Tel Aviv University
Raymond and Beverly Sackler
Faculty of Exact Sciences
School of Mathematical Sciences
Computer Science Department

Designing Multi-Route Synthesis in Combinatorial Chemistry – Complexity and Algorithms

by

Adi Akavia

The research work has been conducted
under the supervision of
Prof. Ron Shamir

Submitted as partial fulfillment of the requirements
towards the M.Sc. degree

September 2002

Abstract

In combinatorial chemistry, one generates a collection of chemical compounds (such as peptides) with desired properties, in order to test their response in disease-related scenarios, as candidate leads for drug development. The process of generating the collection can be represented by a synthesis graph. In this work we address several problems regarding the design of a synthesis graph for a given target set of compounds. First, we consider a few variations of the Min Node Exact Synthesis (MNES), in which given the target set S, a minimum synthesis graph producing S is sought. In particular, we investigate the (d, g)-Flexible Min Node Synthesis ((d, g)-FMNS), in which the goal is to find a g-approximation of a minimum synthesis graph producing S, and no more than d additional compounds. We show that the problem is hard for any $d \leq O(g)$ and $g \leq O(|S|^{\frac{1}{10}-\varepsilon})$, unless NP=ZPP. In particular, this implies that approximating MNES by factor g is hard for any $g = O(|S|^{\frac{1}{10}-\varepsilon})$, unless NP=ZPP.

Next, we concentrate on the Max Strings Synthesis problem (MSS), in which the goal is to find a synthesis graph whose width is constrained, which maximizes the number of produced target compounds, given a constraint on the number of beads. We show that this problem in NP-hard.

Our main focus is presenting two heuristics for solving MSS – one based on a continuous search using a gradient-descent algorithm, and the other based on discrete search. We test the heuristics on several data sets, explore their behavior, and show that they achieve good success rates.

Key words: combinatorial chemistry, hardness of approximation, NP hardness, heuristics.

Acknowledgements

I would like to thank my advisor Prof. Ron Shamir for accompanying me in my first steps in research and providing all that is needed to enable me to concentrate on research. Through his demand for accuracy my vague ideas were sharpened into precise mathematical statements, and with the aid of his brilliant writing skills I learned to communicate these ideas to others.

I would also like to thank Prof. Muli Safra for his photographer's eye that saw the beautiful picture emerging from the many details I brought before him. Working together I learned the meaning of devotion to research and how not to despair by the many obstacles on the way.

I also want to express my thanks to Dr. Hanoch Senderovitch. His vast knowledge and good will has made him a wonderful guide in the paths of Combinatorial Chemistry.

Last but not least, I want to thank Alon Lerner for our long hours near the computer in transforming ideas into actions.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter we first provide background on combinatorial chemistry and introduce the problem that motivated this study. We then provide formal definitions of the algorithmic problems we will focus on. We summarize the thesis results in the last section.

1.1 Combinatorial Chemistry

Drug development is a long and expensive process, hence methods with potential of shortening it are of utmost importance. *Combinatorial-chemistry* is such a method, developed during the last two decades [20, 18, 14, 16, 22, 10, 17, 12, 28, 8, 1, 25]. In the early stages of *drug discovery*, one focuses on *lead finding*. Lead compound is a compound (*e.g.,* a *peptide*, or a small molecule) with the desired biological activity. Its activity is usually insufficient for therapeutic purposes and it may have some additional limitations as a drug. In order to overcome these shortcomings the lead is optimized. Lead optimizations is the process by which lead's activities are optimized and other limitations are overcome. Traditionally, the leads were identified based on historical medicinal research; however, in combinatorial chemistry, one relies on a broad search over large *libraries* (collections of compounds),

by utilizing rapid synthesis methods. These libraries are designed to span a predefined *parent library*, which contains many potentially relevant compounds, but is too large to allow generation or screening of each compound. The produced libraries are normally utilized in high-throughput screening (HTS)[1], with the goal of discovering a lead. The combinatorial chemistry approach, when utilized with a good design procedure, produces libraries that are better (with respect to the diversity criterion discussed in the following paragraph) than the ones obtained by traditional approaches [26], thus enhancing the chances of finding a lead.

The basic approach for designing a compound library depends on the desired type of library. At the lead-finding stage, when there is very little knowledge on the appropriate drug, a library of high diversity is desired. Roughly speaking, each compound in the parent library has some properties, and a subset of high diversity represents a wide range of these properties, by containing highly dissimilar compounds [24]. At later stages, after active compounds were already identified and must be refined, a *focused library* – which contains many compounds similar to the active ones – is desired. In order to quantify these goals, the parent library is embedded in a *property space*. The property space is a multi-dimensional space, where each coordinate represents a molecular descriptor, and each molecule is represented by a point corresponding to the values of its descriptors. The geometric distance in the property space is assumed to measure functional similarity between compounds. This assumption is often referred to as the *Similarity Principle* [23]. The choice of good descriptors is crucial to the success of the design, hence it is a subject of much research (see for example, [6, 7, 5]). Examples of molecular descriptors are fragment-substructure descriptors, in which a molecule is checked for the presence of some fragments, and physical properties descriptors, such as geometric features, electronic charge and mass.

When approaching the design of compounds libraries, one must con-

[1] HTS deals with the rapid screening of a large number of compound against a biological target, in order to identify active compounds, which have therapeutic potential.

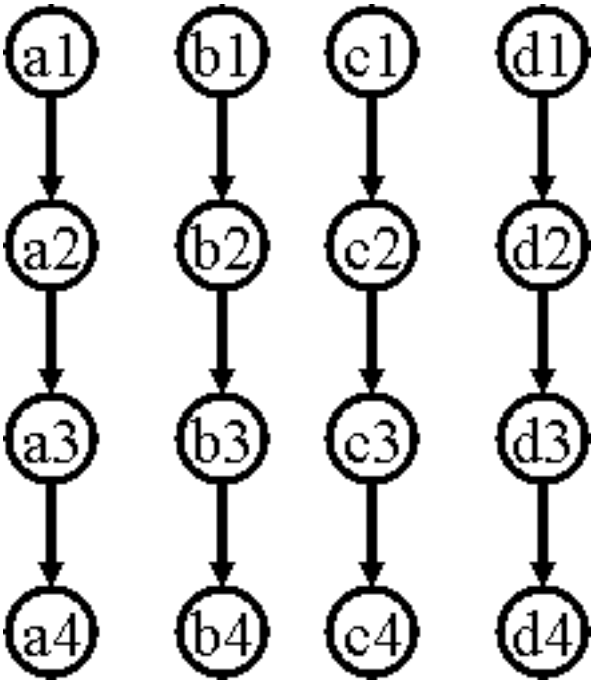

Figure 1.1: Parallel Synthesis. This process produces four strings $a_1a_2a_3a_4, ..., d_1d_2d_3d_4$. Building units (nodes labels) are added according to the order in each chain.

sider the compounds synthesis method at use, as the ability to produce the designed library is constrained by the used synthesis method. One possible synthesis method is the *parallel synthesis*. To describe it, let's assume for simplicity that each compound is a linear chain of building units (*e.g.,* amino-acids), so it can be viewed as a string over a finite alphabet. In a parallel synthesis each compound is grown in a unique vessel (colon) by repetitive addition of its building units (compare Figure 1.1). This method imposes no constraints on the set of compounds in the resulting library due to the fact that each compound is synthesized individually. However, it does impose a constraint on the size of the library, as the number of available synthesis vessels is limited, and the effort of constructing it is linear in the number of compounds.

Another synthesis method, proposed by [13, 21], is the *single-route combinatorial synthesis*, also referred to as split-synthesis or mix-and-split–synthesis. In the following we abbreviate and refer to this method as combinatorial synthesis. In this synthesis method, a large set of compounds is synthesized in a series of steps. Each step is composed of three stages (compare Figure 1.2):

- Split: dividing the mixture into different reaction vessels, one vessel for each desired reaction.
- Grow: in each vessel performing a simple chemical process on all the compounds at once – adding a single building unit to the end of every compound (*e.g.,* peptide) in the mixture.
- Mix: mixing all compounds, produced in the previous steps.

The process proceeds by repeating this step, with each step extending the length of all peptides (strings) by one. Technically, the steps are performed on miniature beads, to which the peptides are chemically attached. In this fashion a 'split' step is just dividing the beads in a tube to random subsets of equal size, which are then put in the tubes of the next layer. Note that libraries produced in this synthesis method are combinatorial libraries, that is, their compounds are composed of the combinations of the building units used in the different positions in the molecules. The size of such combinatorial libraries grows exponentially with the length of the peptides (*i.e.*, the number of building units in the peptide).

The *Multi-Route Synthesis*, proposed by [9], is a generalization of both the combinatorial and the parallel synthesis methods. This method, similarly to the combinatorial synthesis, is a process in which a large set of compounds is synthesized in a combination of mix, split and grow steps. However, the mix step here needs not be of *all* the previously produced compounds, but rather of any desired combination of the previous subsets (compare Figure 1.3). Note that a Multi-Route Synthesis is indeed a generalization of both the combinatorial and the parallel synthesis methods:

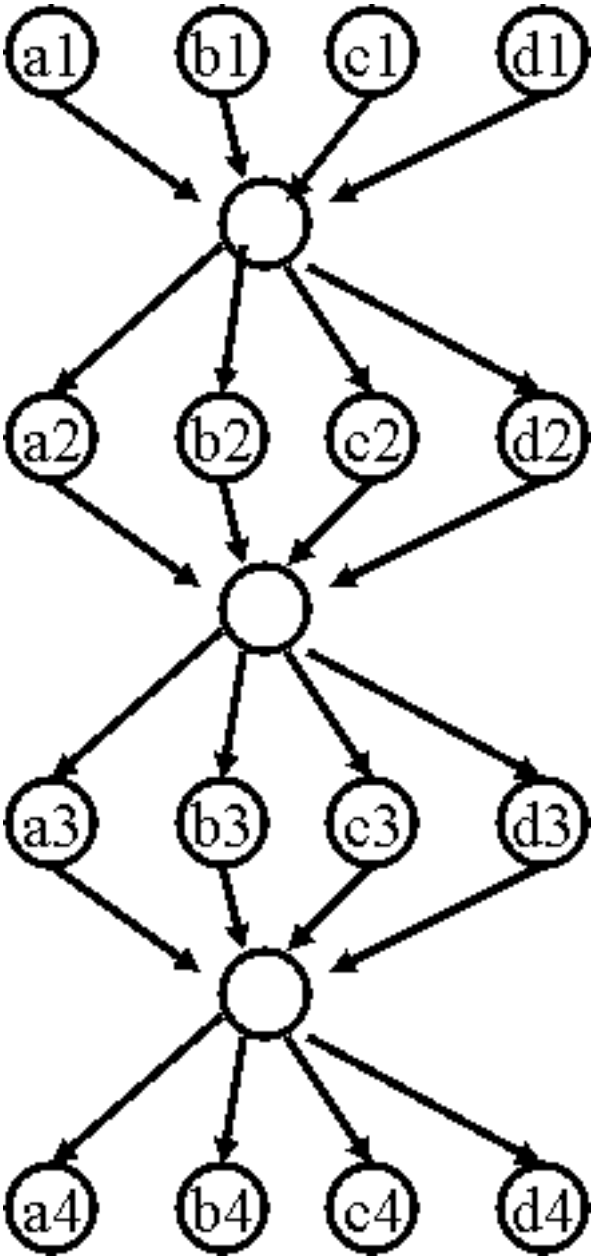

Figure 1.2: Combinatorial synthesis. The empty nodes are 'mix' steps. The nodes with letters in them are 'grow' steps. 'Split' steps are denoted by arrows emanating from a 'mix' node. This process produces 256 strings $a_1a_2a_3a_4, ..., d_1d_2d_3d_4$.

When choosing not to mix at all, it is parallel synthesis; and when choosing to mix *all* compounds previously produced, one gets combinatorial synthesis.

Far more diverse libraries can be produced in the Multi-Route Synthesis than by the traditional combinatorial-synthesis, due to the relaxation of the combinatorial constraint. Additionally, as mixing is allowed, this synthesis method can be used, employing limited resources, to produce far larger libraries than the parallel synthesis. However, unlike the parallel and combinatorial synthesis methods, in Multi-Route Synthesis, given a set of compounds, *it is unclear which is the best way to produce them*: There are many possible synthesis schemes, and many decision points where one can choose whether or not to mix previously produced compounds. Hence, with this synthesis method, when designing a library, one must also specify the synthesis procedure that produces that library.

A description of a Multi-Route Synthesis process may be given in a *synthesis graph*, as presented in [9]. A synthesis graph (see Figure 1.3) is a labelled, directed and acyclic graph. It is composed of layers, each describing the grow operations at one position in the target compounds. The nodes in the graph correspond to the 'grow' operations, and their labels indicate the appended unit. The arcs of the graph correspond to beads transfer between the different reaction vessels, that is, if a node v has in-coming arcs from nodes $u_1, ..., u_k$, then the mixing step is taking compounds from nodes $u_1, ..., u_k$ into node v. The strings obtained by concatenating the labels along paths from the first layer in the graph to the last are called the *theoretical compounds* of the graph. We refer to the theoretical compounds as the *language* of the synthesis graph. They are called *"theoretical"*, as performing the synthesis process described by the graph might not produce all those compounds, due to the random nature of the splitting process, and due to possible failures in the chemical reactions in the 'grow' operations.

Cohen and Skiena [9] consider only unweighted synthesis graphs, we also consider the weighted case, as outlined below: In the Weighted Multi-Route Synthesis, non-uniform distribution of the beads from one node (*i.e.*,

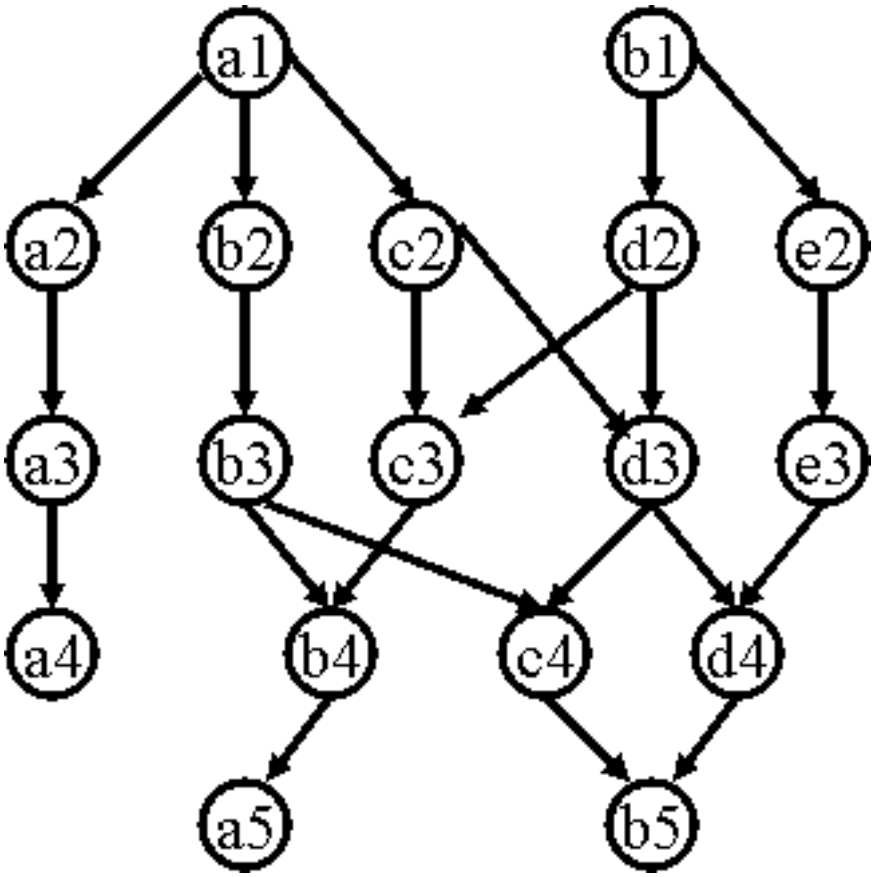

Figure 1.3: Multi-Route Synthesis. The produced strings are $a_1a_2a_3a_4$, $a_1b_2b_3b_4a_5$, $a_1b_2b_3c_4b_5$, $a_1c_2c_3b_4a_5$, $a_1c_2d_3c_4b_5$, $a_1c_2d_3d_4b_5$, $b_1d_2d_3c_4b_5$, $b_1d_2d_3d_4b_5$, $b_1e_2e_3d_4b_5$.

reaction vessel) to its descendants is allowed. The Weighted Multi-Route Synthesis is modelled by a weighted synthesis graph. A weighted synthesis graph is a synthesis graph with non-negative arcs weights (compare Figure 1.4). The weight of an arc (u, v) represents the fraction of substance (beads) from node u that is transferred from u to v.

1.2 Definitions

In this section we give formal definitions and a discussion of the problems that we shall study.

1.2.1 Synthesis Graph

Definition 1.1 *A **Synthesis Graph** (See Figure 1.3) is a layered, vertex-labelled directed acyclic graph $G = (V, E)$. The first layer of G, denoted by $First(G)$, consists of all the nodes with in-degree 0. The last layer of G, denoted by $Last(G)$, consists of all the nodes with out-degree 0. All arcs are directed from one layer to the next one. $label(v)$ denotes the label of node v. We denote by $|G|$ the number of internal nodes in G, that is, $|G| = |V \setminus (First(G) \cup Last(G))|$*

Let the *valid paths* of the graph, denoted by $P(G)$, be the set of all paths in G initiating with a node in $First(G)$ and ending with a node in $Last(G)$. That is,

$$P(G) = \{ v_1, ..., v_k \mid \forall i \ (v_i, v_{i+1}) \in E, \ v_1 \in First(G), \ v_k \in Last(G) \} .$$

We assume that all the paths in the synthesis graph are of the same length. Nevertheless, in Section 5.2 we show how string sets of variable lengths may be produced in spite of this assumption.

Note that in the combinatorial synthesis at each step all the compounds from the previous step are mixed together, and hence by using empty nodes which represent 'mix' steps, the synthesis process can be represented with

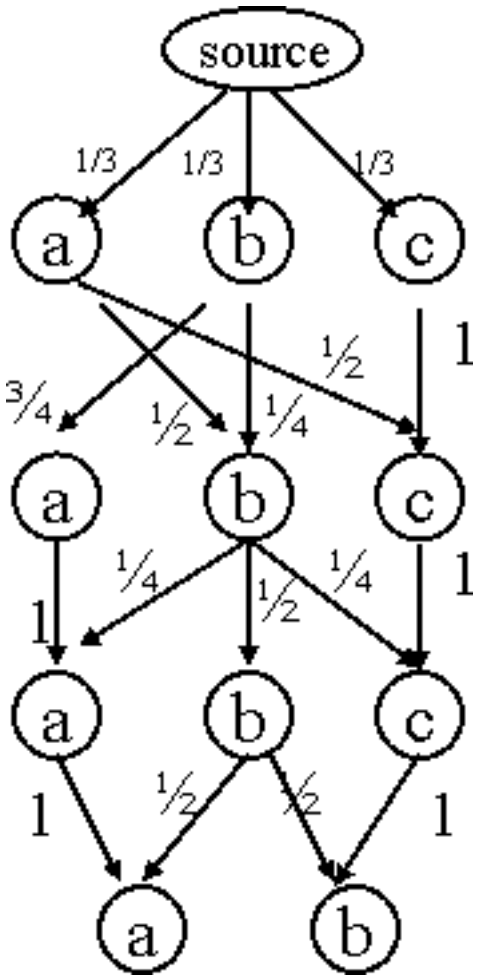

Figure 1.4: Weighted Multi-Route Synthesis. Arcs from 'source' represent the distribution of beads to the first synthesis step.

fewer arcs (see Figure 1.2). Thus arises the problem of finding the *minimum representation* of a synthesis graph. That is, given a synthesis graph G, find another graph G', in which a layer of empty nodes is inserted between each two layers in G, s.t. the same sets of strings are produced by both G and G', and the number of empty nodes in G' is minimum. This problem is NP-hard (see Remark **??** in Chapter **??**). Further study of this problem is left for future research.

Definition 1.2 *The* language *of a synthesis graph G, denoted by $L(G)$, contains all the strings which are a concatenation of node labels in G along a path in $P(G)$. That is, $L(G) = \{\sigma_1 ... \sigma_k \mid \exists p = v_1, ..., v_k \in P(G)\ s.t.\ \forall i\ label(v_i) = \sigma_i\}$*
We say that G produces *the language L, if $L = L(G)$.*

1.2.2 Weighted Synthesis Graph

Definition 1.3 *A **Weighted Synthesis Graph** is a synthesis graph with a weight function $w : E \to \mathbb{R}^+$ assigning a weight to each arc s.t. the sum of weights over all arcs outgoing from each node equals 1. The weighted synthesis graph has one additional node – the* source *node, denoted by 's', which does not represent a grow step. This node is connected by arcs into all the nodes in the first layer. The weights on the outgoing arcs from node v represent the fraction of beads from node v that are used for the subsequent synthesis step.*

Let the *valid paths* of the graph, denoted by $P(G)$, be the set of all paths in G initiating in the source and ending with a node in $Last(G)$. That is,

$$P(G) = \{\, s, v_1, ..., v_k \mid \forall i\ (v_i, v_{i+1}) \in E,\ (s, v_1) \in E,\ v_k \in Last(G)\}\,.$$

Note that this definition coincide with the definition given for the non-weighted version of the synthesis graph, as in a weighted synthesis graph we may assume w.l.o.g. that the only node with in-degree 0 is the source node, and hence $First(G) = \{s\}$.

