Diplomarbeit

Algorithmic Approaches to the String Barcoding Problem

ausgeführt am

Institut für Computergraphik und Algorithmen

der Technischen Universität Wien

unter Anleitung von

Univ.-Prof. Dipl.-Ing. Dr. techn. Günther Raidl

durch

Philipp Neuner

Pfeilgasse 14/2/5

1080 Wien

....

Datum Unterschrift

Abstract

This thesis deals with a heuristic approach based on Lagrangian relaxation to the $s$tring barcoding (SB) problem, a close cousin to the well-known combinatorial set cover (SC) problem. It has recently been proven to be NP-hard and has many real-world applications, particularly in the fields of medicine and biology.

Given a set of sequences over some alphabet, DNA for instance, we aim at finding a set of short sequences, so-called probes, such that we are able to identify an unknown sample sequence as one of the input sequences by determining which probes are subsequences of the sample, and which are not. The problem is twofold: the determination of all possible probes and the selection of a suitable subset of minimum cardinality.

The problem has been dealt with under various other names and has in this form been introduced by Rash and Gusfield in 2002. They proposed an exact approach based on integer linear programming and the use of suffix trees to generate a complete, non-redundant set of candidate probes.

We evaluated several approaches for the SB as well as the SC problem. One of the leading heuristics for the SC problem, based on Lagrangian relaxation, has been proposed by Caprara et al. in 1999. We adapted the algorithm to see if it works equally well when applied to the structurally very similar SB problem. Though the results we obtained are somewhat mixed, the heuristic shows its strength with very complex instances and delivers much better results compared to simpler heuristics.

Contents

1 Introduction

A common task in medical and biological applications is the identification of biological agents like bacteria, viruses, proteins, and the like. Basically, the problem is the same regardless of the exact kind of agent we are dealing with. To keep things simple we will consider only nucleotide sequences hereinafter. Nucleotides are the structural units of deoxyribonucleic and ribonucleic acids, commonly known as DNA and RNA, which carry the genetic code of all living organisms and viruses. They consist of a sugar group (ribose or deoxyribose), one or more phosphate groups, and one base. The four bases you may find in a DNA molecule are adenine (A), cytosine (C), guanine (G), and thymine (T). The same goes for RNA molecules with the only exception being that uracil (U) takes the place of thymine. Each base has a complementary counterpart to which it bonds exclusively. Accordingly, the bases can be divided into the complementary pairs A-T and C-G. The reaction that takes place when two complementary nucleotide sequences bond to each other is called hybridization.

Let us assume we are given a sample containing the DNA or RNA of some unknown virus and a list of viruses that we think may be contained in the sample. The most common techniques to match a genomic sample against a list of candidates rely on sequence alignments to find a perfect or near-perfect match and often incorporate heuristic elements for the sake of efficiency. A very widespread representative of this class of procedures is BLAST [29], which is short for *basic local alignment search tool*. These methods, however, can be very time consuming and require the sequence we want to identify to be available explicitly, i. e. we must perform a comprehensive *sequence analysis* to determine the exact succession of comprising nucleotides. Unfortunately, this is most often technically complex and expensive.

A similar problem arises when one strives to determine the gene expression of a cell, i. e. the genes that are active in the cell and the degree of activity. A gene is basically just a part of the DNA that encodes a particular protein. If a gene is active, it is transcribed into messenger RNA (mRNA) molecules which are complementary the the respective gene. The mRNA molecules are transferred to the cell's ribosome where the protein is actually build. Thus, the kinds and amount of mRNA molecules found in a cell gives insight to its gene expression. When it comes to the identification of a particular mRNA molecule we face the exact same problem as stated above.

Instead of comparing the complete sample sequence with possible candidates, we may alternatively check the sample for the presence or absence of short nucleotide sequences, so-called oligonucleotides, which are typically between five and 50 nucleotides long. Modern microarray technology enables us to efficiently perform many such tests in parallel. Microarrays are small glass microscope slides, sometimes also silicon chips or nylon membranes, onto which many thousands of nucleotide sequences are attached at fixed locations. Before we apply our sample to the microarray, it has to be tagged with a fluorescent marker. This provides for a visible feedback on the hybridization process so that we can easily determine to which oligonucleotides on the array our sample hybridizes. Figure 1.1 exemplifies the outcome of this procedure, where the different colors are due to the different markers used. Special scanner devices are used for computer aided evaluation of the strength of the fluorescent reaction at each spot on the array.

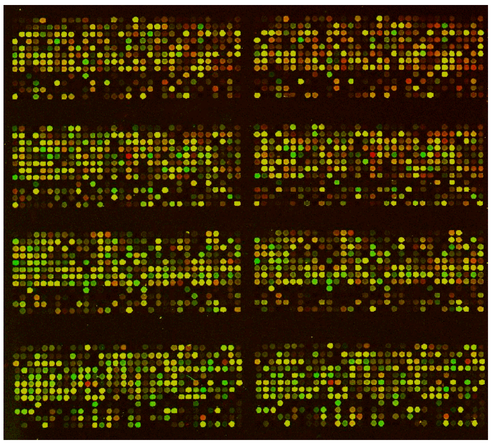
© 2004 Medical Research Council

Figure 1.1: A DNA microarray

From a more formal perspective the outcome of these tests provides some kind of binary signature, each binary digit of which tells us if a particular subsequence occurs as part of the sample sequence or not. The goal is to find a set of such subsequences, which we will call probes further on, so that the corresponding signature associated with each candidate sequence is unique over all sequences. The problem of finding a set of minimal cardinality that satisfies this condition has been introduced by Rash and Gusfield [32] and is referred to as the *string barcoding* (SB) problem. They provide a straight-forward integer linear programming formulation of the problem and propose a sophisticated method for generating a complete set of probes to choose from.

The same problem has been addressed by Borneman et al. [8]. They did, however, factor out the problem of probe generation and assume a given set of candidate probes as part of the input. Moreover, the algorithm they propose for solving the problem is a heuristic based on Lagrangian relaxation. It should be noted, that the problem has been considered well before theses publications, in one form or another under the names probe selection, oligonucleotide fingerprinting, and the like. However, the vast majority of these publications puts the main focus on the biological and application-side aspects of the problem. This thesis focuses on the underlying combinatorial optimization problem and algorithmic approaches to solve it.

SB can be seen as a special case of both the well-known *set covering* (SC) problem and the *minimum test collection* (MTC) problem, which have been shown to be NP-hard combinatorial optimization problems. The question whether SB itself is NP-hard or not was left open in [32]. The authors did only state that the problem was NP-hard if an additional maximum length constraint was imposed on the probes. However, in the meantime it has been proven to be NP-hard [6, 25].

Scope and outline. Chapter 2 introduces some basic theoretical concepts like linear programming and Lagrangian relaxation, which are important for the understanding of later parts. Chapter 3 defines the SB problem and its relatives formally and presents complexity and inapproximability results. Chapter 4 describes in detail how complete probe sets can be generated using suffix trees as has been proposed in [32]. Chapter 5 summarizes previous work related to SB and some state-of-the-art SC approaches, which may be applied to SB instances as well. We decided to adapt one of the most cited and best performing heuristics for the SC problem, hoping that it would be equally successful in solving SB instances. A comprehensive description of the heuristic, including the adaptations we applied to accommodate for the peculiarities of the SB problem, and details on the implementation are given in Chapter 6. Computational results on selected real-world benchmark instances can be found in Chapter 8. Chapter 9 summarizes the results of this work.

2 Preliminaries

The following sections introduce fundamental theoretical concepts, which are essential for the understanding of forthcoming parts. Section 2.1 defines linear programs and the notion of relaxation, Section 2.2 introduces the concept of Lagrangian relaxation, and Section 2.3 presents the subgradient optimization technique.

2.1 Linear Programming

This section defines linear programs and lists some of their properties. A comprehensive discourse on the topic can be found in [28]. Linear programs (LP) are optimization problems with a linear objective function that can be expressed as follows

$$\begin{aligned} \min_x \quad & c^T x && (2.1) \\ \text{s. t.} \quad & Ax \geq b && (2.2) \\ & x, b, c \in \mathbb{R}^n, A \in \mathbb{R}^{m \times n} \end{aligned}$$

Each linear inequality (2.2) defines a half-space of $\mathbb{R}^n$, the intersection of which forms a convex polytope. There are several efficient algorithms to solve problems of this kind to optimality. One of them is the simplex algorithm, which has been introduced by George Dantzig in 1947. The key idea of this method is to move along the polytope's facets from one vertex to the next. If we ensure that the objective function does not decrease when moving to an adjacent vertex, we will eventually reach the optimum. While very efficient in practice, the worst-case complexity is exponential. On the other hand, the more recent interior point methods like the predictor-corrector algorithm [27] have polynomial worst-case complexity and are also in practice competitive with the simplex method.

When dealing with combinatorial optimization problems like those presented in this thesis, we concern ourselves with discrete entities like the probes in an instance of the SB problem. Many of these problems can be formulated as LPs as well, but in this case some or all decision variables are are required to be integer. Accordingly, these problems are called mixed integer linear programs (MIP), or integer linear programs (ILP) if all variables are required to be integer. In contrast to LPs, finding the optimal solution of a MIP is in general a NP-hard problem and therefore no polynomial-time algorithm exists for this task.

2.2 Lagrangian Relaxation

Given some ILP

$$\begin{aligned} \min \quad & c^T x && (2.3) \\ \text{s. t.} \quad & Ax \geq b && (2.4) \\ & x \in \mathbb{Z}^n && (2.5) \end{aligned}$$

we may drop (relax) some of its constraints, thereby enlarging the feasible region of the problem. If we, for instance, relax the integrality constraints (2.5), we get what is called the LP relaxation (LPR) of an ILP. As every feasible solution to the ILP is also a feasible solution to its LPR, the optimal solution of the LPR is guaranteed to be better or exactly as good as the optimal solution of the ILP itself. Thus, the LPR if often used to derive a lower bound for a given ILP.

In practice, the LPR might nevertheless be impractical, either because its optimal solution is too resource consuming to compute, or because the computed solution provides an inferior lower bound to the underlying ILP. In these cases, there are other relaxations which can be used to compute a valid, and potentially better, lower bound, one of them being the Lagrangian relaxation (LGR). This method is based on splitting the constraints into two disjoint sets:

$$\begin{aligned} \min \quad & c^T x && (2.6) \\ \text{s. t.} \quad & Ax \geq b && (2.7) \\ & Cx \geq d && (2.8) \\ & x \in \mathbb{Z}^n && (2.9) \end{aligned}$$

Matrix A together with vector b represent the constraints we want to keep, matrix C in companion with vector d those that make the problem hard to solve and that we want to relax. However, when performing Lagrangian relaxation, we do not just drop the constraints but move them into the objective function. We do this by associating a Lagrangian multiplier $0 \leq u_i \in \mathbb{R}$ with each relaxed constraint and adding a penalty term to the objective function:

$$\begin{aligned} \min \quad & \text{LGR}(u) := c^T x + u^T(d - Cx) && (2.10) \\ \text{s. t.} \quad & Ax \geq b && (2.11) \\ & x \in \mathbb{Z}^n && (2.12) \end{aligned}$$

That the optimal solution to this ILP constitutes a valid lower bound for the originating ILP stems from the fact that, given a valid solution x of the ILP defined by (2.6) - (2.9) with objective value v_{ILP}, we can state two things:

- x is also a feasible solution of (2.10) - (2.12), because only a subset of constraints has to be satisfied
- Let v_{LGR} be the objective value of x with respect to (2.10). Since x satisfies constraint (2.8), $u^T(d - Cx) \leq 0$ holds for any $0 \leq u \in \mathbb{R}^m$, which in turn implies that $v_{\text{LGR}} \leq v_{\text{ILP}}$

Given a set of multipliers, the LGR should be easy to solve to be useful. However, now we need to find a set of multipliers u that yields the highest possible objective value.

This problem is called the Lagrangian dual and is formally defined as

$$\max_{0 \leq u \in \mathbb{R}} \min \mathrm{LGR}(u) \tag{2.13}$$

The optimal objective value v^*_{LGR} of the LG relaxation is guaranteed to be greater than or equal to the optimal objective value v^*_{LPR} of the LP relaxation. This can be proven as follows:

$$\begin{aligned}
v^*_{\mathrm{LGR}} &= \max_{u \in \mathbb{R}^m} \min_{x \in \mathbb{Z}^n} \{c^T x + u^T (d - Cx) : Ax \geq b\} && (2.14)\\
&\geq \max_{u \in \mathbb{R}^m_+} \min_{x \in \mathbb{R}^n} \{c^T x + u^T (d - Cx) : Ax \geq b\} && (2.15)\\
&= \max_{u \in \mathbb{R}^m_+} \max_{y \in \mathbb{R}^m_+} \{b^T y + u^T d : A^T y = c - u^T C\} && (2.16)\\
&= \max_{u,y \in \mathbb{R}^m_+} \{b^T y + d^T u : A^T y = c - C^T u\} && (2.17)\\
&= \min_{z \in \mathbb{R}^n} \{c^T z : Ax \geq b, Cx \geq d\} && (2.18)\\
&= v^*_{\mathrm{LPR}} && (2.19)
\end{aligned}$$

where $\mathbb{R}_+$ denotes the set $\{x \in \mathbb{R} : x \geq 0\}$. In (2.15) we relax the integrality constraints (2.12), which results in a solution with a possibly lower objective value. Equalities (2.16) and (2.18) are valid due to the the strong duality theorem, which is beyond the scope of this work – for an introduction to the topic see Nemhauser and Wolsey [28].

If the relaxation of the integrality constraints does not change the objective value of the solution, the Lagrangian relaxation is said to exhibit the *integrality property.* In this case, we can infer from (2.14) through (2.18) that we cannot derive a lower bound superior to that provided by the LP relaxation.

There are several techniques to solve the Lagrangian dual. One of the most prominent ones is called subgradient optimization and will be described in further detail in Section 2.3. A different and more recent approach is the volume algorithm introduced by Barahona and Anbil [2].

2.3 Subgradient Optimization

Subgradient optimization is an iterative procedure to find Lagrangian multipliers that yield a high lower bound. It is motivated by the fact that the Lagrangian dual is a piecewise linear, convex problem, from which we can infer that the optimum is located at a non-differentiable point. The technique was first introduced by Held and Karp [19, 20], who applied it to the traveling salesman problem.

In each iteration, we try to update the current set of multipliers u so as to reach a higher lower bound than before. To decide on the direction of our move, we use a subgradient. Given some function $f : \mathbb{R}^n \mapsto \mathbb{R}$, a vector $v \in \mathbb{R}^n$ is called a subgradient at point $x_0 \in \mathbb{R}^n$ if

$$f(x) - f(x_0) \geq v \cdot (x - x_0) \tag{2.20}$$

holds for any $x \in \mathbb{R}^n$. Note that a subgradient is defined at non-differentiable points, too.

Consider the generic LGR given in (2.10) through (2.12). Let u be a given multiplier vector and x^* the corresponding optimal solution. Then, we can define the following subgradient

$$s(u) := b - Ax^* \tag{2.21}$$

The basic procedure is depicted in Algorithm 1. The step size parameter λ controls how far we move along the subgradient and it will usually be adapted throughout the procedure to avoid moving past the optimum. Moreover, we have to decide on initial values of the Lagrangian multipliers, define the conditions that need to be satisfied to terminate the process, and provide a heuristic that produces feasible solutions to derive valid upper bounds.

Algorithm 1 Subgradient optimization

```
Initialize step size parameter $\lambda \in (0, 2]$
Initialize the set of multipliers $u$
Find some upper bound $U$
**repeat**
    $L \leftarrow \min \mathrm{LGR}(u)$
    Let $x^*$ be the corresponding solution
    $T \leftarrow \lambda(U - L)/\|s(u)\|^2$
    $u_i \leftarrow \max\{0, u_i + T \cdot s_i(u)\}$
**until** termination criterion met
```

3 Formal Problem Definition

This chapter defines SB and related optimization problems more formally. Section 3.1 provides an ILP formulation of SB and proposes possible extensions, which may arise in real-world applications. Sections 3.2 and 3.3 define the well-known set covering (SC) and minimum test collection (MTC) problems, and state how they relate to SB. Section 3.4 deals with the complexity aspects of SB.

3.1 String Barcoding

Given sequences $s_1, \dots, s_n$ over some finite alphabet Σ and a set of probes $p_1, \dots, p_m$, each probe being a subsequence of at least one sequence s_i. A probe p_k is said to *distinguish* sequences s_i and s_j if and only if p_k is a subsequence of s_i or s_j, but not both. Accordingly, we define

$$\delta_{i,j}(k) = \begin{cases} 1 & \text{if } p_k \text{ distinguishes } s_i \text{ and } s_j \\ 0 & \text{otherwise} \end{cases}$$

and denote the set of probes distinguishing sequences s_i and s_j by $\Delta_{i,j}$.

We aim at finding a minimum cardinality subset of probes such that every pair of sequences is distinguished by at least r probes in this subset. This can be formulated as the following ILP:

$$\begin{aligned}
\min \quad & \sum_{i=0}^{m} x_i && && (3.1)\\
\text{s. t.} \quad & \sum_{i=0}^{m} \delta_{i,j}(k) \cdot x_k \geq r && \forall (i,j) : 1 \leq i < j \leq n && (3.2)\\
& x_i \in \{0,1\} && && (3.3)
\end{aligned}$$

The value of decision variable x_i determines if the corresponding probe p_i is to be part of the solution ($x_i = 1$) or not ($x_i = 0$). The redundancy parameter r defines by how many probes each sequence pair has to be distinguished. In real-world applications a higher redundancy may be suitable, because we cannot expect the outcome of each single subsequence test to be 100% accurate. Setting $r > 1$ compensates, at least to some degree, for false negatives or positives, thereby enlarging our safety margin and making the resulting probe set more robust. One very fortunate effect of requiring a higher redundancy is that the ILP might become easier to solve. This is because, as r increases, the number of possibilities to satisfy constraints (3.2) usually decreases.

