Algorithmic Strategies in Combinatorial Chemistry

Deborah Goldman* Sorin Istrail† Giuseppe Lancia‡ Antonio Piccolboni§
Brian Walenz¶

Abstract

Combinatorial Chemistry is a powerful new technology in drug design and molecular recognition. It is a wet-laboratory methodology aimed at "massively parallel" screening of chemical compounds for the discovery of compounds that have a certain biological activity. The power of the method comes from the interaction between experimental design and computational modeling. Principles of "rational" drug design are used in the construction of combinatorial libraries to speed up the discovery of lead compounds with the desired biological activity.

This paper presents algorithms, software development and computational complexity analysis for problems arising in the design of combinatorial libraries for drug discovery. We provide exact polynomial time algorithms and intractability results for several Inverse Problems – formulated as (chemical) graph reconstruction problems – related to the design of combinatorial libraries. These are the first rigorous algorithmic results in the literature. We also present results provided by our combinatorial chemistry software package OCOTILLO for combinatorial peptide design using real data libraries. The package provides exact solutions for general inverse problems based on shortest-path topological indices. Our results are superior both in accuracy and computing time to the best software reports published in the literature. For 5-peptoid design, the computation is rigorously reduced to an exahustive search of about 2% of the search space; the exact solutions are found in a few minutes.

1 Introduction

1.1 The Combinatorial Chemistry Framework

Chemical Indices and Inverse Design Problems based on them. The area of quantitative structure-activity relationship (QSAR) identified for chemical compounds various measures, or indices, that provide correlations with the likelihood of biological activity. There are 2D measures (at the level of the chemical graph) and 3D measures (at the level of coordinates for its atoms in the 3D space). In our context, "biological activity" is a complex process of molecular recognition, binding, and possible conformation change between one small compound, and a large biological complex (e.g., a protein complex). It is very difficult to capture the notion of biological activity within the framework of numerical measures at the compound level. However, some measures were found that that work well. One notorious example is the Wiener index defined as the sum of pairwise shortest path distances between atoms in the chemical graphs of the compound. It correlates with physicochemical characteristics such as the boiling point. A variety of chemical topological (2D) and topographical (3D) indices were introduced and much research was performed towards the understanding of their correlation with various types of activities.

A chemical index is a map from the set of chemical compounds to the Real numbers. One could think of the co-domain of this function as the "activity space". Compounds with similar activity are mapped "close" in the space. Typically huge numbers of compounds are mapped to identical, or near identical index values. In a natural way, given some activity level/value, or a region in the activity space, one wants to design chemical compounds having that index value, or whose index is in that region. Solving these types of *inverse problems* is the subject of our paper. The input data for these computational problems are laboratory experiments, where some lead compounds were identified. The problem is to generate new laboratory experiments that will accelerate the likelihood of discovering new, more powerful, compounds. In order to do so we have to solve inverse problems based on specific indices. One wants several solutions for the inverse problem that are as "diverse" (different chemcial structure) as possible. Based on them, a new combinatorial library is created, and new lead compounds are discovered.

*UC Berkeley, dgoldman@cs.berkeley.edu
†Sandia National Laboratories, scistra@cs.sandia.gov
‡University of Padova, lancia@dei.unipd.it
§UC Davis, piccolbo@ucdavis.edu
¶Sandia National Laboratories, bwalenz@cs.sandia.gov

Chemical Graph Reconstruction Problems. New types of graph reconstruction problems occur in this area whose solutions are needed for the design of combinatorial libraries. One type involves constructing graphs or trees having a given topological index. A second type involves selecting chemical fragments from a library and creating "artificial proteins", called *combinatorail peptides*, that match a given index.

1.2 Algorithmic Challenges

In this paper we will consider in particular the Wiener index (the sum of the distances in the graph between each pair of vertices), which is probably the most widely known ([1]).

The Wiener index, W, was devised by the chemist Harold Wiener in 1947 [2], who found a strong correlation between W and a variety of physical and chemical properties of alkanes, alkenes and arenes.

With respect to the inverse problem on unrestricted graphs, we will show that in general it has a simple solution both in its decision (does a graph with a given Wiener index exist?) and construction versions. The problem however becomes more complex if we add the constraint that the graph must be a tree. For this case we give a pseudo-polynomial dynamic programming procedure which builds a tree with a given Wiener index (if one exists), but we do not know the complexity of the decision version. While analyzing the inverse problem on trees, we come to the definition of a new interesting topological property, that is the *loads distribution* for the edges. We show that finding a tree whose edges have given load values is NP-complete, and describe a search procedure which solves the problem very quickly in practice.

As far as the construction of peptoids is concerned, our work focuses on inverse problems based on 2D and 3D QSAR descriptors (which include the Wiener index, but also the Atom Pairs, the Bemis-Kuntz histogram of triangles) that have been proven effective in a number of projects for selecting active molecules from large databases. Formulated as graph reconstruction problems, a typical inverse problem is defined as follows. Given a combinatorial library for peptides with N units, with fragment libraries for every position of maximum size L and an integer W, find a set of high diversity peptides whose Wiener index is W.

We present a polynomial time algorithm, based on dynamic programming, for such inverse problem. Further, we describe a software implementation of a search algorithm, capable of finding all possible solutions, that outperform the existing methods proposed in the literature (see e.g. [3, 4, 5, 6]). Our strategy is based on an effective pruning of the search space, via the introduction of a simple computational filter – the flower compression – and show how it can be used to group many graphs which have related Wiener indices and discard, at once, whole families of unfeasable solutions without examining their members in detail. Our algorithms can be easily generalized to find all (or any) feasible molecule whose topological index of interest is within some given range from a specific target. Our software package OCOTILLO contains the implementations of several algorithms that exactly solve inverse problems based on general shortest-paths indices.