Let $\sigma = \sigma_1 ... \sigma_k$ be a string, and $p = s, v_1, ..., v_k$ a path. We say that *the path p corresponds to* σ, if $p \in P(G)$ and $\forall i$ $label(v_i) = \sigma_i$. The *weight of p* is the product of the arcs weights along its arcs, *i.e.*, $weight(p) = w(s, v_1) \cdot \prod_{i=1,...,k-1} w(v_i, v_{i+1})$. Let P_σ be the set of all paths corresponding to σ, then the *weight of* σ is the sum of weights of all paths in P_σ, that is, $weight(\sigma) = \sum_{p \in P_\sigma} weight(p)$. Note, that if σ is not a concatenation of node labels in G along a path from the source node to a node in $Last(G)$, then there is no path corresponding with σ, and hence $weight(\sigma) = 0$.

The *weighted language* of a weighted synthesis graph G, denoted by $WL(G)$, is a set of pairs each composed of a string and its weight: $WL(G) = \{ \langle \sigma, weight(\sigma) \rangle \mid \exists p \in P(G),\ s.t.\ \sigma\ corresponds\ to\ p \}$.

If σ has weight $weight(\sigma)$ and b beads are used, then $weight(\sigma) \cdot b$ is the expected number of beads synthesized with word σ. Hence, we say that a weighted word $\langle \sigma, weight(\sigma) \rangle$ is *produced* by a synthesis process following a weighted synthesis graph G and using b beads, if $weight(\sigma) \cdot b \geq 1$.

Definition 1.4 *The* language $L(G, b)$ *of the synthesis process, defined by the weighted synthesis graph* G *and* b *beads, is the set of all strings* σ *s.t.* $weight(\sigma) \cdot b \geq 1$. *That is,*

$$L(G, b) = \{ \sigma \mid \langle \sigma, weight(\sigma) \rangle \in WL(G),\ weight(\sigma) \cdot b \geq 1 \}.$$

In order to simplify the description of our algorithms, we define a few terms. A *valid string* is string corresponding to some path $p \in P(G)$ (it is not necessarily produced by the graph, as its weight may be too small); a *good path* is a path $p \in P(G)$ corresponding to some target string; and a *bad path* is a path $p \in P(G)$ that does not correspond any target string.

1.2.3 Relevant Parameters

Following Cohen and Skiena, we focus on the problem of finding a synthesis scheme for a given library. However, when considering the real-world version of the problem, there are many different possible formulations to it. These

formulations depend on the choice of parameters to be constrained and those to be optimized. Generally speaking, the relevant parameters we focus on are:

1. $|V|$ - the number of nodes in the graph (which corresponds to the number of 'grow'-steps). This number can be estimated by the width of the graph (which corresponds to the number of parallel 'grow'-steps), and the depth of the graph which is determined by the length of the strings in S.

2. $|L(G,b)|$ (or $|L(G)|$ in the non-weighted version) - the size of the language produced by the graph,

3. $|P(G)|$ - the number of paths in the graph (which roughly corresponds to the number of needed beads in the synthesis process),

4. $|L(G,b) \cap S|$ (or $|L(G) \cap S|$ in the non-weighted version) - the number of *target strings*, *i.e.,* strings from the input target set, which are produced by the graph.

Note that both $P(G)$ and $L(G,b)$ are devised to capture the constraint of the number of beads used in the synthesis process. Each string must be generated on at least one bead, hence the number of paths $|P(G)|$ is closely related to the number of beads. However, the beads are not necessarily evenly distributed among all paths; thus, even when the number of paths does not exceed the number of beads, strings corresponding to some of the paths might not be produced. That is, the number of paths does not capture the constraint on the number of beads well enough. Consequently, we define $L(G,b)$, that directly addresses the language produced by the graph G while using b beads.

In the above list of parameters, the first three parameters are to be minimized, while the fourth should be maximized.

We have therefore four parameters, and in any optimization problem some may be bounded (or set to a fixed value, or penalized) and one (or

a function over a few of them) optimized. These variants give different problems, which might vary greatly in their complexity. In the following we briefly consider examples of such variations.

First, let us consider two such variations which yield easy-to-solve problems. If we must produce all target strings, while there is no penalty on the number of nodes, and the number of paths is to be minimized, the solution is immediate: use parallel synthesis to produce the target strings. If producing all target strings is required, there is no penalty on the number of paths, and the number of nodes is to be minimized, the solution is again obvious – combinatorial synthesis.

Second, let us consider two variations which yield hard problems. One example of such a variation is the Min Node Exact Synthesis, in which the requirements are to produce all target strings, while the size of the produced language is limited, and the goal is to minimize the number of nodes (see Section 1.2.4). Cohen and Skiena [9] show that this problem is NP-hard. Another example, is the Max Strings Synthesis, in which the goal is to maximize the number of produced target strings, while the number of nodes and paths is limited (see Section 1.2.6). In Chapter 2 we show this problem is NP-hard.

1.2.4 Min Node Synthesis

In the Min Node Exact Synthesis, one has to find a synthesis graph for producing a given set of compounds while minimizing the number of needed 'grow' operations, *i.e.*, the number of nodes in the synthesis graph. Formally, the problem is as follows:

Definition 1.5 ***Min Node Exact Synthesis (MNES).*** *Given a set S of target strings, find a synthesis graph G s.t. $L(G) = S$ and $|G|$ is minimum. Recall that $|G|$ is the number of* internal *nodes in G, i.e., the nodes which are not in $First(G)$ or $Last(G)$.*

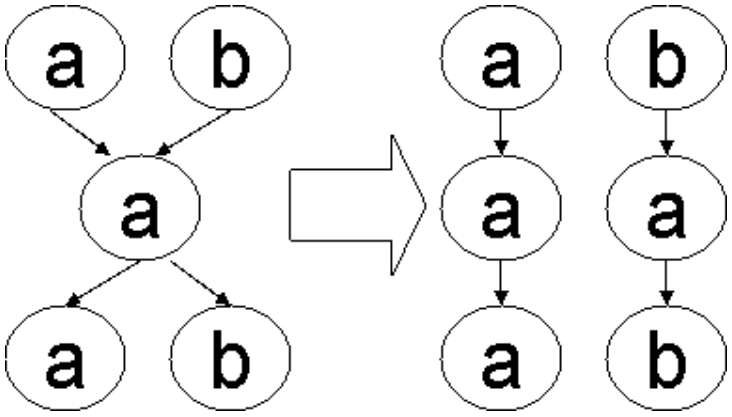

Figure 1.5: Label duplications in internal nodes. Duplicating labels in the first or last layer will not reduce the number of paths in the graph, while duplicating labels in internal layers can possibly reduce the number of paths in the graph (from 4 paths to 2 in this example).

A very similar problem is addressed in [9] (we give here a different name to the problem, for reasons that will become clear soon). While we want to minimize the number of *internal nodes* in G, Cohen and Skiena minimize the *total* number of nodes. Notice, that the first layer of G is trivially defined in a minimum solution: it must contain at least one node for each letter appearing in the first position of the strings in S, and moreover, no duplication of labels (*i.e*, different nodes with the same label) is necessary (see Figure 1.5). The same holds for the last layer. Hence, the natural optimization goal is the number of internal nodes. Of course, this difference is of no importance when discussing optimal solutions or the decision version of the problem. However, it is crucial when addressing approximations. Cohen and Skiena show that the problem is NP-hard.

Consider a synthesis graph producing a super-set S' of the target set S. From the point of view of combinatorial chemistry, such a synthesis graph is a good solution, as long as the super-set S' is of moderate size that may be reasonably generated and screened (in the applications we consider, S' is of reasonable size if $|S' \setminus S| = c \cdot |S|$ for some small constant c). Moreover, a

minimum synthesis graph producing S' might be considerably smaller than a minimum synthesis graph producing S.

The above two observations lead to an extension of MNES to the *Min Node Synthesis* problem, in which one seeks the minimum size synthesis graph out of all graphs producing a super-set of S of reasonable size. The formal definition follows.

Definition 1.6 ***Min Node Synthesis (MNS).*** *Given a set S of target strings, and an integer d, find a synthesis graph G s.t.*

1. $L(G) \supseteq S$,
2. $|L(G)| - |S| \leq d$,

and $|G|$, i.e., the number of internal *nodes in G, is minimum.*

From an algorithmic point of view this problem is very interesting. However, observing that when $d = 0$ this is simply MNES, we obtain that the NP-hardness of MNES implies that MNS is NP-hard as well. Nonetheless, note that for some d values a solution can be easily found. For example, let $d \geq |\Sigma|^l - |S|$, where $|\Sigma|$ is the size of the alphabet, and l is the length of the strings in S, then we may simply take the full combinatorial library, which is trivially a minimum solution.

Cohen and Skiena focus on algorithmic solutions to another variation of MNES, in which the number of paths ($|P(G)|$) is limited (and not the size of the produced language). That is, given a set S and an integer d, find a synthesis graph G s.t. $L(G) = S$, $|P(G)| - |S| \leq d$ and $|G|$ is minimum. They present two heuristics for solving it. Their algorithms start with an initial graph and by repetitive actions of splitting or merging paths they reach a graph producing a super-set of the given set of compounds, that adheres to the given constraint of the number of paths, and has a small number of nodes. The performance analysis of their heuristics is measured by comparison to the *trivial solution*, *i.e.*, the number of 'grow' steps in

a parallel synthesis producing the target set, and not with respect to the optimum solution, which is generally unknown.

The algorithms in [9] perform either only path merging or only path splitting. Similar algorithms, in which merging or splitting of paths are used to improve the current model, are popular in many problems, which can be formulated as a graph minimization problem where merging or splitting of paths are applicable. The *minimum Hidden Markov Model* is one such example. This problem can be formulated by a graph, in which the nodes represent *states* of the model, and the arcs represent stochastic *transitions* between states. The goal is to find a graph with a minimum number of states which gives a satisfying representation of the stochastic process it is designed to describe. Algorithms for solving this problem in which the operations of merging and splitting of paths are used are discussed in [27, 4]. An additional example is the *minimum work-flow model* problem in Industrial Engineering. In this problem, we are given a set of tasks, *i.e.*, sequences of actions and partial order constraints on these actions, and we are interested in the most efficient procedure for executing these tasks, *i.e.*, one with the fewest actions. (The same action may be executed once and used for different tasks). In a graph representation of this problem, actions are represented by nodes, partial order between actions is represented by arcs, and tasks are represented by paths. The problem is to find a directed graph with paths corresponding to the given set of tasks, and with minimum number of nodes. Algorithms for solving this problem, using merging and splitting of paths, are discussed in [19, 29].

1.2.5 Flexible Min Node Synthesis

In this section we consider a *relaxation* of Min Node Exact Synthesis, which we call *d-Flexible Min Node Synthesis (d-FMNS)*.

First, let us make a few observations regarding MNS. It is clear that MNS is at least as hard to solve as MNES: for any input S to MNES, a solution G to MNS on inputs S and $d = 0$ is also a solution to MNES.

Let us consider separately the two parts in the definition of MNS – the constraints on a solution G, and the optimization goal. Note that when only the constraints on a solution are considered (while the optimization goal is omitted), this is a relaxation of MNES, as any solution to MNES fulfill those constraints. However, when the optimization goal is considered as well, MNS is no longer easier than MNES: though a minimum solution to MNES adheres to the constraints of MNS, the size of the resulting graph might not be as small as a *minimum solution* to MNS.

In view of the above observations, we define the following relaxation of MNES.

Definition 1.7 ***d*-Flexible Min Node Synthesis (*d*-FMNS).** *Given a set S of target strings, find a synthesis graph G s.t.*

1. $L(G) \supseteq S$,
2. $|L(G)| - |S| \leq d$,

and $|G| = |G_{MNES}|$, where G_{MNES} is a minimum solution to MNES.

Note that, in d-FMNS the parameter d is not a part of the input, but rather an outside parameter. Due to this change we cannot apply the same argument used earlier for showing that MNS is at least as hard as MNES in case $d \neq 0$. Additionally, in d-FMNS – though the constraints on a solution graph are the same as the constraints in MNS – the optimization goal is as in MNES, that is, a solution to d-FMNS is a graph of the same size as the optimum of MNES. This problem is indeed a relaxation of MNES as any solution to MNES is also a solution to d-FMNS.

When an optimum to MNES is sought, d-FMNS is not an interesting relaxation. Assume we only seek the *size* (*i.e.*, the number of internal nodes) of an optimal solution and not the synthesis graph itself. In this case, a solution to d-FMNS is exactly the same as a solution to MNES (as the solution to both problems is $|G_{MNES}|$).

Nevertheless, when considering approximations and not exact solutions, this is indeed a relaxation as demonstrated in the following discussion. A *g-approximate solution* of d-FMNS is a synthesis graph G, which is (a) a feasible solution to d-FMNS, *i.e.*, $L(G) \supseteq S$ and $|L(G)| - |S| \leq d$, and (b) a g-approximation, that is, $|G| \leq g \cdot |G_{MNES}|$. We denote the problem of finding a g-approximation to d-FMNS by (d, g)-*Flexible Min Node Synthesis*. A formal definition follows.

Definition 1.8 (d, g)-***Flexible Min Node Synthesis*** *(*(d, g)-***FMNS)***. *Given a set S of target strings, find a synthesis graph G s.t.*

1. $L(G) \supseteq S$,
2. $|L(G)| - |S| \leq d$,

and $|G| \leq g \cdot |G_{MNES}|$.

Note, that any graph G, which is a feasible solution to FMNS, is also feasible solution to MNS, as in both problems a solution must adhere to the same constraints (that is, constraints (1) and (2)). Therefore, any solution G to (d, g)-FMNS is at least as large as the minimum solution to MNS, that is, $|G_{MNS}| \leq |G|$, where G_{MNS} is a minimum solution to MNS on inputs S and d. Combining this with the above definition, we obtain that for any a solution G to (d, g)-FMNS:

$$|G_{MNS}| \leq |G| \leq g \cdot |G_{MNES}| .$$

In contrast, let us consider the problem of finding a *g-approximation* of MNES. We denote this problem by g-**MNES**. A g-approximation of MNES is a synthesis graph G, s.t. $L(G) = S$ and $|G| \leq g \cdot |G_{MNES}|$. Combining those two requirements, we obtain that for any solution G to g-MNES

$$|G_{MNES}| \leq |G| \leq g \cdot |G_{MNES}| .$$

A solution G to either one of the above approximation problems ((d, g)-FMNS or g-MNES) is bounded from above by $g \cdot |G_{MNES}|$. However, the

lower bound is not the same for the two problems. A solution to (d, g)-FMNS is bounded from below by $|G_{MNS}|$; whereas a solution to g-MNES is bounded from below by $|G_{MNES}|$. Note that $|G_{MNS}|$ may be considerably smaller than $|G_{MNES}|$. Therefore, a solution to g-MNES is also a solution to (d, g)-FMNS, but the converse does not necessarily hold. Therefore, (d, g)-FMNS is a relaxation of g-MNES as we intended.

In contrast, a g-approximation of MNS on inputs S and d, is a synthesis graph G, s.t. $L(G) \supseteq S$, $|L(G)| - |S| \leq d$, and $|G_{MNS}| \leq |G| \leq g \cdot |G_{MNS}|$. Note that $|G_{MNS}|$ may be considerably smaller than $|G_{MNES}|$, hence a g-approximate solution MNES might be larger than $g \cdot |G_{MNS}|$, *i.e.*, it is not necessarily a g-approximate solution to MNS.

We investigate the complexity of approximating d-FMNS for different values for d. It is clear that when d is very large this problem is in P. For example, when d is large enough so that a minimum solution to MNS can be polynomially found, a minimum solution to d-FMNS can also be polynomially found (as a minimum solution to MNS on inputs S and d is clearly also a minimum solution to d-FMNS on the input S). Additionally, when $d = 0$, d-FMNS, is equivalent to MNES, hence the hardness of approximation results we show for d-FMNS, imply the hardness of approximation of MNES.

1.2.6 Max Strings Synthesis

In many real-life situations the laboratory constraints on the number of available reaction vessels (corresponding to the number of nodes in any single layer) and the number of used beads (related to the number of paths and to $L(G, b)$) are quite rigid. On the other hand, the target set of strings is heuristically designed, so producing all of them is not critical. Moreover, avoiding some target strings can reduce the number of paths and the number of nodes sharply. Hence, we define the *Max Strings Synthesis problem*, in which the goal is to produce as many target strings as possible within the above constraints (see definition 1.9).

Let w be the number of available reaction vessels, and b the number of used beads. Bounding w and b imposes bounds on other parameters, as follows. Let l be the length of the strings in S, then the number of nodes in G is at most $l \cdot w$. Additionally, a string σ is produced if $weight(\sigma) \geq \frac{1}{b}$, hence bounding b imposes a constraint on the produced language.

Note that a synthesis graph (or a weighted synthesis graph) represents a stochastic process, as each 'split' step randomly divides the set of beads into subsets. Maximizing $|L(G,b) \cap S|$ corresponds to maximizing the *expected* number of produced target strings. When applying the synthesis scheme represented by the synthesis graph, one can ensure that the actual number of produced strings is close to the expectation by taking redundant beads. The needed amount of redundancy is explored in [30].

In the applications we address, S is a set of small molecules or peptides – their length does not exceed ten building units, and is usually a lot smaller, say, five building units. Therefore, the length of the strings in S can be regarded as a constant. Nevertheless, whenever this assumption is used to simplify our algorithm we also indicate how the algorithm can be extended to work without this assumption. Additionally, we assume that w is given in unary, that is, the graph width is polynomial in the input. This technical assumption (used in Section 3.5.4) virtually always holds, as the length of the target set is $|S| \cdot \ell \cdot \log(|\Sigma|)$ and in practice $|S| >> w$.

Definition 1.9 ***Max Strings Synthesis (MSS)***. *Given a set S of target strings and two integers $b \in Z^+$ and $w \in Z^+$, find a weighted synthesis graph G of width at most w, that maximizes $|L(G,b) \cap S|$, i.e., the number of produced target strings.*

Note, that assuming only one 'grow' step can be done in a reaction vessel, w is actually the number of vessels that can be used in parallel.

1.3 Summary of Thesis Results

We show that (d, g)-Flexible Min Node Synthesis, when $d \leq O(g)$, is hard for any $g \leq O(|S|^{\frac{1}{10}-\varepsilon})$, unless NP=ZPP. Consequently, Min Node Exact Synthesis is hard to approximate by factor $g \leq O(|S|^{\frac{1}{10}-\varepsilon})$, unless NP=ZPP.

We focus mainly on Max Strings Synthesis. We begin by showing that Max Strings Synthesis is NP-hard, and proceed by suggesting heuristics for solving it.

We explore two heuristic approaches for this problem – a continuous approach and a discrete one. In the continuous approach, we use a gradient-descent algorithm to optimize our score function. The score function is devised as a continuous approximation of the score of a synthesis graph, which is the number of target strings produced by it. In the discrete approach, we search for an optimal synthesis graph by starting with some initial synthesis graph, and using repetitive discrete actions of deleting arcs from the graph with the aim of improving it.

We implemented both algorithms, and performed extensive experiments to evaluate their performances and time requirements. The experiments were done over synthetic and real data sets. The real data we used are data sets of 96 strings and 1000 strings each.

The implementation of the discrete optimization is very efficient – its running time, on each of the data sets, is shorter than one minute. The implementation of the continuous optimization is much slower: although each optimization iteration is very efficient (taking less than a second), a very large number of optimization iterations is required for convergence. Nevertheless, it is still reasonably efficient – its running time is a few hours on the data sets we tested.

The discrete optimization gave better results than the continuous one. Using a graph width $w = 10$, and a number of beads $b = 10,000$, the discrete optimization produced on average 87% of the target strings on the small data sets (96 strings), and 36% on the large ones (1000 strings). In contrast, the

continuous optimization produced on average only 57.5% and 18.5%, respectively. The results of the continuous algorithm can be greatly enhanced by combining it with several types of arcs-deletion heuristics, which are motivated by the discrete approach. With these improvements the results of the discrete algorithm are matched.

Finally, we explore the behavior of the discrete algorithm when different values of the graph width and the number of beads are used. As expected, increasing either the width or the number of beads improves the results. In particular, when increasing the graph width or the number of beads by a factor of 10, the results obtained on the large data sets are about equal to the results on the small ones with the standard parameters. Note that the two types of real data sets differ in size by a factor of 10.

Chapter 2

Hardness of Flexible Min Node Synthesis

In this chapter we investigate the complexity of (d, g)-Flexible Min Node Synthesis.

