The Signature Molecular Descriptor. 4. Canonizing Molecules Using Extended Valence Sequences

Jean-Loup Faulon,*,† Michael J. Collins,‡ and Robert D. Carr§

Departments of Computational Biology, Cryptography, and Discrete Algorithms and Mathematics, Sandia National Laboratories, P.O. Box 969, MS 9951, Livermore, California 94551, and Department of Computer Science MSC01 11301 1, University of New Mexico, Albuquerque, New Mexico 87131-0001

Received August 20, 2003

We present a new algorithm to canonize molecular graphs using the signature molecular descriptor introduced in the previous papers of this series. While developed specifically for molecular structures, the algorithm can be used for any graph and is not limited to acyclic graphs, planar graphs, bounded valence, or bounded genus graphs, for which polynomial time algorithms exist. The algorithm is tested with benzenoid hydrocarbons and a database of 126 705 organic compounds. The algorithm's performances are compared against Brendan Mc Kay's Nauty algorithm, which is believed to be the fastest graph canonization algorithm for general graphs, with five series of graphs each comprising up to 30 000 vertices: 2D meshes (pericondensed benzenoids), 3D cages (fullerenes and nanotubes), 3D meshes (crystal lattices), 4D cages, and power law graphs (protein and gene networks). The algorithm can be downloaded as an open source code at http://www.cs.sandia.gov/∼jfaulon/QSAR.

INTRODUCTION

The molecular descriptor we call *signature* is based on extended valence sequences. An extended valence sequence, or molecular signature, is a vector of occurrence numbers of extended valences, or atomic signatures. The extended valence of an atom is a canonical representation of the topological environment of the considered atom up to a predefined height. The first paper[1] of the series investigates the use of the signature in quantitative structure−activity relationship (QSAR) and quantitative structure−property relationship (QSPR) analyses, the second paper[2] proposes an algorithm for enumerating molecules from signatures, and the third paper[3] presents an example of inverse-QSAR analysis using signatures to design ICAM-1 inhibitory peptides. As described in our first paper, atomic signatures are computed in five steps. (1) A subgraph is constructed containing all atoms and bonds that are at a distance no greater than the signature height from the probed atom, (2) the vertices of the subgraph are labeled in canonical order, (3) a tree spanning all the edges of the canonical subgraph is constructed, (4) all canonical labels that appear only one time are removed, and (5) the signature is written by reading the tree in a depth-first order. We proved in that paper (proposition 1) that atomic signatures can be stored in polynomial space (linear for molecular graphs) and can theoretically be computed in polynomial time (proposition 7). The most computationally intensive step in computing the signature is the canonization step. In our first paper we used Brendan McKay's Nauty algorithm,[4,5] which is believed to be the fastest graph canonization algorithm. In the current paper we propose a novel canonization algorithm de facto replacing steps 2−4 of the above procedure.

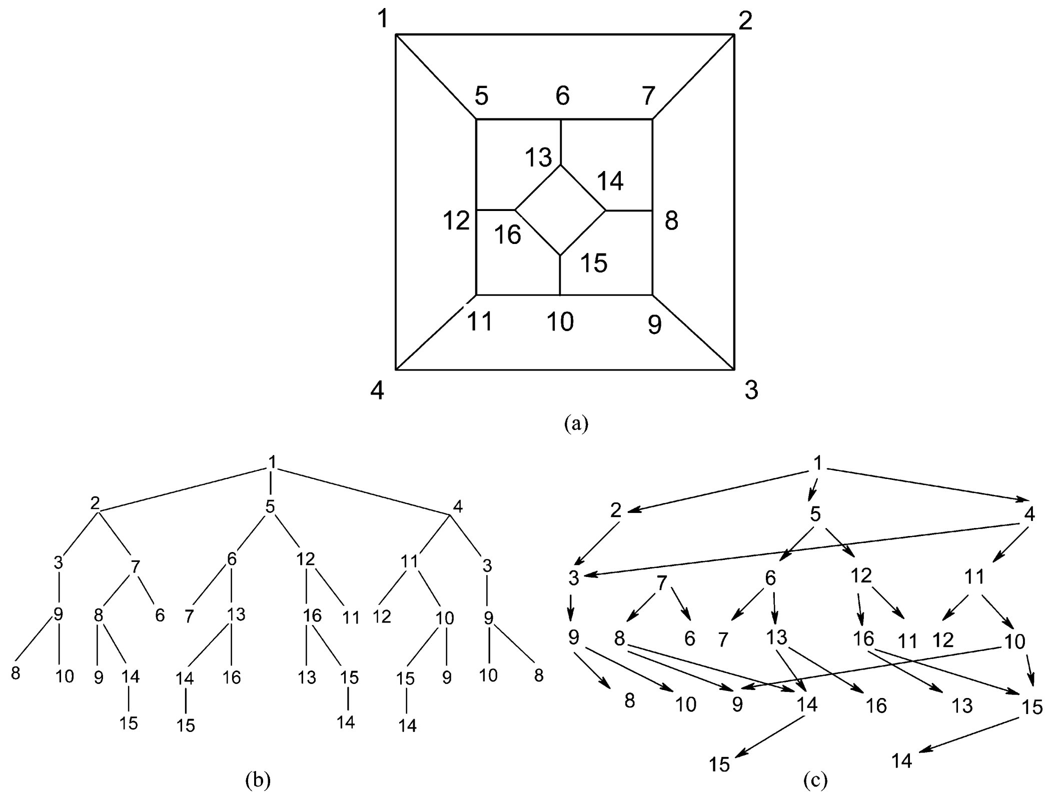
Beyond canonizing signatures, our proposed algorithm can also be used to canonize molecular graphs. According to proposition 4 of our first paper, for any given molecule and any given height greater than the diameter of the molecule, one can reconstruct in linear time the entire molecular graph from any of its atomic signatures. Thus, any algorithm canonizing a signature will also canonize a molecular graph as long as the signature's height is greater than the diameter of the graph.