Another possibility to make the generated probe set more robust, particularly with regard to small mutations, is to require the minimum pairwise distance between probes in the solution to be above a particular threshold. One possible metric for measuring the distance between two strings or probes is the edit or Levenshtein distance. It defines the distance between two strings s and t as the minimum number of unit operations – character insertion, deletion, or substitution – required to transform s into t or vice versa. Note that it is a symmetric metric and that, depending on the particular application, the unit operations may be weighed differently. To enforce a minimum pairwise edit distance, we add one additional constraint to ILP (3.1) - (3.3) for each pair of candidate probes p_i and p_j whose edit distance from one another is below the required threshold:

$$p_i + p_j \leq 1 \qquad (3.4)$$

Thus we assure that at most one probe of each such pair will show up in the solution.

3.2 Set Cover

Set cover (SC) is one of Karp's 21 NP-complete problems [22]. The problem is defined by a set of items $S = \{s_1, \ldots, s_n\}$ and a collection $\mathcal{C} = \{C_1, \ldots, C_m : C_i \subseteq S\}$ of subsets of S with each subset C_i having associated cost $0 < c_i \in \mathbb{R}$. The goal is to determine a collection $\mathcal{C}^* \subseteq \mathcal{C}$ of minimum total cost that covers all items in S:

$$\begin{aligned} \min_{\mathcal{C}' \subseteq \mathcal{C}} \quad & \sum_{C_i \in \mathcal{C}'} c_i \\ \text{s. t.} \quad & \bigcup_{C \in \mathcal{C}'} C = S \end{aligned}$$

We restrict our discussion to the unicost case where $c_i = 1$ for all $1 \leq i \leq m$. It is easy to see that SB is a special case of SC and can therefore be stated in these terms. In this case, set S comprises all sequence pairs and collection $\mathcal{C}$ the sets of sequence pairs distinguished by the candidate probes. This relationship enables us to apply algorithms originally designed to solve SC instances to SB instances as well, which will be demonstrated in Chapter 6.

3.3 Minimum Test Collection

The minimum test collection (MTC) problem is very similar to the SB problem and can even be regarded as a higher-level formulation of the latter. It is defined by a set of items D and a collection $\mathcal{T} = \{T_1, \ldots, T_n : T_i \subseteq D\}$ of subsets of D, called tests. We aim at finding a collection $\mathcal{T}^*$ of minimum size such that for each pair of elements $(d_i, d_j) \in D \times D$ there is at least one test $T_i \in \mathcal{T}^*$ with $|\{d_i, d_j\} \cap T_i| = 1$. Garey and Johnson [17] have shown that the problem is NP-complete by reduction from 3-dimensional matching. As is the case with SC, any SB instance can easily be transformed into an MTC instance in a straight-forward manner.

3.4 Complexity and Inapproximability

Rash and Gusfield [32] state that the max-length string barcoding problem, which adds the additional constraint that no candidate probe exceeds a given maximum length, is NP-complete. However, they did not answer the question for the unconstrained case. In the meantime, Lancia and Rizzi [25] and Berman et al. [6] have proven this. Their findings rely on previous work that has shown that SC cannot be approximated in polynomial time within a factor of $(1 - \varepsilon) \log n$, with $\varepsilon > 0$ and n being the number of items to be covered. More precisely, this means that no polynomial approximation algorithm exists that produces a solution of size $z \leq z^* \cdot (1 - \varepsilon) \log n$, where z^* is the size of the optimal solution. Lancia and Rizzi [25] provide polynomial time transformations, which preserve this property, from SB to a special subclass of MTC and from there to SC.

Borneman et al. [8] claim, though without proof, that NP-completeness can be shown by reduction from vertex cover.

4 Probe Generation

As shown in Section 3.4, SB is a NP-hard optimization problem for which no polynomial time algorithm is likely to exist (unless P=NP). For this reason, it is strongly advisable to keep the number of candidate probes as low as possible. On the other hand, it is important not to leave out any essential probe as this would change the solution space and we may end up with a suboptimal solution. A rather naive approach would be to enumerate all possible substrings of all given input sequences. This would, however, result in a large number of redundant probes that distinguish the same set of sequence pairs. Fortunately, there is a much more sophisticated way to generate a complete set of probes in a very efficient manner by means of a suffix tree. Section 4.1 will outline this method and Section 4.2 presents a clarifying example.

4.1 Suffix Trees

A suffix tree is a data structure which encodes all suffixes of a number of strings; Gusfield [18] provides an in-depth coverage of the topic. The suffix tree for a number of strings $S = \{s_1, \dots, s_n\}$ has the following properties:

- It is a directed tree with a designated root node
- Each edge is labeled with a non-empty string that must be a substring of at least one s_i
- Each leaf node is associated with one input string s_i. The concatenation of the edge labels on the path from the root node to this node is a suffix of s_i
- Each internal node has at least 2 children

- The labels of all outgoing edges of a node start with different characters

As any possible substring is the prefix of some suffix, it can be found on a unique path, starting at the root node and ending either at some node or inside an edge (label). According to the properties of a suffix tree, the substring occurs in every string that is associated with one of the leaf nodes that are located below this point in the tree.

The defining property of a probe p_k is the set of strings $S_k \subseteq S$ in which it occurs as substring and, derived from that, the set of sequence pairs it distinguishes. Thus, two probes p_i and p_j can be considered equivalent if either $S_i = S_j$ or $S_i = S \setminus S_j$. As a consequence, each suffix tree node v represents a whole set of substrings that are equivalent in this regard. This set comprises all substrings that we get by concatenating the labels of the edges on the path from the root node to the parent node of v plus a non-empty suffix of the label of the edge from the parent node to v.

To exploit this fact, we assign a unique binary signature $b_0 b_1 \ldots b_{n-1}$, $b_i \in \{0, 1\}$, to each node, with $b_i = 1$ if the corresponding set of strings occurs in input string s_i, and $b_i = 0$ otherwise. Note that two nodes with different signatures might still be equivalent to each other, namely if one signature is the binary complement of the other. To get rid of this special case, we require the first binary digit of every signature to be 1 and flip signatures starting with 0. Apart from this, we can ignore nodes with a signature $11 \ldots 1$ because probes that occur in every sequence do not distinguish any sequence pairs and are of no use.

The probe generation procedure can be summarized as follows: we build the suffix tree for the given set of sequences; this can be done in linear time. Then, we traverse the tree bottom-up and determine the signature of each node as defined above. For each distinct signature we choose as many probes as necessary. Compared to more naive approaches like the complete enumeration of all distinct substrings this method yields 75% less probes on average, in some cases the reduction exceeds 90%.

4.2 Example

To make things clearer, we will present a simple example. Consider the strings $s_0 =$ `CAGTGC`, $s_1 =$ `CAGTTC`, and $s_2 =$ `CATGGA`. The corresponding suffix tree is depicted in

Fig. 4.1 with the black node being the root. Equivalent nodes, which distinguish the same pairs of strings, are labeled with identical Greek letters. Our example gives rise to three different classes of nodes with the following signatures

Class	Signature
α	1 0 1
β	1 0 0
γ	1 1 0

If we choose one substring per node class, we get three probes. On the other hand there are 41 distinct substrings of s_0, s_2, and s_2, so we save about 93%.

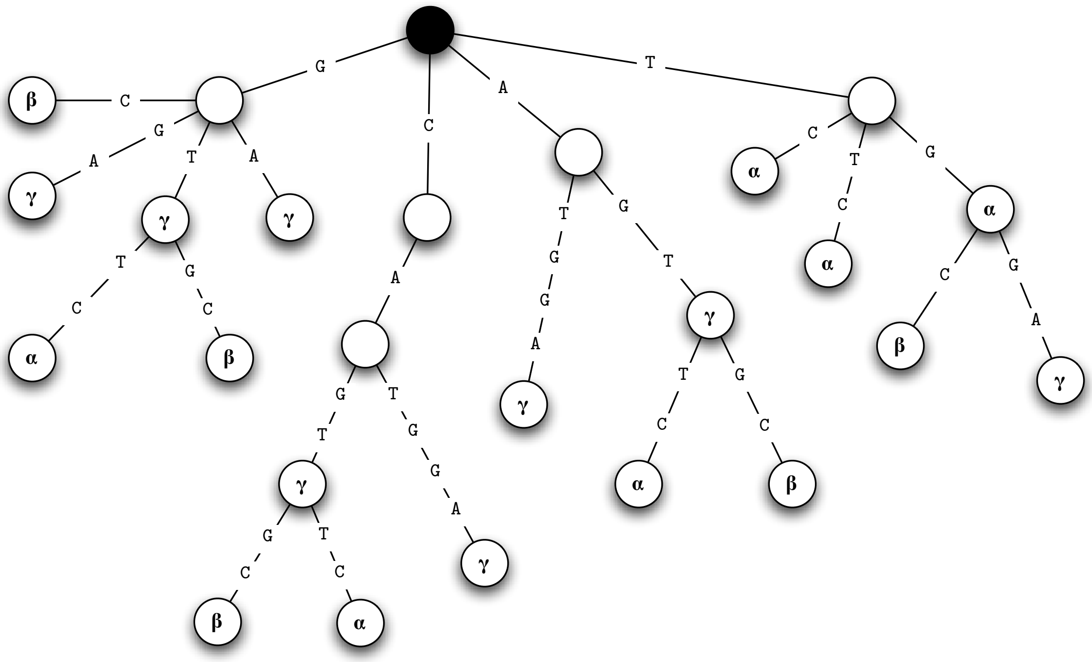

Figure 4.1: Suffix tree for the strings `CAGTGC`, `CAGTTC`, and `CATGGA`

4.3 Pitfalls

There are some issues as to what subsequences to associate with a particular signature. Especially if a redundancy $r \geq 1$ or a minimum edit distance $d \geq 1$ is required, we have to take special care to not artificially restrict the solution space. Otherwise we risk to cut off the optimal solution. If we aim at a solution with a higher redundancy $r \geq 1$, it is not anymore sufficient to choose one probe for each distinct signature as described above. In this case we need to select r or, in case the total number of possible probes for this signature is smaller than r, as many probes as possible. If we do not do this, the problem may still be feasible but incomplete in the sense that the optimal solution may not be reachable anymore. For the sake of completeness, Figure 4.2 displays a counter example comprising six input sequences and the ILP it translates to when we require a redundancy of $r = 2$.

The x_i are binary variables because we selected one only probe per signature. One optimal solution to this ILP with objective value 6 is given by $x_0 = x_1 = x_2 = x_3 = x_6 = x_8 = 1$ and $x_i = 0$ for all other variables. However, if we do not restrict ourselves to one probe per signature but generate $r = 2$ probes per signature instead, the binary constraint changes to $x_i \in \{0, 1, 2\}$ and we are able to achieve a better solution with objective value 5, given by $x_1 = 2, x_2 = x_3 = x_7 = 1$ and $x_i = 0$ otherwise.

$s_1 =$ `GGTAATAATACTACTAGTAGTACTCATCATAGTCCTCGTGATGCTGGTACTAAATAACTACC`
$s_2 =$ `GGTAATAATACTACTAGTAGTACTCATCATAGTCCTCGTGATGCTGGTACTAAATAAGTACG`
$s_3 =$ `GGTAATAATACTACTAGTAGTACTCATCATAGTCCTCGTGATGCTGGTACTAAGTAGA`
$s_4 =$ `GGTAATAATACTACTAGTAGTACTCATCATAGTCCTCGTGATGCTGGTACTAAATAATACATAGC`
$s_5 =$ `GGTAATAATACTACTAGTAGTACTCATCATAGTCCTCGTGATGCTGGTACTAACTAGG`
$s_6 =$ `GGTAATAATACTACTAGTAGTACTCATCATAGTCCTCGTGATGCTGGTACTACATCAA`

$$\Downarrow$$

$$\begin{array}{ll}
\text{minimize} & x_0 + x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8 \\
\text{subject to} & x_0 + x_2 + x_7 \geq 2 \\
& x_0 + x_1 + x_2 + x_7 + x_8 \geq 2 \\
& x_0 + x_2 + x_3 + x_6 \geq 2 \\
& x_0 + x_1 + x_5 \geq 2 \\
& x_0 + x_1 + x_2 + x_3 + x_4 \geq 2 \\
& x_1 + x_8 \geq 2 \\
& x_3 + x_6 + x_7 \geq 2 \\
& x_1 + x_2 + x_5 + x_7 \geq 2 \\
& x_1 + x_3 + x_4 + x_7 \geq 2 \\
& x_1 + x_3 + x_6 + x_7 + x_8 \geq 2 \\
& x_2 + x_5 + x_7 + x_8 \geq 2 \\
& x_3 + x_4 + x_7 + x_8 \geq 2 \\
& x_1 + x_2 + x_3 + x_5 + x_6 \geq 2 \\
& x_1 + x_4 + x_6 \geq 2 \\
& x_2 + x_3 + x_4 + x_5 \geq 2 \\
& x_i \in \{0, 1\}
\end{array}$$

Figure 4.2: SB instance and corresponding ILP, showing effect of redundancy on probe generation

5 Previous Work

This chapter presents a selection of approaches to solve the SB and SC problems, which have been defined in Sections 3.1 and 3.2. For each algorithm, we will give a short summary and try to convey the underlying key idea. Being one of the classic NP-hard combinatorial optimization problems, the extent of scientific coverage of the SC problem is enormous. For this reason, we picked out those approaches that seemed most promising, with respect to both efficiency and solution quality, and suitable for our purpose. Section 5.1 addresses approaches specific to the SB problem, whereas Section 5.2 summarizes several algorithms targeted at the more general SC problem. With the exception of the ILP approach of Rash and Gusfield [32], all algorithms are of heuristic nature because none of the exact approaches we came across appeared to be powerful enough to solve large-scale instances as those presented in Chapter 8.

5.1 String Barcoding

Rash and Gusfield [32] introduced the problem as described in Section 3.1. They describe how a complete set of candidate probes can be generated using suffix trees as a means to ensure that no two probes distinguish the same set of sequence pairs; Chapter 4 explains this procedure in full detail. Applied to real-world instances, the reduction in the number of generated probes, compared to the number of all distinct subsequences, exceeds 75% on average and reaches over 90% in many cases.

Given a set of sequences $s_1, \ldots, s_n$ and a valid solution, comprising probes $p_{i_1}, \ldots, p_{i_k}$, we can assign a unique binary signature $\sigma_i = b^i_1 b^i_2 \ldots b^i_k$ to each sequence s_i with $b^i_j = 1$ if probe p_{i_j} is a subsequence of s_i, and $b^i_j = 0$ otherwise. Obviously, the number of possible signatures of length l is 2^l, so the number of digits, which in turn corresponds to the number of probes, must be greater than or equal to $\log_2(n)$. This can be used to make the ILP formulation slightly stronger by adding the constraint

$$\sum_{i=1}^{m} x_i \geq \lceil \log_2 n \rceil \tag{5.1}$$

Doing so does not affect the problem structure but may help to accelerate the branch-and-bound procedure employed by ILP solvers to find the optimal solution. During our computational experiments (Chapter 8), however, the observed effect on the running time was negligible.

The computational results presented by Rash and Gusfield were generated from real-world instances by solving the ILP given in (3.1) through (3.3) and (5.1). All benchmark instances were derived from biological data available through the GenBank [31] database. They considered only probes between 15 and 40 nucleotides long and required the minimum edit distance and the redundancy r to be greater than 1 for most benchmark instances.

As described in Chapter 4, it is sufficient to take r probes per node class in the suffix tree. Unfortunately, this is no longer true if we want to produce a probe set with a minimum pairwise edit distance greater than 1. In this case there is no way but to include all possible probes at each suffix tree node, which in most cases makes the resulting ILP so large that it is not solvable anymore – at least not with the computational resources available to us. Most probably, Rash and Gusfield made some kind of preselection instead of taking all probes per node. If this is true, they did not state on which criteria this selection was based.

Another issue is the handling of *degenerate bases*. Since sequence analysis techniques are not perfectly accurate, the resulting sequence data usually contains some positions for which we cannot state the nucleotide with certainty. For this reason, real-word data often contains degenerate bases, which represent the non-empty subsets of nucleotides that are possible for a particular sequence position (see Table 5.1 for a complete list). They are, however, not treated differently from simple nucleotides in [32].

Borneman et al. [8] present a heuristic approach based on Lagrangian relaxation. They factor out the problem of probe generation and assume a given set of predeter-

Degenerate base	Set of possible nucleotides
R	$\{A, G\}$
Y	$\{C, T\}$
M	$\{A, C\}$
K	$\{G, T\}$
S	$\{C, G\}$
W	$\{A, T\}$
B	$\{C, G, T\}$
D	$\{A, G, T\}$
H	$\{A, C, T\}$
V	$\{A, C, G\}$
N	$\{A, C, G, T\}$

Table 5.1: List of degenerate bases

mined probes like all distinct subsequences of a particular length. Furthermore, the computational results cannot be compared to those published in [32] because different benchmark instances have been used. Two of the four data sets used are considerably larger with respect to the number of contained sequences than any benchmark instance used in [32].

