using an embedded dialogues. The hyper-heuristic options can be specified on the other form. The termination condition of the search or a type of progress that will be shown can be easily changed. Selecting the detailed progress results in the search status updated on the progress view that is shown on the Figure B.6. This view is divided on three section. In the topmost section every solution that was accepted during the hyper-heuristic search is listed. Every such entry contains a visual overview of the solution, length of the solution, number of oligonucleotides used, an auxiliary search value and the low-level heuristic that created this particular solution. The left bottom section of the progress view contains statistics about low-level heuristics usage, i.e., number of times they were used. The right bottom section contains overall information about the search.

Appendix C

Numerical Results

Results for the benchmark data set are listed in table C.1. Both set and tree methods were analysed. Column i denotes the instance from benchmark data set, i.e., the number corresponds to the index in Table 5.1. Columns M_o and M_a denotes the hyper-heuristics used to obtain the best objective value and the best auxiliary value respectively. If the hyper-heuristic is preceded with star written in superscript then other hyper-heuristics also found the objective value but the one listed was fastest. For set method columns C_o and C_a denote one of four tested configurations used to obtain the best objective and auxiliary value respectively. Values a and b denotes the configurations $\{\mathrm{SI}, \mathrm{SM}, \mathrm{SMC}, \mathrm{SD}\}$ with a using the non-empty initial solution and b using empty initial solution. Values c and d denotes the same configuration respectively with additional SDC low-level heuristic. For tree method columns C_o and C_a denote the configuration used to obtain the best objective and auxiliary values respectively. Configuration C_{43} is not described in Section 5.4. It is equal to $C_{43} = \{\diamond\mathrm{TI}_2, \mathrm{TI}_1, \mathrm{TM}_2^{[0,k]}, \triangledown\mathrm{TM}_3^{[0,k]}, \diamond\mathrm{TS}^{[3,k]}, \mathrm{TS}^{[0,k]}, \triangledown\mathrm{TS}^{[0,k]}, \diamond\mathrm{TD}^{[0,0]}\}$. When star written in superscript precedes the configuration that other configuration also led to the best values but the one listed was the fastest. Columns $f_o^\dagger$ and $f_a^\dagger$ are the best objective and auxiliary values found. Columns $t_o^\dagger$ and $t_a^\dagger$ denote the time in seconds that took to obtain the best solution.

The numerical results obtained for using set and tree hyper-heuristics on biological instances from [1] and [2] are listed in tables C.2 through C.7. Tables C.2, C.3 and C.4 summarises the results for standard instances in data set [1] obtained from sequences without repetitions with 5%, 10% and 20% respectively. Table C.5 summarises results for instance obtained from sequences with repetitions in this data set. In tables C.6 and C.7 are listed the results for instances in data set [2] obtained from random sequences (Table C.6) and real sequences (Table C.7).

Each table has six columns. The first column i is an instance number in a particular data set. Second column o is a number of oligonucleotides from spectrum that are contained in the sequence that was used to generate the particular instance. This is a hint for a theoretical objective value. However, this is not the maximally possible objective value. Some positive errors can lead to better sequences than the one used to generate the particular instance. Third column o' is an objective value obtained running the WrMc set method hyper-heuristic search and t' is the time in seconds when the best solution appeared for a first time during the search. Fifth column o'' is an objective value obtained