Graph canonization, along with graph isomorphism, and automorphism partitioning are problems that have been extensively studied both in computer science and computational chemistry. While none of these problems has been shown to be intractable (NP-complete), no polynomial time general solutions are known. Yet these problems are tractable and can be solved efficiently for several restricted classes of graphs, to one of which molecular graphs belong. In 1998 one of us showed how to transform molecular graphs into simple bounded valence graphs,[6] a class of graphs where isomorphism and canonization have theoretically been shown to be solvable in polynomial time.[7] Even though bounded valence graphs can potentially be canonized efficiently, no practical algorithm based on the above results has ever been published, and computational chemists have continued developing heuristics to canonize molecules.[6,8]

Along these lines, the recent IUPAC project IChI is aimed at producing a canonizer available to every chemist.[9]

To canonize a molecular graph, one has to find a unique representation that is independent of the initial labeling of the atoms. Relabeling the atoms in order to minimize or maximize the bond list or the upper triangle of the adjacency

* Corresponding author phone: (925) 294-1279; fax: (925) 294-3020; e-mail: jfaulon@sandia.gov.
† Computational Biology Department, Sandia National Laboratories.
‡ Cryptography Department, Sandia National Laboratories, and the University of New Mexico.
§ Discrete Algorithms and Mathematics Department, Sandia National Laboratories.



atom no.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
atom types and number of parents	C,0	C,1	C,2	C,1	C,1	C,2	C,2	C,2	C,3	C,2	C,2	C,2	C,2	C,3	C,3	C,2
invariant	1	2	3	2	2	3	3	3	4	3	3	3	3	4	4	3

(d)

Figure 1. Atomic signature: (a) Schlegel diagram of a cage comprising 16 atoms; (b) signature tree of atom 1; (c) DAG representation of the signature tree of atom 1; (d) initial invariants.

matrix of a graph are examples of canonization. While a canonization algorithm can be any procedure producing a unique code representing a graph, it is convenient to generate a code from which adjacency matrices can easily be reconstructed.

The paper is set up as follows. The first section presents the data structure used by the canonization algorithm. The second section describes the canonization algorithm. In the third section the correctness of the algorithm is informally demonstrated. Finally, the performance of the algorithm is compared with that of Nauty in the last section.

SIGNATURE DATA STRUCTURE

As already stated, our proposed algorithm canonizes a signature and canonizes a molecular graph when the signature height is greater than the diameter of the graph. In the remainder of the paper we will consider all signatures to be of heights greater than the diameters of the graphs to be canonized.

Let G be a molecular graph and let x be an atom of G. We use the terminology atoms and bonds when referring to the initial molecular graph, and the terms vertices and edges to describe signatures. As illustrated in Figure 1a,b, for $x = 1$, the signature of x is a tree spanning all the bonds of G. We use the term tree here in a somewhat loose manner as several vertices in the tree may correspond to the same atom. The root of the tree is atom x itself. The first layer of the tree is composed of the neighbors of x; the second layer is composed of the neighbors of the vertices of the first layer except atom x. The construction proceeds one layer at a time until no more layers can be added, that is until all the bonds of G have been considered. Assuming the tree has been constructed up to layer l, layer $l + 1$ is constructed considering each vertex y of layer l. Let z be a neighbor of y in G. Vertex z and edge $[y,z]$ are added to layer $l + 1$ if the edges $[y,z]$ or $[z,y]$ are not already present in the previous layers of the tree. To each vertex added to the tree, one associates an atom type and the initial label or number of the corresponding atom. Note that a given atom number z may appear several times in the tree (such as atom number 3 in Figure 1b) since it can be the neighbor of several atoms present in the previous layer. It is important to note that when an atom appears several times at a given layer, the subtree rooted at this atom will appear the same number of times (such as the subtree attached to atom 3 in Figure 1b). When a subtree appears several times at a given layer, it is not necessary to duplicate its data structure; instead the data structure is stored only once, and references to the stored subtree are added as many times as there are duplicates. As illustrated in Figure 1c, using references instead of duplicated subtrees results in a final representation known as a rooted directed acyclic graph (rooted-DAG). A DAG must not contain cycles when following the directions of the edges.

Table 1. Canonized Molecular Signatures for Familiar Hydrocarbons

name	atoms	bonds	colors[a]	strings[a]	canonical molecular signature[b]
hexane	6	5	0	1	2.0[c_]([c_][c_]([c_]([c_]([c_]))))
					2.0[c_]([c_]([c_]([c_]))[c_]([c_]))
					2.0[c_]([c_]([c_]([c_]([c_]([c_])))))
benzene	6	6	0	1	6.0[cp]([cp]([cp]([cp,1]))[cp]([cp]([cp,1])))
naphthalene	10	11	1	2	4.0[cp]([cp]([cp]([cp]([cp]([cp,1]))[cp,2]([cp]([cp,1]))))[cp]([cp]([cp,2])))
					4.0[cp]([cp]([cp]([cp,1]))[cp]([cp]([cp]([cp,2]))[cp]([cp,1][cp]([cp,2]))))
					2.0[cp]([cp]([cp]([cp,1]))[cp]([cp]([cp,2]))[cp]([cp]([cp,2]) [cp]([cp,1])))
pyrene	16	19	1	2	2.0[cp]([cp]([cp]([cp,2][cp]([cp,3]))[cp]([cp,1][cp]([cp,3])))[cp]([cp]([cp,4])[cp]([cp,1]))[cp] ([cp]([cp,4])[cp]([cp,2])))
					2.0[cp]([cp]([cp]([cp,2]([cp]([cp,1]([cp]([cp,3]))[cp,4]([cp] ([cp,3]))))[cp]([cp]([cp,1])))) [cp]([cp]([cp,2][cp]([cp]([cp,4])))))
					4.0[cp]([cp]([cp]([cp,1]))[cp]([cp]([cp,2]([cp]([cp,3]))))[cp]([cp]([cp,2][cp]([cp,4][cp]([cp,3]))) [cp]([cp,1][cp]([cp,4]))))
					4.0[cp]([cp]([cp]([cp,1]([cp]([cp,2][cp,3]))[cp]([cp]([cp,2]))))[cp]([cp]([cp]([cp,4]))[cp] ([cp,1][cp]([cp,4][cp]([cp,3])))))
					4.0[cp]([cp]([cp]([cp,1]([cp]([cp,2]))))[cp]([cp]([cp]([cp,3]([cp]([cp,4]))))[cp]([cp,1][cp] ([cp,3][cp]([cp,2][cp]([cp,4]))))))
coronene	24	30	1	2	6.0[cp]([cp]([cp]([cp,2]([cp]([cp,3]))))[cp]([cp]([cp,4]([cp]([cp,5]))))[cp]([cp]([cp,4][cp] ([cp,6]([cp]([cp,1][cp,7]))[cp]([cp,5][cp]([cp,1]))))[cp]([cp,2][cp]([cp,6][cp]([cp,3][cp] ([cp,7]))))))
					6.0[cp]([cp]([cp]([cp,2]([cp]([cp,1][cp,3]))[cp]([cp,4][cp]([cp,1])))[cp]([cp,5][cp]([cp,4])))[cp] ([cp]([cp,2][cp]([cp,6][cp]([cp,3])))[cp]([cp,7][cp]([cp,6])))[cp]([cp]([cp,7])[cp]([c p,5])))
					12.0[cp]([cp]([cp]([cp,1]([cp]([cp,2]([cp]([cp,3][cp,4]))[cp, 5]([cp]([cp,4]))))[cp]([cp]([cp,5])))) [cp]([cp]([cp]([cp,6]([c p]([cp,7]))))[cp]([cp,1][cp]([cp,6][cp]([cp,2][cp]([cp,7][cp] ([cp,3])))))))
fullerene C60	60	90	1	2	60.0[cp]([cp]([cp]([cp,2]([cp]([cp,3][cp,4]([cp]([cp,5][cp,6]))))[cp]([cp,7]([cp]([cp,8][cp,9] ([cp]([cp,10][cp,11]))))[cp,3]([cp]([cp,9][cp,6]([cp,11]([cp,12]([cp,13])))))))[cp]([cp,14][cp] ([cp,7][cp]([cp,15][cp,8]([cp]([cp,16][cp,10]([cp,17]([cp,13]))))))))[cp]([cp]([cp,18]([cp] ([cp,19][cp,20]([cp]([cp,21][cp,22]))))[cp]([cp,23]([cp]([cp,24][cp,25]([cp]([cp,26] [cp,27])))) [cp,19]([cp]([cp,25][cp,22]([cp,27]([cp,1]([cp,13])))))))[cp,14]([cp]([cp,23][cp,15]([cp,24] ([cp,16]([cp,26]([cp,17])))))))[cp]([cp]([cp,18][cp]([cp,28][cp]([cp,20][cp]([cp,29][cp,21] ([cp]([cp,1][cp,30]))))))[cp]([cp,2][cp,28]([cp]([cp,4][cp,29]([cp,5]([cp,30]([cp,12]))))))))