1.3 Previous Work

Combinatorial chemistry research started in the early 1990s (see [5, 7, 8, 9] for early develoments and history).

A lot of studies were devoted topological indices and correlations with biological activity [10, 11, 12, 13, 14], including an entire book "Chemical Graph Threory", N. Trinajstic [15].

Heuristic approaches to combinatorial chemistry design problems are discussed in [3, 16].

1.4 An outline of the paper

The remainder of the paper is organized as follows. In section 2 we introduce some suitable notation. Section 3 is devoted to the inverse Wiener index problem for general graphs (subsection 3.1) and trees (subsection 3.2). Section 4 discusses the problem of reconstructing a tree from its set of splits. In section 5 we address the problem of building a peptoid with a given Wiener index. Subsection 5.1 contains a polynomial algorithm, based on dynamic programming, for finding one such peptoid, while subsection 5.2 describes a fast search procedure capable of listing all feasible solutions and reports on our computational results of the OCOTILLO package.

2 Preliminary Definitions

DEFINITION 2.1. *Given a graph $G = (V, E)$, by $d_G(i, j)$ we denote the shortest path (i.e. with the smallest number of edges) between two vertices i and j. If G is a tree, then $d_G(i, j)$ is the length of the unique path between i and j. We simply write $d(i, j)$ if the graph or tree is understood from the context.*

As is customary, we may often denote by n, or $n(G)$, the number of nodes of a graph. We denote by K_n the complete graph on n nodes. S_n is a star on n nodes (all nodes but one are leaves). P_n is a path of n nodes.

For ease of notation, in the following definition and in the remainder of the paper, when we write $\sum_{i,j \in V}$, the summation has to be understood as actually re-

DISCLAIMER

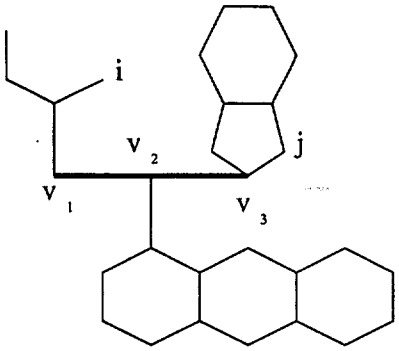

Figure 1: A 3-peptoid; the three fragments are anchored on a linear scaffold at positions v_1, v_2 and v_3.

stricted to pairs of *distinct* vertices.

DEFINITION 2.2. *Given a graph $G = (V, E)$, its* Wiener index *$w(G)$ is the total node–to–node path length. That is, $w(G) = \sum_{i,j \in V} d_G(i, j)$.*

The following graphs are used to describe formally the problem of the combinatorial synthesis of specific molecular structures.

DEFINITION 2.3. *A* (chemical) fragment *is a graph G with a special vertex v denoted as its* anchor, *or* hooking point. *A* peptoid *is a graph obtained by joining in a linear fashion from left to right, k fragments $G_1, \ldots, G_k$ via a path through their hooking points (Figure 1). Note that, when $k = 1$, a fragment is a special case of a peptoid. For a peptoid $D = (V, E)$, by $l(D) := \sum_{i \in V} d_G(i, v_k)$ we denote the total distance of all vertices from the rightmost hooking point v_k. For $k = 1$, $l()$ gives the total distance from all nodes of a fragment to its anchor.*

We can think of a rooted tree as a special case of fragment whose hooking point is its root. Henceforth we have the following definition for rooted trees.

DEFINITION 2.4. *Given a tree $T = (V, E)$ with root $v \in V$, the total distance of its vertices from the root is $l(T) := \sum_{i \in V} d(i, v)$.*

3 The Inverse Wiener Index Problem

We have developed graph theoretic results for the reconstruction problem based on the Wiener index.

3.1 The inverse Wiener index problem for graphs

THEOREM 3.1. *For any $W \neq 2, 5$ there exists a graph G such that $w(G) = W$*

In order to prove this theorem, we need the following lemma:

LEMMA 3.1. *For every graph $G = (V, E)$ with diameter 2 and Wiener index W, the graph $G' = (V, E \cup \{e\})$ for $e \notin E$ has Wiener index $W - 1$.*

Proof. Let $e = (v_1, v_2)$. Clearly $d_G(v_1, v_2) = 2$ and $d_{G'}(v_1, v_2) = 1$. Any other distance is preserved by this transformation. ■

We are now ready to prove Theorem 3.1.

Proof. Let $G_0 = S_n$, the star of size n. We have $w(G_0) = (n-1)^2$ and the diameter of G_0 is two. Let G_1 be the graph obtained by adding to G_0 an edge not already contained in it. G_1 is either K_n or has diameter two, and by the above lemma $w(G_1) = w(G_0) - 1$. It it possible to repeat this procedure until the graph obtained is K_n and $w(K_n) = n(n-1)/2$. At any step, the lemma guarantees that $w(G_k) = w(G_{k-1}) - 1$. Thus each number in the interval $I_n = [n(n-1)/2, (n-1)^2]$ is the Wiener index of G_k for some k.

Since the intervals overlap for $n > 4$, and including the interval values for $n = 4$, we find for $W > 5$ there is a graph G such that $w(G) = W$. 1,3 and 4 are the Wiener index of P_2 (a path of length 2), K_3 and P_3, resp. To prove that there is no graph G such that $w(G) = 2$, it is enough to observe that the graph on n nodes with the smallest Wiener index is K_n, and the one with the largest is P_n, but $w(P_2) = 1$ and $w(K_3) = 3$. ■

The theorem is constructive and leads in a straightforward way to an algorithm solving the search problem, that is outputting a graph with the required given index. Since the size of the graph is polynomial in the Wiener index and a number can be represented with a logarithmic number of bits, this algorithm can be classified as pseudo-polynomial — that is, it is polynomial in the parameters describing the problem but not on the size of the representation of the input. More in detail, the computation time is dominated by the time necessary to output the graph, that is $O(n^2)$. Since for the class of graphs considered $n(n-1)/2 \leq W \leq (n-1)^2$, the time complexity is also $O(W)$.