	Set method								Tree method							
i	M_a	C_a	$f_a^\dagger$	$t_a^\dagger$	M_o	C_o	$f_o^\dagger$	$t_o^\dagger$	M_a	C_a	$f_a^\dagger$	$t_a^\dagger$	M_o	C_o	$f_o^\dagger$	$t_o^\dagger$
1	UcAm	a	0.574	1.6	UcAm	a	35	1.6	UcAm	42	**0.588**	19.3	UcAm	42	**36**	19.3
2	UcAm	$^*b$	0.439	6.6	UcAm	$^*b$	27	5.7	*UcAm	*45	**0.455**	2.0	*UcAm	*42	**28**	4.7
3	UcAm	a	0.652	7.0	UcAm	$^*a$	66	7.0	UcAm	42	**0.672**	10.8	*UcAm	42	**68**	10.8
4	UcAm	a	0.525	21.6	UcAm	a	53	20.9	UcAm	45	**0.552**	71.1	UcAm	45	**56**	71.1
5	UcAm	a	0.393	6.6	UcAm	a	40	6.6	*ScAm	*42	**0.414**	33.1	*TsAm	*42	**42**	4.0
6	UcAm	a	0.518	33.7	UcAm	a	104	14.0	*TsAm	*42	**0.551**	13.3	*TsAm	*42	**111**	13.3
7	UcAm	b	0.287	41.6	UcAm	b	58	41.5	*WrMc	*45	**0.302**	85.1	*TsAm	*42	**61**	34.0
8	UcAm	a	0.548	27.1	UcAm	a	165	27.1	*ScAm	42	**0.600**	119.8	*ScAm	42	**181**	119.8
9	UcAm	a	0.604	35.2	UcAm	a	182	35.8	*TsAm	42	**0.647**	74.6	*TsAm	42	**195**	74.6
10	UcAm	a	0.295	43.8	UcAm	$^*a$	89	43.2	WrMc	*42	**0.312**	111.6	*WrMc	*42	**94**	31.9
11	UcAm	b	0.186	39.5	*UcAm	$^*c$	56	10.9	WrMc	43	**0.195**	65.4	WrMc	*43	**59**	60.5
12	UcAm	a	0.302	73.7	UcAm	a	121	54.4	*WrMc	*43	**0.324**	137.3	*WrMc	*43	**130**	106.8
13	UcAm	a	0.396	119.4	UcAm	a	159	119.2	UcAm	42	**0.460**	474.2	UcAm	42	**185**	407.1
14	UcAm	a	0.194	64.3	UcAm	a	78	57.6	TsAm	43	**0.199**	184.3	*TsAm	*43	**80**	32.3
15	UcAm	a	0.224	137.6	UcAm	a	90	132.8	UcAm	44	**0.246**	320.9	*UcAm	*44	**99**	320.9
16	UcAm	a	0.357	92.5	UcAm	a	179	92.5	*TsAm	42	**0.383**	163.1	*ScAm	42	**192**	138.1
17	UcAm	b	0.235	54.3	UcAm	b	118	54.3	WrMc	*44	**0.247**	82.5	WrMc	*44	**124**	82.6
18	UcAm	a	0.196	62.4	UcAm	a	98	56.3	WrMc	44	**0.201**	195.5	*ScAm	*44	**101**	78.9
19	UcAm	a	0.228	117.7	UcAm	$^*b$	114	66.2	UcAm	43	**0.255**	500.0	UcAm	43	**128**	490.9
20	UcAm	$^*b$	0.140	73.1	UcAm	$^*b$	70	38.9	WrMc	44	**0.141**	205.3	WrMc	44	**71**	205.1
21	WrMc	a	0.329	128.6	*UcAm	a	264	38.0	ScAm	42	**0.367**	372.2	ScAm	42	**294**	372.2

Tab. C.1: Resuts for the hyper-heuristics for benchmark data set using set and tree methods.

from running the UcAm tree method hyper-heuristic search with a configuration C_{42}. Similarly, t'' is the time in seconds when the best solution appeared for a first time during the search. It should be noted that those are not the best results obtained. For some instance UcAm with C_{42} did not lead to the best solution (seven different combinations of hyper-heuristic search and configurations were checked and there are 162 such cases) but it gives the best results on overall.

n = 200

i	o	o'	t'	o''	t''
1	181	181	0.44	181	0.75
2	181	181	0.44	181	1.03
3	181	181	0.48	181	0.69
4	181	181	0.44	181	1.11
5	181	181	0.44	181	0.94
6	181	181	0.61	181	0.75
7	181	181	0.41	181	0.88
8	181	181	0.45	181	1.08
9	181	181	0.42	181	1.02
10	181	181	0.44	181	0.67
11	181	181	0.66	181	0.78
12	181	181	0.44	181	0.88
13	181	181	0.45	181	0.72
14	181	177	0.53	181	1.03
15	181	181	0.42	181	1.00
16	181	181	0.44	181	0.80
17	181	178	0.47	181	0.92
18	181	181	0.42	181	0.77
19	181	181	0.44	181	0.98
20	181	181	0.47	181	0.88
21	181	181	0.45	181	1.11
22	181	181	0.62	181	0.89
23	181	181	0.45	181	1.03
24	181	181	0.44	181	0.84
25	181	181	0.47	181	0.81
26	181	181	0.42	181	0.95
27	181	181	0.44	181	0.86
28	181	181	0.42	181	1.02
29	181	181	0.45	181	0.98
30	181	181	0.42	181	0.80
31	181	181	0.53	181	0.83
32	181	174	0.47	181	0.94
33	181	181	0.41	181	0.91
34	181	181	0.42	181	0.98
35	181	181	0.42	181	0.95
36	181	181	0.44	181	0.77
37	181	181	0.44	181	0.78
38	181	181	0.42	181	0.94
39	181	181	0.41	181	0.86
40	181	181	0.42	181	0.84

n = 400

i	o	o'	t'	o''	t''
1	371	371	2.34	371	3.95
2	371	371	2.38	366	9.92
3	371	371	2.38	371	4.25
4	371	371	2.39	371	3.88
5	371	371	2.38	371	3.42
6	371	371	2.36	371	4.41
7	371	371	2.41	371	3.69
8	371	364	3.17	367	8.17
9	371	365	3.16	367	5.11
10	371	371	3.80	367	5.31
11	371	371	2.33	371	5.00
12	371	367	2.34	371	5.53
13	371	371	2.45	371	5.08
14	371	371	3.31	371	4.16
15	371	371	2.36	371	4.73
16	371	371	2.41	371	4.33
17	371	365	6.50	371	3.25
18	371	371	2.41	371	3.92
19	371	371	2.39	371	4.58
20	371	371	2.34	371	4.16
21	371	371	2.39	371	4.09
22	371	371	2.44	371	4.31
23	371	371	2.45	371	3.58
24	371	369	2.44	371	4.11
25	371	371	2.44	371	3.48
26	371	371	2.45	371	4.59
27	371	371	2.44	371	4.28
28	371	371	2.44	371	3.34
29	371	371	2.44	371	4.31
30	371	371	2.41	371	5.88
31	371	371	2.45	371	3.42
32	371	371	2.47	371	5.95
33	371	371	2.36	371	4.41
34	371	371	2.41	371	3.73
35	371	371	2.33	371	4.78
36	371	371	2.34	371	4.03
37	371	371	2.38	371	3.72
38	371	371	2.42	371	4.16
39	371	371	2.39	371	4.88
40	371	371	2.39	371	4.08