[a] Maximum number of colors used to canonize signature strings and maximum number of signature strings generated for each atom. The number of colors is the number of times the variable *c* in the canonized-signature scheme is incremented (line 12). The number of strings is the number of times the canonized-signature algorithm executes line 6. The strings reported in column 6 are the canonical ones (the lexicographically largest ones). [b] The canonical molecular signature is compiled after calculating the canonical signature string of each atom. All atomic signatures are sorted in decreasing lexicographic order (according to the C language function strcmp). The number preceding each atomic signature is the number of the atom having that signature. Hydrogen atoms have been removed for all compounds; atom type c_ stands for aliphatic carbon, and atom type cp for aromatic carbon.

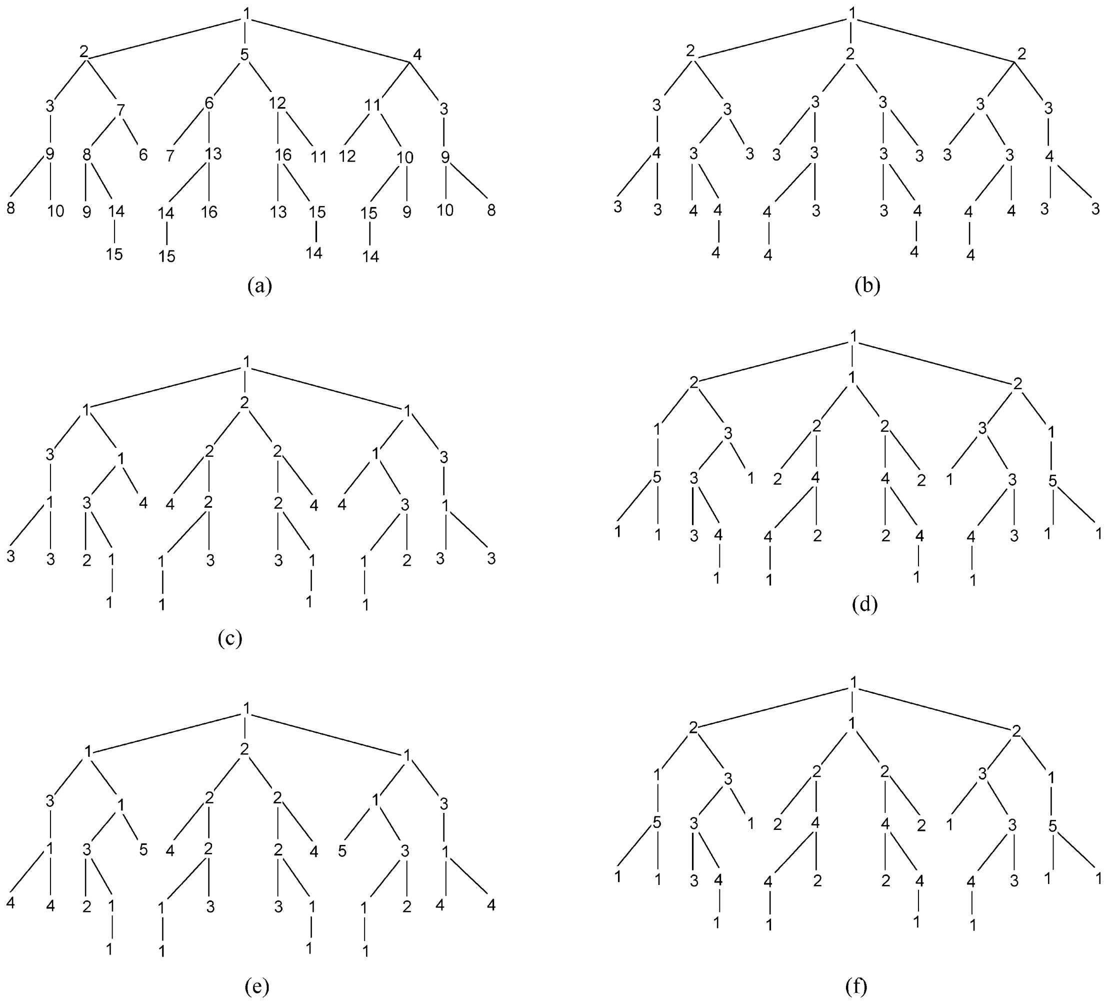
A rooted-DAG representation of a signature tree comprises no more edges than twice the number of bonds of the initial graph. Indeed, as illustrated in Figure 1c with bonds [7,6] and [6,7] in the same layer, a bond may appear twice depending on the direction it is crossed. Consequently, while a signature tree may comprise numbers of vertices and edges that may be exponentially larger than the numbers of atoms and bonds, its rooted-DAG representation has a size linearly proportional to the number of bonds of the molecular graph.

THE SIGNATURE CANONIZATION ALGORITHM

The algorithm described in this section computes and canonizes the signature of a given atom in a molecular graph. The algorithm is repeated for all atoms, and the resulting atomic signatures are compiled into a molecular signature. A canonical representation of the molecular signature is obtained in a postprocess by sorting the atomic signatures in decreasing lexicographic order (cf. examples in Table 1). As mentioned earlier, the molecular graph can be reconstructed from any of its atomic signatures. Consequently, as far as graph canonization is concerned, there is no need to keep all atomic signatures, and the lexicographically largest atomic signature (i.e., the first one in the list) suffices to represent the graph in a unique manner. Yet, to find the largest atomic signature, one has to compute and canonize the signatures for all atoms of the graph. We next explain how atomic signatures are canonized; the reader should keep in mind that the process described below is repeated for all atoms.