3.2 The inverse Wiener problem for trees

The problem we will be concerned with in this subsection is the following: given a positive integer W, find whether there exists a tree T s.t. $w(T) = W$. We will consider also the problem of finding such a tree. Clearly, Theorem 3.1 involves non-trees, and thus it does not apply to this more constrained setting. Indeed, there are many integers that are the Wiener index of some graph but not of any tree. Using an algorithm we

will describe in the following, we checked exhaustively for $W < 10000$ and 159 turns out to be the largest such example. This experimental evidence together with the analogy with the case of graphs leads to the following conjecture:

CONJECTURE 3.1. *Every positive integer but a finite set*[1] *is the Wiener index of some tree.*

The above conjecture appeared first in [17], where it was verified for W up to 1206 by a complete enumeration of all unlabeled non isomorphic trees of up to 20 nodes. If true, it would imply that the decision problem is trivial, but the proof would not necessarily lead to an efficient solution of the search problem.

3.2.1 A recurrence relation for the Wiener index

It is possible to prove a recurrence relation for the Wiener index of trees which is closely related to the one we will prove for peptoids in 5.1. Let $T = (V, E)$ be a tree and (v_1, v_2) an edge. Let $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ be the two trees obtained by removing (v_1, v_2). Let us assume that T and T_1 are rooted in v_1 and T_2 in v_2. We have the following recurrence for $w(\cdot)$, $l(\cdot)$ and $n(\cdot)$:

THEOREM 3.2.

$$
\begin{aligned}
(3.1)\quad n(T) &= n(T_1) + n(T_2) \\
(3.2)\quad l(T) &= l(T_1) + l(T_2) + n(T_2) \\
(3.3)\quad w(T) &= w(T_1) + w(T_2) + l(T_1)n(T_2) + \\
&\quad l(T_2)n(T_1) + n(T_1)n(T_2)
\end{aligned}
$$

Proof. 3.1 is obvious. To prove 3.2 we use the definition of $l(\cdot)$ and rearrange the summations slightly, as follows:

$$
\begin{aligned}
l(T) &= \sum_{v \in V} d(v_1, v) \\
&= \sum_{v \in V_1} d(v_1, v) + \sum_{v \in V_2} d(v_1, v) \\
&= \sum_{v \in V_1} d(v_1, v) + \sum_{v \in V_2} (d(v_2, v) + 1) + 1 \\
&= l(T_1) + l(T_2) + n(T_2)
\end{aligned}
$$

The same technique leads to the proof of 3.3

$$
\begin{aligned}
w(T) &= \sum_{v,w \in V} d(v, w) \\
&= \sum_{v,w \in V_1} d(v, w) + \sum_{v,w \in V_2} d(v, w) + \\
&\quad \sum_{v \in V_1, w \in V_2} d(v, w) \\
&= w(T_1) + w(T_2) + \\
&\quad \sum_{v \in V_1, w \in V_2} (d(v, v_1) + 1 + d(v_2, w)) \\
&= w(T_1) + w(T_2) + l(T_1)n(T_2) + \\
&\quad l(T_2)n(T_1) + n(T_1)n(T_2)
\end{aligned}
$$

■

3.2.2 A dynamic programming algorithm for the inverse Wiener index problem

This recurrence relation leads naturally to a dynamic programming algorithm for the problem of finding a tree T with assigned $w(T)$, $l(T)$ and $n(T)$. The key observation is the following: every tree with at least one edge can be decomposed in the way dictated by the above recurrence, that is by removing an edge. Whatever the edge removed, we obtain two trees $T_i, i = 1, 2$, and for each i, $w(T_i) < w(T)$, $l(T_i) < l(T)$ and $n(T_i) < n(T)$. Let us define a matrix M so that $M_{W,L,N}$ be 1 if there is a tree T such that $w(T) = W$, $l(T) = L$, and $n(T) = N$, 0 otherwise. According to the above recurrence $M_{W,L,N}$ can be computed if $M_{W',L',N'}$ is known for every $W' < W$, $L' < L$ and $N' < N$. This implies that it is possible to compute the entries of M, starting form the initial value $M_{0,0,1} = 1$ and evaluating to 0 all the entries corresponding to W, L, N values outside feasible bounds, proceeding in an orderly fashion.

This algorithm solves as well the inverse Wiener index problem: given W, we compute upper bounds for the largest L and N such that the triple (W, L, N) is feasible. Then we fill the matrix M up to the entry $M_{W,L,N}$. If for any $L' \leq L, N' \leq N$ $M_{W,L',N'} = 0$ then there is no tree T such that $w(T) = W$.

The algorithm can be extended so as to return a tree with the required properties: as it is customary in dynamic programming, it is enough to store, whenever an entry of M is set to 1, the indexes of the two entries to which the recurrence relation has been successfully applied.