n = 500

i	o	o'	t'	o''	t''
1	466	466	4.58	466	11.50
2	466	466	4.59	466	10.53
3	466	457	5.66	466	11.92
4	466	466	4.62	466	7.84
5	466	466	4.59	466	8.23
6	466	466	4.64	466	9.98
7	466	452	5.28	466	8.17
8	466	461	5.25	463	11.98
9	466	461	6.55	466	11.45
10	466	464	4.59	464	11.72
11	466	466	5.94	466	9.62
12	466	461	5.58	466	9.09
13	466	466	4.62	466	8.58
14	466	459	4.58	455	12.19
15	466	466	4.53	466	9.95
16	466	466	4.75	466	7.50
17	466	461	4.62	466	10.16
18	466	466	4.59	466	9.84
19	466	466	4.55	466	9.02
20	466	466	4.56	466	8.44
21	466	466	4.66	466	9.31
22	466	466	4.67	466	7.70
23	466	466	4.59	466	9.58
24	466	464	4.67	465	11.77
25	466	466	4.67	466	8.09
26	466	466	4.62	466	9.11
27	466	466	4.58	466	9.70
28	466	466	4.62	466	8.62
29	466	466	5.92	466	7.42
30	466	466	5.05	466	6.95
31	466	466	4.62	466	10.12
32	466	466	4.61	466	11.59
33	466	466	4.98	466	7.39
34	466	466	4.58	466	6.69
35	466	466	4.61	466	7.06
36	466	466	4.59	466	6.20
37	466	466	4.67	462	10.38
38	466	456	4.75	466	9.05
39	466	466	4.67	466	6.77
40	466	466	4.67	466	9.69

n = 600

i	o	o'	t'	o''	t''
1	561	561	10.89	515	15.19
2	561	561	8.22	561	14.52
3	561	561	8.03	561	11.41
4	561	561	8.03	561	14.16
5	561	561	8.02	537	14.88
6	561	553	9.84	561	14.16
7	561	561	8.06	555	15.00
8	561	547	13.47	561	14.36
9	561	557	13.67	550	14.80
10	561	550	11.23	561	13.28
11	561	561	7.83	561	13.33
12	561	556	7.97	541	14.34
13	561	561	8.00	561	13.02
14	561	561	8.22	555	15.00
15	561	561	8.09	546	14.97
16	561	561	8.16	544	15.03
17	561	555	8.06	544	15.12
18	561	561	8.14	561	15.06
19	561	561	8.22	561	14.91
20	561	558	7.77	561	13.09
21	561	561	7.88	561	12.69
22	561	561	7.86	561	14.44
23	561	561	7.80	544	15.14
24	561	548	8.36	558	12.12
25	561	554	12.30	561	13.30
26	561	561	7.81	555	11.95
27	561	561	10.34	556	14.61
28	561	561	7.83	560	15.14
29	561	561	7.86	561	14.31
30	561	561	7.75	561	13.91
31	561	561	7.89	561	13.91
32	561	561	7.80	550	15.19
33	561	561	7.86	558	15.03
34	561	561	7.70	561	13.56
35	561	561	7.77	561	13.50
36	561	561	7.84	561	15.16
37	561	561	7.89	561	12.84
38	561	561	8.06	561	13.89
39	561	555	9.78	531	15.09
40	561	561	7.98	561	13.41

Tab. C.2: BioServer [1] standard instances without repetitions, 5% errors.

n = 200

i	o	o'	t'	o''	t''
1	171	171	0.44	171	0.97
2	171	171	0.42	171	1.00
3	171	171	0.56	171	0.84
4	171	171	0.42	171	1.05
5	171	171	0.48	171	1.00
6	171	171	1.14	171	1.02
7	171	171	0.44	171	1.06
8	171	171	0.44	171	1.00
9	171	171	0.45	171	1.19
10	171	171	0.44	171	1.11
11	171	171	0.52	171	0.88
12	171	171	0.42	171	0.91
13	171	171	0.47	171	0.81
14	171	167	0.66	171	0.91
15	171	171	0.53	171	0.83
16	171	171	0.42	171	0.92
17	171	169	0.44	171	1.03
18	171	171	0.73	171	0.92
19	171	171	0.45	171	0.91
20	171	171	0.41	171	0.83
21	171	171	0.44	171	0.77
22	171	171	0.44	171	0.81
23	171	171	0.42	171	1.06
24	171	171	0.56	171	0.80
25	171	171	0.42	171	0.86
26	171	171	0.42	171	1.03
27	171	171	0.44	171	0.91
28	171	171	0.42	171	0.73
29	171	171	0.42	171	0.91
30	171	171	0.42	171	0.88
31	171	171	0.55	171	0.81
32	171	171	0.59	171	1.16
33	171	171	0.44	171	1.09
34	171	171	0.62	171	0.83
35	171	171	0.41	171	1.22
36	171	171	0.44	171	0.89
37	171	171	0.42	171	1.06
38	171	171	0.41	171	0.98
39	171	171	0.44	171	0.88
40	171	171	0.53	171	1.00