The heuristic uses Lagrangian relaxation of all constraints (3.2) and subgradient optimization to find Lagrangian multipliers that yield a high lower bound. Due to the integrality property, the solution of the relaxation is integral, albeit not necessarily feasible, so at each step of the subgradient optimization procedure a new partial solution is generated. It is greedily extended by iteratively adding the probe with maximum score to the solution until all sequence pairs are covered. The score of a probe is derived from its Lagrangian cost and the number of undistinguished sequence pairs it distinguishes – the same measure has been used by Caprara et al. [10].

To speed up the procedure, a clustering technique is used. Similar sequences build a cluster, out of which we select one representative sequence. The set of representatives defines a subproblem that can be solved faster and whose solution can be used as an initial solution for the original instance.

They additionally propose a simulated annealing approach for a problem variant where the size of the probe set is fixed and the number of distinguished sequence pairs is to be maximized.

Berman et al. [6] have shown that no polynomial time approximation algorithm for SB can guarantee an approximation factor of $(1-\varepsilon)\log n$ for any $\varepsilon > 0$ unless NP $=$ DTIME$(n^{\log\log n})$ and present an information content heuristic with an approximation factor of $1+\log n$.

Each partial solution defines a partition of the set of sequences into disjoint classes. Two sequences belong to the same class if and only if they cannot be distinguished by any probe in the solution. Let n be the number of sequences and $C_1,\ldots,C_k$ the classes defined by the current solution. The entropy of the solution is defined as

$$H := \frac{1}{n}\log_2\left(\prod_{i=1}^{k}|C_i|!\right)$$

Starting with an empty solution, the algorithm iteratively adds the probe that maximizes the decrease in entropy, until the solution is feasible.

The problem of probe generation is not addressed.

DasGupta et al. [13] describe another greedy algorithm with an approximation factor of $2\log n$ which, compared to [6], allegedly produces results of virtually identical quality. The algorithm has two phases: a probe generation phase and a probe selection phase. To keep the number of generated probes low, those that appear in all sequences are discarded. On the other hand, for each sequence only one probe is generated that appears exclusively in that sequence. While being faster than the suffix tree approach, this procedure may produce redundant candidate probes, which appear in the same set of sequences.

The selection phase is a very simple iterative procedure, which is executed until all sequence pairs are distinguished. At each iteration we select the probe that distinguishes the highest number of yet undistinguished sequences. The source code as well as an online server is publicly available at `http://dna.engr.uconn.edu/~software/DNA-BAR`.

5.2 Set Cover Approaches

As has been noted in Chapter 3, SB is a special case of SC. This gives us the opportunity to use any of the numerous SC approaches to solve SB instances as well. Note, however, that the structure of SB instances, at least of those used by Rash and Gusfield, differs from those commonly used for benchmarking SC algorithms. The instances in Beasley's OR-Library [5], which are often used for benchmarking purposes, range in size from 500 to 5,000 items and from 10,000 to 1,000,000 sets per instance, the density of the ILP matrices ranges from below 1% to up to 20%.

On the other hand, the number of columns of the ILP representation of an SB instance grows quadratically with the number of input sequences. The benchmark instances used by Rash and Gusfield [32] give rise to constraint matrices with 5,000 - 50,000 rows, 500 - 2,000 columns, and a density of about 20%. Note too, that the number of columns depends very much on the length constraints, which may be imposed on the probes. The numbers given result from an allowed probe length of 15 to 40 nucleotides.

Many of the approaches outlined below have been designed to handle weighted SC instances, where each set incurs some cost and the goal is to minimize the total cost instead of the number of sets contained in the solution. The following summary, however, considers the unicost case only, because the SB problem as discussed in this thesis is unweighted, too.

When we refer to an SC instance further on in this chapter, we will denote the set of items to be covered by $S := \{s_1, \ldots, s_n\}$ and the collection of subsets of S as $\mathcal{C} := \{C_1, \ldots, C_m\}$

Caprara et al. [10] propose one of the most cited and seemingly most successful heuristic approaches to date to tackle large-scale SC instances. Like many others, it is based on Lagrangian relaxation and uses several well-known techniques to improve its performance.

The algorithm has three parts, all of which rely on information derived from Lagrangian relaxation. The first part is the most important and produces actual heuristic solutions. The second part examines the best solution produced by the first part and tries to determine sets in the solution that are highly likely to be part of an optimal solution. These will be fixed, which means we will make them a mandatory part of the solution and discard all items that are covered by them. As a result, we get a new subinstance of smaller size to which we reapply the whole procedure. The third part is responsible for keeping the size of an instance manageable. This is achieved by making only a subset of sets visible to the algorithm. This subset is updated regularly.

The complete procedure is used repeatedly until we either exceed a predefined time limit or produce a solution of sufficient quality.

The algorithm's main part uses subgradient optimization to find a near-optimal set of Lagrangian multipliers. Then, a set of multiplier vectors of comparable quality is generated, and for each of them we build a solution by means of a simple greedy heuristic, which is guided by the Lagrangian costs of the sets. Finally, we select some promising sets, fix them as part of the solution, and remove from the instance all items covered by these sets. We repeat this procedure until either all items are covered by fixed sets, or until we can conclude that no further improvement is possible.

This thesis focuses on the adaptation of this algorithm for the SB problem, so a more detailed description will be given in Chapter 6.

Beasley and Chu [3] propose a steady-state genetic algorithm (GA). Inspired by the natural evolutionary process, a genetic algorithm operates on a pool of solutions (population) whose average quality improves over time. In each iteration, solutions are selected based on their quality (fitness) and pairwise recombined into new solutions (parents and children), to which, with a certain probability, slight random modifications are applied (mutation). There are many variants of genetic algorithms which differ from each other in the number of solutions selected, the criteria on which the selection is based, and which solutions go into the population of the next iteration. In case of the steady-state GA at hand, we select only two solutions per iteration, recombine them, apply the mutation operator, and substitute the result for the worst solution in the current pool.

Beasley and Chu represent solutions as binary strings $x_1x_2\ldots x_m$, with $x_i = 1$ if and only if set C_i is part of the solution. They use binary tournament selection, which means that they randomly choose two solutions from the population and select the one with better fitness. The recombination operator randomly takes the value for each x_i from either parent, with a selection probability proportional to the fitness of the respective parent solution. The mutation probability is variable and increases as the algorithm converges. Mutation as well as recombination operators may yield infeasible solutions, so a repair operation is applied if necessary. This is done by greedily extending the partial solution until all items are covered and heuristically removing redundant sets afterward.

Eremeev [14] proposes another steady-state GA, which uses a non-binary representation. A valid solution $\mathcal{C}^* = \{C_{i_1}, \dots, C_{i_k}\}$ is represented as a vector $\langle g_1, g_2, \dots, g_n \rangle$, where $g_i = j$, $j \in \{i_1, \dots, i_k\}$, means that item s_i is covered by set C_j, and maybe some others too. This mode of representation is not unique in that a particular solution may have more than one representation. On the other hand, it has the advantage that both the recombination and mutation operators always yield feasible solutions, so a repair operator is not necessary.

The selection operator picks solutions randomly from the population with each solution having a probability to be chosen proportional to its fitness. Accordingly, this is often called fitness-proportional selection.

The recombination operator constructs a subproblem, which depends on the two solutions on which it is applied, and tries to solve the corresponding LP relaxation. If the number of constraints is above a certain threshold, solving the LP relaxation takes too long, or if the solution is not integral, the first parent solution will be taken without further modification.

Given two solutions $\mathcal{C}_1 \subseteq \mathcal{C}$ and $\mathcal{C}_2 \subseteq \mathcal{C}$, the recombination operator performs the following steps to construct the subproblem:

1. Let $\mathcal{C}' := \mathcal{C}_1 \cup \mathcal{C}_2$ and $S' \subseteq S$ be the set of items that are covered by exactly one set from $\mathcal{C}'$
2. Determine the set $\mathcal{C}'' := \{C \in \mathcal{C}' : C \cap S' \neq \emptyset\}$
3. Remove from the instance all sets in $\mathcal{C}''$ as well as all items covered by $\mathcal{C}''$

The mutation operator takes as input a single element g_i of a solution's representation vector and merely assigns a new value to that element. From all sets covering the corresponding item s_i one is chosen randomly, with more expensive sets having a lower probability to be picked. The mutation operator is applied to each element in turn, although only with a fixed probability. Note, however, that the solution does not necessarily change when its representation does.

The solution produced by the successive application of the recombination and mutation operators is always feasible but may contain redundant sets. Eremeev proposes two different greedy heuristics to eliminate redundant elements from a solution. Both of them are applied independently, and the better result goes into the next population.

Marchiori and Steenbeek [26] introduce an iterative heuristic. The heuristic operates on a dynamic subproblem only, which is recomputed every 100 iterations. Each iteration starts with a partial solution, which is empty in the beginning and is derived non-deterministically from the best solution found so far after each iteration. At each iteration, we alternately add and remove some elements to or from the solution until it is feasible and does not contain any redundant sets.

When adding elements, we do so in a greedy manner by repeatedly adding a set that covers a maximum number of yet uncovered items. The removal of elements is done unconditionally if the solution contains redundant sets, by removing one of them arbitrarily, or otherwise, only with a particular probability, by removing a random set.

Finally, a local optimization procedure identifies and adds sets that make two or more sets of the current solution redundant, which are removed in consequence.

Yagiura et al. [37] describe a local search approach combined with a variable fixing scheme, which is particularly useful when solving large scale instances. Both local search and variable fixing are guided by information derived from Lagrangian relaxation.

The neighborhood of a solution S is defined as the set of solutions that differ from S in at most three sets. The neighborhood has size $O(n^3)$, so its complete exploration is very expensive. Accordingly, the proposed implementation searches only part of the neighborhood, yet without sacrificing solution quality.

As another key point, they allow infeasible solutions by use of a penalized objective function. A penalty weight is assigned to each item to be covered, which determines the extent by which the objective value of a solution increases if the respective item is uncovered.

The procedure repeatedly performs local search, until a local optimum is reached, followed by an update to the penalty weights. This update allows us to control the degree of infeasibility allowed and is used to intensify the search around the border of the feasible region. The rationale behind this is that any optimal solution S^* must be located in this region as removing a single set from S^* would render it infeasible.

Brusco et al. [9] propose a simulated annealing algorithm, supplemented with a *morphing procedure*. They target SC instances that exhibit a high degree of coverage correlation, which measures the extent to which the sets of an instance cover the same items.

Simulated annealing [23, 35] is inspired by a conceptually similar technique used in materials science, which involves heating followed by controlled cooling of a material like metal or glass to improve its quality. It is an iterative procedure which repeatedly determines the best neighbor of the incumbent solution, substituting it for the current solution if it is either superior or otherwise with a certain probability. This probability depends on the current temperature which decreases over time. The probabilistic acceptance of inferior solutions provides a means of escaping local optima.

Brusco et al. define the neighborhood of a solution by means of a non-deterministic algorithm, which first removes a set from the current solution, then adds sets to make the solution feasible again, and finally removes redundant sets. The set to be removed in the first step is either a random one or the set that would leave the least number of items uncovered if removed. During the repair step we greedily add sets that have a low cost and cover many items that are not covered yet.

The proposed morphing procedure involves maintaining for each set in the solution a list of sets that cover similar items. In each iteration we replace a set of the solution with one from the respective morphing list if this leads to an improvement.

Caprara et al. [11] give an extensive survey of further SC approaches, which covers several exact algorithms, too.

6 A Lagrangian Heuristic for the String Barcoding Problem

6.1 Outline

In essence, the heuristic presented in this chapter conforms to the CFT algorithm published by Caprara et al. [10]. Apart from some minor extensions to improve overall performance, we merely made changes that enabled the algorithm to deal with instances of the SB problem as defined in Section 3.1. Note that the original algorithm was designed to handle weighted SC instances, where each subset incurs some fixed cost and the goal is to find a cover of minimum total cost. Although the same concept could be applied to the SB problem as well, we will restrict the discussion to the unicost variant as presented in [32]. At some sites the original algorithm uses set costs as a sorting or selection criterion. In these cases, we use random perturbation or selection instead.

The algorithm, which we will refer to as the CFTSB heuristic, relies heavily on the concept of Lagrangian relaxation, which has been introduced in Section 2.2. Caprara et al. propose the use of subgradient optimization (Section 2.3) to compute good Lagrangian multipliers. However, when starting from scratch, finding satisfactory multipliers can be a very time-consuming process. We have found that an initial set of optimal multipliers can be computed in reasonable time by solving the LP relaxation instead. In general, this approach may not scale very well because the required effort to solve a particular LP is hard to estimate. With regard to our benchmark instances, however, this modification of the original algorithm leads to considerably lower run times without sacrificing solution quality.

The heuristic pursues a two-level, iterated approach. At the top level, we repeatedly generate feasible solutions for the given instance, each time starting from a different partial solution. This partial solution is empty at the beginning of the first iteration and is subsequently derived from the solution generated in the previous iteration. We repeat this procedure until we either reach a predefined time limit or find a solution that is good enough for our purposes.

The actual solutions are generated at the lower level, which is conceptually similar to the top level procedure. We perform a series of iterations, each one starting from a partial solution, which is empty at the very beginning. As the first step of each iteration, we try to find good Lagrangian multipliers for the subproblem defined by the given partial solution. These are used to produce a set of heuristic solutions, each one extending the current partial solution. At the end of each iteration, some further probes are chosen to extend the partial solution, which serves as a starting point for the next iteration. We repeat this procedure until either the partial solution itself has become feasible, or until it is evident that we cannot find a solution better than the best one found so far.

To improve overall performance, especially with regard to large-scale instances, a pricing scheme is applied to the whole procedure. This means that, at any time, the algorithm takes into account only a subset of all available candidate probes; this subset will be dynamically updated at regular intervals. The use of this technique may reduce the run time considerably, even for medium-sized instances. We will discuss this topic in more detail in Section 6.2.4.

Before we delve further into the details, we introduce some notational conventions, which will be used throughout the chapter, and present specifics on the application of Lagrangian relaxation and subgradient optimization to the SB problem.

Each SB instance $\mathcal{I} = (S, P, r, F)$ is defined by a set S of sequences, for which we seek a probe set of minimum size, a set P of possible probes to choose from, the number r of probes by which each sequence pair must be distinguished, and an optional partial solution $F \subseteq P$. Obviously, the set F affects the number probes by which a particular sequence pair must be distinguished; for any sequence pair $(s_i, s_j) \in S \times S$, $i \neq j$, this number is given by:

$$r_{i,j}(F) = \max\left\{ r - \sum_{p_k \in F} \delta_{i,j}(k), 0 \right\} \tag{6.1}$$

Any sequence pair (s_i, s_j) with $r_{i,j}(F) = 0$ is distinguished by the required number of probes and therefore needs not to be taken into consideration anymore. Thus, any partial solution F defines a subinstance $\mathcal{I}_F$ of the original instance. For convenience, we will denote the set of sequence pairs that need further probes to distinguish them by

$$S^2(F) := \{(s_i, s_j) \in S \times S : i \neq j,\ r_{i,j}(F) > 0\} \tag{6.2}$$

We apply Lagrangian relaxation to the ILP formulation of the SB problem given in (3.1) through (3.3) and additionally relax the integrality constraints. This leads to the following minimization problem:

$$\begin{aligned}
\min\ L(u) :=& \sum_{k=1}^{m} x_k + \sum_{1\le i<j\le n} u_{i,j} \left(\sum_{k=1}^{m} \delta_{i,j}(k) \cdot x_k \right) && (6.3)\\
=& \sum_{k=1}^{m} x_k \underbrace{\left(1 - \sum_{1\le i<j\le n} \delta_{i,j}(k) \cdot u_{i,j} \right)}_{\text{Lagrangian cost } c_k(u)} + r \sum_{1\le i<j\le n} u_{i,j} \\
=& \sum_{k=1}^{m} c_k(u) \cdot x_k + r \sum_{1\le i<j\le n} u_{i,j} \\
\text{subject to}\quad & 0 \le x_k \le 1,\ k = 1, \ldots, m && (6.4)
\end{aligned}$$

Each variable x_i corresponds to probe p_i. Recall that the term c_i denotes the Lagrangian cost of variable x_i or probe p_i, respectively.

Given some Lagrangian multiplier vector u, an optimal solution x^* to (6.3) - (6.4) can be derived easily as follows:

$$x_i^*(u) = \begin{cases} 1 & \text{if } c_i(u) < 0 \\ 0 & \text{if } c_i(u) \ge 0 \end{cases}$$

Note that the Lagrangian relaxation defined by (6.3) through (6.4) has the integrality property (see Section 2.2). As a consequence, any optimal solution is integral, which is the reason why we relax the integrality constraints too. Unfortunately, this does also imply that we will not be able to find a lower bound higher than that provided by the optimal solution of the LP relaxation.

When using subgradient optimization as a means to find good Lagrangian multipliers, we define the subgradient s as

$$s_{i,j}(u) := r - \sum_{k=1}^{m} \delta_{i,j}(k) \cdot x_k^*(u) \tag{6.5}$$

Accordingly, the update formula for the Lagrangian multipliers is given by

$$u_{i,j}^{k+1} := \max\left\{ u_{i,j}^{k} + \lambda \frac{U - L(u^k)}{\|s(u^k)\|^2} s_{i,j}(u^k), 0 \right\} \tag{6.6}$$

where U is a valid upper bound, derived from any feasible solution, and λ denotes the step size parameter, which will be updated periodically.