In our implementation we use a technique related to dynamic programming called *memoization*. Instead of filling the matrix M bottom up, this technique applies recursively the recurrence relation. To avoid recomputation of the same entries, intermediate results

[1] Namely:{2, 3, 5, 6, 7, 8, 11, 12, 13, 14, 15, 17, 19, 21, 22, 23, 24, 26, 27, 30, 33, 34, 37, 38, 39, 41, 43, 45, 47, 51, 53, 55, 60, 61, 69, 73, 77, 78, 83, 85, 87, 89, 91, 99, 101, 106, 113, 147, 159}

get stored in M. It can be thought of as a top-down version of basic dynamic programming. It is worth noting that without storage of intermediate results the time complexity would blow-up exponentially, because of the repeated recomputation of the same entries in M. This technique is valuable when an algorithm can be terminated without filling completely the matrix M. Otherwise, the number of entries evaluated is the same, but there is a slight overhead due to function calls and stack management. For our problem, memoization turns out to be much faster for "yes" instances. For example, (524,36,19) is a "yes" instance and requires less than one second to compute, while (525,36,19) is a "no" instance, requiring 145 seconds. This example is rather extreme, but this behavior is absolutely consistent. This evidence prompts for further research along different lines:

- quantify and analyze this asymmetry between "yes" and "no" instances;
- exploit it to make the computation more efficient (is it safe to "give up" after a reasonably short running time? In exploiting the recurrence, is it faster to compute many entries in parallel and stop when the first successful computation is over?)

As a further algorithmic refinement, already exploited in the above mentioned experimental results, we adopt also a divide and conquer strategy, whenever possible. Since there are many possible ways of using the recurrence relation, we try first the ones for which $n(T_1) \simeq n(T_2)$. This way we proceed directly to the smallest possible sub-problems. This approach is ineffective in the worst case (consider $W = (N-1)^2$, $L = N-1$, that is a star on N nodes), but suggests a sensible order in which to proceed.

The pseudo-code is given in Appendix A.

3.2.3 Recurrence relations for the Wiener index of bounded degree trees and k-ary trees

Often the graphs of molecular structures have intrinsic constraints on the degree of the nodes. For instance, when the nodes represent individual atoms and edges chemical bonds between them, we obtain a graph whose maximum degree is not greater than 4 ([15]).

Unfortunately, Theorem 3.1 does not apply to bounded degree graphs. On the contrary, the use of graphs with high degree seems essential to its proof. The situation is better for trees, since we can develop recurrence relations of the same kind of the one in Theorem 3.2, and this recurrences lead to dynamic programming algorithms similar to the one just shown. Let us first deal with bounded degree trees. Besides the quantities used so far — $w(\cdot)$, $l(\cdot)$ and $n(T)$ — we need two more definitions. Let mdeg$(\cdot)$ be the maximum degree of a tree and rdeg$(\cdot)$ the degree of its root. As for Theorem 3.2, let $T = (V, E)$ be a tree and (v_1, v_2) an edge. Let $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ be the two trees obtained by removing (v_1, v_2). Let us assume that T and T_1 are rooted in v_1 and T_2 in v_2. We have the following:

THEOREM 3.3.

$$\begin{aligned} mdeg(T) &= \max(mdeg(T_1), mdeg(T_2), \\ &\quad rdeg(T_1)+1, rdeg(T_2)+1) \\ rdeg(T) &= rdeg(T_1)+1 \end{aligned}$$

Together with Theorem 3.2, Theorem 3.3 characterizes the existence of a tree with the required properties and thus can be used to define a dynamic programming algorithm. This time, though, the matrix M containing the partial solutions will have five dimensions, to account also for mdeg$(\cdot)$ and rdeg$(\cdot)$. The worst case bound for the running time has to be updated accordingly.

We turn now to k–ary trees. To develop a recurrence for the Wiener index in this case, we still rely on the quantities that proved useful so far — namely $w(\cdot)$, $l(\cdot)$ and $n(T)$ —, but we decompose a tree in a different way . Instead of using cuts as before we exploit the definition of k–ary tree. Let $T = (V, E)$ be a k–ary tree and let $T_i = (V_i, E_i)$ be the k subtrees hanging from its root. We can prove a yet more complex recurrence for the Wiener index in this case.

THEOREM 3.4.

$$n(T) = \sum_i n(T_i) + 1 \tag{3.4}$$

$$l(T) = \sum_i (l(T_i) + n(T_i)) \tag{3.5}$$

$$\begin{aligned} w(T) &= \sum_i (w(T_i) + l(T_i) + n(T_i)) + \\ &\quad \sum_{i \neq j} l(T_i) n(T_j) + \sum_{i<j} 2n(T_i) n(T_j) \end{aligned} \tag{3.6}$$

The proof is similar to the one for Theorem 3.2 and will be omitted.

4 The SPLITS reconstruction problem

In this section we address the following tree reconstruction problem: Find a tree such that for each edge the sizes of the two shores of the cut that the edge defines are equal to some given input values, or report that no such tree exists. As we will see, this problem is closely related to the inverse Wiener index problem for trees. We start with some definitions.

DEFINITION 4.1. *For a tree $T = (V, E)$ we define the split an edge $e \in E$, denoted by $s(e)$ as the number of*

nodes on the smallest shore of the unique cut identified by e. The load of the edge, denoted by $l(e)$, is the number $s(e) \times (n - s(e))$ of paths in T which contain the edge e.

By using the loads, we can rewrite the Wiener index for a tree as $w(T) = \sum_{e \in E} l(e)$.

The last bit of the Wiener index and the last bit of n are not independent, as the following proposition shows. This result appears also in [17, 18], where it is derived by considering trees as bipartite graphs and arguing on the parity of paths. We give a much simpler proof.