The approach we have taken to canonize atomic signatures is based on the classical Hopcroft and Tarjan's rooted-tree canonization algorithm.[10] Assuming we want to canonize a rooted tree, we first assign to each vertex of the tree an initial invariant. In our case, the initial vertex invariant is an integer based on the atom type corresponding to the vertex. Next, the vertices are sorted according to their invariants layer by layer starting with those farthest away from the root. After the vertices at layer l are ordered, each vertex in layer $l - 1$ is given a new invariant, computed from its initial invariant and the sorted list of the invariants of its neighbors in layer l. Vertices in layer $l - 1$ are sorted according to the new invariants. The process is repeated until the root is reached. Using this simple procedure, the edges of each vertex of the tree are ordered in a unique manner based on the vertex invariants. The rooted tree obtained following the edge ordering is canonical.

We now describe in some detail how the above algorithm has been implemented to canonize a signature tree. Let x be an atom of a molecular graph G. The signature tree, $T(x)$, is represented by a DAG rooted at x. To each atom a, one



atom no.		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
layer no.	5	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0
	4	0	0	0	0	0	0	0	1	3	1	0	0	2	4	4	2
	3	0	0	0	0	0	1	2	3	5	3	0	0	4	0	0	4
	2	0	0	1	0	0	2	3	0	0	0	3	2	0	0	0	0
	1	0	2	0	2	1	0	0	0	0	0	0	0	0	0	0	0
	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
invariant		10	8	7	8	9	6	5	4	3	4	5	6	2	1	1	2

(g)

Figure 2. Vertex and atom invariants.: (a) signature tree of atom 1 in Figure 1; (b) same signature tree with initial invariants from Figure 1d; (c) vertex invariants after running Hopcroft−Tarjan algorithm from the leaves to the root; (d) vertex invariants after running Hopcroft−Tarjan algorithm from the root to the leaves; (e) vertex invariants after running Hopcroft−Tarjan algorithm for the second time, from the leaves to the root; and (f) vertex invariants after running Hopcroft−Tarjan algorithm for the second time, from the root to the leaves. The algorithm stops as invariants are the same as in the previous run. (g) The corresponding atom vectors and atom invariants. The root is located at layer 0.

associates an atom type, a color, color(a), and an invariant, inv(a). Colors and invariants are integers no greater than n, the total number of atoms. To each vertex v in $T(x)$ one associates a corresponding atom, atom(v), in graph G and an invariant, inv(v). Since the signature tree is represented by a rooted-DAG, for each vertex of any layer l, one can access its parents in layer $l - 1$ and its children in layer $l + 1$.

Prior to running the algorithm, all the invariants and colors are initialized. The initial color is 0 for all atoms. The initial invariant of any atom a is computed from the atom type of a and the number of parents a has in $T(x)$. More precisely, a string of characters is compiled from the atom type and the number of parents, and the string is converted into an integer following lexicographic ordering. The integer is not greater than n since there are no more than n different strings. Examples of initial invariants are given in Figure 1d.

After initialization, the first step of the algorithm is to compute the invariants of the vertices in $T(x)$ from the atom invariants. The vertex invariants are computed twice, first reading the tree layer by layer from the leaves to the root and then from the root to the leaves. Unlike the classical Hopcroft−Tarjan algorithm, the tree must also be read from the root to the leaves because in signature trees, some vertices may have more than one parent; thus, the invariants for these vertices may be different, depending on the invariants of their parents. We first examine the case where the tree is read from the leaves to the root. Starting at the last layer, to each vertex we associate a pair of numbers composed of the color and the invariant of the corresponding atom. Duplicated pairs are removed, and all nonidentical pairs are sorted in decreasing order. The vertex invariant becomes the order of the pair in the sorted list. Going to the layer above, to each vertex one assigns a vector of pairs composed of the color and the invariant of the corresponding atom and the pairs for the children of the vertex. Duplicated vectors are removed, the remaining vectors are sorted in decreasing order, and the vertex invariant becomes the order of the vertex in the sorted list. Note that these vertex invariants range from 1 to *n* since there are no more than *n* vertices in a layer. The above procedure is repeated until the root is reached. The algorithm is then run from the root to the leaves, but this time, for any vertex, the vector invariant is composed of the color and invariant of the corresponding atom and the invariants of the parents of the vertex. The algorithm is given next and illustrated in Figure 2a−f.

```
invariant-vertex(T(x),relative)
Input:  T(x) the signature-tree of atom x
        relative is a parent or child relationship
Output: Updated vertices invariants

1.  List-$V_{inv}$ = ∅
2.  for all layers l of T(x)
3.      for all vertices v of layer l
4.          $V_{inv}$(v) = color(atom(v)), inv(atom(v)),
                      {inv(w) s.t. w is a relative of v}
5.          List-$V_{inv}$ = List-$V_{inv}$ ∪ $V_{inv}$(v)
6.      done
7.      sort List-$V_{inv}$ in decreasing order
8.      for all vertices v of layer l
9.           inv(v) = order of $V_{inv}$(v) in List-$V_{inv}$
10.     done
11. done
```

Once invariants have been computed for all vertices, each atom invariant is compiled from the invariant of all the vertices corresponding to the atom. Precisely, for each atom, an invariant vector is first initialized to zero, and then for each vertex corresponding to the atom, the invariant of the vertex is assigned to the *l*-coordinate of the vector where *l* is the layer the vertex occurs. Once invariant vectors are computed for all atoms, duplicated vectors are removed, the vectors are sorted in decreasing order, and the atom invariant becomes the order of the atom's vector in the sorted list. The above process is repeated until the number of atom invariants remains constant. Note that at each iteration, the number of invariants increases. Indeed, atom invariants are computed from vertex invariants, which in turn are computed from the atom invariants from the previous iteration. Because the number of invariants is at most the number of atoms, the process cannot repeat itself more than *n* times. The invariant-atom algorithm is given next and illustrated in Figure 2g.

```
invariant-atom(T(x),G)
Input:  T(x) the signature-tree of atom x
        G a molecular graph
Output: Updated atoms invariants

1. repeat
2.     invariant-vertex(T(x),child)
3.     invariant-vertex(T(x),parent)
4.     for all vertices v of T(x)
5.         let l be the layer of v
6.         $V_{inv}$(atom(v))(l) = inv(v)
7.     done
8.     List-$V_{inv}$ = ∅
9.     For all atoms a of G do
10.        List-$V_{inv}$ = List-$V_{inv}$ ∪ $V_{inv}$(a)
11.    done
12.    sort List-$V_{inv}$ in decreasing order
13.    for all atom a of G
14.        inv(a) = order of $V_{inv}$(a) in List-$V_{inv}$
15.    done
16. until the number of invariant values remain constant
```