First let us recall some standard graph theoretic definitions: Let $G = (P, Q, E)$ be a bipartite graph. A *biclique* C is a set of vertices $C \subseteq P \cup Q$, which induces a complete bipartite graph. We say that, a biclique C *covers* an arc (u, v), if $u, v \in C$.

A *biclique edge cover* of G is a collection $C_1, ..., C_k$ of bicliques, which covers *all* arcs of G (see Figure **??** for an example). Note that non-edges of G cannot be covered. Let us denote by $\mathcal{B}(G)$ the set of all biclique edge covers of G, and for every biclique edge cover $b \in \mathcal{B}(G)$ let us denote by $|b|$ the number of its bicliques.

Definition 2.1 ***Min Biclique Edge Cover (BEC).*** *Given a bipartite graph $G = (P, Q, E)$, find a minimum biclique edge cover of G, that is, find a biclique edge cover b^* of G s.t. $|b^*| = \min_{b \in \mathcal{B}(G)} |b|$*

Next, we define a relaxation of BEC approximation problem. In this

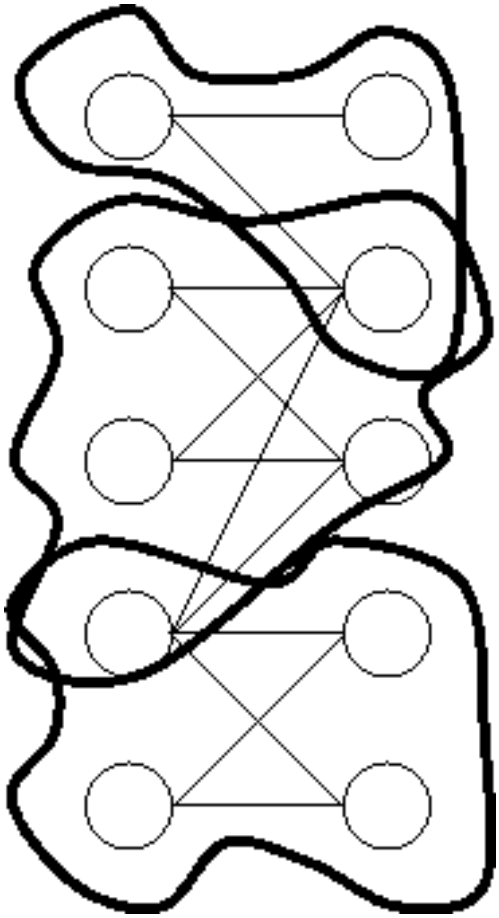

Figure 2.1: A biclique edge cover. The graph is covered by three bicliques – each marked set of nodes is a biclique, and they cover all the arcs (and no non-arcs)

relaxation, the cover may consist of "almost-bicliques", which are bicliques with up to a few missing arcs.

Definition 2.2 (d, g)***-Flexible Min Biclique Edge Cover problem (***(d, g)***-FBEC).*** *Given a bipartite graph $G = (P, Q, E)$, find a biclique edge cover b for a graph G', obtained from G by the addition of no more than d arcs, s.t. $|b| \leq g \cdot \min_{b \in \mathcal{B}(G)} |b|$.*

Similarly to our discussion regarding (d, g)-Flexible Min Node Synthesis (see Section 1.2.4), (d, g)-FBEC is a relaxation of the BEC approximation problem. Additionally, when $d = 0$, a solution to (d, g)-FBEC, is simply a g-approximation of BEC.

The next theorem [**?**] gives parameters for which (d, g)-FBEC is hard.

Theorem 2.1 ***(Akavia and Safra, 2002)*** *For every $\varepsilon > 0$, (d, g)-FBEC is hard for any $g \leq O(|P \cup Q|^{\frac{1}{5}-\varepsilon})$ and $d \leq O(g)$, unless NP=ZPP.*

We now prove our result regarding the (d, g)-FMNS.

Theorem 2.2 *For every $\varepsilon > 0$, (d, g)-Flexible Min Node Synthesis is hard for any $g \leq O(|S|^{\frac{1}{10}-\varepsilon})$ and $d \leq O(g)$, unless NP=ZPP. This holds even if all the strings in S are of length 3, and all have the same second letter.*

Proof. By a gap-preserving reduction from d-FBEC to d-FMNS. The reduction is outlined in Figure **??**. Let $G = (P, Q, E)$ be a bipartite graph, which is the input to d-FBEC. Define a language $S = \{pAq | (p, q) \in E\}$.

First, we assume a solution of size ℓ to d-FBEC, and show how to construct a solution of size ℓ to d-FMNS. Let $G' = (P, Q, E')$, $C_1, \ldots, C_\ell$ be a solution for the instance G. Using this solution, we may define a synthesis

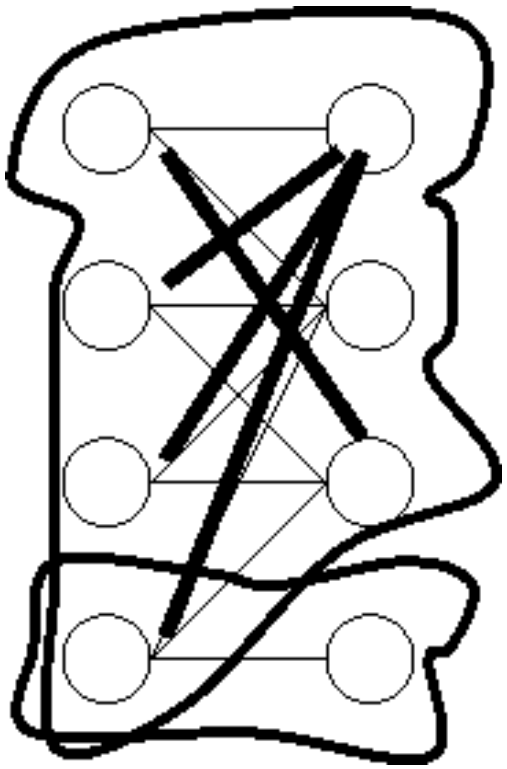

Figure 2.2: A flexible biclique edge cover. The graph is modified and then covered by two bicliques. The bold lines are additional arcs. Each marked set of nodes is a biclique, and together they cover all the arcs in the modified graph.

graph $H = (V_H, E_H)$ in the following way (compare Figure **??**).

$$V_H = P \cup Q \cup \{1, \ldots, \ell\},$$
$$E_H = \{(p, i) \mid p \in C_i\} \cup \{(i, q) \mid q \in C_i\}$$
and the labelling is:
$$\forall v \in V_H,\ label(v) = \begin{cases} v & \text{if } v \in P \cup Q \\ A & \text{if } v \in \{1, \ldots, \ell\} \end{cases}$$

The above construction has the following characteristics:

- $S \subseteq L(H)$: Let $pAq \in S$. By the definition of S it follows that $(p, q) \in E$. Since $\{C_i\}$ is a cover of E' and $E \subseteq E'$, there exists a biclique C_i such that $p, q \in C_i$. Hence, by the definition of H, $(p, i), (i, q) \in E_H$. This implies that $pAq \in L(H)$.

- $|L(H)| - |S| \leq d$: For every $pAq \in L(H)$, there exists i s.t. $(p, i), (i, q) \in E_H$. This implies, by the definition of H, that $\exists i,\ p, q \in C_i$. Hence, $(p, q) \in E'$ (as C_i is a biclique in G'). Moreover, note that every path in $P(H)$ is of the form p, i, q where $p \in P, i \in \{1, ..., \ell\}$ and $q \in Q$ (since $First(H) = P$, $Last(H) = Q$ and there are no arcs connecting P and Q). Therefore, every string in $L(H)$ is of the form pAq and $|L(H)| \leq |E'|$. Furthermore, by the construction of S, $|S| = |E|$. Hence $|L(H)| - |S| \leq |E'| - |E|$. However, $|E'| - |E| \leq d$, since G' is a solution to (d, g)-FBEC. Therefore $|L(H)| - |S| \leq d$.

- $|V_H| = |P| + |Q| + l$ (immediate from the construction).

Hence H is a solution of d-FMNS of size ℓ.

Second, we assume a solution of size ℓ to d-FMNS, and show how to construct a solution of size ℓ to d-FBEC. Let $H = (V_H, E_H)$ be a solution of size ℓ to the instance S of d-FMNS, *i.e.,* V_H contains ℓ internal nodes. Note, that w.l.o.g. we may assume that all internal nodes are labelled with A, as any node labelled differently contributes nothing to S, and may be deleted.

Let us denote by $A_1, \ldots, A_l$ those nodes labelled by A. Using H, we define a solution $G' = (P, Q, E')$, $C_1, ..., C_\ell$ to the instance G of d-FBEC, as follows (compare Figure **??**).

$$E' = \{\, (p, q) \mid pAq \in L(H)\}$$
$$C_i = \{\, p \mid (p, A_i) \in E_H\} \cup \{\, q \mid (A_i, q) \in E_H\} \;\; \forall i = 1, \ldots, \ell$$

The above construction has the following characteristics:

- $G' = (P, Q, E')$ is obtained from G by the addition of no more than d arcs: By the definition of E', $|E'| \leq |L(H)|$. By the definition of S, $|E| = |S|$. Combining those two facts, we obtain that $|E'| - |E| \leq |L(H)| - |S|$. However, H is a solution to d-FMNS, hence $|L(H)| - |S| \leq d$. Therefore $|E'| - |E| \leq d$. Additionally, $S \subseteq L(H)$, therefore $E \subseteq E'$.

- For each i, C_i is a biclique in G': By the definition of C_i, for every $p, q \in C_i$, $(p, A_i), (A_i, q) \in E_H$. Therefore $pAq \in L(H)$. Hence, by the definition of G', $(p, q) \in E'$.

- $\{C_i\}_{i=1,\ldots,\ell}$ is a biclique edge cover of G': For every edge $(p, q) \in E'$, $pAq \in L(H)$ (by the definition of G'). Hence, there exists an internal node A_i in H, s.t. $(p, A_i), (A_i, q) \in E_H$. Therefore, by the definition of C_i, $p, q \in C_i$.

Hence $\{C_i\}_{i=1,\ldots,\ell}$ is a solution of size ℓ to d-FBEC.

We saw that a solution of size ℓ to d-FBEC, implies a solution of size ℓ to d-FMNS, and vice versa. In particular, by considering $d = 0$, we may deduce that the size of a *minimum* solution to BEC and the size of a *minimum* solution to MNES are equal. Hence, $\{C_i\}_{i=1,\ldots,\ell}$ is a g-approximation of d-FBEC if and only if H is a g-approximation of d-FMNS.

Assume H approximates the minimum solution to d-FMNS within a factor $g \leq O(|S|^{\frac{1}{10}-\varepsilon})$. Then, $\{C_i\}_{i=1,\ldots,\ell}$ approximates the minimum solution to d-FBEC within the same factor g. Note, that $|S| \leq |P \times Q| \leq |P \cup Q|^2$;

hence, $g \leq O((|P \cup Q|^2)^{\frac{1}{10}-\varepsilon}) = O(|P \cup Q|^{\frac{1}{5}-\varepsilon})$. This is a contradiction of Theorem **??**. □

Remark 2.3 *Note that by replacing nodes labelled by $'A'$ with empty nodes, the above reduction gives a reduction from BEC to the problem of finding the minimum representation of a synthesis graph (see Section 1.2.1), thus showing that finding the minimum representation of a synthesis graph is NP-hard.*

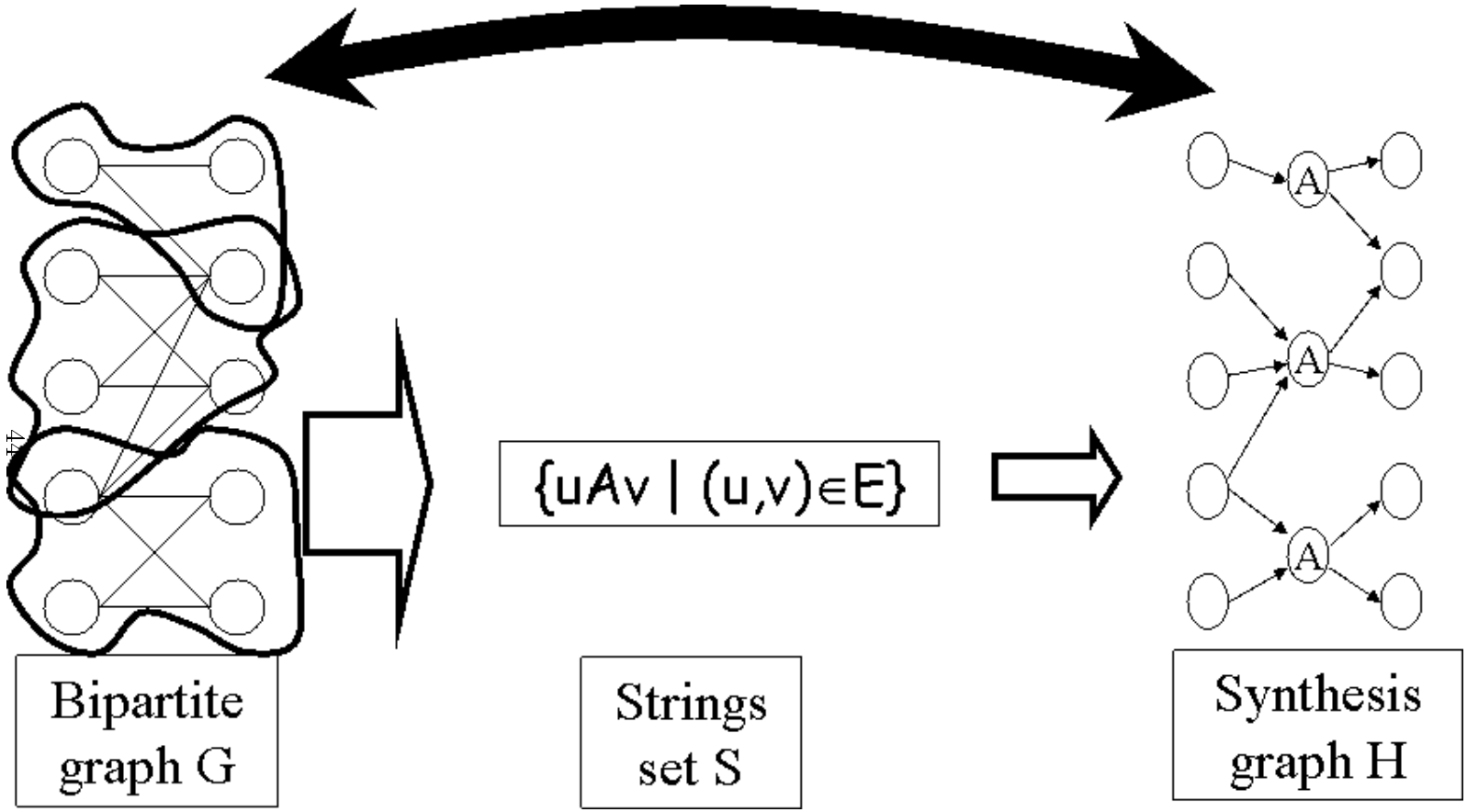

Figure 2.3: Outline of the reduction from d-FBEC to d-MNS

Chapter 3

Hardness of Max Strings Synthesis

In this chapter, we prove that Max Strings Synthesis is NP-hard. In the applications we address the considered molecules are short peptides or small molecules. Hence it is reasonable to assume that the strings in S are of length bounded by $O(1)$. We show that under this assumption the decision problem corresponding to Max Strings Synthesis is NP-complete. Of course, when the strings length is not bounded, the problem is at least as hard, therefore this assumption does not weaken our result. Moreover, this assumption is only used in showing that the problem in polynomially verifiable, that is, showing it is in the class NP.

First, let us define the decision version of the Max Strings Synthesis.

Definition 3.1 ***Max Strings Synthesis – decision version.*** *Given an input $\langle S, b, w \rangle$, and a threshold t, where*

- *S is a set of strings, over an alphabet $\Sigma_1, ..., \Sigma_l$, where l is the length of the strings in S.*
- *$b \in Z^+$ is a limit on the number of beads used in the process.*

- $w \in Z^+$ *is a limit on the width of the weighted synthesis graph.*

Does there exist a weighted synthesis graph G, of width at most w, such that $|L(G,b) \cap S| \geq t$?

Recall the definition of the NP-Complete problem – *Balanced-Biclique* (problem [GT24] in [15]):

Definition 3.2 ***Balanced-Biclique.*** *Given a bipartite graph $G = (V, U, E)$, and a threshold k, does there exist a* balanced k-biclique W, *i.e., a set of nodes $W = V' \cup U'$, such that $V' \subseteq V$, $U' \subseteq U$ and $|V'| = |U'| = k$, where W induces a biclique?*

Theorem 3.1 *Max Strings Synthesis is NP-hard.*

Proof. We show that the corresponding decision problem is NP-complete, even if $b = \infty$ and all strings in S are of length 2.

Clearly the problem is in NP, as given a synthesis graph it is easy to verify it is a valid solution (note that we assume that the length of the strings in S is bounded by a constant, thus the number of paths is polynomial in the width w). Hardness is proven by reduction from Balanced-Biclique:

Given an instance $G = (V, U, E), k$ of Balanced-Biclique, let us define an instance $\langle S, b, w \rangle, t$ of the decision version of Max Strings Synthesis, as follows: $\Sigma_1 = V,\ \Sigma_2 = U$ and

$$\begin{aligned} S &= \{\, vu \mid (v,u) \in E \} \\ w &= k \\ b &= \infty \\ t &= k^2 \end{aligned}$$

Note that since $b = \infty$ we may disregard the weights in the graph, as each path with non-zero weight is produced.

We first argue that we can assume that any synthesis graph H s.t. $L(H) \subseteq S$ is an induced subgraph of G. If H is not an induced subgraph of G, then the following procedure defines a graph H' s.t. $L(H') = L(H)$ and H' is an induced subgraph of G. Unite all the nodes in the same layer with identical labels into one node whose neighbors are the union of the neighbors of all the nodes it was composed of. Delete any node x in layer i s.t. $label(x) \notin \Sigma_i$. Delete any arc (v, u) with no corresponding arc in G,*i.e.*, with no arc $(v', u') \in E$ s.t. $label(v') = label(v)$ and $label(u') = label(u)$. This procedure does not increase the excessive strings (strings that are not in S) produced by the graph, nor does it change the number of target strings (strings in S) produced by the graph, and clearly the obtained graph is an induced subgraph of G.

Moreover, any induced subgraph of G with x arcs is a synthesis graph producing a subset S' of S such that $|S'| = x$ (immediate from the definition of S).

Now, we shall demonstrate that $\langle S, b, w\rangle, t$ is a "YES" instance if and only if G, k is a "YES" instance.

- Assume $\langle S, b, w\rangle, t$ is a "YES" instance, then there exists a synthesis graph H with width at most w (and $b \leq \infty$), producing at least t strings. But since $w = k$, the only way to obtain $t = k^2$ strings is by a biclique with k vertices at each side, s.t. each arc in the biclique corresponds to a string in S. However, as noted earlier, any such synthesis graph is a subgraph of G, and hence G has a balanced biclique with k nodes at each side.
- In contrast, assume $\langle S, b, w\rangle, t$ is a "NO" instance, then G cannot contain a balanced biclique with k nodes at each side: Such a biclique immediately translates to a synthesis graph with width at most $k(= w)$ producing $k^2(= t)$ strings.

□

Remark 3.2 *Note that – unlike the previous reduction (see Remark* **??***)*

– this reduction does not easily extend to allow empty nodes (see Section 1.2.1).

Chapter 4

A Continuous Approach to Solving Max Strings Synthesis

In this chapter we present a continuous approach to solving Max Strings Synthesis. We begin with a general overview of our algorithm, and proceed with a brief background on gradient-descent algorithms. Finally, we describe the specifics of the our algorithm: We give a detailed description of our objective function, and its partial derivatives, and provide a description of our algorithm.

4.1 An Overview of the Algorithm

The algorithm we present here is basically a greedy search algorithm, which searches the best weighted synthesis graph out of all graphs adhering the given constraints on the width and on the number of beads. The algorithm starts with an initial graph and by repetitive actions improves the graph.

The modification may be done either by 'continuous actions', or by 'discrete actions'. In continuous actions the arcs weights are modified following

a gradient-descent algorithm. In discrete actions some arcs are deleted, that is, assigned weight zero. The deleted arcs are chosen according to a scoring function as specified in Section 3.8.

The search is guided by the problem's objective function, *i.e.*, the number of target strings produced by the graph. However, this function is discrete, hence for the continuous actions we use a continuous approximation of this function.

The search begins with a graph of maximum width. Searching for the best graph only over graphs of maximum width suffices, since there exists an optimal solution of maximum width graph (as we may simply have some isolated nodes, in case not the entire width is needed). The nodes labels are chosen in the initialization stage, and they are not modified during the search, in order to narrow the search space we explore.