PROPOSITION 4.1. *Any tree with an odd number of nodes has an even Wiener index.*

Proof. For each edge either $s(e)$ or $n - s(e)$ is even, so the load is even. ■

The problem of finding a tree of a given Wiener index asks therefore to find $n-1$ loads whose sum is W. This prompted us to the following question: assume we are given such loads; can we find the tree? Since for a fixed n the loads uniquely determine the splits, we can rephrase the problem as: given splits $s_1, \ldots, s_{n-1}$ find a tree T such that the edges of T have the given input splits. This is a problem of tree reconstruction and the set of splits can be viewed as yet another topological property that characterizes a family of trees. Furthermore, the problem of reconstructing a tree from its set of splits is interesting on its own. Unfortunately, the reconstruction problem turns out to be NP-complete

THEOREM 4.1. *The problem, SPLITS, of reconstructing a tree from its set of splits is NP-complete.*

Proof. We reduce from the problem, 3-PARTITION. In this problem we are given a bound, B, and $3m$ elements, $s_1, \ldots, s_{3m}$, such that for each $i \in \{1, \ldots, 3m\}, B/4 < s_i < B/2$. The problem asks whether there exists a partition of the $\{s_i\}$ into into 3-element disjoint sets such that the sum of the elements in each set is B.

We map the instance of 3-PARTITION to the following instance of SPLITS: the value $B + 1$ appears m times and, for each i, we include the values, $s_i, s_i - 1, \ldots, 1$. If we are given a yes instance of 3-PARTITION, we can build a tree in the following way: the root has m children (an m-star) each corresponding to a 3-element set in the solution to 3-PARTITION and then departing from each of these there are three paths of length equal to the size of items that belong to that set. It can be easily verified that we obtain the given splits. Conversely, suppose we are given a tree with the set of splits listed above. We show that it is necessarily of the form we just described. Inductively, the tree must necessarily contain $3m$ paths of length $\min_i\{s_i\}$ consisting of edges with splits $min\{s_i\}, min\{s_i\} - 1, \ldots, 1$ (for example, each edge with a split of two must necessarily be connected to an edge with a split of one). At this point, we conclude that, in fact, we must have $3m$ similar paths of length s_i each starting from an edge with split s_i (which contain the former paths) since we are now only able to attach loads $\geq min\{s_i\}$ and, by assumption on the s_i sizes, $\max\{s_i\} < 2\min\{s_i\}$: for each edge with split s between $min\{s_i\}+1$ and $max\{s_i\}$, the only edge with smaller load that we can attach must have load exactly $s - 1$. Finally, also from the bounds on the s_i, we infer that exactly three paths depart from each leaf of an initial m-star, the edges of which all have split size $B + 1$. As above, the s_i values of the edges departing from the star edges provide a solution to the instance of of 3-PARTITION. Since the reduction is clearly polynomial time computable, this completes the proof of the theorem. ■

The problem of reconstructing a tree from its set of splits can be solved by the following enumerative algorithm. Sort the splits so as to have $s_1 \geq \ldots \geq s_{n-1} = 1$. Starting with a tree consisting of a single node of weight n, we insert the edges one at a time, ending up with a tree on n nodes, each of weight 1. At step k we

1. look –exaustively– for a node i whose weight w_i is larger than s_k
2. *augment:* attach to node i a new node node j, setting $w_j := s_k$ and decreasing w_i to $w_i - s_k$.

Note that at step 1 we may have to break ties. The presence of these ties is what makes the algorithm exponential, since we may have to backtrack from a wrong choice. It is not immediate that this algorithm does indeed work. For instance, the sorting of the s_i is crucial, as the following example shows: Take $n = 4$ and $s_1 = s_2 = 1$, $s_3 = 2$. Then there is no way of placing the split 2 after having placed the two splits 1. So we need to show that it is enough to consider the sorted permutation of the splits out of the $(n - 1)!$ possibilities.

PROPOSITION 4.2. *If $s_1 \geq \ldots \geq s_{n-1}$ is a YES instance of SPLITS, then the algorithm terminates with a feasible solution.*

Proof. We may reason backwards by starting from the tree and finding the correct sequence of nodes to augment. Let T be a feasible solution. Give weight 1 to each node of T and repeat the following operation, for

$k = 1$ to $n-1$, until T has only one node. Take a leaf i of T of minimum weight among the leaves. Let $(i, j(k))$ be the unique edge out of i. Delete node i and increase $w_{j(k)}$ as $w_{j(k)} := w_{j(k)} + w_i$. By looking backwards at the sequence of trees thus obtained, we see a possible run of the algorithm which augments on $j(k-1), \ldots, j(1)$ creating edges of decreasing splits. ■

This argument also implies that for YES–instances there always exists a choice of nodes to augment which requires no backtrack, and indeed this is what happened on the vast majority of small examples which we tried initially, before proving that the problem is NP-complete. We then performed a more exaustive testing in the following way. We generate an unlabeled tree, uniformly at random (as described in [19]), then compute its splits and try to reconstruct it (or a different feasible solution).

Ten instances for each value of $n = 10, 20, \ldots, 100$ were solved immediately, while for $n \geq 110$ the algorithm started incurring in some long runs every once in a while. By performing the selection at step 1 in an orderly fashion (i.e. try the available nodes by increasing order of weight) we solved all generated problems, for n up to 300, in less than 1 second each. The good average performance raises an interesting theoretical question on the probability that the search algorithm may find a solution withouth backtracking (or within a small number of tries) on a tree generated u.a.r.

5 Inverse Problems for Peptoid Design

In this section we consider the following problem. In the framework of combinatorial chemistry we are given a fragment library, and values (lists, histograms) for some index. We want to find combinatorial peptoids (a compound of elements from the given library) that match exactly that index.