Algorithm 2 depicts the top level procedure in pseudocode. In line 7, we produce a valid solution for subinstance $\mathcal{I}_F$ defined by partial solution F. This is the most important part of the complete heuristic and will be addressed in Section 6.2. In addition to a feasible probe set, function CoreOpt returns the best set of Lagrangian multipliers it has found. Observe that the multipliers returned are associated with subinstance $\mathcal{I}_F$ and do not necessarily yield a good lower bound for the original problem. Therefore, we use the returned multiplier vector only if F is empty, in which case $\mathcal{I} = \mathcal{I}_F$ holds, and if it results in a better lower bound for $\mathcal{I}$ than the best multiplier vector found so far (lines 12 - 14). Parameter π controls the size of subinstance $\mathcal{I}_F$ derived from F and is updated in lines 15 - 19. Section 6.2.5 is dedicated to the inner workings of function Refine that computes partial solution F at the end of each iteration.

Algorithm 2 CFTSB heuristic – top level perspective

```
1:  function CFTSB($S, P, r$)
2:      $Q \leftarrow P$                                    ☞ *Best known solution*
3:      $F \leftarrow \emptyset$                                    ☞ *Fixed probes*
4:      $u \leftarrow 0$                                    ☞ *Lagrangian multiplier vector*
5:      $\pi \leftarrow \pi_{\min}$
6:      repeat
7:          $(Q', u') \leftarrow$ CoreOpt($S, P, r, F, u$)
8:          $Q' \leftarrow Q' \cup F$
9:          if $|Q'| < |Q|$ then
10:             $Q \leftarrow Q'$
11:         end if
12:         if $F = \emptyset$ then
13:             $u \leftarrow u'$
14:         end if
15:         if $|Q'| = |Q|$ or $F = \emptyset$ then
16:             $\pi \leftarrow \pi_{\min}$
17:         else
18:             $\pi \leftarrow 1.1 \cdot \pi$
19:         end if
20:         $F \leftarrow$ Refine($Q', \pi$)
21:     until $L(u) = |Q|$ or time limit exceeded
22:     return $Q$
23: end function
```

6.2 Core Algorithm

Actual solutions for a given instance, or a subinstance thereof, are generated at the lower level of the CFTSB heuristic. The complete process is illustrated in Algorithm 3 and can be broken down into three steps:

Step 1 In lines 4 - 7, we attempt to find a near-optimal set of Lagrangian multipliers. In the first iteration, we achieve this by solving the LP relaxation to optimality. Subsequently, a slightly modified subgradient optimization procedure is employed for this purpose. Details on this step will be presented in Section 6.2.1.

Step 2 In the next step (line 8) we attempt to generate feasible solutions for the given problem instance using a simple greedy heuristic that iteratively adds probes to an initially empty set of probes until it constitutes a feasible solution. The decision which probe to add next is guided by the number of still insufficiently distinguished sequence pairs a probe covers in relation to its Lagrangian cost. A detailed rundown will be given in Section 6.2.2.

Step 3 During the third step (line 16), we select probes which have a presumably high likelihood of appearing in an optimal solution and add them to partial solution F. Details on how the selection is performed can be found in Section 6.2.3. Before we loop back to step one, we perturb the Lagrangian multipliers randomly so as to facilitate the generation of diverse solutions (line 17).

Function CoreOpt terminates when either F itself constitutes a feasible solution – observe that we never remove a probe from it – or when we can infer that we will not be able to construct a solution that is better than the best one found so far; this is the case if the number of probes in F plus the lower bound for the remaining subproblem is equal to or greater than the current best solution.

6.2.1 Lagrangian Multiplier Optimization

In this section, we describe two possible methods that can be used to find good or even optimal Lagrangian multipliers for a given problem instance $\mathcal{I} = (S, P, r, F)$. An initial set of optimal multipliers is computed by solving the LP corresponding to instance $\mathcal{I}$.

Algorithm 3 Core component of CFTSB heuristic

```
function CoreOpt($S, P, r, F, u$)
    $Q \leftarrow P$                                                ☞ Best known solution
    repeat
        $u' \leftarrow$ MultOpt($u$)                           ☞ Find (near-)optimal multipliers
        if $L(u') < L(u)$ then
            $u \leftarrow u'$
        end if
        $(Q', u') \leftarrow$ HeurOpt($S, P, r, F, u$)         ☞ Generate heuristic solution
        $Q' \leftarrow Q' \cup F$
        if $|Q'| < |Q|$ then                                      ☞ Update solution
            $Q \leftarrow Q'$
        end if
        if $L(u') < L(u)$ then
            $u \leftarrow u'$
        end if
        Extend $F'$ by fixing additional probes
        Choose $r \in [-0.1, 0.1]$ randomly                          ☞ Perturb multipliers
        $u_{i,j} \leftarrow (1 + r) \cdot u_{i,j}$
    until $F = P$ or $|F| + L(u) > |Q|$
    return $(Q \setminus F, u)$
end function
```

Subgradient optimization is used to compute ensuing multipliers, starting from the most recent set of multipliers.

LP Relaxation Method

This method computes optimal Lagrangian multipliers. We have shown in Section 2.2 that, if the integrality property holds, the dual values associated with an optimal solution to the LP relaxation pose an optimal solution to the Lagrangian relaxation as well. Given some optimal solution to the LP relaxation, let y^* be the corresponding dual solution. If we assign the value of $y^*_{i,j}$ to Lagrangian multiplier $u_{i,j}$, both relaxations yield the same objective value. This is optimal because the objective value of the Lagrangian dual cannot exceed that of the LP relaxation due to the integrality property.

The main disadvantage of this approach is that it is hard to estimate the computational effort required to solve the LP relaxation to optimality. State-of-the-art LP solvers, however, may be able to solve the LP relaxation faster than the subgradient optimization process takes. At least this is the case with the benchmark instances we use.

Subgradient Method

Caprara et al. propose the use of subgradient optimization, which has been introduced in Section 2.3, to find high quality Lagrangian multipliers. It is particularly useful when tackling very large instances but, on the other hand, there is no guarantee as to the quality of the multipliers it produces. For the most part, the procedure conforms to the pseudocode given in Algorithm 1.

As has been noted above, we do not use this technique to compute Lagrangian multipliers from scratch. Instead, we take the best multipliers found during the previous iteration, perturb them randomly, and use the result as a starting point.

In each iteration the current multiplier vector u will be updated according to (6.6). As for the upper bound U, we use the size of the best known solution for the current subproblem. Prior to the first iteration, we compute an initial solution by means of the greedy heuristic described in Section 6.2.2. Caprara et al. propose to use $u = 0$ as input when computing the initial solution but we decided to initialize the Lagrangian multipliers beforehand, as this seems more reasonable to us.

The step size parameter λ controls how far we move along the subgradient. If we are still far from the optimum, its value should be high. If we come closer to the optimum, its value should get smaller because otherwise we risk to overshoot. We use an initial value of $\lambda = 0.1$. In the classical approach, the step size is halved if the lower bound cannot be improved for a particular number of iterations. We use a slightly different method which, according to Caprara et al., makes for faster convergence. Every $p = 20$ iterations we compare the best and worst lower bounds, $L_{\max}$ and $L_{\min}$, achieved in this period. If these two values are within 0.1% of other, we assume that we either are very close to an optimum or the low improvement is due to the step size being to small. Either way, we will enlarge the step size by a factor of 1.5. Even if we were already very close to the optimum, we do not loose much, because near-optimal multipliers are good enough for our purposes. If, on the other hand, $L_{\min}$ and $L_{\max}$ differ by more than 1%, we decrease the step size by a factor of 2 to prevent moving past the optimum. In summary, this amounts to the following updating scheme:

$$\lambda = \begin{cases} 1.5 \cdot \lambda & \text{if } L_{\max} \leq 1.001 \cdot L_{\min} \\ 0.5 \cdot \lambda & \text{if } L_{\max} \geq 1.01 \cdot L_{\min} \\ \lambda & \text{otherwise} \end{cases}$$

In practice, a large number of probes will have Lagrangian costs very close to zero, which usually implies the existence of many almost optimal solutions each of which can be obtained by including different subsets of those probes into the solution. Correspondingly, there are just as many subgradients (6.5) to choose from. Although it is theoretically possible to determine the subgradient providing the steepest ascent, it is computationally onerous to do so.

Instead, Caprara et al. [10] propose the use of a heuristically selected small-norm subgradient direction, which can be computed quickly and, compared to the subgradient defined in (6.5), leads to faster convergence of the subgradient optimization procedure in most cases.

We start with a partial solution $G := \{p_i \in P : c_i(u) \leq 0.001\}$. Let Q_G be the set of

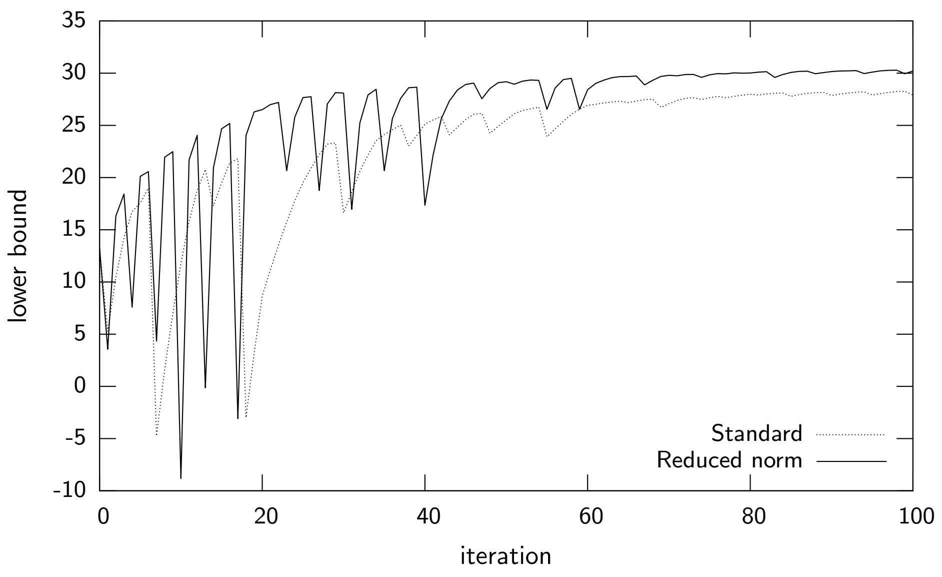

Figure 6.1: Comparison between alternative subgradient definitions

sequence pairs that are distinguished by at least one probe in G. Let R denote the set of redundant probes in G, i. e. those which can be removed safely from G without affecting set Q_G. We walk through set R in arbitrary order, removing a probe from G if this does not affect set Q_G. Thus, we need to check for each probe in turn if its removal would leave any sequence pair in Q_G undistinguished. However, for reasons of efficiency we do not update set R after having deleted a probe from G. The small-norm subgradient is defined with respect to set G, after removing redundant elements as described above, as follows:

$$\dot{s}_{i,j}(u) := r - \sum_{p_k \in G} \delta_{i,j}(k) \tag{6.7}$$

Note that, strictly speaking, $\dot{s}(u)$ may not qualify as a subgradient anymore. Figure 6.1 contrasts the standard subgradient as defined in (6.5) with the reduced-norm variant (6.7). It exemplifies the evolution of the lower bound for both subgradient variants, when applied to one of our benchmark instances. As one can see, the use of the small-norm subgradient causes the subgradient procedure to converge to a higher lower bound. When applied to other benchmark instances, the results look similar.

We terminate the subgradient optimization when the lower bound does not improve anymore, or only to a very small extent. This is the case if the absolute and relative lower bound improvement over the the last 300 iterations is below 1.0 and 0.1%, respectively. Furthermore, to be on the safe side we impose a hard limit of $10 \cdot |Q_G|$ on the number of iterations.

6.2.2 Greedy Heuristic

Given a set of Lagrangian multipliers, the greedy heuristic presented in this section produces a feasible solution for a given SB instance. More precisely, we compute a series of Lagrangian multipliers, starting with the best known multiplier vector. For each multiplier vector in this series, we generate a feasible solution and return the best solution found.

We perform a fixed number of iterations of the subgradient optimization procedure given in Algorithm 1 to generate a series of Lagrangian multiplier sets. Apart from the fixed number of iterations, we also use the standard subgradient definition given in (6.5). Note that, as we are basically continuing the subgradient optimization procedure, we may as well find some better Lagrangian multipliers. See Algorithm 4 for the corresponding pseudocode.

Algorithm 4 Iterative heuristic

```
 1: function HeurOpt($S, P, r, F, u$)
 2:     $Q \leftarrow P \setminus F$
 3:     $u^* \leftarrow u$
 4:     for $i = 1, \ldots, 250$ do
 5:         $Q' \leftarrow$ Greedy($S, P, r, F, u$)
 6:         if $|Q'| < |Q|$ then
 7:             $Q \leftarrow Q'$
 8:         end if
 9:         Update $u$ according to (6.6)
10:         if $L(u) > L(u^*)$ then
11:             $u^* \leftarrow u$
12:         end if
13:     end for
14:     return $(Q, u^*)$
15: end function
```

The greedy procedure we use to compute a feasible solution for a given set of Lagrangian multipliers is depicted in Algorithm 5. We start with a given, possibly empty, partial solution Q, which we extend probe by probe until we reach a feasible solution. The key point is the way we select the probes to add to the partial solution. For this, we assign to each candidate probe p_i a score σ_i, which is based on the number sequence pairs not yet completely distinguished by the probes in Q that p_i distinguishes and its Lagrangian cost. More formally:

$$\mu_i := \sum_{(s_j,s_k)\in S^2(Q)} \delta_{j,k}(i)$$

$$\gamma_i := 1 - \sum_{(s_j,s_k)\in S^2(Q)} \delta_{j,k}(i) \cdot u_{j,k}$$

$$\sigma_i := \begin{cases} \frac{\gamma_i}{\mu_i} & \text{if } \gamma_i > 0 \\ \gamma_i \mu_i & \text{if } \gamma_i \leq 0 \end{cases}$$

In each iteration we add the probe p_i with minimum score σ_i to Q.

As soon as Q has become feasible, we try to remove as many redundant probes from it as possible. A probe is considered redundant if Q is still feasible after its removal. As long as there are more than 10 of them, we choose one of them randomly, remove it from the solution and update the set of redundant probes. As soon as there are only 10 or less left we perform an exhaustive search to find a subset of redundant probes with maximum cardinality that can be removed safely from our solution without affecting its feasibility.

6.2.3 Probe Fixing

Especially when tackling large instances, the solution available at the end of an iteration of function CoreOpt can be further improved by fixing some probes, that is we require some probes to be part of the solution, and re-applying the function to the resulting subinstance.

Our goal is to fix probes which are highly likely to be member of an optimal solution. To find probes that match this criterion, we employ a simple greedy procedure.

Algorithm 5 Greedy heuristic

```
1:  function GREEDY($S, P, r, F, u$)
2:      $Q \leftarrow F$
3:      while $S^2(Q) \neq \emptyset$ do
4:          Let $p^* \in P \setminus Q$ be the probe with minimum score $\sigma$
5:          $Q \leftarrow Q \cup \{p^*\}$
6:      end while
7:      Let $R$ be the set of redundant probes in $Q$
8:      while $|R| > 10$ do
9:          Remove random probe $r \in R$ from $Q$
10:         Update $R$
11:     end while
12:     Remove the maximum number of redundant probes from $Q$
13:     return $Q \setminus F$
14: end function
```

First, we compute the Lagrangian costs of all probes with respect to the best Lagrangian multipliers found for the current subproblem. Let F' be to set of probes with Lagrangian costs below -0.001 and Q the set of sequence pairs distinguished by at least one probe in F'. From set F' we remove those probes that do not distinguish at least one sequence pair that is not distinguished by any other probe in F'. In other words, we remove a probe from F' if and only if this does not affect set Q.

We apply the greedy heuristic presented in Section 6.2.2 to the subinstance defined by F'. Let F'' be the generated solution. In addition to the probes in F', we fix at most $\lfloor |Q|/200 \rfloor$ probes of F''; if F'' contains more probes, we select a random subset.

6.2.4 Pricing

The computationally most expensive parts of the overall algorithm are the multiplier optimization (Section 6.2.1) and the greedy heuristic (Section 6.2.2). Especially in case of very large instances, these parts should be implemented as efficient as possible to keep the run time low. Inspired by similar techniques used to solve very large LPs, we use a pricing technique to achieve this goal. According to Caprara et al., this has not been done before in combination with Lagrangian relaxation.

Basically, pricing makes only a subset of the available probes visible to the two aforementioned parts of the algorithm. This subset, which we will refer to further on as the *core*, will be updated regularly. The dynamic nature of the core is meant to avoid missing very good or even optimal solutions due to this artificial restriction.

The choice of the probes that make up the core is once more based on their Lagrangian costs. In the beginning, we randomly choose five distinguishing probes for each sequence pair. When updating the core subsequently, the selection process works like this: Let $\mathcal{I} = (S, P, r, F)$ be the instance we are operating on. We start by computing the Lagrangian costs based on the current multiplier vector u^k and initialize the new core with set $C_1 := \{p_i \in P : c_i(u) < 0.1\}$. Additionally, we select for each sequence pair five distinguishing probes at random, just like we did during initialization, and denote this set by C_2. If C_1 contains more than $l := 5 \cdot |S^2(F)|$ elements, we keep only the l probes with the lowest Lagrangian costs and remove all others. The updated core is defined as the union of C_1 and C_2.