A combination of continuous and discrete actions is designed with the purpose of escaping local optima reached by the gradient-descent, and trying to find a better optimum. Their combination is done as follows: continuous gradient-descent search is executed until a local optimum is reached, then a discrete action is applied, and a continuous search begins again. These iterations between gradient-descent and arcs deletion heuristic continue until a graph with a number of paths that is at most b (the number of beads) is reached. The motivation to this stopping criterion is the fact that once such a graph is reached, all the strings corresponding to its paths can be produced, assuming the arcs weights are properly assigned (see Section 3.9).

A general description of the algorithm and its stopping condition is as follows.

1. Initialize G, an initial synthesis graph.

2. Initialize `best`, the best synthesis graph found so far, to be G.

3. While the number of paths in G is at most b (*i.e.*, the number of beads) do

(a) Run the gradient-descent algorithm until one of the following conditions holds:

- a plateau is reached, that is, there is no improvement in the continuous score over a pre-given number of iterations,
- the number of iterations exceeds a pre-given limit.

(b) If the current local optimum is better than `best`, update `best` to be the current graph.

(c) Escape local maximum: delete x arcs, where x is a parameter given to the procedure.

4. Recalculate the arcs weights so that all the paths would be produced by the graph, that is, the weight of every path $p \in P(G)$ is at least 1.

5. If the obtained graph is better than `best`, update `best` to be the current graph.

6. Return `best`.

We ran several variations of our algorithm: one in which only the continuous actions are executed, and three variations in which a combination of continuous and discrete actions are executed. The discrete actions differ in the scoring function they follow for the choice of the arcs to be deleted. The results of those experiments are presented in Chapter 6.

In the following sections we describe the algorithm in more details.

4.2 Background – Gradient-Descent

Consider an optimization problem, in which one wants to minimize a function $f(x)$. In gradient methods of optimization, the process is initiated with some (possibly arbitrary) point x_0, and then in iteration i x_i is altered to x_{i+1} according to the gradient of f in the current point. In the Steepest Descent method (*e.g.*, [3]), the choice of direction towards which x is altered is

done according to where f decreases most sharply, which is in the direction of $\nabla f(x_i)$. Precisely, we get the following iterative definition:

$$x_{k+1} = x_k + \lambda_k \nabla f(x_k),$$

where λ_k is the *learning rate*, that is, the size of the step in changing x_k.

The learning rate is usually taken as to decrease over time, in order to avoid skipping over an optimum due to too large a step (compare Figure **??**). Guaranteeing convergence to a local minimum (if one exists) can be done by setting the learning rate s.t. $\lambda_k \to 0$ and $\sum_{k=0}^{\infty} \lambda_k = \infty$ [2]. For example, $\lambda_k = \frac{1}{k}$ fulfills those constraints. Nevertheless, the convergence rate tends to be very slow. This drawback can be somewhat overcome by initiating the search at proximity to a minimum. In practice λ_k is often either set to be a constant, or decreases exponentially over time (in order to accelerate the convergence).

The Steepest Descent method is intuitive, simple, easy to apply, and each iteration is fast.

4.3 Objective Function

The objective function in the Max Strings Synthesis is the expected number of produced target strings, that is, $|L(G,b) \cap S|$. In the following we refer to this function as the *discrete score*. In order to calculate this score efficiently, we define auxiliary variables, denoted by $q(v,\sigma)$. $q(v,\sigma)$ indicates the expected number of beads entering the node v which holds a prefix of the string σ (q stands for quantity). The method of computing these variables is described later in this section. With these notations, the discrete score can be expressed as follows:

$$D(G) = \left| \left\{ \sigma \in S \mid \sum_{v \in Last(G)} q(v,\sigma) \geq 1 \right\} \right|.$$

Since we aim to use the gradient-descent algorithm for solving Max Strings Synthesis, we need a continuous approximation of the discrete score.

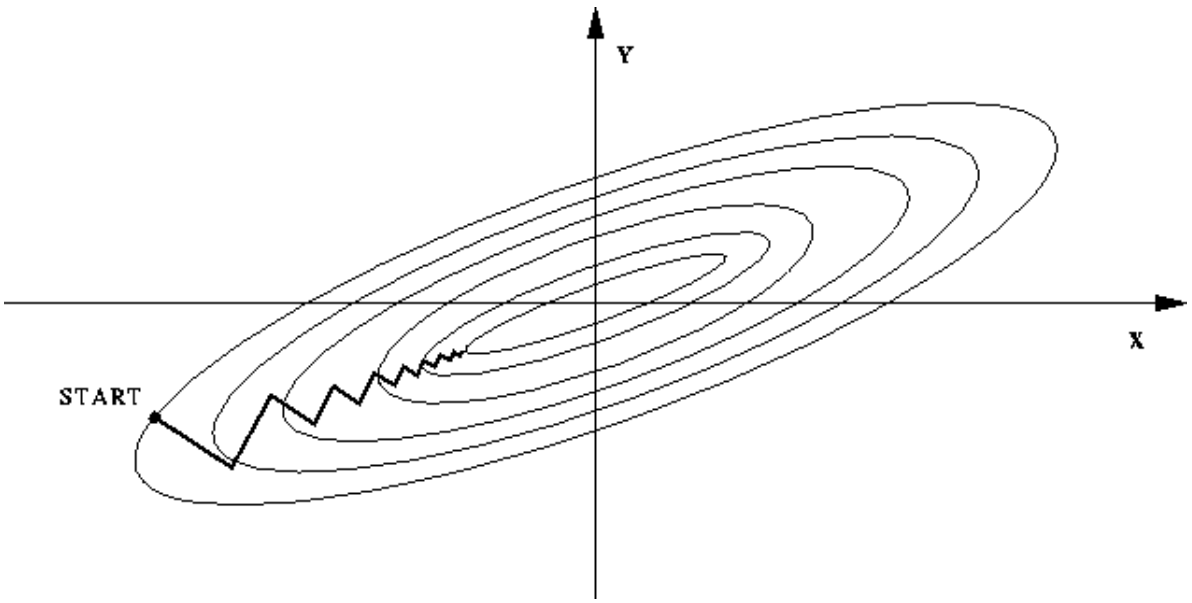

Figure 4.1: Convergence of Steepest Descent. The step size gets smaller and smaller, crossing and recrossing the valley (shown as contour lines), as it approaches the minimum. (source: www.gothamnights.com/trond/Thesis/node26.html)

Such an approximation can be obtained by changing the discrete step-function "jump" between $\sum q(v,\sigma) \geq 1$ and $\sum q(v,\sigma) < 1$, to a continuous approximation – the *logistic sigmoid* (*logsig* in short) function. That is,

$$\begin{aligned} C(G) &= \sum_{\sigma \in S} logsig(\sum_{v \in Last(G)} q(v,\sigma)),\ where \\ logsig(x) &= \frac{1}{1+\exp^{-\beta(x-\alpha)}} \end{aligned}$$

The logistic sigmoid approximates a step function between zero and one. The position of the step is determined by α, and its sharpness is determined by β (compare Figures 3.2). The step function we approximate jumps from zero to one at position $x = 1$, therefore we use $\alpha = 1$. Assigning larger values for β improves the obtained approximation; nevertheless, we use $\beta = 1$. The value for β was determined after extensive experiments with our algorithm, which indicates that the gradient-descent algorithm gives better results with small values for β.

Recall, that arcs weights in a synthesis graph are non-negative reals, which represent the fraction of beads transferred from one node to the next. When using the gradient-descent algorithm for finding the best weights to the arcs of the graph, the weights may either be increased or decreased. Unfortunately, when decreasing an arc's weight, it might become negative. For using the gradient-descent algorithm, we chose to ignore the arc weight non-negativity constraint and perform a filtering step after each iteration, in order to restore non-negativity. This is done by adjusting arc weights $w(u,v)$ using the *exponent normalization* function:

$$filter(w(u,v)) = \frac{\exp(T \cdot w(u,v))}{\sum_{\{u' \mid (u',v) \in E\}} \exp(T \cdot w(u',v))} \qquad (4.1)$$

This function, applied to any real values (positive or negative), returns values between 0 and 1 (compare Figure 3.3). Additionally, it ensures that the sum of arcs weights over all arcs originating from the same node equals 1. The parameter T is a positive real that was experimentally chosen to be 1 for arcs emanating from the source, and 5 for all other arcs. Note that for a

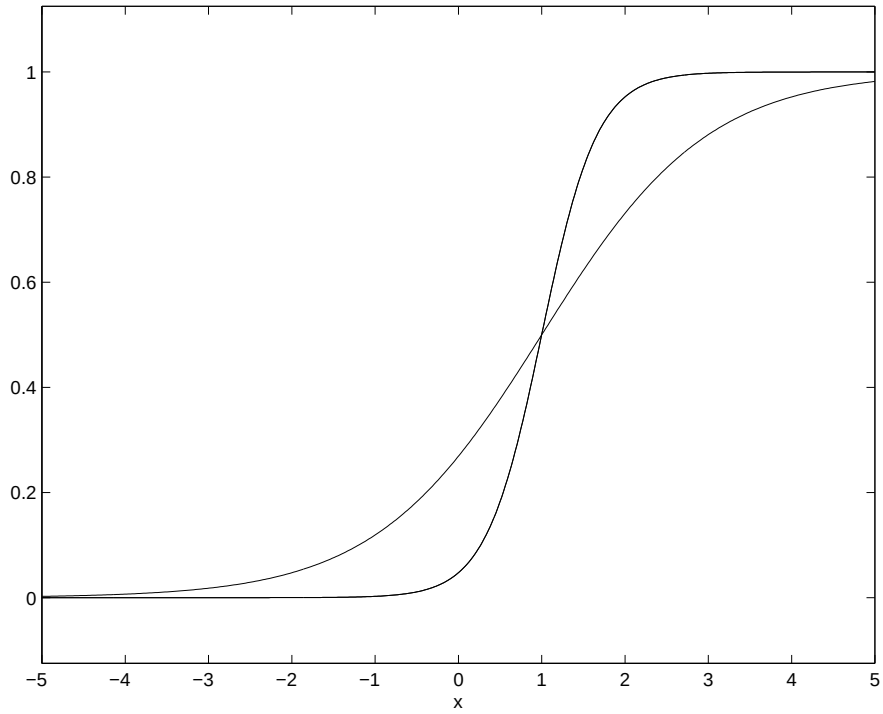

Figure 4.2: An overlay of two graphs of the Logistic sigmoid function, one with parameters $\alpha = 1$, $\beta = 1$ and the other with parameters $\alpha = 1$, $\beta = 3$. The values grow from 0 to 1, reaching $\frac{1}{2}$ at α. When $\beta = 3$ the gradient is sharper than when $\beta = 1$.

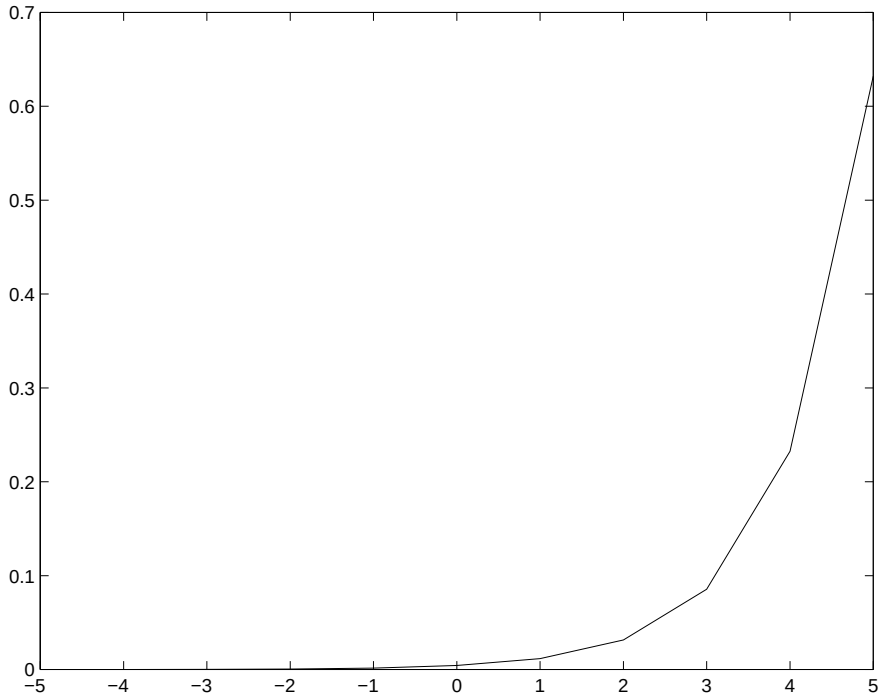

Figure 4.3: Exponent normalization with parameter $T = 1$. The x-axis represents the weights of 11 arcs emanating from the same node. The weights are -5,-4,...,5. The y-axis represents the value of the exponent normalization applied on each of the arcs.

fixed input weights distribution, the distribution of weights obtained as an output of this filter becomes less uniform as T becomes larger.

In order to complete the above definitions, we must give the formal definition of $q(v, \sigma)$. But first, let us define some notation. Recall that a synthesis graph is a layered graph. For each node v, let $layer(v)$ be the index of the layer containing v. For each layer index k, let V_k be the set of all nodes in this layer. Let l be the length of the strings in S.

$$q(v, \sigma = \sigma_1 ... \sigma_l) = \begin{cases} b & if\ v = s \\ \sum_{u \in V_{layer(v)-1}} filter(w(u,v)) \cdot q(u, \sigma) & \\ \qquad if\ v \in V_k,\ and\ label(v) = \sigma_k & \\ 0 & otherwise \end{cases} \tag{4.2}$$

(4.3)

4.4 Derivatives

In order to use the gradient-descent algorithm, we must calculate the partial derivative of the continuous score with respect to an arbitrary arc weight. In the following we present a recursive formula of those partial derivatives, and analyze the complexity of calculating them.

The partial derivative of the score is composed of a derivative of the logistic sigmoid ($logsig$) function and a partial derivative of $q(v, \sigma)$ for any string $\sigma \in S$:
Let $z = \sum_{v \in Last(G)} q(v, \sigma)$, then

$$\frac{\partial continuous\ score}{\partial w(x,y)} = \sum_{\sigma \in S} \frac{d\ logsig(z)}{dz} \cdot \sum_{v \in Last(G)} \frac{\partial q(v, \sigma)}{\partial w(x,y)}$$

The derivative of the $logsig$ function has a simple formula:

$$\frac{d\ logsig(z)}{dz} = \beta \cdot logsig(z) \cdot [1 - logsig(z)]$$

Calculating the partial derivatives of $q(v,\sigma)$ is more complicated. In the following we present a recursive formula of this calculation.

For each $v \in V_k$,

$$\frac{\partial q(v,\sigma=\sigma_1...\sigma_l)}{\partial w(x,y)} = \delta_{\sigma_k,label(v)} \cdot \sum_{j \in layer(y)} \frac{\partial q(v,\sigma)}{\partial filter(w(x,j))} \cdot \frac{\partial filter(w(x,j))}{\partial w(x,y)}$$

$$where\ \delta_{\sigma_k,label(v)} = \begin{cases} 1 & \sigma_k = label(v) \\ \\ 0 & otherwise \end{cases}$$

It remains to specify the partial derivative of $q(v,\sigma)$ according to $filter$, and of $filter$ according to the arc weight. First, we present the formula for calculating the partial derivative according to an arc emanating in the previous layer:

$$\forall v \in V_k,\ and\ \forall x \in V_{k-1}$$

$$\frac{\partial q(v,\sigma=\sigma_1...\sigma_l)}{\partial filter(w(x,j))} = \begin{cases} q(x,\sigma) & if\ label(v) = \sigma_k \\ \\ 0 & otherwise \end{cases}$$

Second, we present a recursive formula for calculating the partial derivative

according to an arc not emanating from the previous layer:

$$\forall v \in V_k, \ and \ \forall x \notin V_{k-1}$$

$$\begin{aligned}
\frac{\partial q(v, \sigma = \sigma_1 ... \sigma_l)}{\partial filter(w(x,j))} &= \delta_{\sigma_k, label(v)} \cdot \sum_{u \in V_{k-1}} [\frac{\partial q(u,\sigma)}{\partial filter(w(x,j))} \cdot filter(w(u,v)) \\
&\qquad + q(u,\sigma) \cdot \frac{\partial filter(w(u,v))}{\partial filter(w(x,j))}] \\
&= \begin{cases} \sum_{u \in V_{k-1}} filter(w(u,v)) \cdot \frac{\partial q(u,\sigma)}{\partial filter(w(x,j))} \\ \qquad\qquad if \ layer(x) < k-1 \ and \ label(v) = \sigma_k \\ 0 \qquad\qquad otherwise \end{cases} \\
\frac{\partial filter(w(x,j))}{\partial w(x,y)} &= \begin{cases} T \cdot filter(w(x,j)) \cdot [1 - filter(w(x,j))] \\ \qquad\qquad if \ j = y \\ -T \cdot filter(w(x,j)) \cdot filter(w(x,y)) \\ \qquad\qquad if \ j \neq y, \ layer(y) = layer(j) \end{cases}
\end{aligned}$$

For each string $\sigma \in S$, this calculation requires (in the worst case) going over all the arcs in the graph (due to its recursive nature). Thus the complexity of calculating one partial derivative is $O(|S| \cdot m)$, where m is the number of arcs in the graph. At first glance, this seems to imply complexity of $O(|S| \cdot m^2)$ for calculating *all* the partial derivatives. However, by using forward propagation the computation for all the partial derivatives can be preformed in $O(|S| \cdot m)$. By 'forward propagation' we mean that the value of each partial derivative used in the recursion is calculated once, instead of being recalculated for each partial derivative. This calculation is done in a bottom-up manner. Moreover, at each stage of the computation we only need to recall the derivatives according to one layer – the previous one. Therefore, the increase in the space complexity is only by factor $O(w)$, and not by factor $O(m)$ (recall, $m \leq (l-1)w^2$, where l is the length of the strings in S).

4.5 Initialization

As discussed in Section 3.2, the gradient-descent method is prone to slow convergence. This drawback can be partially overcome by choosing a good initialization point. We devised an initialization algorithm that considers frequencies of letters when labelling the nodes, and frequencies of transitions between letters when determining the initial arcs weights. We compared our initialization algorithm with random initialization, and obtained clear results indicating that the frequency-based initialization is much better than the random one.

Initializing the synthesis graph requires choosing the nodes labels and the arcs weights. First, let us concentrate on initializing the nodes labels.

4.5.1 Assigning Nodes Labels

In assigning labels, we first choose a layer in the graph, and then assign labels to all the nodes of this layer. We repeat the process until the nodes in all the layers are labelled.

Note that by labelling the nodes of some layer i, it may be the case that some of the strings in S cannot be produced by the graph. This happens if a string has a letter in position i, which is not the label of any node in this layer. We use an auxiliary variable called `considered-strs`, which holds the set of target strings that can be produced by some labelling of the nodes in the remaining unlabelled layers. `considered-strs` is updated after the labelling of each layer of the graph.

The order in which the layers are labelled is greedily chosen. The next layer to be labelled is one such that the labelling we assign it causes the elimination from `considered-strs` of as few strings as possible. That is, one whose labelling leaves the `considered-strs` as large as possible. Note that in the applications we address, the strings length is short, therefore this process is extremely fast.

The labelling of layer i is done according to the letters frequencies in the i^{th} position of the strings in the `considered-strs`: First, the nodes are divided equally between the relevant labels, that is, each label is assigned to $\lfloor \frac{w}{j} \rfloor$ nodes. The remaining $(w \ mod \ j)$ nodes are assigned the $(w \ mod \ j)$ most frequent labels. Thus, letters with high frequency are assigned to one more node than letters with low frequency.

Note that this initialization prefers variety in labels over clear preference to the frequent ones. This approach was chosen after examining the success of our algorithm on both approaches. Nevertheless, in Section 3.6 we show how the success of the algorithm can be further improved by preprocessing the set S of target strings, to obtain a subset $S' \subseteq S$, where – roughly speaking – strings with rare letters are eliminated.

This procedure is summarized as follows.

1. Let the `considered-strs` be the set of all the target strings.
2. Repeat until nodes labels of all layers are defined
   (a) For each layer, check how many strings remain in the `considered-strs` set, after labelling it (as described in 2b). Let i be a layer with the highest number of remaining strings.
   (b) Sort the letters appearing in the i^{th} position in descending order of their number of appearances in the `considered-strs` set. Let j be the number of different letters in position i of the `considered-strs`. Label $\lfloor \frac{w}{j} \rfloor$ nodes in layer i with each letter in position i in the `considered-strs` set, and add one more node labelled with each of the $(w \ mod \ j)$ most frequent letters.
   (c) Remove from the `considered-strs` set irrelevant strings, *i.e.,* strings that can no longer be produced by the graph.