5.1 A dynamic program for peptoid construction

THEOREM 5.1. *One can compute in polynomial time whether there exists a peptoid with a given Wiener index, W, and, if so, output a solution.*

Proof. We use a dynamic programming algorithm similar to the algorithm given to find a tree of a particular Wiener index. Note that W is bounded by a polynomial in the size of the peptoid, N, and the library size, L. Assume we have precomputed the Wiener indices of the fragments in the library. Number the anchors along the peptoid, say from left to right, by 1 through N. We build up our peptoid from left to right by adding a fragment from the library to each anchor sequentially. Let $l(\cdot)$ denote the sum of the distances to the rightmost anchor in a peptoid, or the sum of the distances to the anchor of a fragment from our library (which is just a peptoid with one anchor). Remove the edge linking the rightmost two anchors of a peptoid, P, leaving a smaller peptoid, P', and a fragment, F. The dynamic programming algorithm follows from the recurrences which we present below. Note that by storing one solution (if one exists) in each entry of the table we build, we can output a solution with Wiener index W if one exists. The recurrences follow.

$$\begin{aligned} n(P) &= n(P') + n(F) \\ l(P) &= l(P') + n(P') + l(F) \\ w(P) &= w(P') + w(G) + n(P')l(F) + \\ &\quad n(F)l(P') + n(P')n(F). \end{aligned}$$

■

5.2 A Fast Enumerative Algorithm

In this section, we present a general method for inverse problems based on shortest-paths topological indices. We also present results from our software package OCOTILLO on actual combinatorial libraries.

THEOREM 5.2. *The Wiener index of a linear-scaffold peptoid constructed with fragments (Figure 1) is*

$$W = \sum_{i=1}^{N} \sum_{j=i+1}^{N} [n_i l_j + (j-i) n_i n_j + n_j l_i] + \sum_{i=1}^{N} w_i$$

where n_i is the number of nodes in fragment i, w_i is the Wiener index of the fragment and l_i is the sum of the distance from each node to the anchor.

Proof. Consider the compound in Figure 1. When we compute the shortest path between any two atoms, there are two cases: either the two atoms are in the same fragment, or they are not. For all pairs of atoms that are in the same fragment we pre-compute the sum of the distance between each pair and denote this value w — it is just the Wiener index of the fragment.

For pairs of atoms that are in different fragments, the shortest path between the two atoms is always through the two anchors associated with the fragments. We break this path into three components:

1. The shortest path from atom i to its anchor.
2. The shortest path along the scaffold.
3. the shortest path from atom j to its anchor.

The sum of distances between all pairs of atoms in two different fragments is:

$$\begin{aligned} P(a,b) &= \sum_{i\in F_a}\sum_{j\in F_b} d(i,v_a)+d(v_a,v_b)+d(v_b,j) \\ &= n_b\sum_{i\in F_a} d(i,v_a)+n_an_bd(v_a,v_b)+ \\ &\quad n_a\sum_{j\in F_b} d(v_b,j) \\ &= n_bl_a+n_an_bd(v_a,v_b)+n_al_b \end{aligned}$$

where v_a is the anchor atom of fragment a and n_a is the number of atoms in fragment a.

The Wiener index is now the sum of P over all pairs of fragments, plus the sum of the Wiener index of the individual fragments.

$$\begin{aligned} W &= \sum_{i=1}^{N}\sum_{j=i+1}^{N} P(F_i,F_j)+\sum_{i=1}^{N} w_i \\ &= \sum_{i=1}^{N}\sum_{j=i+1}^{N} [n_il_j+n_in_jd(v_i,v_j)+n_jl_i]+\sum_{i=1}^{N} w_i \end{aligned}$$

If, as in our case, the scaffold is a linear chain, then the distance from the anchor of fragment i to the anchor of fragment j is $|j-i|$, and,

$$W=\sum_{i=1}^{N}\sum_{j=i+1}^{N}[n_il_j+(j-i)n_in_j+n_jl_i]+\sum_{i=1}^{N} w_i$$

■

We can rewrite the Wiener index equation as

$$W=\sum_{i=1}^{N} n_i\sum_{i=1}^{N} l_i+\sum_{i=1}^{N}\sum_{j=i+1}^{N}(j-i)n_in_j+\sum_{i=1}^{N}(w_i-n_il_i)$$

Suppose that we treat the entire scaffold as a single vertex, for example, in Figure 1, we would compress v_1, v_2 and v_3 to a single vertex. In effect, we have constructed a peptoid with an unordered set of fragments, rather than an ordered list of fragments. This is called the *flower compression*, and is at the heart of our fast search method.

THEOREM 5.3. *The Wiener index of a flower-compressed peptoid is*

$$F=\sum_{i=1}^{N}\sum_{j=i+1}^{N}[n_il_j+n_jl_i]+\sum_{i=1}^{N} w_i$$

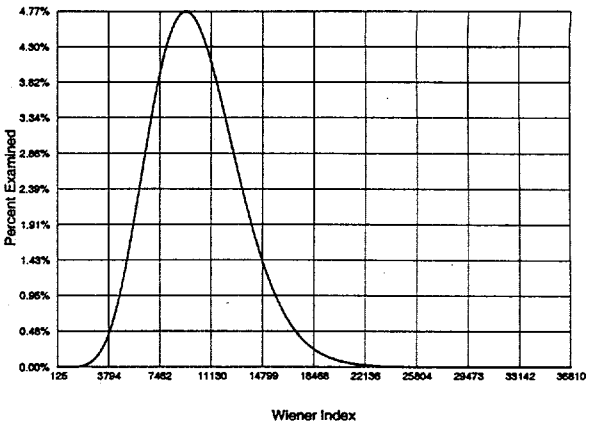

Figure 2: The percent of constructed flower-configuration peptoids that need to be examined in detail. A histogram of the number of peptoids with a specific Wiener index follows almost the same curve, which explains why the pruning algorithm does not prune uniformly for all Wiener index — there are just more peptoids that match.

The proof closely follows that for Theorem 5.2 and is omitted.