The canonization algorithm illustrated in Figure 3 first computes the invariants for all atoms running the invariant-atom algorithm (step 1). Then in step 2, the atoms are partitioned into orbits such that all the atoms in a given orbit have the same invariant. In the next step (3) one searches for an orbit containing atoms having at least two parents in *T*(*x*). Note that these atoms have different invariants than atoms having only one parent, since the initial atom invariants embrace the number of parents. When several such orbits exist, those with the maximum number of atoms are selected, and if several orbits have the same number of atoms, one takes the one with the minimum invariant. For instance, according to Figure 2g, orbits {14,15}, {13,16}, {10,8}, and {6,12} all contain two atoms having more than one parent. Orbit {14,15} is chosen in Figure 3, since it has the minimum invariant. If no orbit can be found or the selected orbit contains only one atom, then the process ends and the signature is printed (steps 4−9). When an orbit is found containing more than one atom, all the atoms of the orbits are colored one after another with the current color (steps 10−14). In Figure 4, atoms 14 and 15 are colored one at a time with color 1. In both cases the algorithm stops after the next iteration since each atom becomes singularized in its own orbit. The current color starts at 1 and is incremented by 1 each time the algorithm calls itself (step 12). Note that each time an atom is colored, at the next iteration the atom will be alone in its orbit. Since there are no more than *n* atoms to be colored, the algorithm cannot call itself more than *n* times.

```
canonize-signature(T(x),G,c,$S_{max}$)
Input:  T(x) the signature-tree of atom x
        G a molecular graph
        c a color
Output: $S_{max}$ a canonical string (initialized to empty string)

1.  invariant-atom(T(x),G)
2.  partition the atoms of G into orbits according to their invariants
3.  let O be the orbit with the maximum number of atoms and the minimum
    invariant value such that all the atoms of O have at least two
    parents.
4.  if |O| < 2 then
5.     color all uncolored atoms having two parents
       according to their invariant
6.     S = print-signature-string(root(T(x)),T(x),∅)
7.     if (S > $S_{max}$) $S_{max}$ = S fi
8.     return $S_{max}$
9.  fi
10. for all atom a in O do
11.     color(a) = c
12.     $S_{max}$ = canonize-signature(T(x),G,c+1,$S_{max}$)
13.     color(a) = 0
14. done
15. return $S_{max}$
```

As described in our first paper,[1] signature strings are printed by reading the signature tree in a depth-first order. Prior to printing signature strings, the children of all vertices are sorted according to their invariants taken in decreasing order. To avoid printing several times duplicated subtrees, any subtree is printed only the first time it is read. This operation necessitates maintaining a list of printed edges. The algorithm is given next and depicted in Figure 4, where atom 14 has been colored. The same signature string is obtained when coloring atom 15.