The calculations for each layer are composed of sorting (which requires $O(|S|)$ time, if bucket sort is used, provided that the size of the alphabet is fixed), assigning labels (which requires $O(w)$ time) and updating the

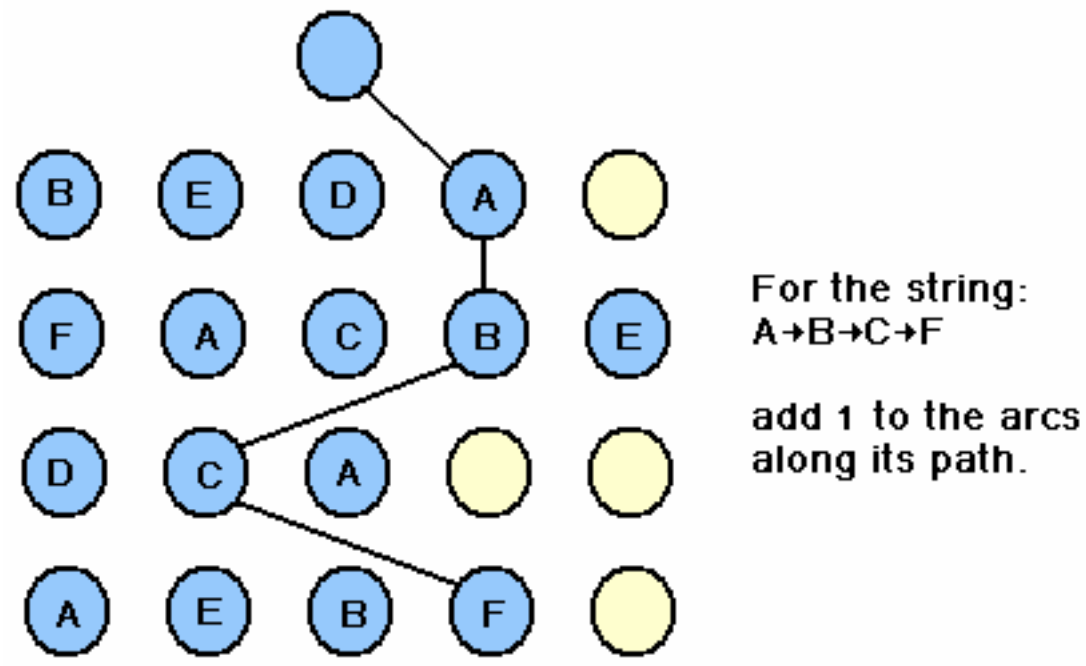

Figure 4.4: Initializing arcs weights in the gradient-descent algorithm

`considered-strs` set (which requires $O(|S|)$ time). Thus, the complexity of the entire procedure is $O(l^2 \cdot (|S| + w))$.

4.5.2 Assigning Arcs Weights

Once the nodes labels are set, we initialize the arcs weights as follows. For each pair of nodes v, u in two consecutive layers $i,\ i+1$, let the weight of the arc (v, u) between them be the number of strings out of the `considered-strs` that have letters $label(v), label(u)$ in positions $i,\ i+1$ respectively, *i.e.*, the number of strings that correspond to a path that may pass through this arc (compare Figure 3.4). Note that if a letter appears more than once in the same level we might get more than one path corresponding to a given string. Weights are then normalized so that the sum of weights outgoing from each node is one.

4.5.3 Assessing the Performance of Our Initialization Algorithm

In order to assess the performance of our initialization algorithm, we compared the results of the gradient-descent algorithm obtained with our initialization with the results obtained by assigning random arcs weights. For each of the first 10 small data sets of real data, we chose 100 random initializations of arcs weights, and ran the gradient-descent algorithm on these initial graphs, without the heuristic of escaping local maxima. The random initialization of arcs weights is done by first assigning a random integer in the range $0, ..., 99$ to each arc corresponding to some string in S, and then normalizing the weights so that the sum of weights outgoing from each node is one. In Table 3.1 we give a summary of the results of these experiments. The results show a clear advantage of our initialization. The poor results obtained by the random initialization may indicate that the potential space of our objective function has many local optima.

	average	std	max	min	our init
96 real data 1	39.59	1.59	43	35	57
96 real data 2	43.94	1.95	48	39	63
96 real data 3	42.15	1.48	47	39	59
96 real data 4	42.72	2.03	47	38	62
96 real data 5	39.83	1.65	44	37	57
96 real data 6	37.31	1.50	42	34	55
96 real data 7	40.67	1.7	45	35	59
96 real data 8	36.85	1.41	41	33	53
96 real data 9	42.8	1.87	46	38	63
96 real data 10	41.95	1.53	46	38	60

Table 4.1: Impact of the initialization scheme. Comparison of our initialization vs. random initialization on small sets of real data. Each row summarizes the average, standard deviation, maximum and minimum scores of 100 random initializations as well as the score obtained by our algorithm.

4.5.4 Complexity of the Obtained Graph

The width of the graph is polynomial in the input (recall, that we assumed that w is given in unary). Let $l+1$ be the length of the graph. The number of nodes is bounded by $l \cdot w + 1$, and the number of arcs is bounded by $(l-1) \cdot w^2 + l$. That is, they are both polynomial. The number of paths is bounded by w^{l-1}, which is polynomial under our assumption that the length of the graph is $O(1)$. However, in the following we describe how to change the initialization procedure, so that our algorithms would be polynomial even when l is not bounded by $O(1)$.

Our algorithm can be extended to handle target sets with strings of polynomial length as follows. The graph is partitioned into 'slices': Each slice has no more than one copy of each label in each layer; and the slices are disconnected, that is, there are no arcs connecting nodes in different slices. Each string corresponds to at most one path in each slice (as there is at most one copy of each label in each layer). Therefore, the number of good paths in each slice is at most $|S|$. (Recall that a path $p \in P(G)$ is good if it corresponds to a string $\sigma \in S$ – see Section 1.2.2). Additionally, the number of slices in no more than w. Therefore, the number of good paths in the obtained graph is polynomial. The good paths can be held in a data structure (say, a list) that allows accessing them without encountering bad paths. Since the size of the graph is polynomial, for each good path such a list can be constructed by simply going over all the slices in the graph and checking if all the arcs between the nodes corresponding to this path exist. Since the number of good paths is polynomial (at most $|S| \cdot w$), the entire list can be polynomially constructed.

In the following sections, the complexity analysis is measured with respect to the number of good paths in the graph. We show that the algorithms we use are polynomial in the number of good paths in the graph and in the input. By the above discussion, this indicates that with proper initialization, our algorithms are indeed polynomial in the input even when l is not bounded by $O(1)$.

This initialization with slices has more advantages than merely the reduction in the complexity. A discussion of these advantages is given in Section 4.1.

4.6 Similarity Score

In this section we describe preliminary ideas on combining the steps of target library construction and synthesis graph design. As discussed in the introduction, the target set of strings is chosen so as to represent the relevant parent library. There are many different sets of compounds, which may represent the parent library equally well. Appropriately choosing the right set of compounds may greatly affect the ability to synthesize it. In particular, our experiments show that though all tested data sets of the same type were chosen by the same diversity selection, on some of them our algorithms give much better results than on the others. On different real sets of 96 strings, scores ranged from 82 to 91, and on 1000-strings data sets they ranged from 339 to 377 (detailed results appear in Section 6).

A natural question is how to combine the diversity selection with the synthesis design, so that many target strings can be synthesized under the given synthesis constraints. As a first step toward this goal, we define a scoring function – the *similarity score* – to direct our search. The scoring function we define measures the similarity in the *sequence* of building-units (say, amino acids) of the compounds in the target set. The similarity of two compounds measures the number of arcs in a synthesis graph, that can possibly be shared by both compounds. Precisely, for each two strings, $a = a_1 \dots a_l$ and $b = b_1 \dots b_{l'}$ with $l' \geq l$, we define

$$Sim(a,b) = \frac{1}{l-1} \cdot \sum_{i=1}^{l-1} \chi(a_i a_{i+1}, b_i b_{i+1}), \text{ where}$$

$$\chi(a_i a_{i+1}, b_i b_{i+1}) = \begin{cases} 1 & \text{if } a_i = b_i, a_{i+1} = b_{i+1} \\ \\ 0 & \text{othewise} \end{cases} .$$

(the $\frac{1}{l}$ factor, normalizes this score, when compounds of different lengths are handled.)

Using the definition of similarity between two sequences, we may now define the similarity of a string to a set of strings, as follows.

$$Sim(a, S') = \sum_{b \in S'} Sim(a, b),$$

When $S' \subseteq S$ is very close in size to S, $Sim(a, S') \approx Sim(a, S)$ (as most of the compound are common to both sets). Therefore, in order to choose a substantial subset S' with large similarity score, we simply take the strings $'a'$ with the highest $Sim(a, S)$ scores.

4.7 Learning Rate

As discussed in Section 3.2, it is a common practice to use a learning parameter λ that decreases over time in order to avoid oscillating around the optimum without the ability to make fine enough movements to reach it. Hence we decrease λ as follows. We initialized λ to be 0.01. As long as the score improves, we keep λ constant. However, when a "zigzag" behavior is encountered we decrease λ by dividing it by two. Thus, $\lambda(t) = 0.01 \cdot (\frac{1}{2})^{t'}$, where t' is the number of times a "zigzag" behavior is encountered. This values were experimentally determined.

4.8 Escaping Local Maxima

The gradient descent algorithm is aimed at achieving a local optimum; however it is the global optimum which naturally interests us. Enhancing the chance of finding the global optimum can be done by running the algorithm from multiple starting points and choosing the best local optimum that was found. Another alternative is directing the search itself by imposing small changes which escape a local optimum and enable a continuation of the

search. While doing so, one must record the best local optimum found so far, so that eventually, the best local optimum that was encountered in the search is returned.

In this work we applied a combination of both methods – we examined multiple starting points and from each starting point we sought multiple local optima. The starting points are either chosen randomly, or by our initialization algorithm (see Section 3.5), or by our similarity measure (Section 3.6). The local changes we apply are arc deletions operations. The deletion of an arc changes the graph so that it is usually no longer a local optimum, and consequently, when the search (using the gradient-descent algorithm) continues, another local optimum is found.

The choice of arcs to be deleted can be done in many different ways. We checked three options, which we refer to as the *"Prob"*, *"Path"*, and *"Lookahead"* heuristics. The general scheme of all the different heuristics is the same: a score is attached to each arc, the arc with the lowest score is deleted, and the entire graph is amended (for example, other arcs may become irrelevant due to the arc deletion, so they are also deleted). However, the three heuristics differ in the way they score the arcs. These scoring methods are described in the following sections. Since each scoring method implies a different order of arcs deletion, it gives a different synthesis graph.

Combining the arc deletion heuristic with the gradient-descent algorithm is done according to the following scheme: as long as there are more paths than the number of beads (b), iterate between (1) running the gradient-descent algorithm until a local optimum is found, and (2) applying an arc deletion heuristic followed by the deletion of irrelevant arcs, that is, arcs which are not on any path corresponding to a target string.

4.8.1 Prob Score

In the *prob score* each arc is assigned a score according to the total path probability of good paths using it. This score is calculated using the normalized arcs weights, and equals the sum of paths' weights over all good

paths passing through that arc.

$$prob\ score(u_i, u_{i+1}) = \sum_{p=(s,v_1,...,v_l) \in Good(u_i,u_{i+1})} w(s, v_1) \cdot \prod_{i=1}^{1-1} w(v_i, v_{i+1}),$$

where

$$Good(u_i, u_{i+1}) = \{p = (s, v_1, ..., v_l) \mid p\ is\ a\ good\ path,\ v_i = u_i, v_{i+1} = u_{i+1}\}$$

For each given path this calculation is linear in l. All good paths can be stored in a list of size t (see Section 3.5.4). Thus, calculating this score for all the arcs in the graph can be done in time $O(|E| \cdot t \cdot l)$. This is indeed polynomial as our assumptions imply that t is polynomial in the input length (see Section 3.5.4).

Note that this heuristic takes into account the arcs weights and therefore running the gradient descent algorithm affects the arcs that are deleted in the heuristic. In Sections 3.8.2 and 3.8.3 we shall show procedures that do not use the weights.

4.8.2 Path Score

The *path score* of an arc is the ratio between the number of good paths and the number of bad paths passing through it.

$$path\ score(v, u) = \frac{\#\text{good paths passing through } (v, u)}{\#\text{bad paths passing through } (v, u)}$$

Although there may be an exponential number of such bad paths, their number can be efficiently computed observing that the number of bad paths through an arc equals the total number of paths passing through it minus the number of good paths passing through it. The number of good paths through an arc is polynomial (see Section 3.5.4).

The total number of paths passing through an arc can be calculated as follows. For each node v we calculate two values: the number of paths outgoing from v ($outgoing(v)$), and the number of paths incoming to it

($incoming(v)$). $outgoing(v)$ can be recursively calculated for all the nodes in the graph starting from the last layer and going backwards: $outgoing(v) = 1$ if $v \in Last(G)$, $outgoing(v) = \sum_{\{ u \,|(v,u)\in E\}} outgoing(u)$ otherwise. Working layer-by-layer backwards implies that $outgoing(u)$ is previously calculated as u is in a layer preceding the layer of v. $incoming(v)$ can be calculated in a similar manner, when starting from $First(G)$. The total number of paths passing through a node is the product of the number of incoming paths and the number of outgoing paths, that is, $incoming(v) \cdot outgoing(v)$.

4.8.3 Lookahead Score

Recall that a string is valid (see Section 1.2.2) if it corresponds to some path $p \in P(G)$ (it is not necessarily produced by the graph, as its weight may be too small). When an arc is deleted, any string σ that was created only through that arc, is no longer valid in the graph. Hence, all the arcs which were used only by σ or other bad paths may also be deleted. The deletion of those arcs may in turn cause the elimination of some additional bad paths (and no additional good paths). In the *lookahead score* we take into account the total effect of the deletion of the arc and not only the counts of good and bad paths passing through it.

Similar to the path score, the lookahead score of an arc is the ratio between good and bad paths eliminated by the deletion of the arc. The difference is that here we regard *all* paths eliminated as a result of the arc deletion, that is, both paths that pass through that arc, and paths that pass through other arcs that are deleted as a result of the deletion of that arc. The number of such paths is calculated by summing the numbers of bad paths through the arc and all through all other arcs that could be deleted as a result of its deletion.

$$lookahead\ score(v,u) = \frac{\#\text{good paths eliminated as a result of deleting } (v,u)}{\#\text{bad paths eliminated as a result of deleting } (v,u)}$$

Note that both the path and the lookahead scores are not influenced by arcs weights (they are only affected by the existence or non-existence of

the arcs). Additionally, the gradient-descent algorithm does not eliminate arcs (it only gradually changes their weights). Hence, the scores they assign are not affected by applying gradient-descent on the graph before using them. Nevertheless, they often achieve better results than the prob score, as discussed in Chapter 6.

4.9 Recalculating Arcs Weights

Once the number of paths in the graph is no more than the number of beads, the arcs weights are reassigned to be:

$$w(v,u) = \frac{outgoing(u)}{outgoing(v)},$$

where $outgoing(v)$ is the number of paths outgoing from node v. This number is calculated as described in Section 3.8.1.

In the following we show that this is a weighting scheme by which all the paths in the graph are produced, *i.e.*, the weight of each path is at least $\frac{1}{b}$.

First, note that $w(v,u)$ is a normalized weight, as: $0 \leq w(v,u) \leq 1$ and

$$\begin{aligned}
\sum_{\{u\,|(v,u)\in E\}} w(v,u) &= \sum_{\{u\,|(v,u)\in E\}} \frac{outgoing(u)}{outgoing(v)} \\
&= \frac{1}{outgoing(v)} \cdot \sum_{\{u\,|(v,u)\in E\}} outgoing(u) \\
&= 1.
\end{aligned}$$

It remains to show that indeed the weight of each path in the graph is at least $\frac{1}{b}$. This is easily proved by incorporating the following three facts: the weight of a path is a telescopic product; for each $v_l \in Last(G)$, $outgoing(v_l) = 1$; and $outgoing(s) \leq b$ (as the number of paths in no more

than the number of beads). The formal calculation follows:

$$\begin{aligned} weight(p = (s, v_1, ..., v_l)) &= w(s, v_1) \cdot \prod_{i=1}^{l-1} w(v_i, v_{i+1}) \\ &= \frac{outgoing(v_1)}{outgoing(s)} \cdot \prod_{i=1}^{l-1} \frac{outgoing(v_{i+1})}{outgoing(v_i)} \\ &= \frac{outgoing(v_l)}{outgoing(s)} \\ &\geq \frac{1}{b} \end{aligned}$$

Chapter 5

A Discrete Approach to Max Strings Synthesis

An alternative approach to solving the Max Strings Synthesis is by discrete search over the space of synthesis graphs. This is done by starting from an initial graph and performing local changes with the aim of gradually improving the graph. In the following we describe both the initialization method, and the local changes we used.

5.1 Overview of the Algorithm

In order to simplify our discussion, let us begin with a few definitions. Recall that a target string is a string which appears in the input S. We say that a strings is *unused*, if it is a target string with no corresponding path in the graph. We say that a string is *unrequested*, if it is not a target string.

Initializing the graph requires setting the nodes labels and the arcs weights. When doing so, we are confronted with two opposing goals: on the one hand, we want as many paths corresponding to target strings as possible, but on the other hand, we want as few paths for unrequested strings as possible. We want to limit the number of paths for unrequested strings,

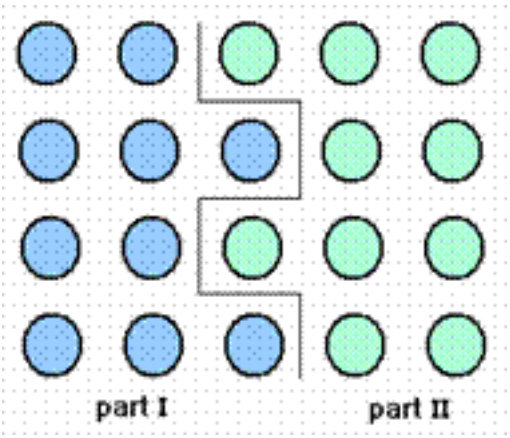

Figure 5.1: Initialization in "slices". Arcs between light nodes and dark nodes are not allowed.

since the number of beads is limited: having more unrequested strings leads to less requested strings being produced by the graph. To balance these two opposite goals, we initialize the graph in "slices" (compare Figure 4.1) as follows. Each slice has no more than one copy of each label (letter) in each layer; and the slices are unconnected, that is, there are no arcs connecting nodes in different slices (arcs between nodes of the same slice are allowed). This type of initialization reduces the connectivity of the graph, thus reducing the number of paths in the graph.

The number of slices depends on the constraint of the graph width. If the graph width is smaller than the size of the alphabet, then only one slice is initialized. In general, once the previous slices were already determined, the next slice takes the next x_k nodes in each layer k, where $x_k = \min\{|\Sigma_k|, w'_k\}$, with Σ_k – the alphabet at position k in the unused strings, and w'_k – the remaining width of layer k of the graph, that is, the number of unlabelled nodes in layer k.

In our algorithm, the initialization and the arc-deletion heuristic are *interleaved* together – we iterate between initializing the next slice and using the arc-deletion heuristic on the current graph (*i.e.*, all slices initialized so far). After using the arc-deletion heuristic we check which of the strings are

not produced by the graph built so far, that is, which are the unused strings. When initiating the next slice, we concentrate only on those strings.

The general scheme of the algorithm is as follows.

1. Initially the set of unused strings is the set S of all target strings.

2. Define the current slice: in each layer k, the current slice contains the first x nodes, where $x = \min\{|\Sigma_k|, w\}$ (we assume the nodes in each layer are ordered from 1 to w).

3. Repeat until

   - the entire width of the graph is used in at least one level, or
   - all the target strings are produced by the graph.

   (a) Initialize current slice: label nodes and assign arcs weights in the current slice of the graph according to the set of unused strings,

   (b) Arcs-deletions heuristic: repeat until the number of paths is at most b:

      i. attach a score to each arc, and delete the arc with the lowest score,

      ii. delete irrelevant arcs (*i.e.*, arcs that are no longer on any path corresponding to a target string).

   (c) Recalculate the arcs weights so that each path is of weight at least $\frac{1}{b}$.

   (d) Let the unused strings be the strings in S that are not produced by the current graph.

   (e) Define the current slice: in each layer k, the current slice contains the next x nodes, where $x = \min\{|\Sigma_k|, w'_k\}$

In the following sections, the algorithm is explained in detail.

5.2 Initialization

Each slice is initialized according to the set of unused strings. The nodes are labelled as follows: Each letter in the unused strings is assigned as a label of one node in the layer corresponding to its position in the string. The arcs are assigned weights as follows: Let u, v be nodes in layers $i, i+1$ respectively. The weight of the arc (u, v) is assigned according to the frequency of strings in the set of unused strings with letters corresponding to the labels of u and v in positions i and $i+1$ respectively. For a more detailed description of this initialization the reader is referred to Section 3.5. Note that the initialization is done with respect to the set of unused strings and not with respect to the target set S (as in Section 3.5).

5.3 Arc Deletion

Once a new slice is initialized, we usually obtain a graph with far too many paths (with respect to b – the constraint over the number of beads). Therefore we use the arc-deletion heuristic to transform this graph into a graph with no more than b paths. Then we recalculate the arcs weight in such a way, that each valid path in the graph is produced by it. Note that, we would like not only to reduce the number of paths in the graph, but also to maintain as many paths corresponding to target strings as possible. We would like to eliminate many paths corresponding to unrequested strings, and as few paths corresponding to target strings as possible.