We let D be the difference between F on a set of fragments and W on a set of fragments given the ordering π:

$$D(\pi)=W-F=\sum_{i=1}^{N}\sum_{j=i+1}^{N}(j-i)n_{\pi(i)}n_{\pi(j)}.$$

For a given set of fragments, the ordering, π_{min}, with the smallest Wiener index is also the ordering with the smallest value of D. This forms our pruning search. If the Wiener index we are looking for is smaller than $minD+F$ we do not need to check *any* orderings for the correct Wiener index.

for each set of N fragments **do**
  if $minD+F\le W_{target}\le maxD+F$ **then**
    examine all orderings of the set of fragments for W_{target}.
  else
    discard all orderings of the set of fragments.

As a first approximation to the minimum and maximum, we can replace each n_i with the smallest or largest value of n in the peptoid, for example,

$$minD\ge\sum_{i=1}^{N}\sum_{j=i+1}^{N}(j-i)n_{min}^2=\frac{N^3-N}{6}n_{min}^2.$$

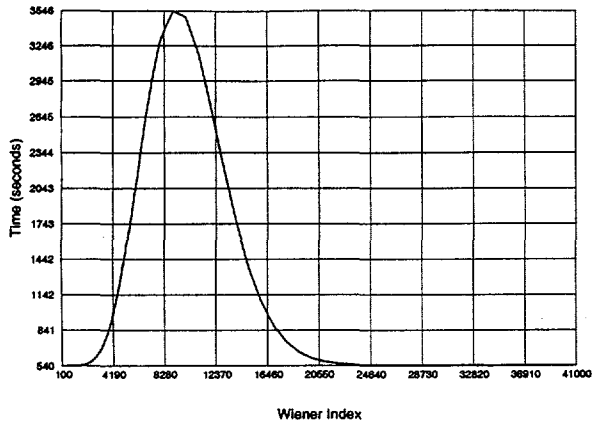

Figure 3: The CPU time required to search.

However, this bound is very weak — at Wiener index 9000, we need to examine 32.8% of the peptoids in detail, compared to 4.7% when using the optimal values for $minD$ and $maxD$.

CONJECTURE 5.1. *Given $n_1 \leq n_2 \leq \ldots \leq n_N$, the ordering for the optimal minimum value of D is:*

$$\pi_{min}(i) = \begin{cases} 2i-1 & \text{if } i \leq N/2 \\ 2(N-i+1) & \text{if } i > N/2 \end{cases}$$

CONJECTURE 5.2. *An algorithm to compute the ordering for the optimal maximum value of D, given $n_1 \leq n_2 \leq \ldots \leq n_N$, is:*

$L_p := 1$; $L := 0$;
$R_p := N$; $R := 0$;
for $i := N$ *downto* 1 ***do***
  if $R \geq L$ ***then***
    $\pi_{max}(L_p) := i$; $L_p := L_p + 1$; $L := L + n_i$;
  else
    $\pi_{max}(R_p) := i$; $R_p := R_p - 1$; $R := R + n_i$;

For example,

i	1	2	3	4	5	6	7	8
n_i	2	5	8	13	17	18	19	28
$n_{\pi_{min}(i)}$	2	8	17	19	28	18	13	5
$n_{\pi_{max}(i)}$	28	18	8	2	5	13	17	19

Both conjectures have been extensively tested.

As seen in Figure 2, not many flower peptoids pass the test.

Computational Results. We tested the performance of the pruning algorithm by searching for a five-fragment peptoid using a fragment library with 350 amine fragments (resulting in 164 different w,l,n values) for each position. This configuration results in 2.6e12 possible peptoids.

A brute force enumeration using the w, l, n computation explained earlier required 51,786 cpu-seconds, or 50.5e6 peptoids per second. For comparison, we can estimate that without the w, l, n computation, the enumeration would be at least $(350/165)^5 \approx 43$ times as long — about one month of cpu time — without even considering that the Wiener index computation is also more difficult. Applying the flower-compression pruning algorithm achieves a significant speedup as can be seen in Figure 3, requring anywhere from 540 seconds (4,860e6 peptoids per second) to 3,500 seconds (750e6 peptoids per second).

6 Acknowledgements

The authors would like to thank Jean-Loup Faulon and Diana Roe for useful discussions regarding this paper. This work was supported in part by Sandia National Laboratories, operated by Lockheed Martin for the U.S. Department of Energy under contract No. DE-AC04-94AL85000 and by the Mathematics, Information, and Computational Science Program of the Office of Science of the U.S. Department of Energy. D. Goldman was supported by an American Fellowship from the American Association of University Women Educational Foundation.

References

[1] Fiftieth Anniversary of the Wiener Index, Discrete Applied Mathematics Special Issue, Vol. 80, no. 1 Gutman, I., Klavzar, S. and Mohar, B. eds., 122 pages, 1997

[2] Wiener, H., Structural determination of paraffin boiling points, *J. Amer. Chem. Soc.*, 69 (1947) 17–20

[3] Sheridan, R., P. and Kearsley, S., K., Using a Genetic Algorithm To Suggest Combinatorial Libraries, *J. Chem. Inf. Comput. Sci.*, 35 (1995) 310–320

[4] Venkatasubramanian, V., Chan, K. and Caruthers, J. M., Evolutionary Design of Molecules with Desired Properties Using the Genetic Algorithm, *J. Chem. Inf. Comput. Sci.*, 35 (1995) 188–195

[5] Gordon, Douglas J., Bellott, Emile M., and Tenenbaum, Boris, Using a Genetic Algorithm to Select an Optimum Combinatorial Library Using a Subset of Available Input Materials, Exploiting Molecular Diversity: Refining Small Molecule Libraries, La Jolla, California, February 1–5, 1999.