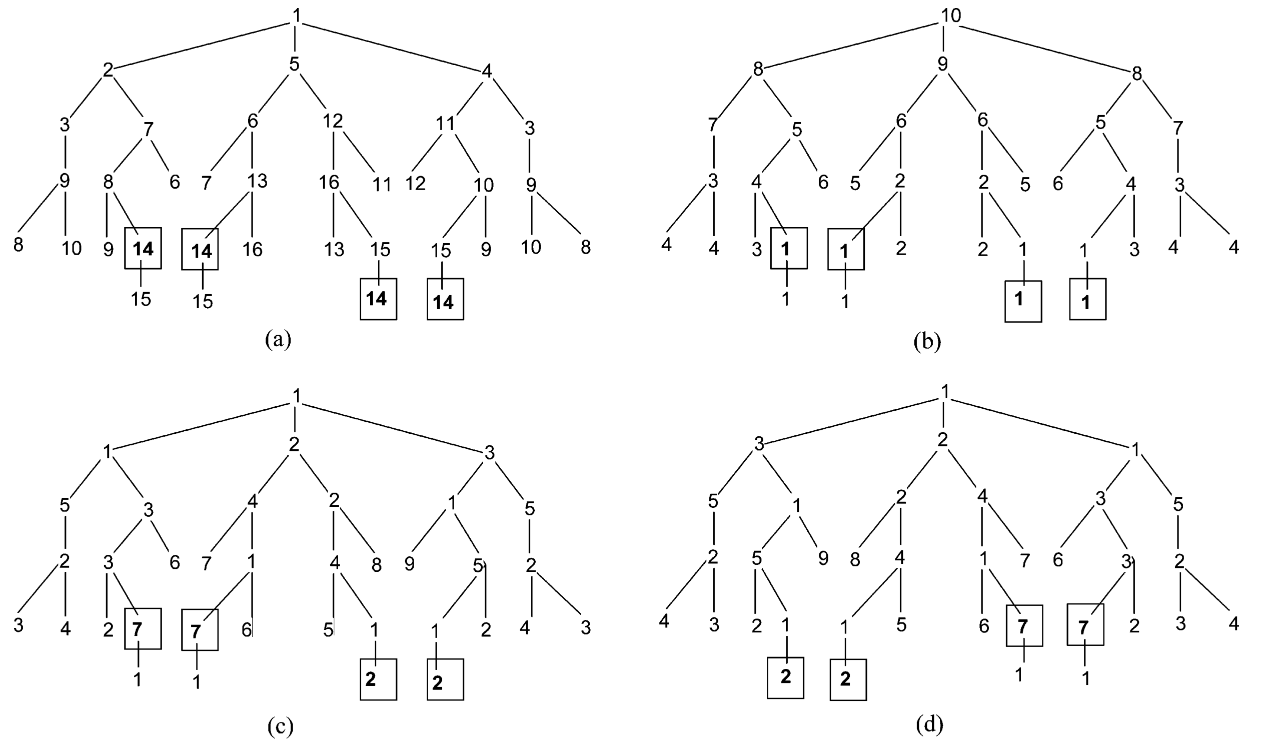


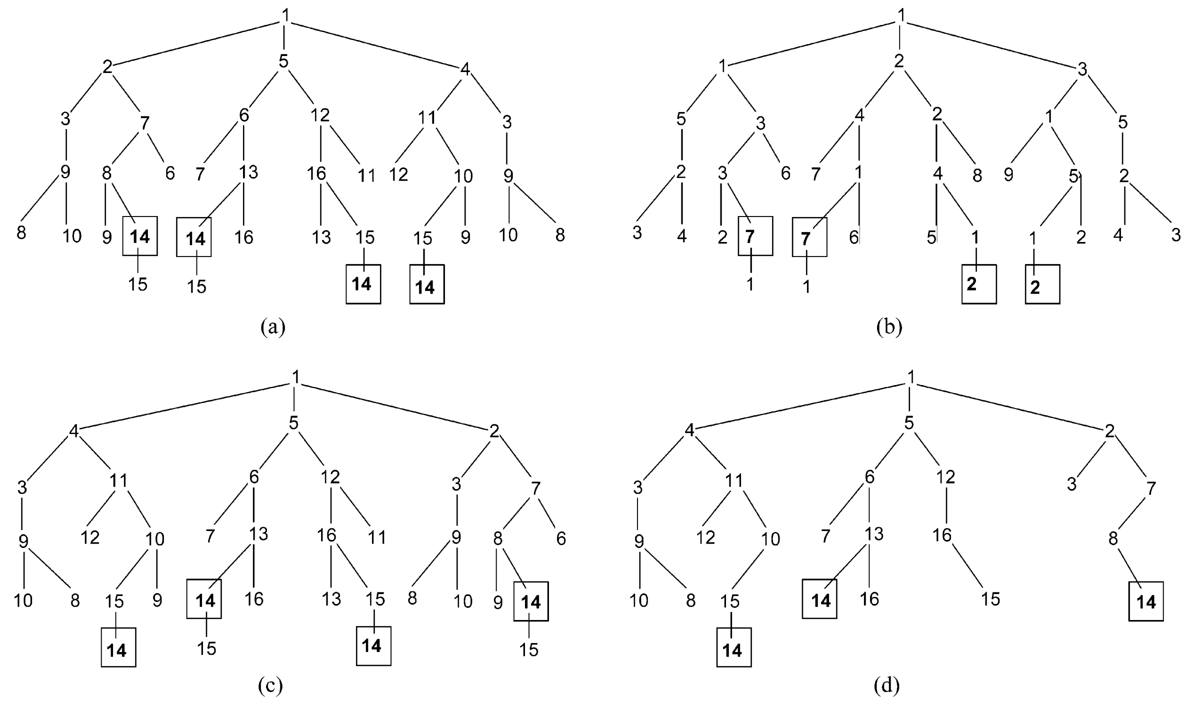
Figure 3. Signature-canonization algorithm: (a) signature tree of atom 1 in Figure 1, where atom 14 is colored; (b) same signature tree with initial invariants of Figure 2d; (c) vertex invariants after running the invariant-atom algorithm, where atom 14 is colored (note that every orbit contains only one atom and the canonization algorithm stops after coloring atom 14); (d) vertex invariants after running the invariant-atom algorithm where atom 15 is colored. The canonization algorithm stops after coloring atom 15.

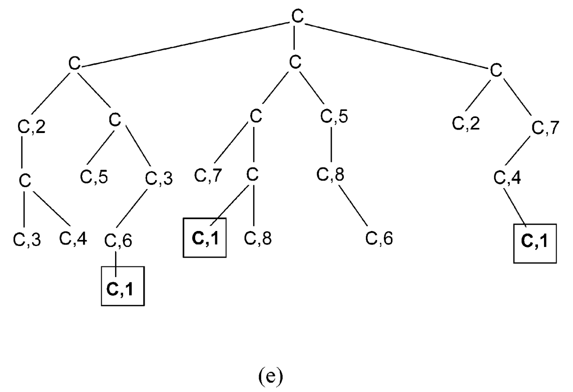
```
print-signature-string(v,T(x),E)
Input:  v a vertex of T(x)
        T(x) the signature-tree of atom x
        E a set of edges
Output: a string of characters

1.  print '['
2.  print atom-type(atom(v))
3.  if (color(atom(v)) ≠ 0) print ',' color(atom(v))
4.  print ']'
5.  if child(v) = ∅ return
6.  sort child(v) according to their invariants
7.  print '('
8.  for all child w of v in decreasing order do
9.      if [v,w] is not already in E then
10.         E = E ∪ [v,w]
11.         print-signature-string(w,T(x),E)
12.     fi
13. done
14. print ')'
15. return
```

CORRECTNESS OF THE ALGORITHM

The canonization algorithm is deterministic and is independent of the initial labeling of the atoms of the graph. In fact, the initial labels are never used by any of the canonization subroutines. Consequently, in any given graph, two automorphic atoms will produce the same canonized signature string. We also need to show that when two atoms have the same canonized signature, they necessarily are automorphic. In a typical brute-force canonization algorithm color 1 is assigned to all the atoms of the graph one after another. For each resulting coloring, color 2 is assigned to the remaining $n - 1$ uncolored atoms, color 3 is assigned to $n - 2$ atoms, and so on, resulting in a total of $n!$ different colorings. Applied to a signature tree, a brute-force algorithm would generate at most $(n - 1)!$ colorings since the root of the tree does not have to be colored. Because for every atom all possible colors are tested, all signature trees isomorphic to the initial one are generated and the coloring maximizing the signature string is kept as the canonical one. Thus, when using a brute-force algorithm, if two atoms have the same canonized signature string, they are automorphic. The latter statement is true with every algorithm generating $(n - 1)!$ colorings. Assume the signature canonization algorithm given in the previous section is run with a graph such that all atoms, except the root, belong to the same orbit and have more than one parent. While such a case may not exist, one can easily construct graphs (cages for instance) with orbits comprising a number of atoms proportional to the molecular graph size; in such a case n is smaller than the number of atoms but is nonetheless polynomially proportional to the graph size. In the worst-case scenario, the first color will be assigned to $n - 1$ atoms one after another. Once a color has been given to an atom, the atom cannot be colored again since its orbit contains only one element. However, all other atoms may still be in the same orbit, and if such is the case, all the $n - 2$ remaining atoms will be colored one after another. In the next iteration $n - 3$ atoms will be colored, and so on, until no more atoms can be colored. With the above scenario the algorithm will call itself $n - 1$ times and will produce $(n - 1)!$ colorings. In other words, the algorithm will generate all $(n - 1)!$ isomorphic representations of the initial signature tree, and thus two atoms having the same canonized signature string are necessarily automorphic. The only difference between our algorithm and a brute-force algorithm is that when a coloring splits the remaining vertices into several orbits, the vertices that belong to different orbits will be colored with different colors. Atoms belong to different orbits when their corresponding vertices in the signature tree have different invariants (cf. invariant-atom algorithm in the





```
[C]([C]([C,2]([C]([C,3][C,4]))
[C]([C,5][C,3]([C,6]([C,1]))))
[C]([C]([C,7][C]([C,1][C,8]))
[C,5]([C,8]([C,6])))[C]([C,2]
[C,7]([C,4]([C,1]))))
```

(f)

Figure 4. Printing the signature string: (a) signature tree of atom 1 in Figure 1, where atom 14 has been colored; (b) same signature tree with atom invariants of Figure 3c; (c) signature tree with branches reordered according to atom invariants computed in b; (d) signature tree where duplicated edges have been removed; (e) signature tree with atom type and colors. Atom 14 is colored 1; all other colors are entered in the order the atoms appear reading the tree in a depth-first order. Colors are added only to vertices appearing more than one time in the signature tree given in d. (f) The corresponding signature string reading the tree in a depth-first order.

previous section). Now, according to the Hopcroft–Tarjan algorithm (cf. invariant-vertex algorithm in the previous section), two vertices with different invariants are not automorphic, and for a given color, the signature string obtained by coloring the vertex with the smallest invariant will always be smaller than the string obtained coloring the other vertex. It is thus unnecessary to color vertices belonging to different orbits with the same color. Finally, atoms that occur only one time in the signature tree do not have to be colored. According to proposition 1 in our first paper, any graph can be reconstructed from any of its canonical signature strings after the colors that appear only one time in the string have been removed.