Similarly to the escape from local maximum (see Section 3.8), the changes we impose on the graph are arc deletions. Our general strategy in choosing the next deleted arc is to delete the worst arc according to our scoring function; that is:

1. Repeat until the number of path is at most b
    - Score all arcs according to the used scoring function.
    - Delete an arc with the worst score.

It remains to describe the scoring function. Here, again, we tried the three scoring functions described in Section 3.8, that is, the prob score (see Section 3.8.1), the path score (see Section 3.8.2) and the lookahead score (see Section 3.8.3). As presented later (see Chapter 6), the lookahead score gave the best results, and hence it was chosen as our scoring function.

Note that the arcs weights affect the score only when the prob score is used. Otherwise, we may simply mark the arcs as existent vs. non-existent.

5.4 Recalculating Arcs Weights

The recalculation of the arcs weights is done as described in Section 3.9.

Chapter 6

Implementation

In this chapter we discuss some implementation related topics. This includes details on the software we developed, handling sets with strings of variable lengths, and the graphical interface we implemented.

6.1 Software

Our algorithms are implemented using the $C++$ language. In total, the software contains approximately 12,000 lines of code. The programs are quite fast. For example, while running on 500MHz Pentium III, the *look-ahead* algorithm runs (on average) within 45 seconds on a large data set with 1000 strings, and only 0.0032 seconds on a smaller data set with 96 strings. Exact times are given in Chapter 6.

6.2 Handling Strings Sets with Variable Lengths

So far, we described our algorithms and the synthesis graph under the assumption that all target strings had the same length. This was done in order to simplify the description. In practice we also handle strings sets with varying lengths. This is incorporated in our algorithm in the following

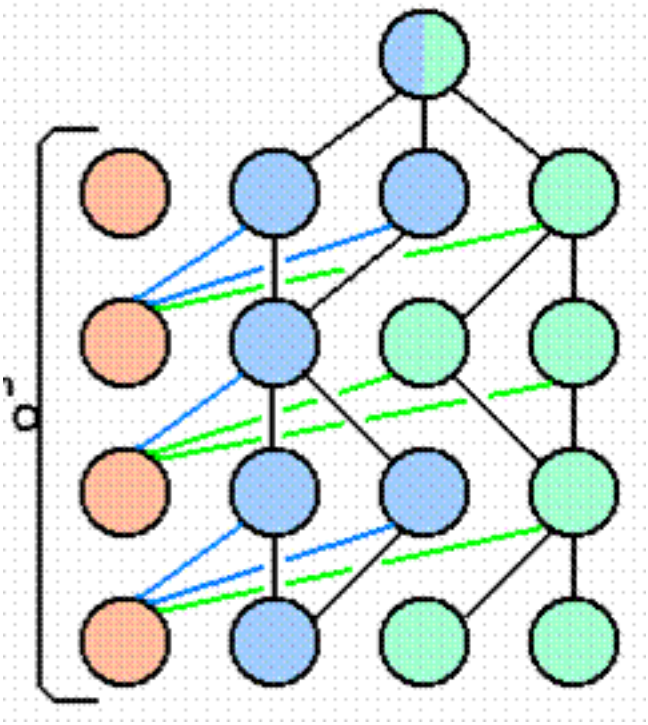

Figure 6.1: Phantom nodes. The left column of nodes represents the Phantom nodes. The beads that end up on these nodes hold short requested strings.

way. We define an additional column of nodes, which we refer to as *phantom nodes*. The phantom nodes have no labelling, and they represent a path termination: an arc from node v into a phantom node p corresponds to a short string that ends in node v. Hence, an arc (v, p) indicates that, in the synthesis process, $w(v, p)$ fraction of beads should be taken out of v so that no more grow operations would be applied to them.

6.3 Graphics

As means of illustrating the synthesis graph obtained as an output of our algorithm, we implemented a graphical interface. In our graphical interface

the output graph (labels and arcs weights) is presented as a picture, with some additional features such as the list of strings produced by the graph, their weights and the paths through which they were produced. An example of such an illustration is given in Figure 5.2

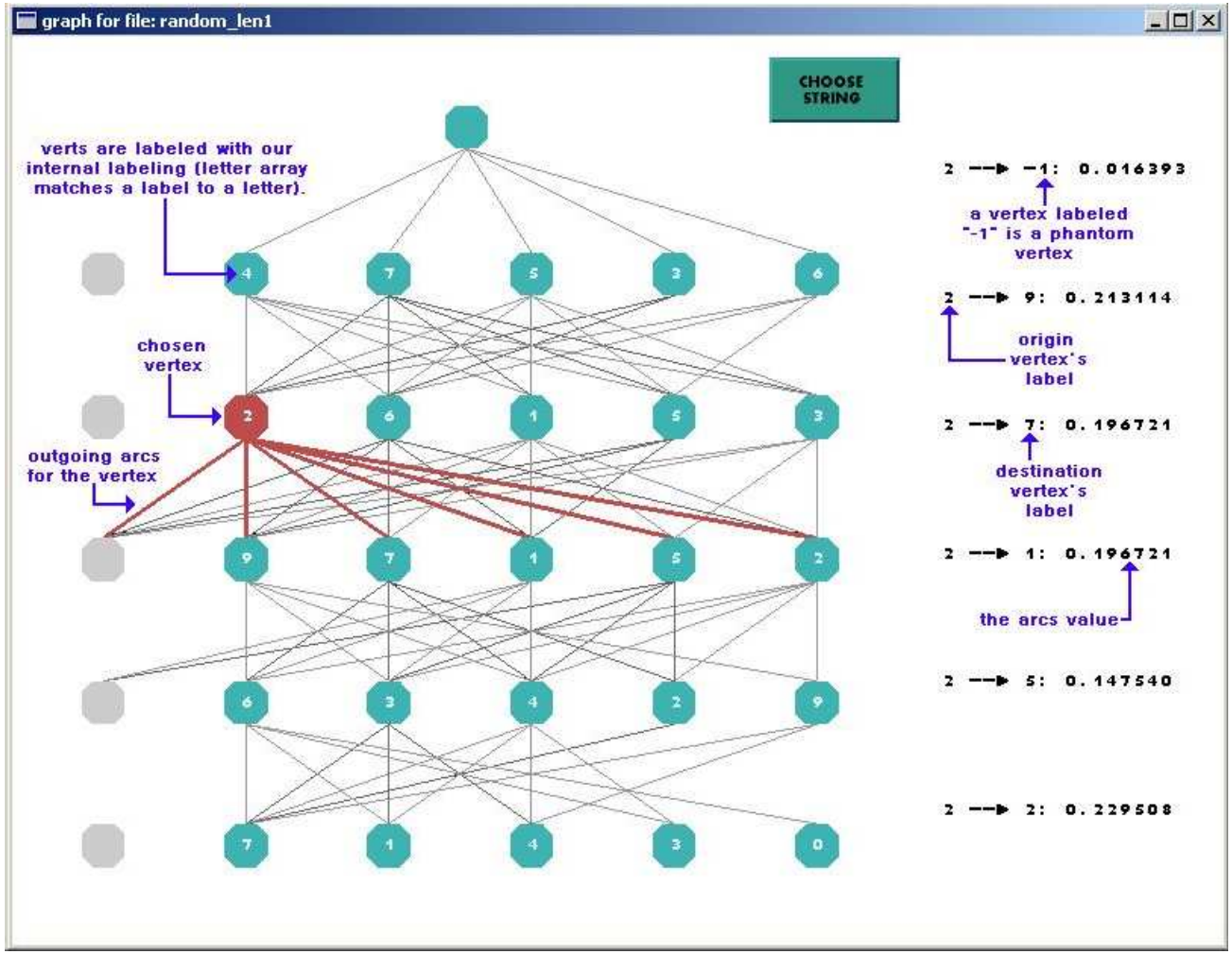

Figure 6.2: Graphical representation of a synthesis graph and the strings produced by it.

Chapter 7

Computational Results

In this chapter we present computational results of our algorithms for solving the Max Strings Synthesis problem. In order to estimate both the success of our different algorithms, and their time requirements, we performed extensive experiments with all algorithms on different types of data sets. In this chapter, we first describe the data sets we used, and then give comparative results of all the algorithms we designed. Those comparative results show that the *lookahead* algorithm was the best both with respect to time requirements and to the synthesis graphs it found. Hence, we continue by running extensive experiments on this algorithm alone, and present their results.

7.1 Data Sets

The data sets we used may be divided into 3 types:

- small sets of real data,
- large sets of real data,
- synthetic data.

We worked with a *parent library*, which contained all possible sequences of length 5 generated by the 10 natural amino acids (in parenthesis, single letter code): Alanine(A), Arginine(R), Asparagine(N), Aspartic Acid(D), Cysteine(C), Glutamine(Q), Glutamic Acid(E), Glycine(G), Histidine(H) and Isoleucine(I).

Each sequence was characterized by a set of 30 descriptors and following Principal Component Analysis (PCA) it was found that six principal components covered more than 90% of the variance in the original data set. Sets of sequences were diversity selected from the space defined by the above six PC's by the MaxMin[1] function using 100000 Monte Carlo steps with 10000 idle steps as a termination criterion. All calculations were performed with Cerius2 version 4.5[11].

By this diversity selection method, we chose 50 small data sets, each containing 96 compounds. We also chose 20 large data sets, each containing 1000 compounds. Additionally, we used synthetic data sets. The synthetic data sets are based on a set of 60 compounds from the parent library, from which 55 compounds can be produced by a synthesis graph with the *basic parameters* of 10000 beads and width 10. The 60 compounds are chosen such that a minimum synthesis graph of width 10 producing all of them has out-degree 6 in each node. Thus, it has $10 \cdot 6^4$ paths, which is more than the number of beads (10,000). By eliminating 5 paths, we can obtain a synthesis graph producing 55 compounds, in which the out-degree of half of the nodes is reduced to 5. This graph has only $5 \cdot 5^4 + 5 \cdot 6^4$ paths (which is less than the number of beads). This construction gave us a lower bound on the number of paths in the optimal solution, and hence a way to evaluate the performances of our algorithms in an absolute (and not only relative) manner. On top of theses 60 compounds, we added different amounts of "noise", that is, extra compounds, which were randomly chosen from the parent library.

[1]The maxmin function is defined as follows: Let D_{ij} be the distance in the property space between the two compounds i and j. This function maximizes the minimum squared distance between two points in the selected subset of compounds. This objective aims to generate a subset of the parent library that is as diverse as possible.

In summary, this is a list of the data sets we used:

(i) 50 sets of 96 compounds, diversity selected from the parent library.

(ii) 20 sets of 1000 compounds, diversity selected from the parent library.

(iii) 10 sets of synthetic data with 10 "noise compounds".

(iv) 10 sets of synthetic data with 20 "noise compounds".

(v) 10 sets of synthetic data with 30 "noise compounds".

(vi) 10 sets of synthetic data with 40 "noise compounds".

Note that the results on the synthetic data are encouraging (see Tables 6.5,6.6,6.7 and 6.8).

7.2 Comparison of the Continuous and Discrete Approaches

Our results comparing the performances of the algorithms we have presented on the different data sets are given in Tables 6.2-6.8. In each table, we present the performance on a different type of data set. All algorithms were run with the basic parameters: number of beads 10000 and width 10. The gradient-descent algorithms were run with the following parameters for the logistic sigmoid ($logsig$) function: $\alpha = 1$, $\beta = 1$.

The following is a brief key to reading the tables. In each line, we present the performances of all algorithms on a single data set. Each column gives the number of target molecules that were produced by the solution of the used algorithm (in the followings this number is referred to as the score of the algorithm).

The synthesis graphs normally utilize (roughly) all the given beads, thus they produce many other molecules, in addition to those in the data set. In most of our experiments the synthesis graphs had unique labels in each layer,

thus the produced strings were distinct. Consequently, when using 10000 beads, the graph produced approximately 10000 distinct strings. However, the excessive strings are not presented in the results we give here as they are not part of the problem definition.

The first column gives the outcome of the gradient descent algorithm, described in Section 3. The second column describes the outcome of the algorithm combining the gradient descent algorithm with the prob score heuristic of arcs-deletion (see Section 3.8). The next three columns, "prob", "path" and "lookahead", describes the performances of the different arcs-deletion heuristics that we use – the prob score heuristic (see Section 3.8.1), the path score heuristic (see Section 3.8.2) and the lookahead heuristic (see Section 3.8.3).

Table 6.1 compares the time requirements of the different algorithms.

7.3 Discussion of Algorithms Comparison

From the comparison results, it is clear that the lookahead algorithm performed consistently better than all the other algorithms we have presented.

It is expected that the lookahead heuristic would be better than the prob and path heuristics. All those three heuristics work by the same principle of performing arcs deletions which locally improve the ratio of target strings vs. unrequested strings produced by the graph. The "lookahead" heuristic has an advantage over the two other heuristics, since it chooses the deleted arcs by testing the results of these actions a few steps ahead and not only the immediate result. Even so, it could be that occasionally the other heuristics outperform the lookahead heuristic, since all the heuristics are limited to a *local* estimation, and neither is guarantied to identify the optimum arcs-deletion ordering.

A more surprising result is the superiority of the lookahead heuristic over the gradient-descent algorithm. It is clear that the gradient-descent is liable to reach a local minimum, and indeed this shortcoming explains

the poor results achieved by the gradient-descent. However, by and large, when combined with the prob heuristic this shortcoming is overcome, as the results show. Nevertheless, the lookahead algorithm outperforms the combined gradient-descent and prob heuristic algorithm.

In order to explore the reasons for the superiority of the lookahead algorithm over the gradient-descent algorithm, we observed the progress of both algorithms. This progress is presented in Figures 6.1 and 6.2. We can see that both algorithms progress in a similar manner of gradually increasing the number of produced compounds, except in the last step. In the last step, we see a great leap in the number of produced strings in the lookahead algorithm, with only a minor improvement for the gradient-descent algorithm. This great leap in the number of produced strings is a result of the recalculation of the arcs weights, which is applied to the graph by the lookahead algorithm, when the number of paths in the graph decreases below the number of beads. By this recalculation, we obtain a graph that produces all strings with a corresponding path in the graph, and hence we have a significant increase in the number of produced strings. However, in the graph obtained by the gradient-descent, the number of paths does not usually decrease below the number of beads, as the gradient-descent algorithm hardly ever eliminates arcs, but rather gradually decreases their weights. Therefore, the gradient-descent algorithm does not usually achieve a graph with less paths than the number of beads, and hence does not enjoy the leap at the number of produced strings, observed in the lookahead algorithm. Note however that even without the "leap" the lookahead algorithm is superior.

When the gradient-descent is combined with an arcs-deletion heuristic, arcs are deleted until the number of paths decreases below the number of beads. Once this occurs, the arcs weights are recalculated so that all strings with corresponding paths are produced. This causes a leap in the score, similar to the one in the lookahead algorithm. Nevertheless, when the gradient-descent is combined with the prob heuristic, the achieved scores are still inferior to the one achieved by the lookahead heuristic. This is due

to the superiority of the lookahead heuristic over the prob heuristic.

The lookahead algorithm is also faster than the gradient-descent algorithm. In Table 6.1 we present the average running times of the gradient-descent and the lookahead algorithms. The running time of the gradient-descent algorithm is mostly determined by the number of iterations. The number of needed iterations depends on the extent of convergence desired by the user.

	gradient descent (50,000 iterations)	lookahead
1000 real data	100 min	45.15 sec
96 real data	6 min	0.0032 sec

Table 7.1: Running time comparison. Numbers are average over all data sets of the corresponding types. The running time of the gradient-descent algorithm is expressed by the average number of *minutes* needed for completing 25,000 gradient-descent iterations (on the data sets we examined, 25,000 iterations sufficed for convergence). The running time of the lookahead algorithm is expressed by the average number of *seconds* needed for a complete run.

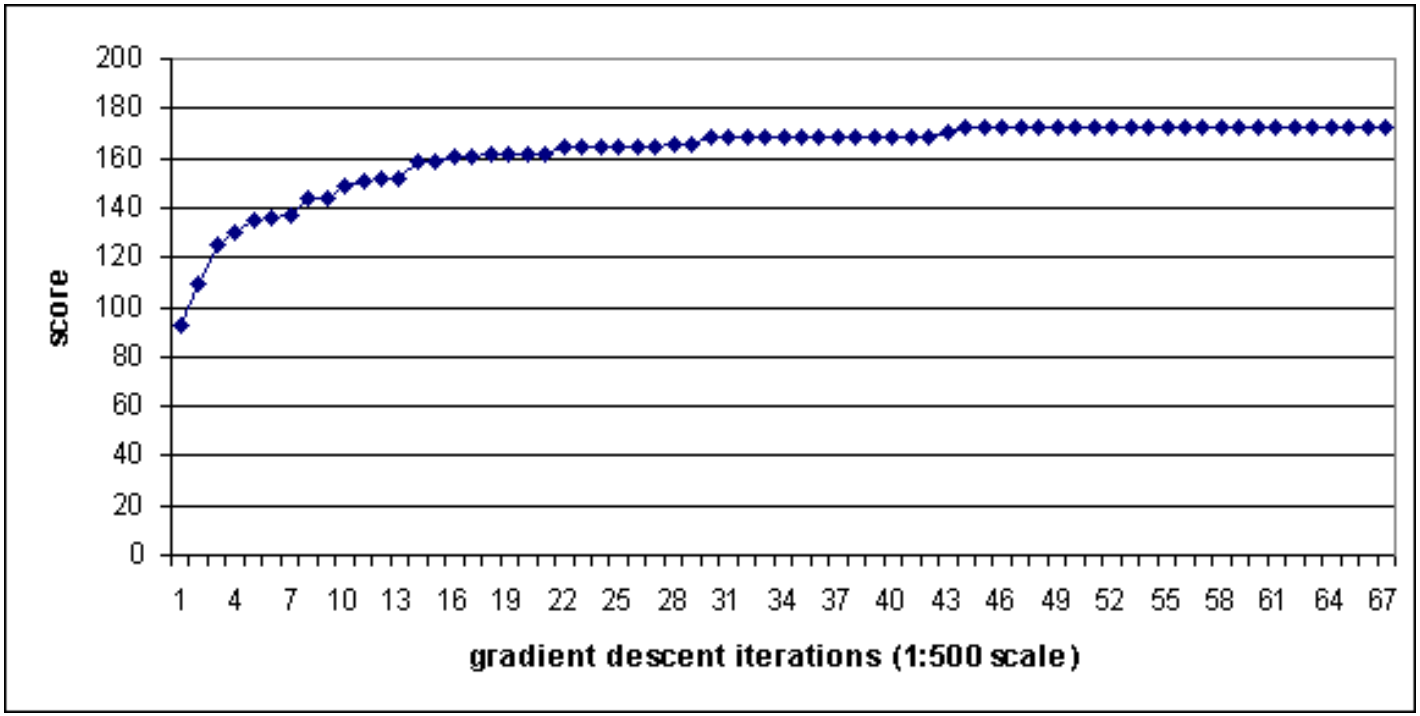

Figure 7.1: Progress of the gradient-descent algorithm. The results show the improvement of the score as the iterations progress, on one real 1000 strings data set. The iterations are presented in a scale of 1:500.