We also have to decide on the pricing frequency T, which tells us after how many subgradient iterations to update the core. We initially set $T := 10$. After each core update, we compute the lower bound, once with respect to all available probes and once with respect to the probes that are part of the current core C^k, subtract the first from the second, and divide the result by the best known upper bound U, i. e. the size of the best known solution for instance $\mathcal{I}$:

$$\Theta := -\frac{\sum\limits_{p_i \in P \setminus C^k} \min\{c_i(u^k), 0\}}{U}$$

A small Θ indicates that the core has settled, and that changing it will most probably not lead to any improvements. Thus, increasing T is the right thing to do. On the other hand, if Θ is large, it is advisable to make the update interval smaller. This leads to the following update formula:

$$T := \begin{cases} 10 \cdot T & \text{if } \Theta \leq 10^{-6} \\ 5 \cdot T & \text{if } \Theta \leq 0.02 \\ 2 \cdot T & \text{if } \Theta \leq 0.2 \\ 10 & \text{otherwise} \end{cases}$$

6.2.5 Solution Refinement

Although the solutions produced at the lower level procedure as implemented by function CoreOpt are mostly of high quality, we can often find an even better solution by fixing part of the returned solution and rerun the complete algorithm on the resulting subinstance. Similar to the probe fixing procedure described in Section 6.2.3, we intent to determine those probes of the solution that are most likely to be part of an optimal solution.

The procedure has one parameter π that affects the size of the subinstance that results from fixing the probes we are going to select. Let u^* be the best Lagrangian multiplier vector, Q^* the best solution found for the original input instance $\mathcal{I} = (S, P, r, \emptyset)$, without any fixed probes, and let $\zeta_{i,j}$, $1 \leq i < j \leq n$, denote the number of probes in Q^* that distinguish sequence pair (s_i, s_j). The gap Δ between the corresponding lower and upper bound, that is the size of the best overall solution found so far, is given by:

$$\begin{aligned}
\Delta &= |Q^*| - \left(\sum_{1 \leq i < j \leq n} u^*_{i,j} + \sum_{i=1}^{m} \min\{c_i(u^*), 0\} \right) \\
&= \sum_{p_i \in Q^*} \left(\underbrace{c_i(u^*) + \sum_{1 \leq j < k \leq n} \delta_{j,k}(i) \cdot u^*_{i,j}}_{=1} \right) - \sum_{1 \leq i < j \leq n} u^*_{i,j} - \sum_{i=1}^{m} \min\{c_i(u^*), 0\} \\
&= \sum_{p_i \in Q^* : c_i(u^*) > 0} c_i(u^*) - \sum_{p_i \notin Q^* : c_i(u^*) < 0} c_i(u^*) + \sum_{1 \leq i < j \leq n} u^*_{i,j} \left(\zeta_{i,j} - 1 \right)
\end{aligned}$$

We estimate the extent to which any single probe $p_i \in Q^*$ contributes to this gap, denoted by η_i, as follows:

$$\eta_i := \max\{c_i(u^*), 0\} + \sum_{1 \leq j < k \leq n} \delta_{j,k}(i) \cdot u^*_{j,k} \left(1 - \frac{1}{\zeta_{j,k}} \right) \tag{6.8}$$

Based on the assumption that probes with a small η are more likely to be part of an optimal solution, we consider the elements of solution $Q^* = \{p_{q_1}, \ldots, p_{q_l}\}$ in ascending order with respect to their gap estimate, i. e. $\eta_{q_1} \leq \ldots \leq \eta_{q_l}$. Depending on parameter π, the set of probes to fix is the smallest subset $F := \{p_{q_1}, \ldots, p_{q_i}\}$, $i \leq l$, such that the percentage of sequence pairs fully distinguished by F is greater than or equal to parameter π:

$$1 - \frac{2 \cdot |S^2(F)|}{|S| \cdot (|S| - 1)} \geq \pi \tag{6.9}$$

7 Implementation

This chapter provides details on our implementation of the CFTSB heuristic. We implemented the complete algorithm in C++ under the Linux operating system, with notable use of the standard template (STL) and the Boost [1] libraries. Section 7.1 summarizes the overall workflow of computing a probe set for a given set of sequences, Section 7.2 specifies the data formats of the input files, Section 7.3 addresses some implementation details, and Section 7.4 gives an overview of the classes of our design.

7.1 Workflow

Given a set of sequences, the workflow to compute a high-quality probe set is as follows. A separate program, which we will refer to as *probe generator* hereinafter, is used to compute an exhaustive set of candidate probes according to the description in Chapter 4. The probe generator also enforces any length constraints. The generated probes, along with the information in which sequences each probe occurs, will be handed as input to the actual implementation of the CFTSB heuristic. The separation of these two stages has the advantage that we do not need to repeat the probe generation phase when running the CFTSB heuristic multiple times on the same instance. However, when we change the length constraints or the redundancy parameter, we have to call the probe generator again.

Figures 7.1 and 7.2 summarize the command line options accepted by the probe generator and the main heuristic, respectively.

`--help`	Display list of supported command line options along with a short description.
`--ifile` *file*	Read instance from *file*.
`--ofile` *file*	Write output to *file*.
`--redundancy` n	Set the number of probes in the solution by which each sequence pair has to be distinguished to n. As pointed out in Section 4.3, we have to generate at least this number of candidate probes per signature (or as many as possible).
`--pmin` n	Set the minimum probe length to n. If omitted, assume $n = 1$.
`--pmax` n	Set the maximum probe length to n.
`--compress`	Automatically gzip compress the output file.

Figure 7.1: Command line options of the probe generator

`--help`	Display list of supported command line options along with a short description.
`--seed` n	Seed the pseudo-random number generator with n
`--ifile` *file*	Read instance from *file*.
`--redundancy` n	Set the number of probes in the solution by which each sequence pair has to be distinguished to n.
`--maxrt` n	Set the maximum run time to n seconds.
`--optimum` n	Set the size of the optimal solution, if known, to n. If not omitted, the algorithm terminates as soon as it finds a solution of this size. Chiefly useful for benchmarking.

Figure 7.2: Command line options of the CFTSB implementation

```
>gi|2769620|emb|Y16151.1|HIVY16151 HIV-1 pol gene, isolate RO-BCI9
ATGGGTGCGAGAGCGTCAGTATTAAGTGGGGGAAAATTAGATGCATGGGAAAAAATTCGGTTACGGCCAG
GGGGAAAGAAAAAATATAGGATAAAACATTTAGTATGGGCAAGCAGAGAGTTAGAAAGATTCGCTCTTAA
CCCTGGCCTTTTAGAAACAGCAGAAGGATGTCAACAAATACTAGAACAGTTACAGTCAACTCTCAAGACA
GGATCAGAAGAACTTAAATCATTATTTAATACAATAGCAACCCTCTGGTGCGTACACCAAAGGATAGATG
TAAAAGACACCAAGGAAGCTTTAGATAAAATAGAGGAAGTACAAAAGAAAAGCCAGCAAAAGACACAGCA
GGCAGCAGCTGGCCCAGGAAGCAGCAGCAAAGTCAGCCAAAATTATCCTATAGTGCAAAATGCACAACCA
CAAATGGTACAC
```

Figure 7.3: Single DNA sequence in FASTA format

7.2 Data Formats

The probe generation application operates on plain text files that comply with the FASTA format [30]. Each file contains a set of sequences, and each sequence starts with a single line description, followed by lines of sequence data. Description lines always start with a > symbol. Figure 7.3 provides an example.

The probe generator produces a plain text file with the following structure. The first line contains the number of input sequences found in the original input file. As described in Section 4.1, each node in the suffix tree is associated with a binary signature that tells us which sequences contain the probes that correspond to this node. For each such set of probes the output file contains a sequence of lines, each containing a probe and its minimum length, followed by the indices of the sequences, which these probes occur in. An example can be found in Figure 7.4. The first line represents 15 distinct probes, namely all prefixes of length at least 22:

- `AGCCTGGGAGCTCTCTGGCTAG`
- `AGCCTGGGAGCTCTCTGGCTAGC`
- `AGCCTGGGAGCTCTCTGGCTAGCT`
- ...
- `AGCCTGGGAGCTCTCTGGCTAGCTAGGGAACCCACT`

The same goes for the second line. The third line tells us, that all these probes are contained exclusively in sequences 0, 8, and 19.

7.3 Implementation Details

One task that we have to accomplish over and over again during the course of the algorithm is to iterate over all sequence pairs distinguished by a particular probe or, vice versa, over all probes distinguishing a particular sequence pair. To make this as efficient as possible, we have to keep this information in memory. Unfortunately, this information changes whenever we are fixing or hiding some probes. To solve this problem, we represent each probe as a *dynamic vector* containing references to those sequence pairs it distinguishes. Correspondingly, each sequence pair is represented as a dynamic vector containing references to those probes by which it can be distinguished. The set of all candidate probes and sequence pairs are held in globally available dynamic vectors, which are implemented as singleton objects [36].

A dynamic vector offers the possibility to hide and unhide elements at any time. It is implemented as a continuous array with all hidden elements located at the end; we maintain the index I_{hidden} of the first hidden element. When hiding an element, we actually exchange it with the last active element, i. e. the predecessor of the first hidden element, and decrement I_{hidden}. Reactivation of a previously hidden element works accordingly. While these operations can be accomplished in constant time, the overall process takes linear time. This is due to the fact that we know the containing vector of an element but have to traverse all elements to find its actual position therein. The main disadvantage of this approach is that hiding or reactivating an element changes the order of elements in the data structure. In contrast to the implementation of the original CFT heuristic, the possibility to efficiently hide probes and sequence pairs temporarily frees us from the necessity to construct subinstances explicitly in memory. Since memory usage is an important issue, especially as instances become larger, this is a great advantage.

At any point in time, we operate on exactly one subinstance, defined by the set of hidden probes and sequence pairs. This subinstance changes

```
AGCCTGGGAGCTCTCTGGCTAGCTAGGGAACCCACT 22
AGCTAGGGAACCCACT 15
0 8 19
```

Figure 7.4: Sample probe specification

- during probe fixing (Section 6.2.3) and solution refinement (Section 6.2.5), where we choose some probes to be fixed as part of the forthcoming solution. In this case, we need to hide the respective probes as well as all sequence pairs that are fully distinguished by these probes. After doing so, we need to recompute the Lagrangian costs and the lower bound.

- when we initialize or update the core during the pricing procedure (Section 6.2.4). In this case, we hide some probes to make the algorithm faster. However, as we do not hide any sequence pairs, only the lower bound is invalidated and needs to be recomputed.

Contrary to the original CFT algorithm, some of the decisions inside the CFTSB heuristic are random. In these cases, we use a SIMD-oriented Fast Mersenne Twister [33] as pseudo-random number generator (PRNG). This kind of PRNG is known for its speed and provides a very high degree of equidistribution.

7.4 Class overview

This section gives an overview of the most important classes of our implementation. An UML class diagram depicting the inheritance relations between these classes is shown in Figure 7.5. Below, we describe briefly the capabilities of each class. Note that, out of all listed classes, class Instance is missing from Figure 7.5. Although it bears no inheritance relation to any of the other classes it is used by the majority of them, so its inclusion would have cluttered the diagram unnecessarily.

DynamicElement Abstract superclass representing objects that can be active or hidden. Objects that are to be stored in a DynamicList must be instances of a subclass of this class. Accordingly, both Probe and SequencePair derive from it.

DynamicList A vector of instances of DynamicElement or a subclass thereof. When the state of any contained element changes, either from hidden to active or vice versa, we need to call `updateActive` if we hid elements, `updateHidden` if we reactivated elements, or `update` if we did both. It is not until the appropriate update method has been called,that the actual masking takes effect. This class offers the possibility to iterate over either all, only the hidden, or only the active elements.

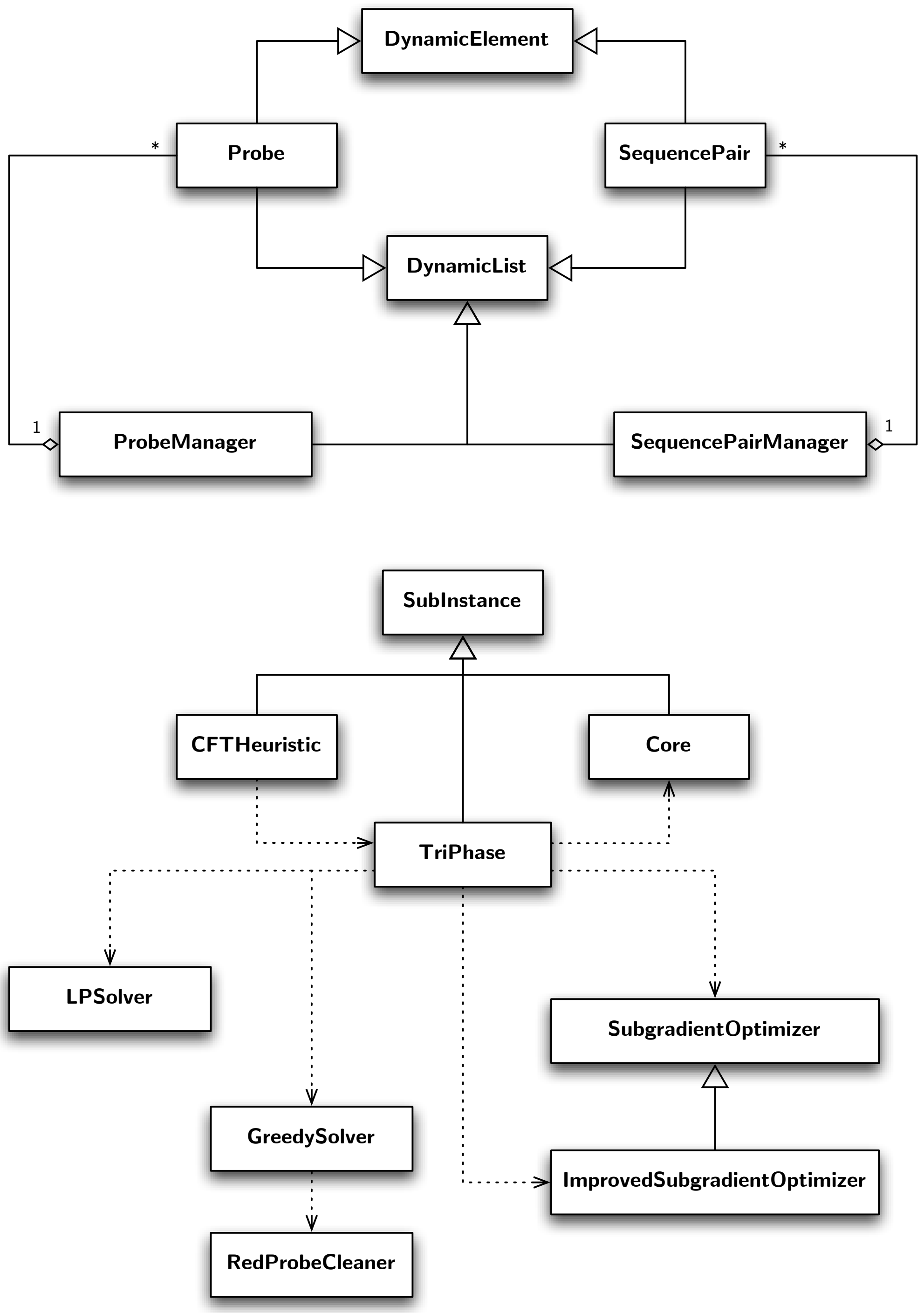

Figure 7.5: UML class diagram

Probe Derives from DynamicElement and DynamicList. Each instance of this class represents a particular candidate probe and holds a list of all sequence pairs that this probe distinguishes. Additionally provides a method to query its Lagrangian cost.

SequencePair Derives from DynamicElement and DynamicList. Each instance of this class represents a particular sequence pair. Provides methods to query the currently assigned Lagrangian multiplier and the number of probes that are required to distinguish the sequence pair. Note that this number changes whenever probes are fixed as part of the solution.

ProbeManager Derives from DynamicList. There is exactly one instance that holds a list of all candidate probes of an instance. This instance is globally available as singleton object.

SequencePairManager Derives from DynamicList. There is exactly one instance that holds a list of all sequence pairs of an instance. This instance is globally available as singleton object.

Instance This class maintains important data pertaining to the instance as a whole. This includes the required number of probes by which each sequence pair must be distinguished, the minimum and maximum probe length, and the number of sequences. Moreover, it provides methods to recompute Lagrangian costs and the lower bound. Moreover, this classes handles the parsing of input instances. There is exactly one instance, globally available as singleton object.

SubInstance Represents a subinstance. Provides methods `fixProbes`, `restoreFixed`, `hideProbes`, and `restoreHidden` to fix, hide, or reactivate a subset of probes.

Core This class implements all aspects, that is initialization and update, of the pricing technique addressed in Section 6.2.4.

GreedySolver Implements the greedy heuristic described in Section 6.2.2.

RedProbeCleaner Implements the removal of redundant probes from a given solution as described in Section 6.2.2. A recursive method enumerates all subsets of redundant probes and eventually removes one with maximum cardinality.

LPSolver Computes an optimal solution to the current subinstance, we are operating on. This is accomplished by constructing the corresponding ILP and solving its LP relaxation using the ILOG CPLEX LP solver.