Examples of canonized signature strings along with the number of colors used to generate them are given in Table 1 for familiar hydrocarbons. For all the compounds listed in Table 1 the computations were duplicated with structures having different initial atom labels. The resulting canonized signatures were found to be identical. The same exercise was carried out for some difficult graphs given in Figures 3 and 5 of the Rucker et al. paper.[11] These graphs are nonisomorphic but are isopectral and have the same Balaban *J* index.[12] The nonisomorphic graphs were found to have different signatures. As in Table 1, no more than one color was used to canonize these graphs.

ALGORITHM PERFORMANCE

A database of 126 705 compounds was downloaded from the National Cancer Institute website.[13] This database comprises various organic and organometallic compounds up to 94 atoms. For each of the compounds the signature was computed and compared with the signature obtained after randomly renumbering the atoms. As expected from the previous section the signatures corresponding to the same compound were found to be identical. The CPU times taken to compute the molecular signature of each compound of the database are plotted in Figure 5. The entire database was processed in 26 min CPU time, with a rate of about 80 compounds/s. All runs were performed on a SGI/O2 workstation. While the running time increased with the number of atoms, we also observed that highly symmetrical compounds were slower to process.

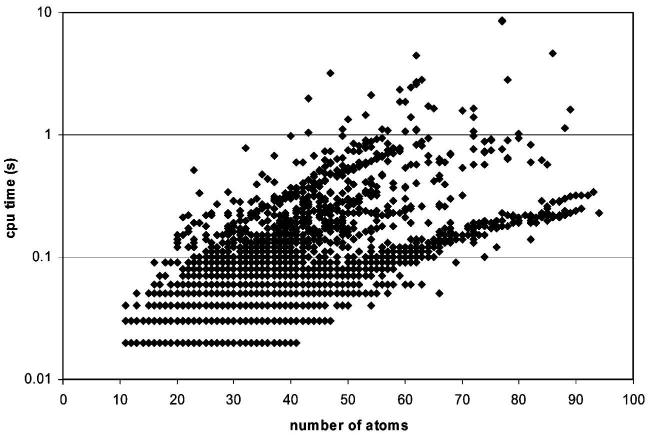


Figure 5. CPU times distribution for the National Cancer Institute's open database (opennci) of 126 705 organic compounds. Compounds processed in less than 0.02 s CPU time are not shown.

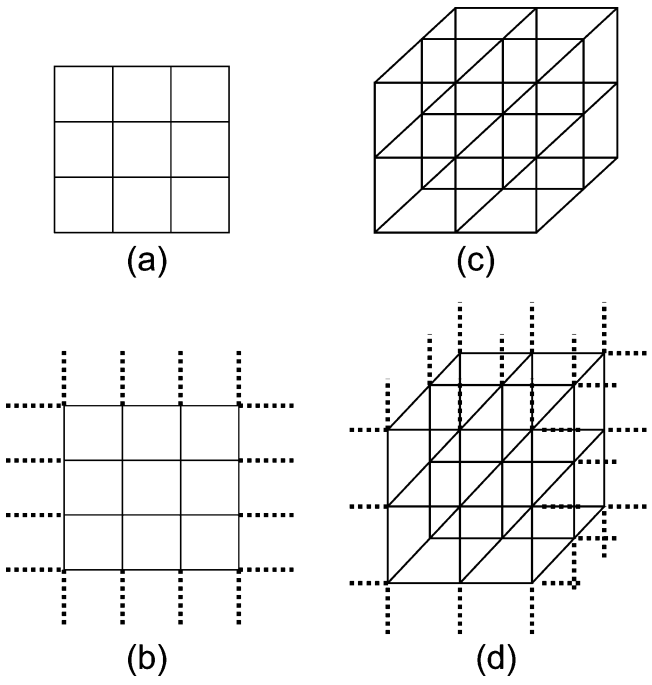


Figure 6. Examples of 2D mesh (a), 3D cage (b), 3D mesh (c), and 4D cage (d). For 3D cages the bonding sites (dashed lines) on the left side are merged with the right bonding sites, and the top bonding sites are merged with the bottom ones. For 4D cages bonding sites of the left face are merged with those of the right face, bonding sites of the top face are merged with those of the bottom face, and bonding sites of the front face are merged with those of the back face.

To further assert our algorithm, we compared its performances with those of Nauty on five series of graphs, 2D meshes, 3D cages, 3D meshes, 4D cages, and power law graphs. All series of graphs contained up to 30 000 atoms. Figure 6 depicts the general structure of the 2D and 3D meshes and the 3D cage series. While these series do not represent chemical compounds, they have the same topological characteristics, and the canonization results obtained on the series should be similar to those obtained on compounds of the same size. Furthermore, meshes and cages are highly symmetrical graphs and thus should be difficult for our algorithm. The series of meshes were chosen since benzenoids and crystal lattices are particular types of 2D and 3D meshes. The series of 3D cage was selected because fullerenes and nanotubes are 3D cages. The series of 4D

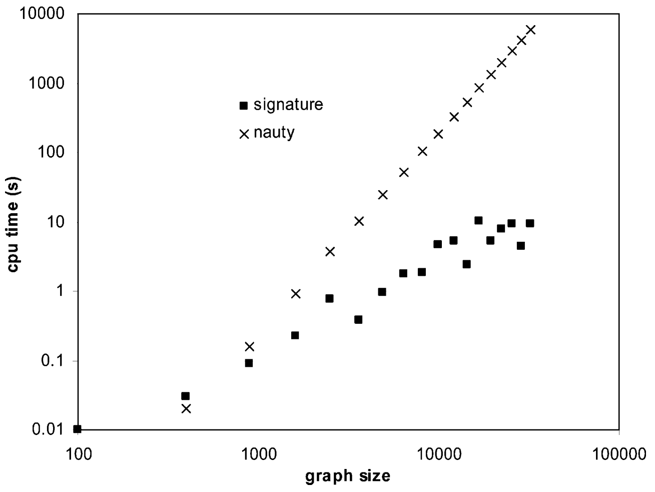


Figure 7. Signature and Nauty canonization CPU times for 2D meshes.