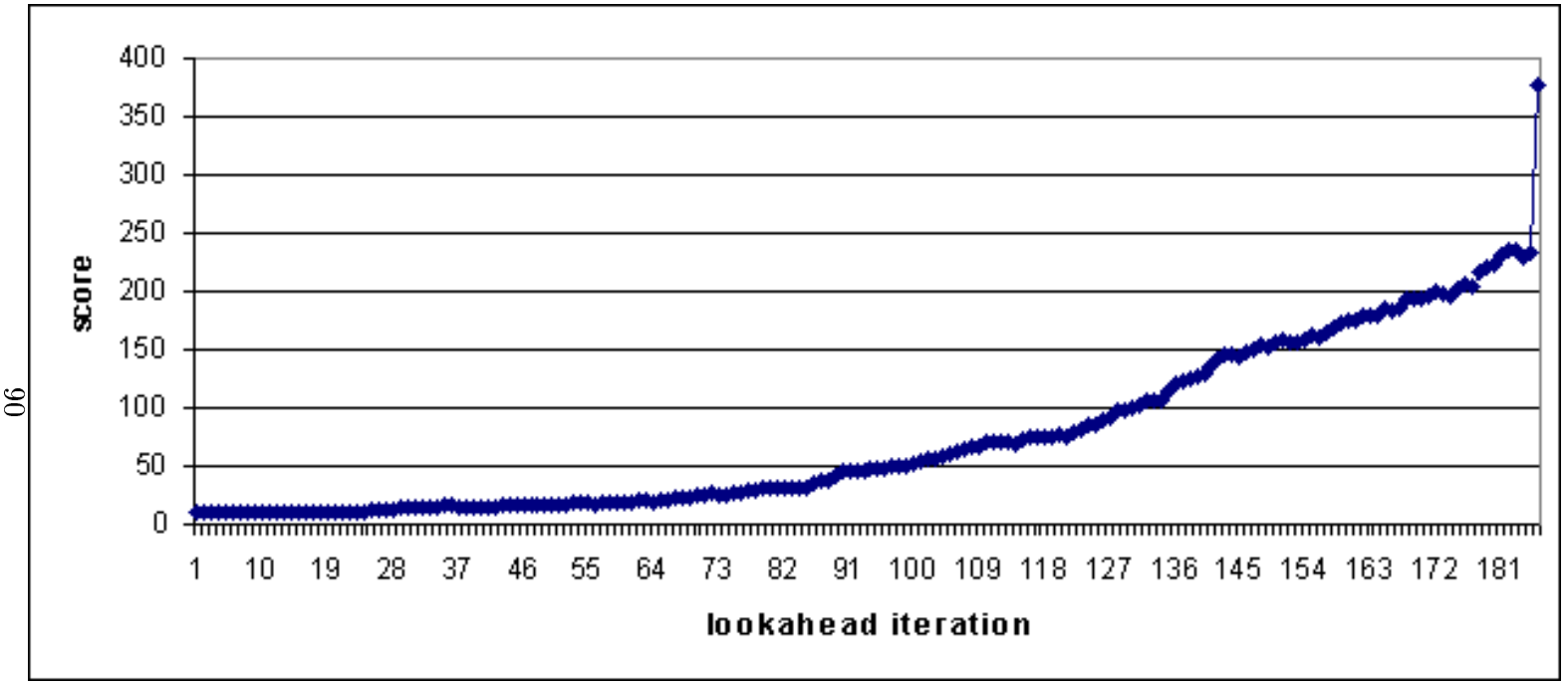

Figure 7.2: Progress of the lookahead algorithm. The results show the improvement of the score as the iterations progress, on one real 1000 strings data set. The iterations are presented in a scale of 1:1.

7.4 Detailed Results for the Lookahead Algorithm

Having established the superiority of the lookahead algorithm, we would like to explore its performance more deeply. We present results giving the number of target compounds produced by the lookahead algorithm, when different constraints were imposed on the synthesis graph.

First, we would like to explore the difference between the results on the two real data sets. At first, it might seem surprising that while on the 96 strings data sets we reach an average success of 87%, on the 1,000 strings data sets, we reach an average success of only 36%. This phenomenon is easily understood when we recall that the graph parameters (width 10, and number of beads 10,000) were kept the same for both data sets in spite of the great difference between their sizes. Figure 6.3 shows that indeed when the width of the graph is increased by the factor of 10, the success over the large data sets (1,000 strings) becomes close to 80% as expected. Additionally, when the number of beads in increased 10-fold, the success trivially becomes 100%, as in a graph of width 10 and length 5, the full combinatorial library requires no more than 100,000 beads.

Next, we explore the behavior of our algorithm when different values of the graph width and the number of beads are taken. In Figure 6.3 we present the results of the lookahead algorithm on the large sets of real data with different widths between 2 and 100. In Figure 6.4 we present the results of the lookahead algorithm on the small sets of real data with widths between 5 and 20.

In Figures 6.6 and 6.5 we give the number of produced strings on graphs with width 10 as a function of the number of beads.

	gradient descent	gradient & prob hue.	prob heuristics	path heuristics	lookahead heuristics
1000 real data1	184	364	367	382	377
1000 real data2	194	348	341	369	368
1000 real data3	173	324	327	349	350
1000 real data4	195	357	350	368	371
1000 real data5	184	340	340	356	357
1000 real data6	191	341	349	360	360
1000 real data7	198	354	361	373	373
1000 real data8	182	341	348	370	368
1000 real data9	181	328	335	346	341
1000 real data10	172	325	319	347	347
1000 real data11	170	326	322	339	339
1000 real data12	173	329	332	342	342
1000 real data13	194	334	334	354	355
1000 real data14	191	349	352	366	369
1000 real data15	193	343	345	359	358
1000 real data16	178	340	333	356	356
1000 real data17	188	334	343	358	357
1000 real data18	196	357	358	367	368
1000 real data19	189	329	323	343	346
1000 real data20	175	337	338	352	355
average	185.05	340	340.85	357.8	357.85
std	9.09	11.80	13.18	11.61	11.32

Table 7.2: Performance of all algorithms on 1000 strings real data

	gradient descent	gradient & prob hue.	prob heuristics	path heuristics	lookahead heuristics
96 real data 1	57	85	85	83	86
96 real data 2	63	89	89	90	91
96 real data 3	59	87	87	85	88
96 real data 4	62	87	88	89	89
96 real data 5	58	86	86	86	87
96 real data 6	55	84	84	84	86
96 real data 7	59	86	85	86	86
96 real data 8	53	84	84	83	84
96 real data 9	63	89	89	87	89
96 real data 10	60	86	87	86	88
96 real data 11	55	86	87	87	88
96 real data 12	58	88	88	87	88
96 real data 13	57	86	87	83	87
96 real data 14	58	85	84	82	85
96 real data 15	54	86	86	85	87
96 real data 16	57	85	84	82	86
96 real data 17	59	87	88	88	89
96 real data 18	59	88	87	87	88
96 real data 19	55	83	84	82	85
96 real data 20	57	85	85	84	86
96 real data 21	56	85	85	84	86
96 real data 22	62	89	89	88	90
96 real data 23	58	85	85	85	87
96 real data 24	60	87	87	87	88
96 real data 25	58	86	86	86	87
96 real data 26	59	85	85	82	86
96 real data 27	57	86	86	85	85
96 real data 28	57	85	84	85	86
96 real data 29	49	80	81	78	82
96 real data 30	58	87	88	86	89

Table 7.3: Performance of all algorithms on 96 strings real data (part 1)

	gradient descent	gradient & prob hue.	prob heuristics	path heuristics	lookahead heuristics
96 real data 31	57	86	86	87	87
96 real data 32	57	86	86	85	87
96 real data 33	58	85	85	84	86
96 real data 34	57	85	86	84	87
96 real data 35	58	86	85	85	87
96 real data 36	57	87	88	86	88
96 real data 37	57	84	87	85	88
96 real data 38	57	87	88	86	88
96 real data 39	58	86	87	85	87
96 real data 40	52	83	82	82	84
96 real data 41	58	87	87	86	88
96 real data 42	58	85	85	83	86
96 real data 43	55	83	85	83	86
96 real data 44	62	87	87	87	88
96 real data 45	62	86	86	86	87
96 real data 46	56	86	87	87	88
96 real data 47	54	86	86	84	86
96 real data 48	57	87	87	85	88
96 real data 49	55	85	86	85	87
96 real data 50	58	86	87	84	88
average	57.5	85.8	86.06	85.02	87
std	2.68	1.62	1.68	2.12	1.57

Table 7.4: Performance of all algorithms on 96 strings real data (continued)

	gradient descent	gradient & prob hue.	prob heuristics	path heuristics	lookahead heuristics
70 synth data1	35	61	61	60	61
70 synth data2	36	61	62	61	62
70 synth data3	36	60	61	61	61
70 synth data4	35	60	61	60	61
70 synth data5	36	61	61	61	62
70 synth data6	36	63	63	62	63
70 synth data7	36	61	61	61	62
70 synth data8	35	61	61	61	61
70 synth data9	37	63	63	63	63
70 synth data10	37	62	62	62	63
average	35.9	61.3	61.6	61.2	61.9
std	0.73	1.05	0.84	0.91	0.87

Table 7.5: Performance of all algorithms on 70 strings synthetic data

	gradient descent	gradient & prob hue.	prob heuristics	path heuristics	lookahead heuristics
80 synth data1	42	68	68	68	69
80 synth data2	40	68	69	67	69
80 synth data3	44	70	70	69	70
80 synth data4	44	69	69	67	69
80 synth data5	39	65	66	65	67
80 synth data6	44	69	70	69	70
80 synth data7	41	67	68	67	68
80 synth data8	42	68	69	68	69
80 synth data9	42	67	67	66	68
80 synth data10	41	67	67	66	68
average	41.9	67.8	68.3	67.2	68.7
std	1.72	1.39	1.33	1.31	0.94

Table 7.6: Performance of all algorithms on 80 strings synthetic data

	gradient descent	gradient & prob hue.	prob heuristics	path heuristics	lookahead heuristics
90 synth data1	46	76	77	75	77
90 synth data2	46	74	75	72	75
90 synth data3	47	75	75	74	76
90 synth data4	44	74	74	72	75
90 synth data5	47	74	75	73	75
90 synth data6	44	72	74	72	74
90 synth data7	46	74	73	71	75
90 synth data8	45	74	75	73	76
90 synth data9	50	77	77	77	78
90 synth data10	47	75	75	72	76
average	46.2	74.5	75	73.1	75.7
std	1.75	1.35	1.24	1.79	1.15

Table 7.7: Performance of all algorithms on 90 strings synthetic data

	gradient descent	gradient & prob hue.	prob heuristics	path heuristics	lookahead heuristics
100 synth data1	52	82	82	79	83
100 synth data2	49	81	81	78	81
100 synth data3	49	81	81	79	82
100 synth data4	51	81	80	78	82
100 synth data5	46	78	80	77	80
100 synth data6	51	82	81	79	82
100 synth data7	48	78	78	76	79
100 synth data8	49	78	78	77	80
100 synth data9	52	82	82	80	83
100 synth data10	50	81	83	81	83
average	49.7	80.4	80.6	78.4	81.5
std	1.88	1.71	1.64	1.50	1.43

Table 7.8: Performance of all algorithms on 100 strings synthetic data

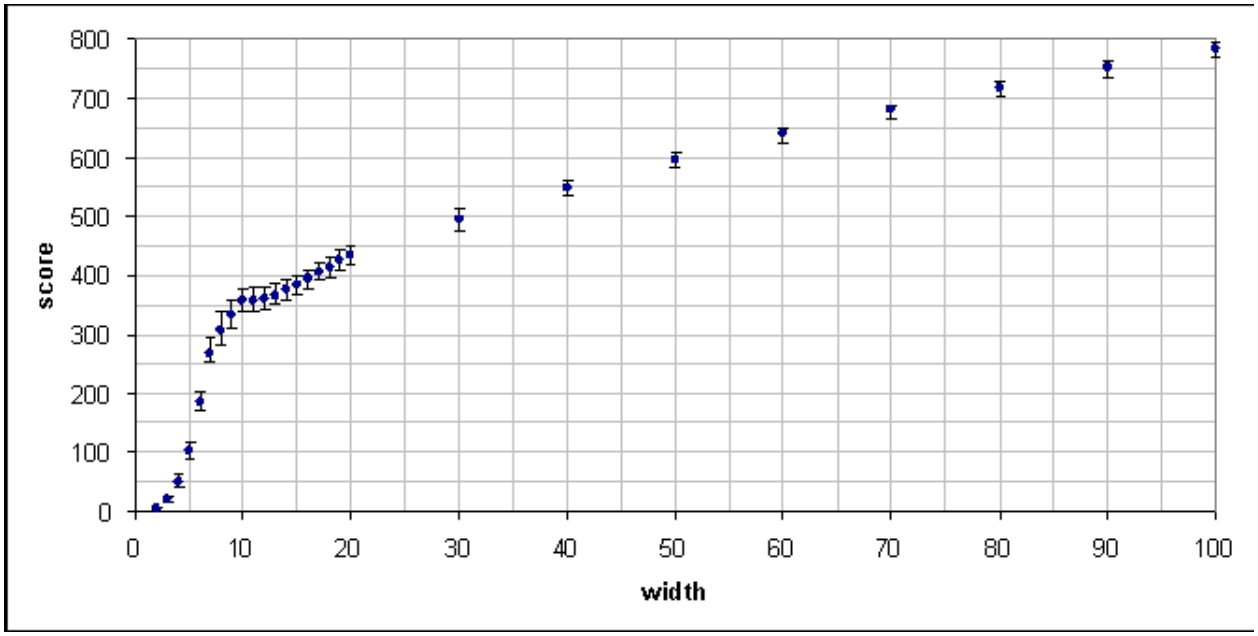

Figure 7.3: Impact of the width on performance. The graph summarizes the results of the lookahead algorithm with different widths on real data sets of 1000 strings. The average (dot), minimum and maximum (bars) are shown for each width.

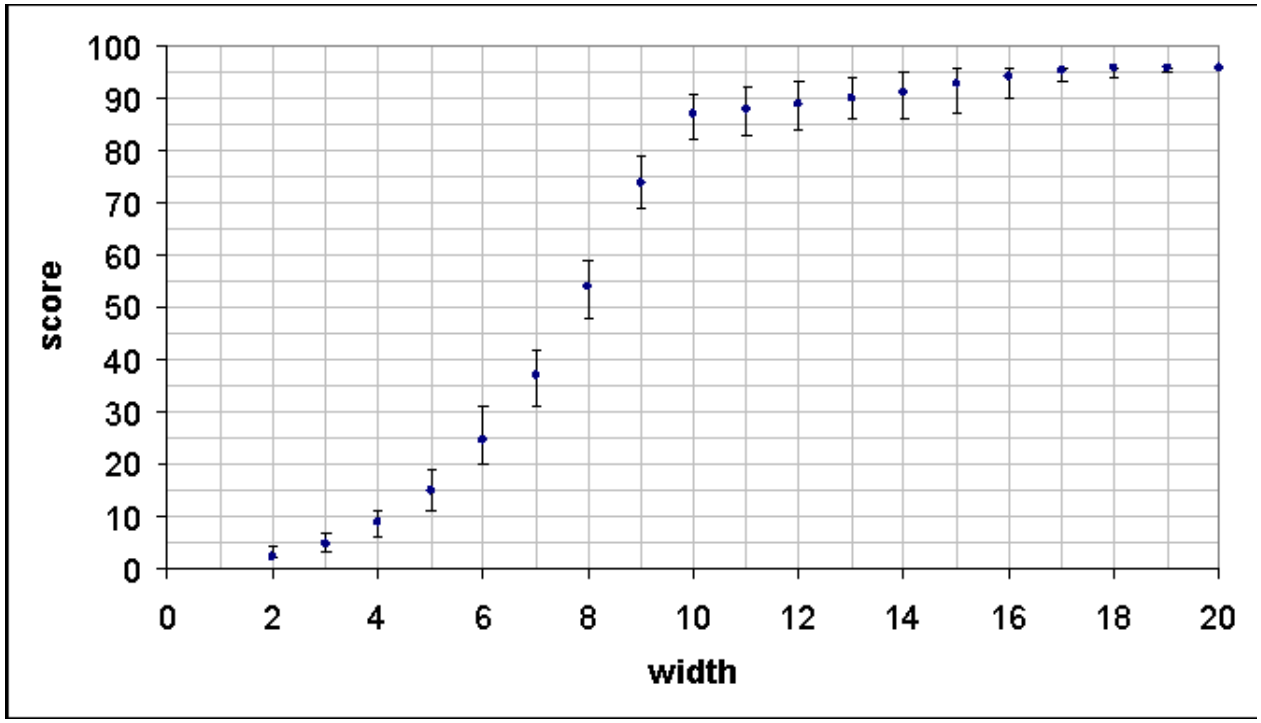

Figure 7.4: Impact of the width on performance. The graph summarizes the results of the lookahead algorithm with different widths on real data sets of 96 strings. The average (dot), minimum and maximum (bars) are shown for each width.

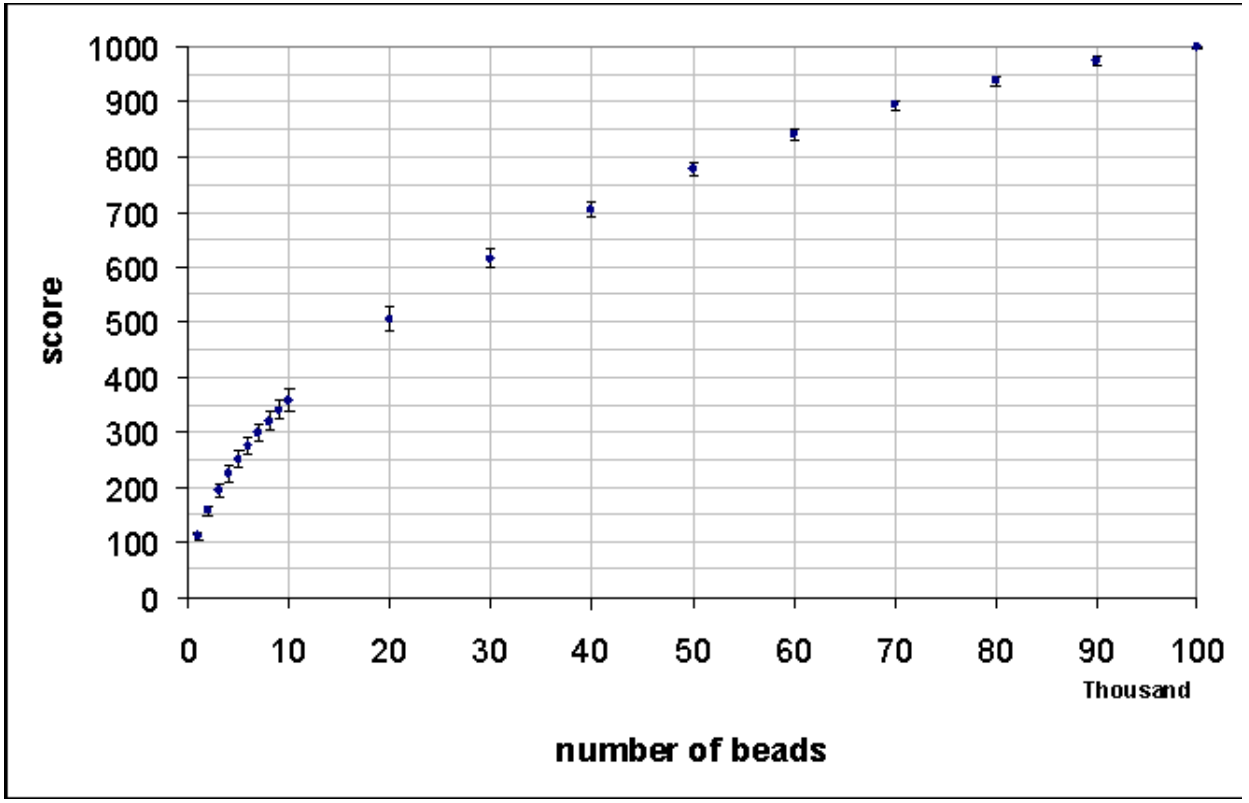

Figure 7.5: Impact of the number of beads on performance. The graph summarizes the results of the lookahead algorithm with different number of beads on real data sets of 1000 strings. The average (dot), minimum and maximum (bars) are shown for each number of beads.

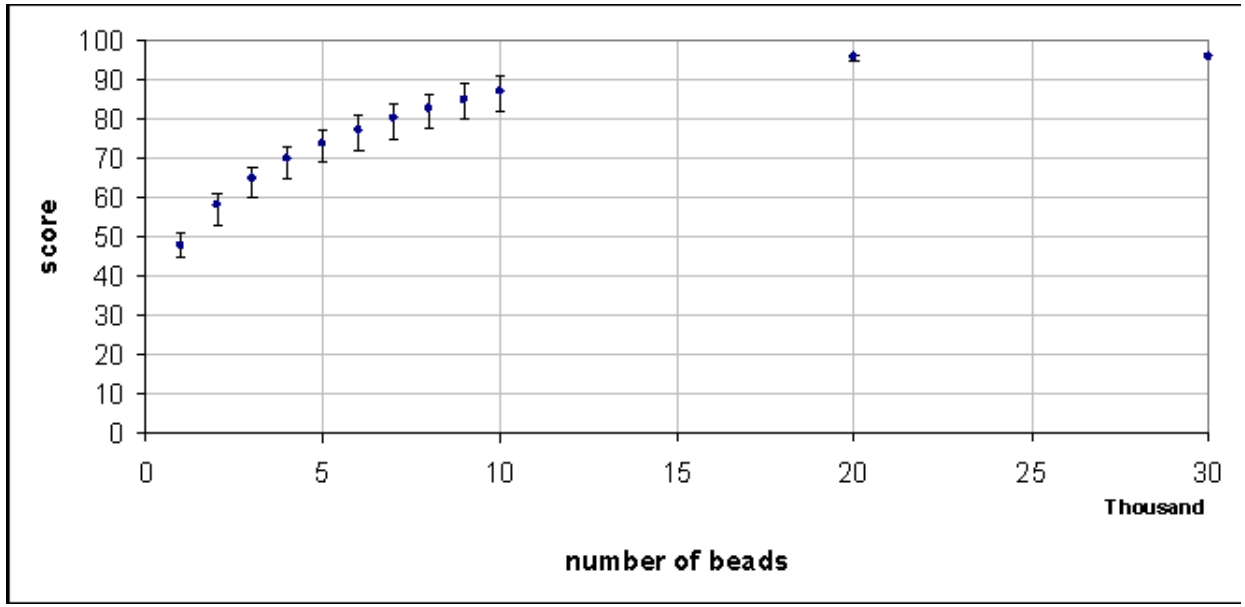

Figure 7.6: Impact of the number of beads on performance. The graph summarizes the results of the lookahead algorithm with different number of beads on real data sets of 96 strings. The average (dot), minimum and maximum (bars) are shown for each number of beads.