n = 400

i	o	o'	t'	o''	t''
1	351	351	2.41	344	5.78
2	351	351	2.42	351	3.92
3	351	344	4.08	351	4.20
4	351	351	2.42	351	4.52
5	351	351	2.38	351	3.75
6	351	341	4.00	348	3.80
7	351	351	2.38	351	3.27
8	351	347	3.91	351	5.30
9	351	346	2.67	351	5.12
10	351	339	8.41	348	6.50
11	351	351	2.38	351	3.64
12	351	348	3.03	348	10.14
13	351	351	2.33	351	4.34
14	351	351	2.95	351	4.67
15	351	347	2.33	351	4.53
16	351	351	2.36	351	5.42
17	351	344	3.66	351	4.80
18	351	351	2.42	351	4.88
19	351	351	2.38	351	4.48
20	351	351	2.36	351	5.59
21	351	351	2.41	351	4.06
22	351	351	2.41	351	4.75
23	351	351	2.59	351	9.64
24	351	344	5.94	351	4.80
25	351	351	2.41	351	4.45
26	351	351	2.39	351	5.97
27	351	351	2.36	351	4.12
28	351	351	2.39	351	5.34
29	351	351	2.39	351	3.97
30	351	347	2.48	351	4.67
31	351	351	2.39	351	5.36
32	351	351	2.36	351	5.95
33	351	351	2.39	351	3.81
34	351	351	2.34	351	4.14
35	351	351	2.33	351	3.97
36	351	351	2.50	346	4.14
37	351	351	2.38	351	4.67
38	351	351	2.44	351	4.77
39	351	351	3.53	351	5.23
40	351	338	3.39	351	7.20

n = 500

i	o	o'	t'	o''	t''
1	441	441	4.53	441	8.84
2	441	441	4.52	441	8.81
3	441	437	8.27	441	8.33
4	441	441	4.48	441	10.31
5	441	441	4.53	441	8.41
6	441	436	11.52	439	11.03
7	441	429	8.80	441	8.36
8	441	434	7.67	436	9.38
9	441	429	7.00	435	9.09
10	441	439	4.47	437	8.69
11	441	441	4.50	441	7.17
12	441	437	5.14	441	7.62
13	441	441	4.53	441	7.91
14	441	433	4.56	432	11.08
15	441	431	7.70	441	9.17
16	441	441	5.66	441	6.38
17	441	438	6.27	439	9.61
18	441	441	5.41	441	6.58
19	441	440	4.47	441	7.50
20	441	441	4.50	441	9.05
21	441	441	4.50	441	8.16
22	441	441	4.53	441	6.91
23	441	441	4.48	441	9.19
24	441	441	4.58	441	7.38
25	441	434	4.70	441	7.88
26	441	441	4.52	441	6.03
27	441	441	4.41	441	8.09
28	441	441	4.53	441	6.50
29	441	441	4.62	441	8.61
30	441	424	4.33	441	6.73
31	441	441	5.16	441	7.27
32	441	441	4.47	441	7.28
33	441	441	4.52	441	7.98
34	441	441	4.50	441	5.02
35	441	441	4.53	441	6.28
36	441	435	5.89	441	10.72
37	441	441	4.59	441	8.17
38	441	434	4.66	441	7.20
39	441	434	4.52	441	9.33
40	441	438	7.02	441	9.17

n = 600

i	o	o'	t'	o''	t''
1	531	531	10.80	531	13.06
2	531	531	7.81	531	10.75
3	531	531	7.77	531	11.59
4	531	531	7.83	531	12.78
5	531	531	12.84	531	11.41
6	531	526	10.22	531	12.88
7	531	531	7.81	531	11.59
8	531	520	11.47	523	15.06
9	531	525	8.50	529	11.56
10	531	530	14.41	530	11.20
11	531	531	7.75	528	14.78
12	531	531	7.91	524	12.64
13	531	531	7.81	531	14.91
14	531	531	8.00	525	14.86
15	531	531	7.83	531	14.94
16	531	523	10.58	524	11.17
17	531	517	13.91	522	14.72
18	531	526	7.80	531	15.09
19	531	524	7.73	531	12.91
20	531	521	10.56	531	11.44
21	531	531	7.98	531	14.86
22	531	531	7.91	531	14.66
23	531	525	11.12	526	14.69
24	531	531	15.08	528	13.09
25	531	518	7.89	531	13.20
26	531	531	7.80	531	11.72
27	531	531	7.77	527	12.91
28	531	531	7.91	531	10.98
29	531	519	14.47	531	13.91
30	531	531	7.81	531	13.11
31	531	531	7.89	531	13.05
32	531	531	7.83	531	10.25
33	531	531	8.97	531	12.95
34	531	522	9.66	526	11.48
35	531	531	7.75	531	14.67
36	531	531	7.66	531	12.89
37	531	531	7.81	531	11.45
38	531	505	14.94	531	14.09
39	531	526	12.34	531	11.38
40	531	531	7.86	531	12.75

Tab. C.3: BioServer [1] standard instances without repetitions, 10% errors.