SubgradientOptimizer This class encapsulates the implementation of the subgradient optimization procedure using the standard subgradient definition (6.5) as it is used by the greedy heuristic, which has been presented in Section 6.2.2. Method `run` launches the subgradient optimization procedure, whereas method `step` performs a single iteration only. The latter will be used when we are computing heuristic solutions (Section 6.2.2). The constructor accepts an optional Core object that will be used throughout its lifetime.

ImprovedSubgradientOptimizer Subclass of SubgradientOptimizer that employs the small-norm subgradient definition given in Section 6.2.1.

TriPhase Encompasses the complete lower level part of the CFTSB heuristic according to its description in Section 6.2.

CFTHeuristic Realization of the top level part of the CFTSB heuristic as outlined in Algorithm 2.

8 Computational Results

This chapter presents the computational results obtained from a selection of benchmark instances and compares the quality of the solutions computed by the CFTSB heuristic to those of the simpler DNA-BAR heuristic [13] as well as to optimal solutions. Section 8.1 describes the benchmark instances we used, where we obtained them from or how they have been generated. The actual results are presented in Section 8.2 with a detailed discussion following in Section 8.3.

8.1 Instances

For our benchmark runs, we selected some of the instances on which the results presented by Rash and Gusfield [32] were based. These are derived from real-world genomic data of 4,548 different HIV strains publicly available through the GenBank [31]. Each instance class (hiv0, hiv1, hiv4) contains 25 instances. To assess the quality of the solutions produced by the CFTSB heuristic, we compared our results to the optimal solutions, which we computed by solving the corresponding ILPs. We found that the computational effort to solve these instances to optimality tends to grow as the required level of redundancy r decreases. For $r = 1$ we were not able to compute optimal solutions for any of the given instances within reasonable time or had to abort the runs due to excessive memory consumption. Therefore, we additionally constructed a set of smaller instances from those in instance class hiv1 – we chose this instance class arbitrarily. From each original instance, we selected a random subset of sequences, containing between 30 and 75% of the sequences of the original instance. For each instance we tried different percentages until the resultant instance could be solved to optimality in reasonable time. The resulting 24 instances constitute instance class hiv1a . All instances are available for download at `http://www.ads.tuwien.ac.at/phil/mthesis/instances.tar.bz2`.

CPU	2 × Intel Xeon 5150 *Clock speed . . . 2.66GHz* *L2 cache . . . . . . 4MB* *Cores 2*
Memory	8GB
Operating System	Debian Linux 4.0 amd64
Compiler	GCC 4.1.2 20061115 (prerelease) (Debian 4.1.1-21)
Optimization Flags	`-O2 -march=nocona -funroll-loops -ftree-vectorize -msse2 -mfpmath=sse -fomit-frame-pointer`
LP Solver	ILOG CPLEX 10.0.1

Table 8.1: Hard- and software environment

8.2 Results

In addition to the results provided by the CFTSB heuristic, we computed an optimal solution for each instance using an ILP solver and compared the result to that of the simpler DNA-BAR heuristic [13]. We chose the latter primarily because out of the SB approaches summarized in Section 5.1, this is the only one for which the source code is publicly available. Due to the non-deterministic nature of the CFTSB heuristic, we performed ten runs per instance, each time with a different PRNG seed between 0 and 9. Accordingly, the reported values below are averages over these runs.

Like Rash and Gusfield, we required the generated probes to be at least 15 and at most 40 nucleotides long. This corresponds to restrictions as they are applied in practice and results in instances that can still be solved optimally. As noted in Section 5.1, requiring a minimum pairwise edit distance greater than one increases the number of candidate probes to an extent that makes computing an optimal solution impossible. For this reason, we refrained from doing so.

For details on the used hard- and software, we refer to Table 8.1. Note that, though the benchmark runs have been performed on a multi-core machine, our implementation uses only one core at a time. The computational results are presented in Tables 8.2 through 8.6. The column labels and their meaning are:

seqs	number of sequences
len	average sequence length

pgen	CPU time in seconds spent for probe generation
opt	size of the optimal solution
dbar	size of the solution produced by the DNA-BAR heuristic
cft^0	size of the first solution encountered by the CFTSB heuristic
cft*	size of the best solution found by the CFTSB heuristic

We restricted the maximum run time for each CFTSB run to one hour. For each solution, we also provide the CPU time spent to compute it (hours, minutes, and second, separated by colons) and, in case of the heuristic solutions, the relative distance to the optimal solution in percentages (gap). Given the size ω^* of an optimal solution, the gap for a heuristic solution of size ω is given by $(\omega - \omega^*)/\omega$.

We also provide the standard deviation (column sdev) for the CFTSB results. The last row of each table shows the mean, or median in case of run times, over all instances of a class.

Figures 8.1 through 8.5 provide a more visual comparison of the quality of the solutions computed by each algorithm. Each bar represents a single instance, and its total height corresponds to the size of the solution produced by the DNA-BAR heuristic. Each algorithm is associated with a different color, black for the optimal solution and different shades of gray for the other algorithms. The size of a solution produced by an algorithm is given by the total height of the part of the bar with the color that corresponds to the respective algorithm and all the parts below that. Due to this layout, it is easy to see that the the following relation holds for all results: opt $\leq$ cft* $\leq$ cft^0 $\leq$ dbar.

8.3 Discussion

It is noticeable that the solutions generated by the CFTSB heuristic are less than satisfactory if we require a redundancy greater than one. For many instances, finding the best solution takes longer than computing the optimal solution by solving the ILP. The average gap for benchmark runs with a redundancy greater than one is about 2%. With the exception of instance class hiv4, redundancy 2, the median run times for computing optimal solutions lie between 44.99 and 187.97 seconds. For the same instances, the median run times of the CFTSB heuristic lie between about 10 and 20 minutes, which is nearly 4 to 26 times as much. Moreover, the algorithm reaches an optimal solution

inst	seqs	len	pgen	opt	time	dbar	gap	time	cft^0	sdev	gap	time	cft*	sdev	gap	time
00	96	657.2	0.22	102	0:00:22.54	167	63.73	0:00:00.42	126.1	2.92	23.63	0:00:01.32	104.9	0.99	2.84	0:15:27.13
01	95	576.3	0.19	103	0:00:52.63	176	70.87	0:00:00.42	127.9	2.47	24.17	0:00:01.02	105.0	0.82	1.94	0:20:27.14
02	75	1328.1	1.06	81	0:01:00.69	152	87.65	0:00:00.44	99.8	2.04	23.21	0:00:00.74	82.5	0.53	1.85	0:18:17.91
03	88	696.1	0.25	102	0:00:12.66	172	68.63	0:00:00.49	126.4	3.37	23.92	0:00:00.83	103.3	0.95	1.27	0:20:34.43
04	88	843.5	0.36	90	0:00:44.99	155	72.22	0:00:00.40	110.6	2.41	22.89	0:00:00.80	91.8	0.92	2.00	0:13:01.74
05	95	688.8	0.28	97	0:00:23.44	165	70.10	0:00:00.49	115.9	3.31	19.48	0:00:01.47	98.9	0.88	1.96	0:29:11.66
06	92	561.1	0.17	96	0:00:25.72	171	78.13	0:00:00.36	115.9	2.64	20.73	0:00:00.91	97.2	0.92	1.25	0:07:40.68
07	88	624.5	0.2	101	0:00:11.29	173	71.29	0:00:00.43	119.8	3.68	18.61	0:00:00.92	102.1	0.74	1.09	0:22:31.85
08	110	2332.0	18.73	108	0:29:40.71	171	58.33	0:00:01.30	136.3	3.40	26.20	0:00:02.44	110.9	1.29	2.69	0:26:19.71
09	97	578.9	0.19	118	0:00:39.83	212	79.66	0:00:00.62	145.6	3.75	23.39	0:00:00.76	121.4	0.97	2.88	0:26:02.67
10	94	581.2	0.18	93	0:04:10.42	171	83.87	0:00:00.36	112.0	3.16	20.43	0:00:01.16	94.4	0.70	1.51	0:19:51.89
11	93	773.7	0.3	103	0:00:42.04	171	66.02	0:00:00.61	127.2	5.27	23.50	0:00:01.04	105.5	0.71	2.43	0:21:57.73
12	86	560.1	0.12	94	0:00:12.48	150	59.57	0:00:00.34	114.6	1.65	21.91	0:00:00.77	95.4	1.17	1.49	0:20:40.18
13	92	760.9	0.38	94	0:00:41.33	168	78.72	0:00:00.56	120.6	2.67	28.30	0:00:00.98	96.2	0.92	2.34	0:14:50.02
14	92	525.4	0.18	94	0:51:26.99	145	54.26	0:00:00.29	115.5	3.81	22.87	0:00:00.88	96.2	0.92	2.34	0:21:06.73
15	78	2078.1	5.64	73	0:01:43.13	126	72.60	0:00:00.60	87.5	3.60	19.86	0:00:00.79	74.3	0.48	1.78	0:22:37.93
16	108	802.2	0.43	105	1:21:26.90	167	59.05	0:00:01.06	125.8	3.65	19.81	0:00:02.10	107.6	1.07	2.48	0:30:44.90
17	99	2172.9	18.6	104	0:13:36.97	175	68.27	0:00:00.98	125.8	2.10	20.96	0:00:01.48	107.9	1.45	3.75	0:19:13.61
18	100	607.3	0.2	112	0:12:40.77	177	58.04	0:00:02.58	134.2	3.16	19.82	0:00:01.14	113.2	0.42	1.07	0:17:18.03
19	93	3058.2	29.26	99	0:20:11.32	164	65.66	0:00:01.12	125.1	3.00	26.36	0:00:01.20	101.3	0.48	2.32	0:14:40.73
20	88	528.7	0.12	93	0:05:10.06	154	65.59	0:00:00.38	109.6	3.63	17.85	0:00:00.83	94.2	0.42	1.29	0:15:35.02
21	80	666.7	0.23	96	0:00:21.08	154	60.42	0:00:00.29	113.3	2.95	18.02	0:00:00.48	97.9	0.74	1.98	0:30:07.94
22	92	616.3	0.15	93	0:02:14.39	151	62.37	0:00:00.48	110.9	2.56	19.25	0:00:01.83	94.8	0.42	1.94	0:19:39.71
23	77	561.0	0.11	94	0:00:02.42	144	53.19	0:00:00.27	109.9	2.38	16.91	0:00:00.47	95.2	0.42	1.28	0:10:57.03
24	90	803.3	0.34	90	0:00:26.64	152	68.89	0:00:00.62	107.1	2.18	19.00	0:00:01.94	91.8	0.63	2.00	0:28:05.90
∅	91.4	959.3	0.23	97.4	0:00:44.99	163.3	67.68	0:00:00.48	118.5		21.70	0:00:00.98	99.4		2.01	0:20:27.14

Table 8.2: Results for instance class hiv0, probe length $\in [15, 40]$, redundancy 4

inst	seqs	len	pgen	opt	time	dbar	gap	time	cft^0	sdev	gap	time	cft*	sdev	gap	time
00	87	494.3	0.1	48	0:14:54.53	73	52.08	0:00:00.26	54.9	1.91	14.37	0:00:00.89	49.1	0.32	2.29	0:05:20.76
01	83	489.1	0.1	52	0:00:39.01	85	63.46	0:00:00.20	60.0	1.70	15.38	0:00:00.51	52.3	0.67	0.58	0:11:15.31
02	76	576.5	0.14	45	0:02:51.60	68	51.11	0:00:00.35	52.5	1.78	16.67	0:00:00.59	45.4	0.52	0.89	0:21:43.80
03	103	448.5	0.1	53	34:35:56.16	87	64.15	0:00:00.34	64.7	2.71	22.08	0:00:02.53	54.4	0.52	2.64	0:14:42.31
04	90	748.5	0.29	50	1:05:31.25	75	50.00	0:00:00.42	62.8	2.10	25.60	0:00:01.40	51.5	0.71	3.00	0:16:04.47
05	89	693.8	0.22	50	0:00:47.85	77	54.00	0:00:00.40	61.0	1.94	22.00	0:00:01.08	51.5	0.53	3.00	0:06:31.22
06	81	586.0	0.12	43	0:10:24.22	68	58.14	0:00:00.23	50.3	1.70	16.98	0:00:00.87	43.1	0.32	0.23	0:05:49.93
07	92	524.7	0.12	53	0:19:06.49	90	69.81	0:00:00.26	63.7	1.89	20.19	0:00:01.46	54.6	0.52	3.02	0:06:48.61
08	73	473.9	0.08	51	0:00:07.98	92	80.39	0:00:00.12	60.2	2.15	18.04	0:00:00.24	51.9	0.32	1.76	0:13:55.37
09	97	473.0	0.12	55	2:55:15.47	79	43.64	0:00:00.34	64.8	1.87	17.82	0:00:02.14	55.6	0.70	1.09	0:18:05.17
10	81	672.8	0.2	49	0:00:34.31	74	51.02	0:00:00.26	60.0	2.71	22.45	0:00:00.64	50.1	0.57	2.24	0:05:59.59
11	76	743.9	0.2	39	0:05:31.39	69	76.92	0:00:00.22	49.7	2.95	27.44	0:00:00.98	40.2	0.42	3.08	0:05:55.69
12	93	764.1	0.3	49	0:44:58.06	78	59.18	0:00:01.07	61.8	2.53	26.12	0:00:01.74	50.3	0.48	2.65	0:10:24.67
13	89	1167.1	1.64	50	0:00:23.26	77	54.00	0:00:00.38	63.2	1.48	26.40	0:00:00.98	50.9	0.74	1.80	0:14:54.17
14	93	903.9	0.42	49	1:39:03.30	77	57.14	0:00:00.43	61.8	2.44	26.12	0:00:01.27	49.5	0.53	1.02	0:12:37.82
15	91	514.3	0.11	50	3:40:33.89	81	62.00	0:00:00.20	62.2	1.32	24.40	0:00:01.18	51.4	0.84	2.80	0:16:35.63
16	89	623.7	0.19	53	0:11:00.86	83	56.60	0:00:00.40	64.5	1.65	21.70	0:00:00.74	54.3	1.16	2.45	0:12:56.54
17	83	930.1	1.03	50	0:00:39.55	77	54.00	0:00:00.21	59.5	1.58	19.00	0:00:00.50	50.7	0.82	1.40	0:16:12.18
18	96	848.3	0.37	58	0:03:06.29	107	84.48	0:00:00.51	71.3	2.75	22.93	0:00:01.56	59.5	0.71	2.59	0:13:23.94
19	99	830.1	0.4	53	0:03:07.97	87	64.15	0:00:00.56	67.5	1.58	27.36	0:00:01.42	53.8	0.79	1.51	0:13:03.40
20	104	746.5	0.38	56	0:01:28.67	89	58.93	0:00:00.62	71.2	2.10	27.14	0:00:01.77	58.2	1.14	3.93	0:19:28.28
21	98	594.4	0.23	56	2:00:34.94	94	67.86	0:00:00.39	69.2	1.32	23.57	0:00:01.36	57.7	0.48	3.04	0:11:07.62
22	86	653.4	0.19	52	0:00:26.14	85	63.46	0:00:00.31	65.0	2.21	25.00	0:00:00.85	53.3	0.67	2.50	0:07:13.43
23	92	804.0	0.34	60	0:00:15.41	103	71.67	0:00:00.40	71.9	2.13	19.83	0:00:00.96	61.3	0.48	2.17	0:15:34.75
24	91	776.0	0.3	49	0:01:47.81	82	67.35	0:00:00.92	62.1	1.91	26.73	0:00:01.48	49.6	0.70	1.22	0:11:42.61
∅	89.3	683.2	0.2	50.9	0:03:07.97	82.3	61.59	0:00:00.35	62.2		22.22	0:00:01.08	52.0		2.14	0:12:56.54

Table 8.3: Results for instance class hiv1, probe length $\in [15, 40]$, redundancy 2

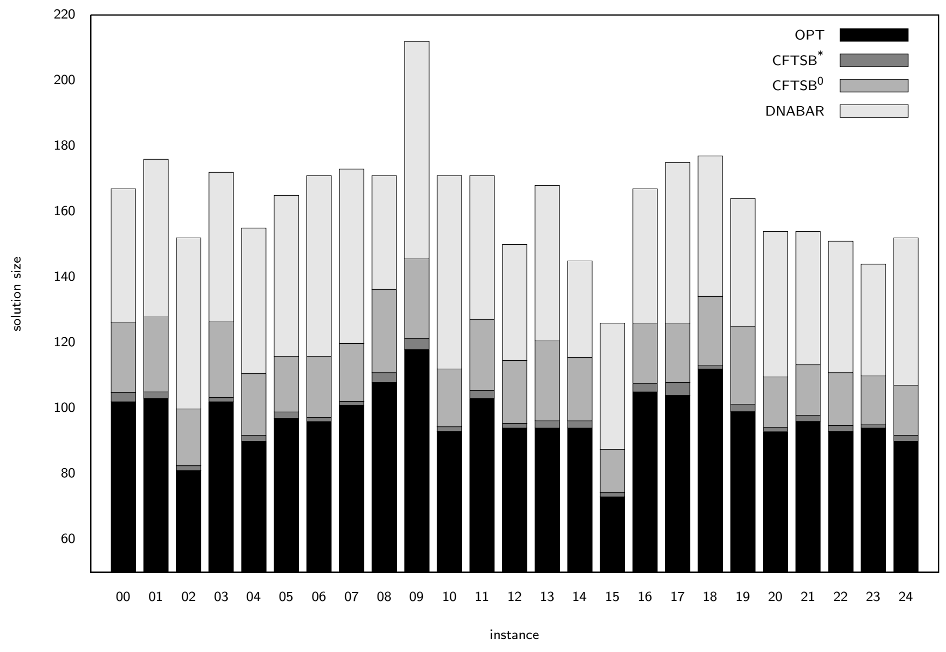

Figure 8.1: Results for instance class hiv0, probe length $\in [15, 40]$, redundancy 4