From the above results, it is clear that increasing either the width or the number of beads improves the obtained results. The effect of increasing the number of beads is sublinear, while increasing the width first causes a rapid exponential-like rise of the score, and then a slow, sublinear rise. The rapid rise continues until the alphabet size is reached, as each new letter which is added for the first time accommodates many of the target sequences. After all the letters are present, the effect of additional copies of the same letters is more modest. In Figures 6.7 and 6.8 we explore the tradeoff between those two parameters. Such plots allow the experimentalist to make the most convenient choice of the two parameters in order to achieve a desired number of target strings.

Finally, as our algorithm is deterministic and greedy, we explored its stability by trying different starting points of the algorithm. In order to achieve this goal, we ran the algorithm using subsets of the large sets of real data as input. Each subset contained 900 strings that were randomly chosen out of the full set of 1000 strings. Note that the solution of the algorithm on the full set of 1000 strings had no more than 400 strings, thus taking only 900 strings as input does not necessarily imply a decrease in the obtained result. The initialization according to each subset gives a different initial graph, and hence a different order of arc deletions, which results in a different output. This experiment was repeated for each of the first 10 large data sets with 950 different randomly chosen subsets of 900 strings. The results of those experiments show that our algorithm is very stable, at it always performed better than all other initializations. A summary of the results is presented in Table 6.9 and Figure 6.9.

In order to test the utility of using the similarity score, we compared the success of our algorithm on randomly chosen subsets of the given data sets and on the subsets chosen by their high similarity. The results of this test are presented in Table 6.9 and Figure 6.9. We can see that subsets chosen by their similarity *always performed better than randomly chosen subsets*; and more surprisingly, in 8 out of 10 cases, they also give better results than the algorithm on the *full set* of 1000 strings.

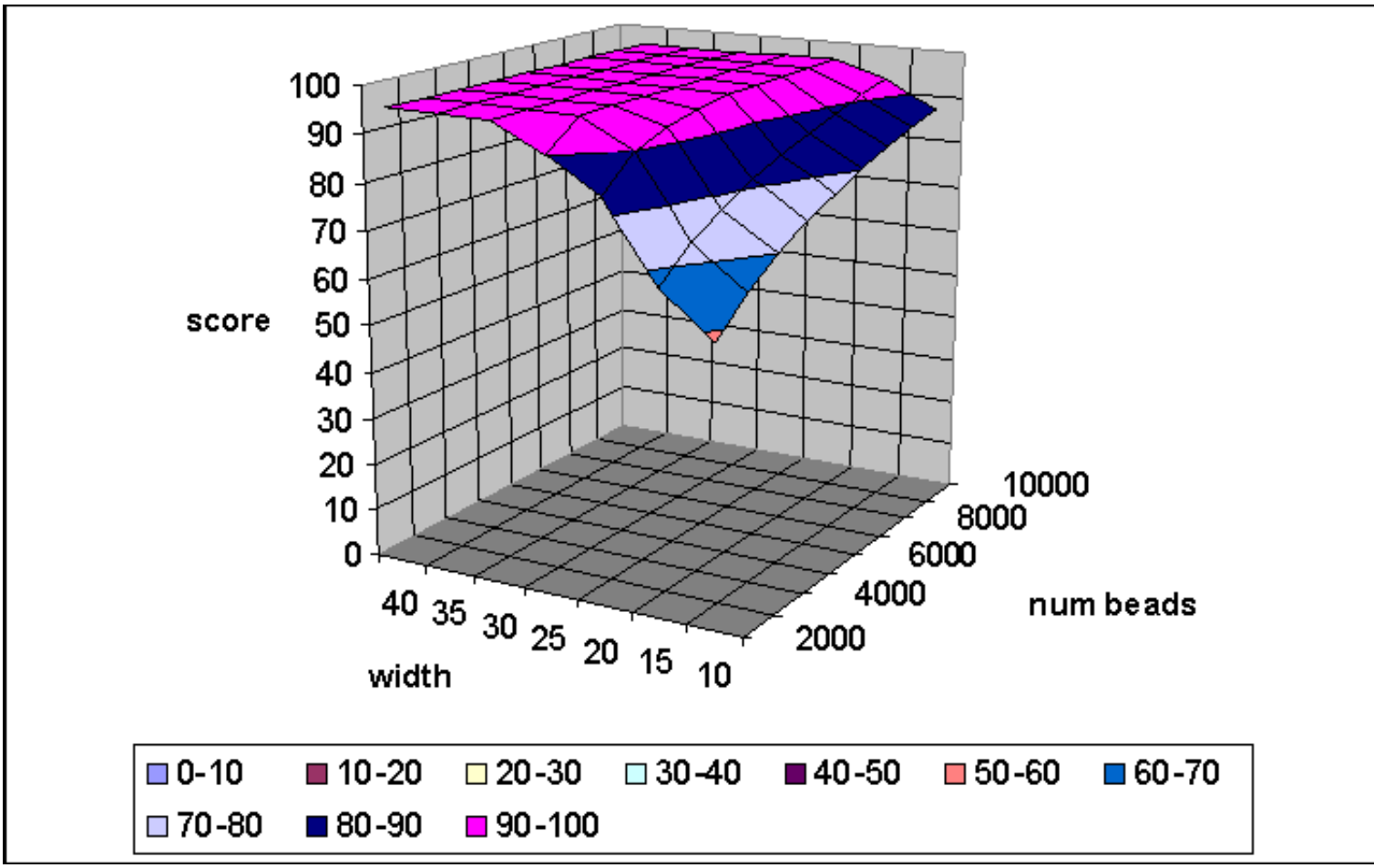

Figure 7.7: Tradeoff between the width and the number of beads. Performance of the lookahead algorithm with different widths and different number of beads on 96 strings real data sets.

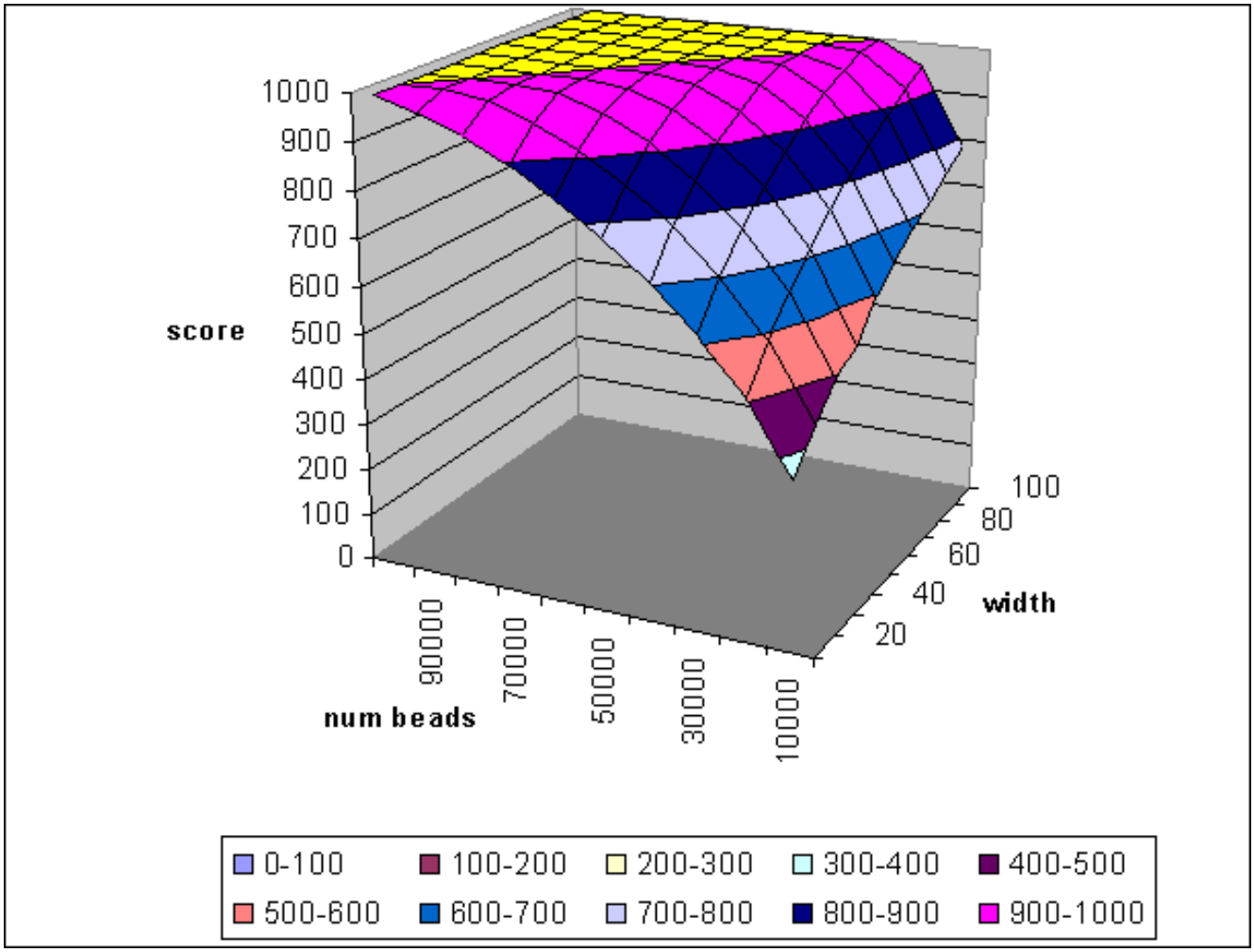

Figure 7.8: Tradeoff between the width and the number of beads. Performance of the lookahead algorithm with different widths and different number of beads on 1000 strings real data sets.

	average	max	min	1000	similarity
1000 real data1	349.59	366	337	377	381
1000 real data2	344.01	362	330	368	375
1000 real data3	326.18	340	313	350	349
1000 real data4	341.76	358	327	371	373
1000 real data5	329.08	342	314	357	354
1000 real data6	333.47	347	316	360	364
1000 real data7	345.65	358	328	373	376
1000 real data8	336.98	350	322	368	369
1000 real data9	325.39	339	309	341	355
1000 real data10	323.39	337	309	347	352

Table 7.9: Impact of partial sets. In order to achieve different starting points, the algorithm was applied to 950 different subsets of 900 target strings, which were chosen out of each 1000 strings real data set, and statistics were collected. Columns 2,3,4 give the average, maximum and minimum score obtained. Column 5 repeats for comparison the result on the full set. Column 6 gives the result on a set chosen according to the similarity score.

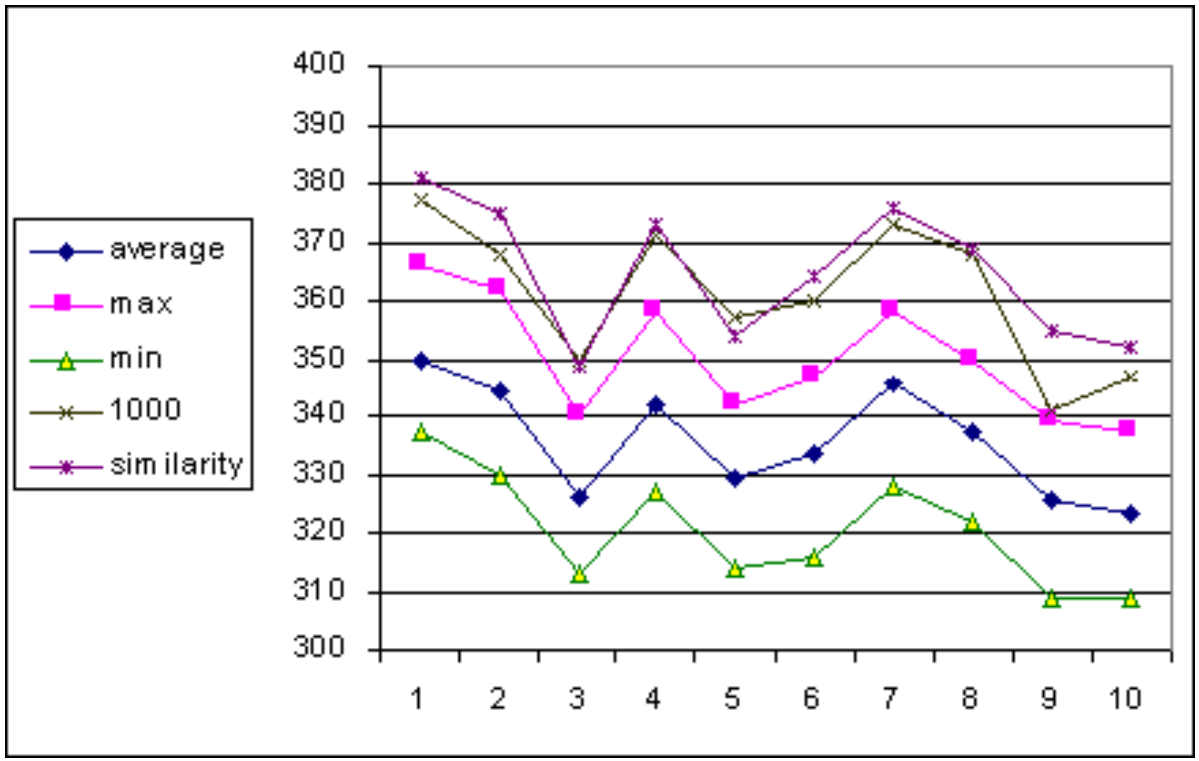

Figure 7.9: Impact of partial sets – graphical summary. The setup is as for Table 6.9. The x axis gives the ten large data sets in arbitrary order. The y axis gives the number of produced target strings.

Chapter 8

Future Work

In this chapter we suggest two directions for future work. One is combining library selection with the design of a synthesis graph producing it; and the other is establishing tight upper bounds for the solutions to the various problems we explored.

8.1 Combining Diverse Library Selection with Synthesis Graph Design

As discussed in Section 3.6 there are many target sets which represent the parent library equally well. However, these target sets may vary greatly in their synthesis complexity. Hence, we suggest exploring possibilities of combining library selection with the design of a synthesis graph producing it. That is, choosing a library that not only represents the parent library, but can also be efficiently synthesized.

8.2 Upper Bounds

Another important issue is the establishment of tight upper bounds for the solutions to the various problems we explored.

Worst case upper bounds are easily obtained in some cases. Consider a target set S in which the letters in each of the strings' positions are distinct. In the applications we address this is indeed a realistic example: For example, when the alphabet is composed of unnatural amino acids its size can reach a few thousands. Clearly the strings in S must be synthesized individually, *i.e.*, by parallel synthesis. Therefore, the minimum solution to MNES is of size (*i.e.*, a number of internal nodes) $(l-2)\cdot|S|$, where l is the length of the strings in S. Likewise, a maximum solution to MSS produces w strings out of S, where w is the constraint on the width of the solution graph (assuming w is at most $|S|$).

A more interesting question is the establishment of average case upper bounds. However, even identifying the average case is a difficult problem. The target set which is an input to the problems we explored is designed to represent some parent library. It is definitely not a random set. In fact the properties of such a set are still the subject of extensive research.

The ultimate goal is the establishment of specific tight upper bounds. That is, for each target set S bounding its synthesis requirements: in terms of the number of required nodes for MNES, and in terms of the number of strings which can possibly be synthesized for MSS.

Bibliography

[1] F. Balkenhohl, C. von dem Bussche-Hunnefeld, A. Lansky, and C. Zechel. Kombinatorische synthese. *Angrew. Chem. Int. Ed. Engl.*, 35:2288–2337, 1996.

[2] D.P. Bertsekas and J.N. Tsitsiklis. Gradient convergence in gradient methods with errors. *SIAM Journal in Optimization.*, 10(3):627–642, 2000.

[3] C.M. Bishop. *Neural Networks for Pattern Recognition.* Oxford University Press, Oxford, England, 1995.

[4] T. Brants. Estimating HMM topologies. *In Proceedings of the Tbilisi Symposium on Language, Logic, and Computation. Tbilisi, Georgia.*, 1995.

[5] R.D. Brown. Descriptors for diversity analysis. *Perspect. Drug Discovery*, 7-8:31–49, 1997.

[6] R.D. Brown and Y.C. Martin. Use of structure-activity data to compare structure based clustering methods and descriptors for use in compound selection. *J. Chem. Inf. Comput*, 36:572–584, 1996.

[7] R.D. Brown and Y.C. Martin. The information content of 2D and 3D structural descriptors relevant to Ligand-Receptor binding. *J. Chem. Inf. Comput*, 37:1–9, 1997.

[8] I.C. Choong and J.A. Ellman. Solid-phase synthesis: Application of combinatorial libraries. *Annu. Rep. Med. Chem.*, 31:309–318, 1996.

[9] B. Cohen and S. Skiena. Efficient split synthesis for targeted libraries. *J. Comb. Chem.*, 2(1):10–18, 2000.

[10] J.A. Ellman. Strategy and tactics in combinatorial organic synthesis. applications to drug discovery. *Acc. Chem. Res.*, 29:132–143, 1996.

[11] Molecular Simulations Inc., Cerius2 Modeling Environment, 1999, Release 4.0, San Diego.

[12] J.S. Fruchtel and G. Jung. Organic chemistry on solid supports. *Angrew. Chem. Int. Ed. Engl.*, 35(1):17–42, 1996.

[13] A. Furka, F. Sebestyen, M. Asgedom, and G. Dibo. General method for rapid synthesis of multicomponent peptide mixtures. *Int. J. Pept. Protein Res.*, 37:487–493, 1991.

[14] M.A. Gallop, R.W. Barrett, W.J. Dower, S.P.A. Fodor, and E.M. Gordon. Applications of combinatorial technologies to drug discovery. 1. background and peptide combinatorial libraries. *J. Med. Chem.*, 37:1233–1251, 1994.

[15] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Co., San Francisco, 1979.

[16] E.M. Gordon, R.W. Barrett, W.J. Dower, S.P.A. Fodor, and M.A. Gallop. Applications of combinatorial technologies to drug discovery. 2. combinatorial organic synthesis, library screening strategies, and future directions. *J. Med. Chem.*, 37:1385–1399, 1994.

[17] E.M. Gordon, M.A. Gallop, and D.V. Patel. Strategy and tactics in combinatorial organic synthesis. applications to drug discovery. *Account. Chem. Res.*, 29:144–154, 1996.

[18] E.M. Gordon and J.F. Jr. Kerwin. *Combinatorial Chemistry and Molecular Diversity in Drug Discovery.* Wiley, Ney York, 1998.

[19] J. Herbst. An inductive approach to the acquisition and adaptation of workflow models. *16th International Joint Conference on Artificial Intelligence, (IJCAI '99), Stockholm*, July 31 - August 6 1999.

[20] G. Jung. *Combinatorial Peptide and Nonpeptide Libraries.* Wiley, Europe, 1996.

[21] K.S. Lam, S.E. Salmon, E.M. Hersh, V.J. Hruby, W.M. Kazmierski, and R.J. Knapp. A new type of synthetic peptide library for identifying ligand- binding activity. *Nature*, 354:82–84, 1991.

[22] D. Madden, V. Krchnak, and M. Lebl. Synthetic combinatorial libraries: Views on techniques and their application. *Persp. Drug Discovery and Design*, 29:269–285, 1995.

[23] G.M. Maggiora and M.A. Johnson. *Concepts and Applications of Molecular Similarity.* Wiley, Ney York, 1990.

[24] E.J. Martin, J.M. Blaney, M.A. Siani, D.C. Spellmeyer, A.K. Wong and, and H. Moos. Measuring diversity: experimental design of combinatorial libraries for drug discovery. *J. Med. Chem.*, 38:1431–1436, 1995.

[25] A. Nefzi, J.M. Ostresh, and R.A. Houghten. The current status of heterocyclic combinatorial libraries. *Chem. Rev.*, 97:449–472, 1997.

[26] M.R. Pavia, T.K. Sawyer, and W.H. Moos. The generation of molecular diversity. *Bioorg.Med.Chem.Lett.*, 3:387–396, 1993.

[27] A. Stolcke and S. M. Omohundro. Best-first model merging for hidden Markov model induction. Technical Report TR-94-003, Computer Science Division, University of California at Berkeley and International Computer Science Institute, 1994.

[28] L.A. Thompson and J.A. Ellman. Synthesis and applications of small molecule libraries. *Chem. Rev.*, pages 555–600, 1996.

[29] D. Woelk, P. Attie, P. Cannata, G. Meredith, A. Sheth, M. Singh, and C. Tomlinson. Task scheduling using intertask dependencies in Carnot. pages 491–494, 1993.

[30] P. Zhao, R. Zambias, J.A. Bolognese, D. Boulton, and K. Chapman. Sample size determination in combinatorial chemistry. *Proc. Natl. Acad. Sci USA*, 92:10212–10216, 1995.