$n = 200$

i	o	o'	t'	o''	t''
1	152	152	0.42	152	1.11
2	152	152	0.41	152	1.05
3	152	152	0.72	152	1.09
4	152	149	0.72	152	0.94
5	152	152	0.39	152	0.83
6	152	152	0.86	152	1.11
7	152	152	0.41	152	0.91
8	152	152	0.41	152	0.92
9	152	152	0.41	152	0.94
10	152	152	0.41	152	1.20
11	152	152	1.12	152	1.08
12	152	146	0.56	152	1.27
13	152	152	0.39	152	0.98
14	152	152	0.41	152	0.92
15	152	152	0.41	152	1.02
16	152	152	0.81	152	0.81
17	152	150	0.42	152	1.17
18	152	152	0.56	152	1.11
19	152	152	0.39	152	1.05
20	152	148	0.44	152	1.03
21	152	152	0.39	152	0.98
22	152	152	0.38	152	1.16
23	152	150	0.73	152	0.83
24	152	152	0.67	152	1.11
25	152	152	0.41	152	1.66
26	152	152	0.55	152	1.28
27	152	152	0.48	152	1.33
28	152	152	0.62	152	0.83
29	152	152	0.42	152	0.86
30	152	147	0.42	152	1.17
31	152	152	0.41	152	0.92
32	152	152	0.69	152	1.03
33	152	152	0.42	152	1.03
34	152	152	0.61	152	1.00
35	152	152	0.42	152	1.16
36	152	152	0.39	152	1.14
37	152	152	0.67	152	0.84
38	152	152	0.53	152	1.17
39	152	152	0.41	152	0.78
40	152	152	0.58	152	1.00

$n = 400$

i	o	o'	t'	o''	t''
1	312	312	2.23	312	4.39
2	312	312	2.97	312	3.69
3	312	312	5.80	312	4.17
4	312	307	2.28	312	4.41
5	312	312	2.22	312	4.78
6	312	312	2.25	312	4.73
7	312	312	2.52	312	5.80
8	312	303	3.27	312	5.38
9	312	309	6.31	312	4.11
10	312	303	5.14	309	6.22
11	312	307	5.88	312	4.88
12	312	309	3.22	309	9.66
13	312	309	2.19	312	4.45
14	312	312	2.19	312	4.47
15	312	309	2.17	312	5.27
16	312	307	3.61	312	6.25
17	312	295	4.09	312	5.14
18	312	312	2.25	312	5.48
19	312	312	2.25	312	5.97
20	312	312	3.02	312	6.47
21	312	311	2.23	312	5.42
22	312	307	3.22	312	4.12
23	312	312	3.33	312	4.70
24	312	291	4.50	312	4.89
25	312	312	2.20	312	5.66
26	312	312	2.27	312	4.00
27	312	312	2.25	312	4.09
28	312	312	2.30	312	4.64
29	312	304	5.39	312	4.47
30	312	312	2.25	312	4.06
31	312	312	4.05	312	5.33
32	312	295	5.27	312	6.55
33	312	312	2.16	312	5.69
34	312	312	2.16	312	4.56
35	312	312	3.95	312	5.38
36	312	312	2.16	312	5.50
37	312	305	5.75	312	4.86
38	312	312	8.12	312	6.11
39	312	308	6.84	312	4.83
40	312	303	7.50	308	4.92

$n = 500$

i	o	o'	t'	o''	t''
1	392	386	8.67	391	8.97
2	392	390	8.05	392	9.86
3	392	385	8.20	392	9.53
4	392	388	9.23	392	8.48
5	392	392	4.17	392	9.48
6	392	380	4.16	392	9.44
7	392	382	4.12	386	11.88
8	392	386	5.28	392	9.08
9	392	382	7.61	392	7.88
10	392	377	7.81	392	9.98
11	392	392	4.16	392	9.41
12	392	382	5.00	392	9.23
13	392	392	10.78	392	11.56
14	392	372	5.00	389	11.16
15	392	374	11.92	392	10.62
16	392	389	4.16	392	10.56
17	392	387	4.16	388	11.05
18	392	382	4.09	388	12.19
19	392	392	4.25	392	9.55
20	392	390	4.17	389	10.88
21	392	392	4.41	392	8.89
22	392	391	4.33	392	8.23
23	392	392	6.88	392	9.34
24	392	381	4.23	390	10.67
25	392	389	8.28	392	8.81
26	392	391	4.25	391	11.27
27	392	392	4.23	392	10.34
28	392	387	6.56	392	12.16
29	392	388	7.42	392	11.03
30	392	377	4.59	387	12.09
31	392	380	6.89	392	11.88
32	392	373	4.03	392	9.95
33	392	388	12.12	392	10.78
34	392	376	10.52	392	7.97
35	392	392	4.23	392	11.34
36	392	392	4.19	387	11.55
37	392	375	11.48	392	7.53
38	392	385	5.62	392	7.39
39	392	372	8.53	392	11.70
40	392	387	10.02	392	8.16