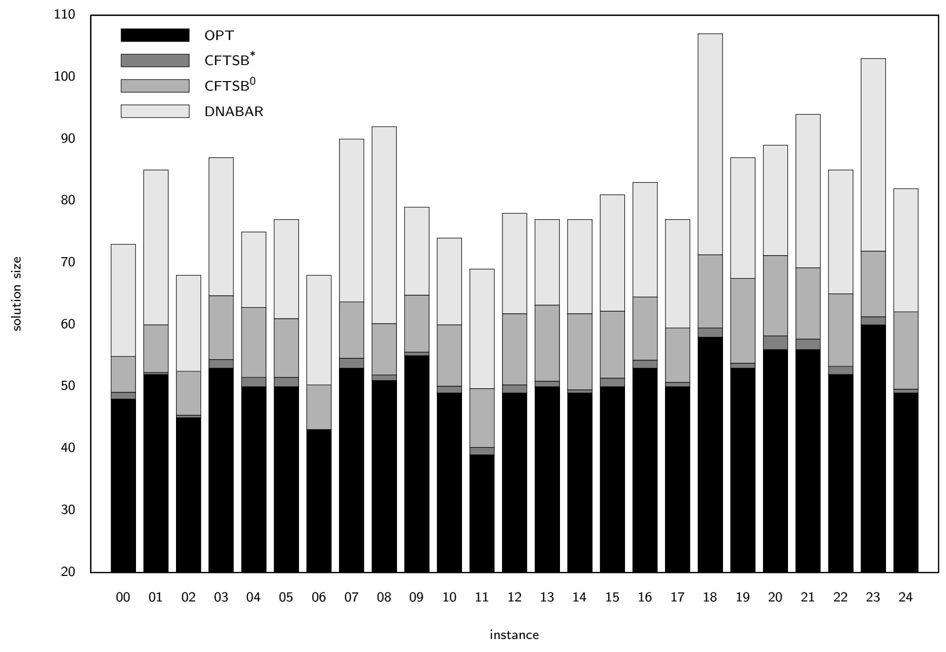

Figure 8.2: Results for instance class hiv1, probe length $\in [15, 40]$, redundancy 2

inst	seqs	len	pgen	opt	time	dbar	gap	time	cft^0	sdev	gap	time	cft*	sdev	gap	time
00	92	652.9	0.21	48	0:26:23.49	83	72.92	0:00:00.32	57.9	1.91	20.62	0:00:03.12	49.5	1.08	3.13	0:12:36.68
01	94	619.3	0.2	52	4:26:16.49	81	55.77	0:00:00.36	63.6	3.17	22.31	0:00:01.51	54.3	1.06	4.42	0:13:17.96
02	95	926.7	0.48	51	0:07:29.96	83	62.75	0:00:00.55	61.5	1.51	20.59	0:00:01.54	52.1	0.32	2.16	0:04:07.34
03	115	536.4	0.15	63	95:58:15.04	97	53.97	0:00:00.60	78.2	2.15	24.13	0:00:02.19	65.2	0.79	3.49	0:17:41.48
04	97	737.6	0.34	57	0:08:38.17	99	73.68	0:00:00.36	68.8	1.03	20.70	0:00:01.12	58.5	0.71	2.63	0:18:22.72
05	85	539.7	0.16	48	0:22:23.19	74	54.17	0:00:00.22	55.4	1.58	15.42	0:00:00.92	48.3	0.48	0.62	0:20:00.60
06	97	658.6	0.28	56	0:17:51.22	92	64.29	0:00:00.39	69.6	2.67	24.29	0:00:00.87	56.8	1.23	1.43	0:24:27.80
07	87	565.2	0.13	46	0:59:33.26	74	60.87	0:00:00.30	54.4	1.35	18.26	0:00:01.01	47.1	0.32	2.39	0:12:25.49
08	79	689.3	0.19	44	0:03:44.24	70	59.09	0:00:00.26	54.8	1.75	24.55	0:00:00.77	45.0	0.00	2.27	0:07:30.27
09	75	728.3	0.24	43	0:00:40.04	71	65.12	0:00:00.28	51.9	1.29	20.70	0:00:00.51	43.1	0.32	0.23	0:01:43.60
10	101	566.0	0.15	55	22:41:35.07	91	65.45	0:00:00.40	66.7	2.63	21.27	0:00:02.71	55.3	0.48	0.55	0:10:32.14
11	83	782.5	0.28	46	0:01:41.43	73	58.70	0:00:00.26	56.5	1.78	22.83	0:00:00.98	46.6	1.07	1.30	0:11:43.08
12	106	760.4	0.38	56	0:47:39.89	88	57.14	0:00:00.61	68.3	3.13	21.96	0:00:02.46	56.8	0.42	1.43	0:17:21.57
13	86	636.1	0.19	47	14:30:38.43	83	76.60	0:00:00.28	59.0	1.70	25.53	0:00:01.12	47.7	0.67	1.49	0:22:35.52
14	95	706.8	0.29	50	0:40:54.60	77	54.00	0:00:00.31	62.7	2.11	25.40	0:00:01.51	51.0	0.67	2.00	0:14:15.77
15	88	576.2	0.18	48	0:04:14.55	71	47.92	0:00:00.26	55.6	1.96	15.83	0:00:01.30	48.9	0.32	1.87	0:05:24.79
16	90	674.7	0.24	44	0:00:25.12	70	59.09	0:00:00.31	51.6	1.71	17.27	0:00:01.63	44.8	1.03	1.82	0:08:30.36
17	105	1645.5	5.68	62	0:37:57.02	110	77.42	0:00:00.90	76.4	2.17	23.23	0:00:01.69	64.8	0.42	4.52	0:12:33.13
18	78	626.6	0.16	47	0:00:16.01	74	57.45	0:00:00.25	55.3	2.21	17.66	0:00:00.56	47.4	0.52	0.85	0:26:29.76
19	87	553.3	0.17	51	0:01:36.19	83	62.75	0:00:00.40	65.4	2.46	28.24	0:00:00.86	52.8	0.42	3.53	0:28:12.31
20	109	949.6	1.04	57	0:22:24.60	85	49.12	0:00:00.72	70.6	2.12	23.86	0:00:02.44	58.6	1.17	2.81	0:18:40.10
21	84	918.0	0.41	49	0:01:29.06	77	57.14	0:00:00.34	58.1	1.37	18.57	0:00:01.07	50.0	0.47	2.04	0:05:59.97
22	71	640.3	0.16	44	0:00:20.40	73	65.91	0:00:00.20	51.8	1.87	17.73	0:00:00.33	44.0	0.00	0.00	0:08:10.54
23	78	474.5	0.07	48	1:02:19.06	75	56.25	0:00:00.16	54.8	1.99	14.17	0:00:00.51	48.0	0.00	0.00	0:02:25.11
24	93	708.1	0.32	51	0:03:37.63	85	66.67	0:00:00.40	63.8	2.30	25.10	0:00:00.92	51.1	0.32	0.20	0:22:52.22
∅	90.8	714.9	0.21	50.5	0:17:51.22	81.6	61.44	0:00:00.32	61.3		21.35	0:00:01.12	51.5		1.96	0:12:36.68

Table 8.4: Results for instance class hiv4, probe length $\in [15, 40]$, redundancy 2

inst	seqs	len	pgen	opt	time	dbar	gap	time	cft^0	sdev	gap	time	cft*	sdev	gap	time
00	92	652.9	0.22	93	0:25:07.05	159	70.97	0:00:00.48	107.6	1.90	15.70	0:00:02.49	94.3	0.48	1.40	0:28:21.76
01	94	619.3	0.2	100	0:06:51.91	165	65.00	0:00:00.56	122.2	3.61	22.20	0:00:01.38	102.9	0.74	2.90	0:14:23.12
02	95	926.7	0.47	99	0:01:36.95	162	63.64	0:00:00.89	126.7	1.42	27.98	0:00:01.58	101.1	0.57	2.12	0:22:12.12
03	115	536.4	0.15	119	0:06:31.80	185	55.46	0:00:00.88	151.7	1.70	27.48	0:00:02.68	123.7	1.34	3.95	0:23:11.58
04	97	737.6	0.34	110	0:00:15.86	199	80.91	0:00:00.60	132.7	2.58	20.64	0:00:01.54	112.6	1.17	2.36	0:17:32.86
05	85	539.7	0.16	92	0:01:55.03	149	61.96	0:00:00.33	105.0	3.13	14.13	0:00:00.92	93.0	0.00	1.09	0:08:28.43
06	97	658.6	0.26	108	0:00:20.42	177	63.89	0:00:00.65	138.0	2.58	27.78	0:00:00.98	111.7	0.95	3.43	0:29:44.17
07	87	565.2	0.12	88	9:36:20.02	143	62.50	0:00:00.42	106.0	2.58	20.45	0:00:01.07	90.1	0.57	2.39	0:15:49.24
08	79	689.3	0.19	84	0:00:26.06	139	65.48	0:00:00.39	102.6	1.96	22.14	0:00:00.80	85.2	0.63	1.43	0:15:16.14
09	75	728.3	0.25	82	0:00:09.21	142	73.17	0:00:00.44	95.5	1.35	16.46	0:00:00.51	82.7	0.67	0.85	0:19:45.35
10	101	566.0	0.16	104	0:00:24.36	174	67.31	0:00:00.63	127.5	3.31	22.60	0:00:02.87	106.3	0.48	2.21	0:35:19.27
11	83	782.5	0.28	87	0:03:54.52	150	72.41	0:00:00.42	108.7	2.98	24.94	0:00:01.19	88.9	0.74	2.18	0:15:04.00
12	106	760.4	0.37	106	0:02:51.23	178	67.92	0:00:00.95	138.3	3.20	30.47	0:00:01.99	111.3	0.82	5.00	0:30:47.00
13	86	636.1	0.2	87	0:49:25.43	162	86.21	0:00:00.44	106.7	2.21	22.64	0:00:01.03	89.2	0.79	2.53	0:26:57.59
14	95	706.8	0.29	96	0:12:59.81	161	67.71	0:00:00.49	118.2	3.22	23.13	0:00:01.68	97.6	0.70	1.67	0:14:23.68
15	88	576.2	0.17	95	0:00:29.24	141	48.42	0:00:00.40	106.3	1.89	11.89	0:00:01.43	95.1	0.32	0.11	0:07:49.93
16	90	674.7	0.24	87	0:01:01.05	142	63.22	0:00:00.46	100.2	3.05	15.17	0:00:01.81	87.3	0.48	0.34	0:09:47.37
17	105	1645.5	5.61	120	0:02:38.05	197	64.17	0:00:01.46	152.5	4.06	27.08	0:00:01.73	123.3	1.06	2.75	0:32:08.67
18	78	626.6	0.17	90	0:00:25.10	143	58.89	0:00:00.41	109.3	3.40	21.44	0:00:00.42	91.0	0.00	1.11	0:10:16.64
19	87	553.3	0.17	96	0:00:20.94	154	60.42	0:00:00.62	117.7	3.62	22.60	0:00:00.73	99.9	1.10	4.06	0:33:02.01
20	109	949.6	1.01	110	0:01:21.06	168	52.73	0:00:01.09	134.8	3.82	22.55	0:00:02.39	112.7	0.48	2.45	0:20:03.86
21	84	918.0	0.43	94	0:00:03.57	155	64.89	0:00:00.55	111.7	2.06	18.83	0:00:00.84	95.5	0.71	1.60	0:23:56.81
22	71	640.3	0.17	85	0:00:01.26	141	65.88	0:00:00.31	96.8	1.93	13.88	0:00:00.48	85.2	0.42	0.24	0:19:37.59
23	78	474.5	0.08	91	0:00:13.60	157	72.53	0:00:00.26	110.0	2.71	20.88	0:00:00.47	92.6	0.84	1.76	0:08:57.95
24	93	708.1	0.32	99	0:00:14.06	169	70.71	0:00:00.64	120.0	2.79	21.21	0:00:00.93	101.6	0.97	2.63	0:23:46.87
∅	90.8	714.9	0.22	96.9	0:01:01.05	160.5	65.65	0:00:00.49	117.9		21.66	0:00:01.19	99.0		2.18	0:19:45.35

Table 8.5: Results for instance class hiv4, probe length $\in [15, 40]$, redundancy 4

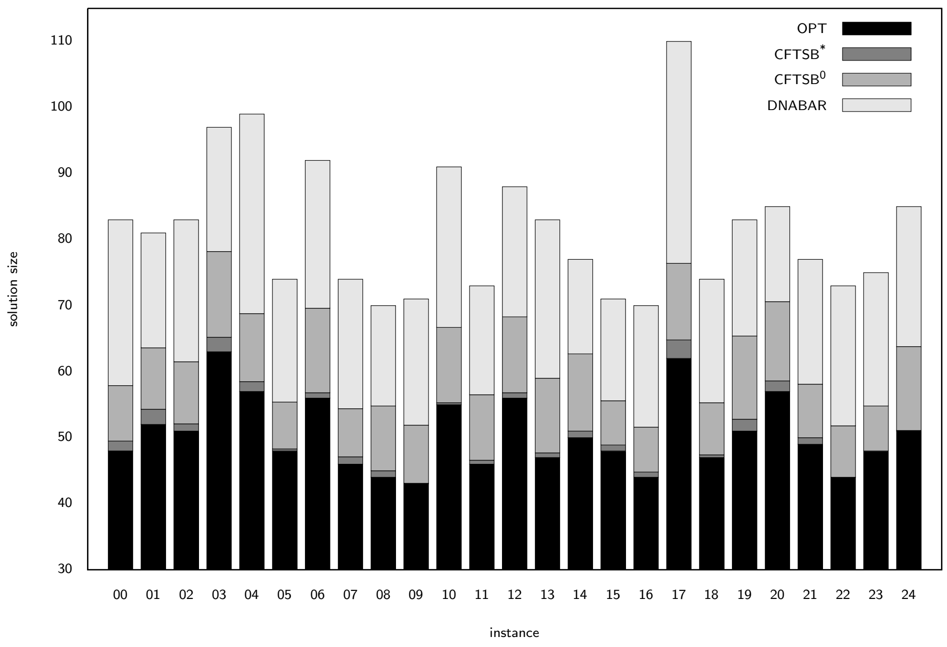

Figure 8.3: Results for instance class hiv4, probe length $\in [15, 40]$, redundancy 2