[6] Singh, Jasbir et. al., Application of Genetic Algorithms to Combinatorial Synthesis: A Computatorial Synthesis: A Computational Approach to Lead Identification and Lead Optimization, *J. Am Chem. Soc.*, Vol. 118, (1996), 1669–1676

[7] Gallop, Mark A., Barrett, Ronald W., Dover, William J., Fodor, Stephen P. A., Gordon, Eric M., Applications of Combinatorial Technologies to Drug Discovery. Background and Peptide Combinatorial Libraries, *Journal of Medicinal Chemistry*, Vol. 37, No. 9 (1994) 1233–1251

[8] Zheng, Weifan, Cho, Sung Jin, and Tropsha, Alexander, Rational Combinatorial Library Design. 1. Focus-2D: A new Approach to the Design of Targeted Combinatorial Chemical Libraries, *J. Chem. Inf. Comput. Sci.*, Vol. 38, (1998) 251–258

[9] Zheng, Weifan, Cho, Sung Jin, and Tropsha, Alexander, Rational Combinatorial Library Design. 2. Rational Design of Targeted Combinatorial Peptide Libraries using Chemical Similarity Probe and the Inverse SAR Approaches, *J. Chem. Inf. Comput. Sci.*, Vol. 38, (1998) 259–268

[10] Carhart, Raymond E., Smith, Dennis H., and Venkataraghavan, R., Atom Pairs as Molecular Features in Structure-Activity Studies: Definition and Applications, *J. Chem. Inf. Comput. Sci.*, Vol. 25, No. 25 (1985) 64–73

[11] Bemis, Guy W. and Kuntz, Irwin D., A fast and efficient method for 2D and 3D molecular shape description, *J. Computer-Aided Molecular Design*, Vol. 6 (1992) 607–628

[12] Good, Andrew C. and Kuntz, Irwin D., Investigating the extension of pairwise distance pharmacophore measures to triplet-based descriptors, *J. Computer-Aided Molecular Design*, Vol. 9, (1995) 373–379

[13] Rouvray, D.H., The Search for Useful Topological Indices in Chemistry, *American Scientist*, Vol. 61, No. 6, (1973), 729–735.

[14] Sabljić, Aleksandar and Trinajstić, Nenad, Quantitative structure-activity relationships: the role of topological indices, *Acta Pharm. Jugosl.*, Vol 31, (1981), 189–214.

[15] Trinajstic, N., Chemical Graph Theory, CRC Press, 1992.

[16] Gillet, V. J. and Willett, P. and Bradshaw, J. and Green, D.V.S, Selecting Combinatorial Libraries to Optimize Diversity and Physical Properties, *J. Chem. Inf. Comput. Sci.*, Vol. 39, No. 1 (1999) 169–177

[17] Lepović, M. and Gutman, I., A Collective Property of Trees and Chemical Trees, *J. Chem. Inf. Comput. Sci.*, Vol. 38, No. 5 (1998) 823–826

[18] Bonchev, D., Gutman, I. and Polansky, O., Parity of the Distance Numbers and Wiener Numbers of Bipartite Graphs, *Commun. Math. Chem.*, 22 (1987) 209–214

[19] Wilf, H. S., The Uniform Selection of Free Trees, *Journal of Algorithms* 2 (1981) 204–207

[20] Brown, Robert D. and Martin, Yvonne C., Use of Structure-Activity Data to Compare Structure-Based Clustering Methods and Descriptors for Use in Compound Selection, *J. Chem. Inf. Comput. Sci.*, Vol. 25, (1985) 64–73

[21] Brown, Robert D. and Martin, Yvonne C., The Information Content of 2D and 3D Structural Descriptors Relevant to Ligand-Receptor Binding, *J. Chem. Inf. Comput. Sci.*, Vol. 37, (1997) 1–9

[22] Needham, Diane E., Wei, I-Chen, and Seybold, Paul G., Molecular Modeling of the Physical Properties of Alkanes, *J. Am Chem. Soc.*, Vol. 110, (1998), 4186–4194

[23] Plunkett, Matthew J. and Ellman, Jonathan A., Combinatorial Chemistry and New Drugs, *Scientific American*, April 1997, 69–73

[24] Mohar, Bojan, A Novel Definition of the Weiner index for Trees, *J. Chem. Inf. Comput. Sci*, Vol. 33, (1993), 153-154

A Appendix: pseudo-code for dynamic programming algorithm for the inverse Wiener index problem

In the pseudo-code description of the algorithm that follows, we assume that the matrix M has been initialized to a value "undefined" but for $M_{0,0,1} = 1$.

---

tree (W,L,N)
if $N^3 - N < 6W \bigvee (N-1)^2 > W \bigvee L < N-1 \bigvee L > N(N-1)/2$ **then**
  return 0
if $M_{W,L,N} \neq undefined$ **then**
  return $M_{W,L,N}$
if $N = 1$ **then**
  return 0
for $N_1 := N/2$ to $N-1$ **do**
  $N_2 := N - N_1$
  for $L_1 := N_1 - 1$ to $L - N_2$ **do**
    $L_2 := L - L_1 - N_2$;
    for $W_1 := L_1$ to $W - L_1N_2 - L_2N_1 - N_1N_2$ **do**
      $W_2 := W - W_1 - L_1N_2 - L_2N_1 - N_1N_2$
      if tree$(W_1, L_1, N_1) = 1 \bigwedge$ tree$(W_2, L_2, N_2) = 1$ **then**
        $M_{W,L,N} := 1$
        return 1
for $L_1 := N_1 - 1$ to $L - N_1$ **do**
  $L_2 := L - L_1 - N_1$
  for $W_1 := L_1$ to $W - L_1N_2 - L_2N_1 - N_1N_2$ **do**
    $W_2 := W - W_1 - L_1N_2 - L_2N_1 - N_1N_2$
    if tree$(W_1, L_1, N_1) = 1 \bigwedge$ tree$(W_2, L_2, N_2) = 1$ **then**
      $M_{W,L,N} := 1$
      return 1
$M_{W,L,N} := 0$
return 0