$n = 600$

i	o	o'	t'	o''	t''
1	472	472	9.92	466	14.17
2	472	472	12.91	443	14.98
3	472	472	7.30	457	14.95
4	472	472	7.30	443	15.06
5	472	472	12.80	451	14.84
6	472	462	7.23	463	14.95
7	472	472	7.30	464	14.69
8	472	446	6.89	454	14.89
9	472	462	12.20	462	15.03
10	472	452	13.89	458	15.12
11	472	462	15.03	419	14.86
12	472	472	13.55	459	14.70
13	472	472	7.28	429	15.17
14	472	456	11.34	450	15.02
15	472	472	13.61	462	15.06
16	472	469	14.98	471	14.58
17	472	457	14.28	450	14.67
18	472	469	13.02	450	15.12
19	472	465	11.31	472	15.16
20	472	467	9.17	472	14.28
21	472	466	12.75	452	15.02
22	472	467	7.92	468	15.19
23	472	430	6.64	457	14.84
24	472	459	11.02	463	15.05
25	472	447	12.09	453	14.20
26	472	472	7.17	456	14.70
27	472	472	7.12	459	14.36
28	472	472	10.77	464	14.73
29	472	457	12.64	449	15.05
30	472	465	11.81	469	13.89
31	472	472	11.80	472	13.44
32	472	469	10.23	461	14.45
33	472	450	13.20	468	14.12
34	472	468	10.84	472	13.91
35	472	464	7.22	468	14.81
36	472	472	7.11	468	15.06
37	472	466	11.22	468	14.34
38	472	436	6.69	462	15.19
39	472	461	14.69	455	15.22
40	472	460	14.75	472	13.72

Tab. C.4: BioServer [1] standard instances without repetitions, 20% errors.

0% errors						5% errors					
i	o	o'	t'	o''	t''	i	o	o'	t'	o''	t''
1	589	583	7.83	589	14.72	1	561	555	7.95	555	13.25
2	588	567	7.88	583	14.84	2	561	556	10.88	561	12.73
3	587	577	7.72	587	10.52	3	561	537	7.88	561	12.22
4	587	587	7.98	587	13.44	4	561	561	8.11	542	14.36
5	589	585	8.27	581	11.75	5	561	551	8.22	552	14.28
6	586	572	7.97	580	12.23	6	561	561	8.05	560	12.89
7	586	581	7.94	584	11.64	7	561	556	14.94	556	13.39
8	587	583	7.86	585	15.06	8	561	548	8.14	553	12.28
9	589	589	7.80	589	13.97	9	561	561	8.19	561	10.48
10	589	577	8.05	581	11.91	10	561	546	7.94	552	13.77
11	584	580	11.70	577	13.53	11	561	550	12.06	561	13.27
12	589	584	8.44	581	14.88	12	561	561	7.95	560	14.28
13	585	585	7.86	585	11.67	13	561	561	8.08	561	11.94
14	585	583	9.92	585	13.11	14	561	557	8.20	561	13.62
15	584	584	7.81	583	11.77	15	561	561	8.05	560	13.75
16	574	574	7.38	574	10.55	16	561	562	8.86	561	11.50
17	583	583	7.89	583	14.34	17	561	561	8.06	561	12.83
18	589	589	7.94	588	11.05	18	561	561	7.94	560	14.91
19	589	582	9.86	583	14.33	19	561	561	7.95	561	13.31
20	580	580	9.62	578	15.09	20	561	561	12.55	561	9.58
21	589	585	10.28	582	12.94	21	561	561	9.39	548	14.78
22	584	584	7.27	584	11.73	22	561	552	13.03	558	13.64
23	589	589	7.98	589	11.83	23	561	561	7.92	552	10.84
24	589	583	8.09	576	12.75	24	561	549	7.84	556	12.36
25	585	581	7.83	582	12.09	25	561	557	8.12	561	14.73
26	587	587	8.16	587	12.12	26	561	561	11.86	561	11.89
27	585	585	8.11	581	11.05	27	561	556	11.42	557	13.08
28	587	587	7.98	583	12.83	28	561	561	8.05	555	14.66
29	586	586	7.98	585	14.17	29	561	561	11.31	554	15.00
30	589	577	8.02	569	13.17	30	561	541	13.19	553	13.56
31	579	579	7.58	579	11.86	31	561	561	7.97	561	13.30
32	590	590	7.97	590	11.97	32	561	561	7.97	561	13.72
33	590	585	9.56	582	12.12	33	561	543	8.66	559	15.14
34	586	581	7.84	578	10.33	34	561	557	7.92	555	11.50
35	589	584	8.95	581	14.55	35	561	561	7.86	552	14.70
36	588	588	7.78	588	12.64	36	561	561	7.83	561	12.33
37	587	581	7.80	576	14.92	37	561	551	7.73	557	14.27
38	590	586	8.03	569	15.16	38	561	558	7.92	553	14.91
39	586	580	10.52	572	12.50	39	561	550	9.75	552	14.20
40	584	583	10.41	579	11.73	40	561	559	10.48	560	13.30