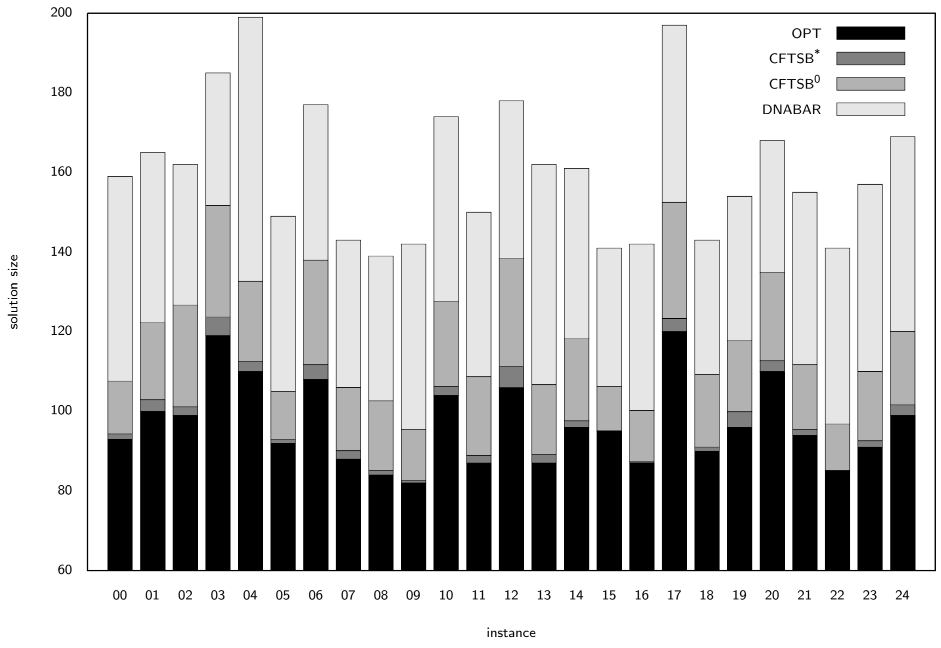

Figure 8.4: Results for instance class hiv4, probe length $\in [15, 40]$, redundancy 4

inst	**seqs**	**len**	**pgen**	**opt**	**time**	**dbar**	**gap**	**time**	**cft^0**	**sdev**	**gap**	**time**	**cft***	**sdev**	**gap**	**time**
00	43	402.4	0.02	24	6:36:47.04	27	12.50	0:00:00.02	26.3	0.48	9.58	0:00:00.04	24.0	0.00	0.00	0:00:00.05
01	64	468.6	0.06	36	1:53:22.61	43	19.44	0:00:00.09	39.8	0.79	10.56	0:00:00.13	36.8	0.42	2.22	0:00:30.28
02	50	643.3	0.12	26	66:13:10.02	31	19.23	0:00:00.05	28.9	0.99	11.15	0:00:00.09	26.3	0.48	1.15	0:00:00.96
03	45	496.8	0.04	22	3:32:58.60	24	9.09	0:00:00.03	23.4	0.52	6.36	0:00:00.09	22.0	0.00	0.00	0:00:00.14
04	48	976.1	0.21	23	0:44:11.32	28	21.74	0:00:00.11	26.4	1.07	14.78	0:00:00.10	23.7	0.48	3.04	0:00:00.47
05	33	896.9	0.12	19	6:59:44.16	21	10.53	0:00:00.04	20.4	0.52	7.37	0:00:00.02	19.0	0.00	0.00	0:00:00.06
06	48	553.8	0.04	19	1:52:51.78	22	15.79	0:00:00.04	20.7	0.67	8.95	0:00:00.12	19.0	0.00	0.00	0:00:02.49
07	43	571.0	0.05	25	0:20:20.14	27	8.00	0:00:00.03	26.1	0.74	4.40	0:00:00.05	25.0	0.00	0.00	0:00:00.07
08	55	453.0	0.05	33	0:28:04.42	38	15.15	0:00:00.05	36.5	1.27	10.61	0:00:00.07	33.2	0.42	0.61	0:00:03.91
09	48	485.0	0.05	23	8:59:37.85	26	13.04	0:00:00.06	25.3	0.67	10.00	0:00:00.13	23.0	0.00	0.00	0:00:00.19
10	36	549.3	0.04	20	0:00:08.92	25	25.00	0:00:00.02	22.8	0.63	14.00	0:00:00.03	20.0	0.00	0.00	0:00:00.38
11	33	1013.5	0.13	18	3:05:28.87	23	27.78	0:00:00.02	19.9	0.88	10.56	0:00:00.03	18.0	0.00	0.00	0:00:00.05
12	38	589.0	0.05	21	2:48:41.65	24	14.29	0:00:00.02	22.0	0.82	4.76	0:00:00.04	21.0	0.00	0.00	0:00:00.05
13	40	1527.6	1.4	22	0:51:43.07	25	13.64	0:00:00.04	23.6	0.52	7.27	0:00:00.03	22.0	0.00	0.00	0:00:00.05
14	50	856.7	0.18	23	46:49:45.38	28	21.74	0:00:00.06	25.3	0.82	10.00	0:00:00.10	23.0	0.00	0.00	0:00:00.11
15	44	592.8	0.06	26	10:58:25.97	29	11.54	0:00:00.02	28.3	0.67	8.85	0:00:00.05	26.0	0.00	0.00	0:00:00.41
16	27	733.7	0.08	17	14:29:58.50	19	11.76	0:00:00.01	17.6	0.52	3.53	0:00:00.01	17.0	0.00	0.00	0:00:00.02
17	35	1630.1	0.8	21	0:06:49.98	25	19.05	0:00:00.02	23.4	0.70	11.43	0:00:00.02	21.0	0.00	0.00	0:00:00.03
18	30	741.0	0.08	21	0:05:57.03	23	9.52	0:00:00.01	21.4	0.52	1.90	0:00:00.02	21.0	0.00	0.00	0:00:00.02
19	36	908.5	0.12	21	79:09:09.99	23	9.52	0:00:00.02	22.5	0.53	7.14	0:00:00.03	21.0	0.00	0.00	0:00:00.35
20	50	493.3	0.04	25	5:15:40.32	30	20.00	0:00:00.05	27.6	0.97	10.40	0:00:00.09	25.0	0.00	0.00	0:00:00.42
21	40	399.8	0.02	22	0:21:58.55	24	9.09	0:00:00.01	23.7	0.67	7.73	0:00:00.04	22.0	0.00	0.00	0:00:00.05
22	38	690.7	0.1	24	0:27:22.11	28	16.67	0:00:00.03	26.0	0.82	8.33	0:00:00.03	24.0	0.00	0.00	0:00:00.08
23	48	708.8	0.12	28	1:56:54.87	35	25.00	0:00:00.05	32.2	1.48	15.00	0:00:00.05	28.0	0.00	0.00	0:00:00.06
∅	42.6	724.2	0.07	23.3	2:22:48.26	27	15.92	0:00:00.03	25.4		9.14	0:00:00.04	23.4		0.36	0:00:00.10

Table 8.6: Results for instance class hiv1a, probe length $\in [15, 40]$, redundancy 1

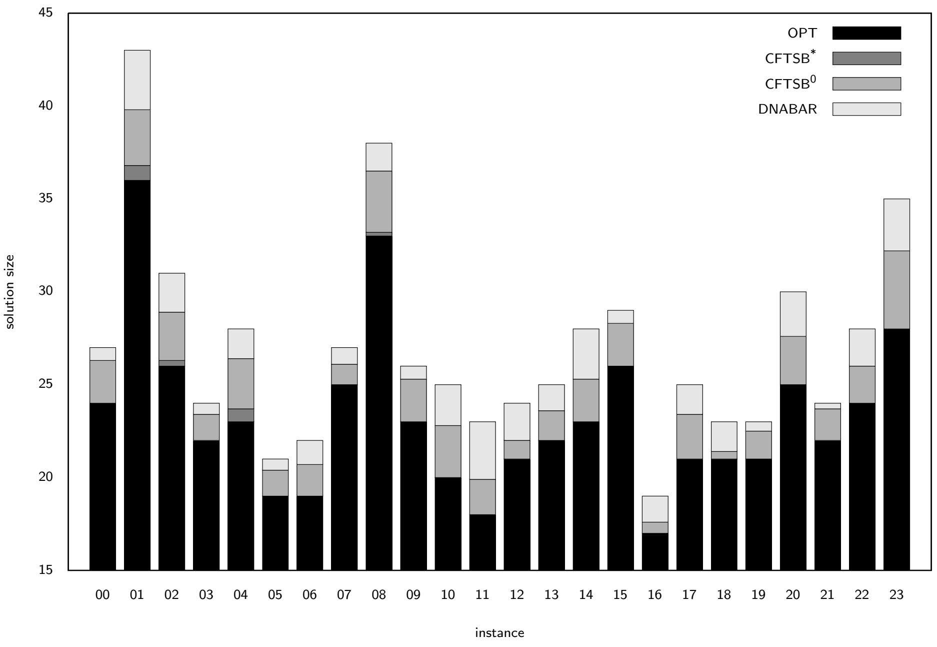

Figure 8.5: Results for instance class hiv1a, probe length $\in [15, 40]$, redundancy 1

in few cases only. On the other hand, the run times for the CFTSB heuristic differ to a much lesser extent. Whereas the run times for computing optimal solutions vary wildly, even within one instance class, the run times of the CFTSB heuristic are all within the same order of magnitude. The low variance of the results over several runs is another advantage. Thus, for complex instances, where solving the ILP takes very long, the CFTSB heuristic provides a viable alternative.

If we look at the runs for instance class hiv1a, where we lower the redundancy to one, the picture changes. The CFTSB heuristic is able to compute an optimal solution for the majority of the instances in a very short time. Computing optimal solutions takes more than one hour for more than half of the instances. For some instances (2, 14 – 16, and 19) we experienced run times between 10 and 80 hours. In these cases, the CFTSB shines as it is able to produce optimal results (with the exception of instance 2) in under one second.

We assume that the heuristic works best when used with a redundancy parameter of one. A reason for this might be that the original approach has been designed for the standard set cover problem, which does not address the aspect of solution redundancy. We extended the original algorithm in a very straight-forward manner to handle this additional requirement, but as it seems, the performance seems to deteriorate with increasing redundancy values.

The DNA-BAR heuristic stands out by its speed, which seems to be independent of the particular instances it operates on. On the other hand, the solutions it produces are considerably worse than the first ones found by the CFTSB heuristic. Since in most cases it takes seldom more than a few seconds to find the first solution, it seems to be a better alternative wherever the focus does not lie on speed alone. Note, however, that you have to take into account the time spent for probe generation, which increases with the number and length of the input sequences.

9 Conclusions

The string barcoding problem is a variant of the well-known set cover problem. Like the latter, it is provably NP-hard and has many practical applications, especially in biology and medicine. In general, we want to identify an unknown sample sequence as one of a set of known sequences without inspecting the sample as a whole. Instead, we are seeking a set of short subsequences (probes) whose presence or absence in the sample we can check efficiently. We are seeking a set of probes of minimum cardinality such that the outcome of these tests allows us to identify the sample sequence uniquely. For each pair of input sequences, a feasible solution contains a predefined number (redundancy) of probes that is part of exactly one of the two sequences.

The problem can be divided into two parts. First, we need to identify the set of possible candidate probes. Each probe must be subsequence of at least one input sequence and must satisfy optional minimum or maximum length constraints. The set of all subsequences is in most cases much too large and contains lots of redundant elements, which appear in the same set of input sequences. A suffix tree can be used to identify a complete set of probes without redundant elements. It can be constructed in linear time and also has linear space complexity.

Few exact and many heuristic approaches address the problem. We adapted a very successful algorithm from literature that targets the set cover problem. It grounds on Lagrangian relaxation and proposes subgradient optimization for finding good Lagrangian multipliers. It is an iterative procedure which uses a Lagrangian heuristic to repeatedly solve different subproblems. Each subproblem is the result of setting up a partial solution of probes with a presumably high probability to appear in an optimal solution. Contrary to the original design, we solve the LP relaxation to find optimal initial Lagrangian multipliers for each subproblem.

We tested our implementation on a set of real-world instances derived from genomic data.

Unfortunately, the heuristic did not deliver results as good as the original approach did for the set cover problem. Especially when a higher solution redundancy was required, the solution quality deteriorated with optimal solutions being found in rare cases only.

On the other hand, our algorithm scales well with regard to instance size and offers a much greater degree of run time predictability. Though the computation of an optimal solution was sometimes even faster than the CFTSB heuristic, the run time over all instances showed an enormous variance. In contrast, the CFTSB heuristic produced good, though in most cases suboptimal, solutions in reasonable time. It was able to compute high quality solutions for very hard instances for which the computation of an optimal solution was excessively resource consuming, with respect to time as well as memory usage. Compared to the much simpler DNA-BAR heuristic proposed by DasGupta et al., our adaptation consistently produced results of superior quality within the same time frame.

10 Bibliography

[1] Boost C++ libraries, 2007. 🌍 `http://www.boost.org`. [Online; accessed 09/08/2007].

[2] Francisco Barahona and Ranga Anbil. The volume algorithm: producing primal solutions with a subgradient method. *Mathematical Programming*, 87(3):385–399, 2000.

[3] J.E. Beasley and P.C. Chu. A genetic algorithm for the set covering problem. *European Journal of Operational Research*, 94:392–404, 1996.

[4] John E. Beasley. *Modern heuristic techniques for combinatorial problems*, chapter Lagrangian Relaxation, pages 243–303. John Wiley & Sons, Inc., New York, NY, USA, 1993.

[5] John E. Beasley. OR-library, 2005. 🌍 `http://people.brunel.ac.uk/~mastjjb/jeb/info.html`. [Online; accessed 10/03/2007].

[6] Piotr Berman, Bhaskar DasGupta, and Ming-Yang Kao. Tight approximability results for test set problems in bioinformatics. In Torben Hagerup and Jyrki Katajainen, editors, *Algorithm Theory - SWAT 2004: 9th Scandinavian Workshop on Algorithm Theory, Humlebaek, Denmark, July 8-10, 2004. Proceedings*, volume 3111 of *Lecture Notes in Computer Science*, pages 39–50. Springer, 2004.

[7] Piotr Berman, Bhaskar DasGupta, and Eduardo Sontag. Randomized approximation algorithms for set multicover problems with applications to reverse engineering of protein and gene networks. In Klaus Jansen, Sanjeev Khanna, José D. P. Rolim, and Dana Ron, editors, *Approximation, Randomization, and Combinatorial Optimization: Algorithms and Techniques*, volume 3122 of *Lecture Notes in Computer Science*, pages 39–50. Springer, 2004.

[8] James Borneman, Marek Chrobak, Gianluca Della Vedova, Andres Figueroa, and Tao Jiang. Probe selection algorithms with applications in the analysis of microbial communities. *Bioinformatics*, 17(90001):39–48, 2001.

[9] Michael J. Brusco, Larry W. Jacobs, and Garry M. Thompson. A morphing procedure to supplement a simulated annealing heuristic for cost- and coverage-corrleated set covering problems. *Annals of Operations Research*, 86:611–627, 1999.

[10] Alberto Caprara, Matteo Fischetti, and Paolo Toth. A heuristic method for the set covering problem. *Operations Research*, 47(5):730–743, 1999.

[11] Alberto Caprara, Paolo Toth, and Matteo Fischetti. Algorithms for the set covering problem. *Annals of Operations Research*, 98:353–371, 2000.

[12] Sebastián Ceria, Paolo Nobili, and Antonio Sassano. A lagrangian-based heuristic for large-scale set covering problems. *Mathematical Programmming*, 81(2):215–228, 1998. doi: http://dx.doi.org/10.1007/BF01581106.

[13] Bhaskar DasGupta, Kishori M. Konwar, Ion I. Mandoiu, and Alex A. Shvartsman. Highly scalable algorithms for robust string barcoding. *International Journal of Bioinformatics Research and Applications*, 1(2):145–161, 2005.

[14] Anton Eremeev. A genetic algorithm with a non-binary representation for the set covering problem. In *Operations Research Proceedings 1998*, pages 175–181. Springer, 1999.

[15] Markus Finger, Thomas Stützle, and Helena Lourenço. Exploiting fitness distance correlation of set covering problems. In *Proceedings of the Applications of Evolutionary Computing on EvoWorkshops 2002*, pages 61–71. Springer, 2002.

[16] Marshall L. Fisher. The Lagrangian relaxation method for solving integer programming problems. *Management science*, 27(1):1–18, 1981.

[17] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1990.

[18] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press, New York, NY, USA, 1997.

[19] Michael Held and Richard M. Karp. The traveling-salesman problem and minimum spanning trees. *Operations Research*, 18(6):1138–1162, 1970.

[20] Michael Held and Richard M. Karp. The traveling-salesman problem and minimum spanning trees: Part II. *Mathematical Programming*, 1(1):6–25, 1971.

[21] Karla L. Hoffman and Manfred Padberg. Solving airline crew scheduling problems by branch-and-cut. *Management Science*, 39(6):657–682, 1993.

[22] Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller and James W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, New York, NY, USA, 1972.

[23] Scott Kirkpatrick, C. D. Gelatt Jr., and Mario P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671 – 680, 1983.

[24] Gunnar W. Klau, Sven Rahmann, Alexander Schliep, Martin Vingron, and Knut Reinert. Optimal robust non-unique probe selection using integer linear programming. *Bioinformatics*, 20 (Suppl. 1):i186–i193, 2004.

[25] Giuseppe Lancia and Romeo Rizzi. The approximability of the string barcoding problem. *Algorithms for Molecular Biology*, 1, 2006.

[26] Elena Marchiori and Adri Steenbeek. An evolutionary algorithm for large scale set covering problems with application to airline crew scheduling. In *Real-World Applications of Evolutionary Computing, EvoWorkshops 2000: EvoIASP, EvoSCONDI, EvoTel, EvoSTIM, EvoROB, and EvoFlight*, pages 367–381. Springer, 2000.

[27] Sanjay Mehrotra. On the implementation of a primal-dual interior point method. *SIAM Journal on Optimization*, 2(4):575–601, 1992. doi: 10.1137/0802028.

[28] George L. Nemhauser and Laurence A. Wolsey. *Integer and Combinatorial Optimization.* Wiley-Interscience Series in Discrete Mathematics and Optimization. Wiley, 1999.

[29] National Center of Biotechnology Information. BLAST – basic local alignment search tool. 🌍 `http://www.ncbi.nlm.nih.gov/blast`. [Online; accessed 09/26/2007].

[30] National Center of Biotechnology Information. Fasta format description. 🌍 `http://www.ncbi.nlm.nih.gov/blast/fasta.shtml`. [Online; accessed 09/02/2007].

[31] National Center of Biotechnology Information. GenBank overview, 2006. 🌍 `http://www.ncbi.nlm.nih.gov/Genbank`. [Online; accessed 09/03/2007].

[32] Sam Rash and Dan Gusfield. String barcoding: Uncovering optimal virus signatures. In *RECOMB '02: Proceedings of the sixth annual international conference on Computational biology*, pages 254–261, New York, NY, USA, 2002. ACM Press.

[33] Mutsuo Saito. An application of finite field: Design and implementation of 128-bit instruction-based fast pseudorandom number generator. Master's thesis, Hiroshima University, February 2007.

[34] Kenneth Steiglitz and Christos H. Papadimitriou. *Combinatorial Optimization: Algorithms and Complexity.* Dover, 1998.

[35] V. Černý. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of Optimization Theory and Applications*, 45(1):41 – 51, 1985.

[36] Wikipedia. Singleton pattern — Wikipedia, the free encyclopedia, 2007. 🌍 `http://en.wikipedia.org/wiki/Singleton_pattern`. [Online; accessed 08/08/2007].

[37] Mutsunori Yagiura, Masahiro Kishida, and Toshihide Ibaraki. A 3-flip neighborhood local search for the set covering problem. *European Journal of Operational Research*, 2001.