Tab. C.5: BioServer [1] instances with repetitions of desired length $n = 600$.

n = 109					
i	o	o'	t'	o''	t''
1	80	80	0.22	80	0.56
2	80	80	0.19	80	0.41
3	80	80	0.20	80	0.45
4	80	80	0.20	80	0.53
5	80	80	0.19	80	0.50
6	80	80	0.23	80	0.45
7	80	80	0.23	80	0.45
8	80	80	0.28	80	0.47
9	80	80	0.19	80	0.53
10	80	80	0.20	80	0.47
11	80	80	0.30	80	0.47
12	80	80	0.33	80	0.55
13	80	80	0.20	80	0.44
14	80	80	0.28	80	0.39
15	80	80	0.30	80	0.56
16	80	74	0.20	80	0.64
17	80	80	0.20	80	0.44
18	80	80	0.19	80	0.50
19	80	80	0.20	80	0.44
20	80	80	0.25	80	0.45
21	80	80	0.23	80	0.56
22	80	80	0.33	80	0.52
23	80	79	0.28	80	0.55
24	80	80	0.19	80	0.42
25	80	80	0.19	80	0.42
26	80	80	0.30	80	0.52
27	80	80	0.19	80	0.42
28	80	80	0.22	80	0.50
29	80	80	0.25	80	0.42
30	80	80	0.19	80	0.47
31	80	80	0.20	80	0.42
32	80	80	0.19	80	0.47
33	80	80	0.30	80	0.48
34	80	80	0.20	80	0.47
35	80	80	0.33	80	0.52
36	80	80	0.28	80	0.53
37	80	80	0.20	80	0.42
38	80	80	0.19	80	0.39
39	80	80	0.20	80	0.47
40	80	80	0.19	80	0.48

n = 209					
i	o	o'	t'	o''	t''
1	160	156	0.73	160	1.11
2	160	152	0.67	160	1.62
3	160	154	0.53	160	1.02
4	160	160	0.44	160	1.56
5	160	160	0.59	160	1.25
6	160	160	2.20	160	1.11
7	160	160	0.45	160	1.11
8	160	156	0.50	160	1.31
9	160	160	0.50	160	1.09
10	160	158	4.44	160	1.02
11	160	159	0.42	160	1.22
12	160	160	0.64	160	1.12
13	160	160	0.44	160	0.95
14	160	160	0.42	160	1.38
15	160	160	0.45	160	1.06
16	160	160	0.62	160	1.19
17	160	157	0.91	160	0.92
18	160	160	0.44	160	0.94
19	160	160	0.41	160	1.12
20	160	160	0.44	160	1.66
21	160	160	0.44	160	1.14
22	160	160	0.44	160	1.05
23	160	160	0.48	160	1.16
24	160	156	1.08	160	1.05
25	160	160	0.70	160	0.95
26	160	154	0.81	160	1.05
27	160	160	0.70	160	1.09
28	160	158	0.61	160	1.48
29	160	160	0.45	160	1.61
30	160	160	0.42	160	1.08
31	160	158	1.02	160	1.22
32	160	160	0.47	160	0.89
33	160	160	0.64	160	0.88
34	160	160	0.44	160	0.91
35	160	160	0.41	160	0.91
36	160	160	0.44	160	1.31
37	160	160	0.41	160	1.19
38	160	160	0.47	160	1.66
39	160	160	0.44	160	1.03
40	160	160	0.44	160	1.08

n = 309					
i	o	o'	t'	o''	t''
1	240	235	1.77	240	2.56
2	240	231	1.78	240	2.58
3	240	240	1.08	240	2.48
4	240	240	1.41	240	2.48
5	240	236	1.08	240	2.16
6	240	240	1.09	240	2.77
7	240	240	1.48	240	2.67
8	240	240	2.59	240	2.23
9	240	240	1.92	240	2.39
10	240	240	1.11	240	2.41
11	240	234	1.73	240	2.02
12	240	240	1.86	240	2.48
13	240	240	1.11	240	2.70
14	240	240	1.86	240	2.31
15	240	234	3.44	240	2.41
16	240	237	1.67	240	2.47
17	240	240	1.86	240	3.38
18	240	240	2.41	240	2.38
19	240	240	1.25	240	3.62
20	240	240	1.12	240	2.77
21	240	240	1.58	240	5.80
22	240	240	1.66	240	3.14
23	240	240	1.08	240	2.11
24	240	240	1.62	240	3.00
25	240	236	1.64	240	2.08
26	240	240	1.59	240	2.45
27	240	238	1.88	240	3.17
28	240	240	1.12	240	2.30
29	240	240	1.05	240	2.44
30	240	240	1.06	240	3.91
31	240	240	1.08	240	2.38
32	240	240	1.34	240	3.02
33	240	240	1.91	240	2.44
34	240	240	1.08	240	2.23
35	240	240	1.09	240	2.56
36	240	240	1.27	240	3.28
37	240	240	1.09	240	2.20
38	240	240	1.88	238	2.98
39	240	240	1.05	240	2.44
40	240	236	1.67	238	3.59

Tab. C.6: BioServer [2] instances derived from random sequences with 20% errors.

n = 109					
i	o	o'	t'	o''	t''
1	80	80	0.39	80	0.55
2	80	80	0.22	80	0.44
3	80	80	0.19	80	0.45
4	80	80	0.30	80	0.42
5	80	80	0.19	80	0.50
6	80	80	0.22	80	0.64
7	80	80	0.20	80	0.53
8	80	80	0.31	80	0.44
9	80	80	0.33	80	0.47
10	80	80	0.28	80	0.52
11	80	80	0.20	80	0.77
12	80	80	0.33	80	0.42
13	80	80	0.31	80	0.45
14	80	80	0.34	80	0.42
15	80	80	0.36	80	0.41
16	80	80	0.31	80	0.48
17	80	76	0.36	80	0.41
18	80	80	0.30	80	0.41
19	80	80	0.31	80	0.42
20	80	80	0.20	80	0.41
21	80	80	0.20	80	0.47
22	80	80	0.20	80	0.39
23	80	80	0.20	80	0.42
24	80	80	0.20	80	0.48
25	80	80	0.22	80	0.47
26	80	80	0.34	80	0.44
27	80	78	0.36	80	0.47
28	80	80	0.22	80	0.45
29	80	79	0.22	80	0.47
30	80	80	0.36	80	0.47
31	80	80	0.22	80	0.50
32	80	80	0.31	80	0.48
33	80	80	0.20	80	0.53
34	80	80	0.20	80	0.45
35	80	80	0.23	80	0.41
36	80	80	0.30	80	0.38
37	80	80	0.38	80	0.41
38	80	80	0.33	80	0.42
39	80	80	0.28	80	0.50
40	80	80	0.19	80	0.50

n = 209					
i	o	o'	t'	o''	t''
1	160	158	2.34	160	0.97
2	160	160	0.66	160	0.98
3	160	157	0.62	160	1.16
4	160	160	0.44	160	0.86
5	160	160	0.44	160	1.00
6	160	160	0.45	160	1.34
7	160	156	0.42	157	2.81
8	160	160	0.70	160	1.16
9	160	160	1.98	160	1.09
10	160	160	2.81	160	1.06
11	160	152	1.92	160	0.98
12	160	160	0.45	160	1.19
13	160	154	0.83	160	0.92
14	160	160	0.45	160	1.11
15	160	160	0.44	160	1.05
16	160	160	0.72	160	1.31
17	160	160	0.86	160	1.28
18	160	160	0.47	160	1.25
19	160	160	0.45	160	1.05
20	160	160	0.42	160	1.14
21	160	160	0.45	160	0.95
22	160	160	0.44	160	1.00
23	160	160	0.45	160	1.12
24	160	157	1.09	160	2.58
25	160	160	0.69	160	0.97
26	160	160	0.44	160	0.95
27	160	156	0.48	160	0.97
28	160	155	0.70	160	4.31
29	160	160	0.44	160	1.33
30	160	156	0.72	158	3.58
31	160	155	0.91	160	1.09
32	160	160	0.44	160	1.06
33	160	160	0.69	160	1.84
34	160	160	0.67	160	1.00
35	160	156	0.45	160	1.31
36	160	160	0.69	160	1.17
37	160	160	0.61	160	2.05
38	160	158	0.64	160	1.28
39	160	153	0.56	160	1.19
40	160	160	0.62	160	0.83

n = 309					
i	o	o'	t'	o''	t''
1	240	232	1.55	240	2.88
2	240	240	1.28	240	2.16
3	240	234	3.16	240	2.86
4	240	240	1.09	240	2.52
5	240	235	2.02	240	2.81
6	240	240	1.56	240	5.08
7	240	230	1.11	240	2.47
8	240	237	1.64	240	2.30
9	240	234	1.67	240	3.62
10	240	238	2.09	240	2.61
11	240	217	1.14	237	3.73
12	240	231	1.66	240	2.47
13	240	235	1.11	240	2.36
14	240	240	1.09	240	3.06
15	240	222	2.50	240	2.58
16	240	235	3.09	240	2.91
17	240	236	1.62	240	2.11
18	240	240	1.12	240	4.94
19	240	234	5.31	237	1.62
20	240	240	1.31	240	1.89
21	240	240	1.11	240	2.31
22	240	240	1.12	240	2.11
23	240	240	1.09	240	3.00
24	240	240	1.12	240	2.20
25	240	240	2.16	237	2.00
26	240	240	1.16	240	4.81
27	240	240	1.67	240	3.25
28	240	240	2.14	240	2.58
29	240	236	1.70	240	3.06
30	240	240	1.69	240	2.20
31	240	240	1.56	240	2.56
32	240	236	7.00	239	3.16
33	240	238	1.98	240	1.88
34	240	240	1.08	240	2.66
35	240	240	1.17	240	2.62
36	240	240	1.42	240	3.58
37	240	237	1.67	240	2.50
38	240	235	2.53	240	3.39
39	240	234	1.11	238	2.78
40	240	240	1.28	240	2.22

Tab. C.7: BioServer [2] instances derived from real sequences with 20% errors.