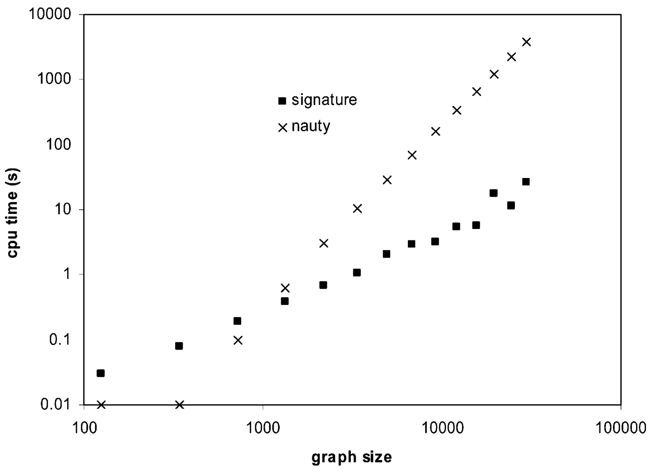


Figure 8. Signature and Nauty canonization CPU times for 3D meshes.

cages was chosen because it is a particularly hard class for our algorithm, as initially all atoms are equivalent (i.e., automorphic) and all atoms have six neighbors. The series of power law graphs was chosen because gene and protein networks have recently been found to follow power laws.[14,15] A power law graph is a graph in which the number of vertices of degree k is $N_k = N_0 1/k)^{\gamma}$, where N_0 is a constant and γ is a scaling exponent. We chose $\gamma = 3$, as this exponent has been found to be between 2 and 3 for gene and protein networks. Note that power law graphs are not necessarily planar, and the maximum degree of the vertices is not bounded as with the previous series. In Figures 7−11 we report the CPU times taken to canonize the signature tree of one selected atom. For 2D and 3D meshes the selected atom was chosen in one of the corners of the mesh. Note that the corner atoms have a signature of greater height than the others, thus maximizing the computational complexity for our algorithm. With 3D and 4D cages all atoms are equivalent, therefore an atom was selected at random. Finally with power law graphs the atom with the largest degree was selected. The CPU times of our algorithm are plotted along with the times obtained with Nauty, prior to running, Nauty atoms were partitioned, the selected atom was inserted into its own orbit, and all other atoms were added to a second orbit. Partitioning the atoms is an option offered by Nauty that enabled us to make a fair comparison between the two

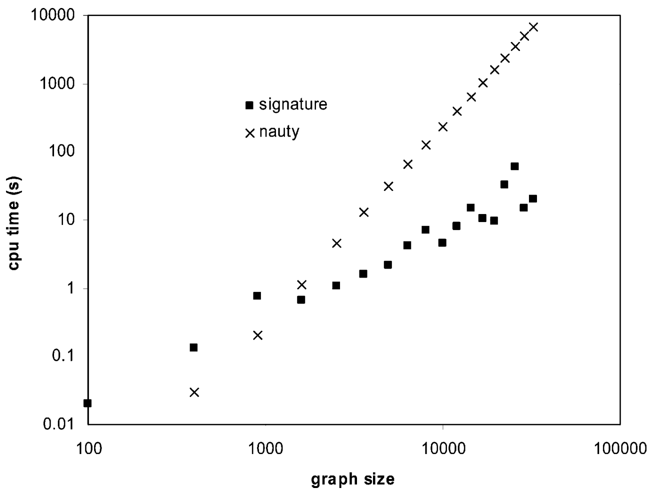


Figure 9. Signature and Nauty canonization CPU times for 3D cages.

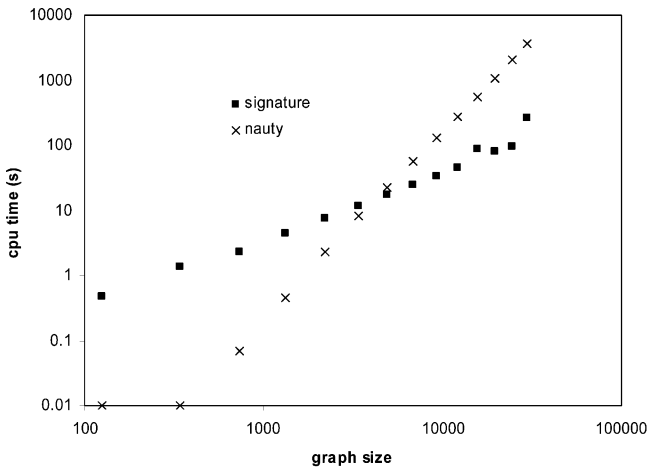


Figure 10. Signature and Nauty canonization CPU times for 4D cages.

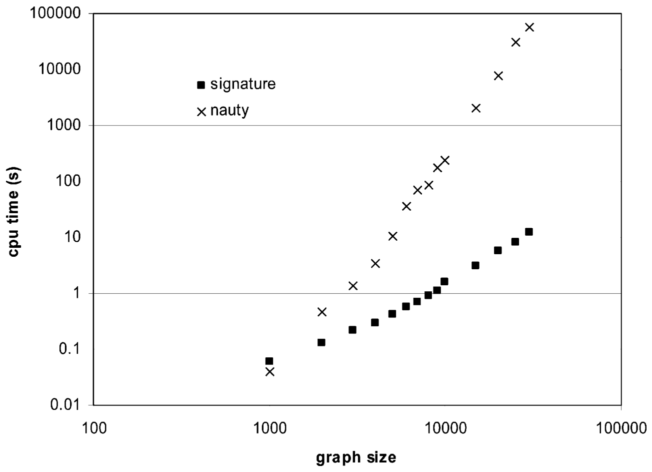


Figure 11. Signature and Nauty canonization CPU times for power law graphs.

codes as in both cases at the beginning of the process one atom was selected as the root of a graph to be canonized.

For all series except power law graphs, exactly one color was sufficient. Independently of the number of atoms, for 2D meshes 4 signature strings were generated, 6 strings were constructed for 3D meshes, 8 strings for 3D cages, and 48 strings for 4D cages. Power law graphs generally have few symmetries, and it is unlikely that these symmetries occur at the same layers of a signature tree. It is therefore not surprising to find out that from 1000 to 30 000 atoms no colors are needed to canonize these graphs. From Figures 7−11 one concludes that Nauty is generally faster than our algorithm for graphs of small sizes, but, for all series, the reverse trend is observed as the graph sizes increases.

CONCLUDING REMARKS

While we have shown earlier that our canonization algorithm might have the worst-case complexity of the brute-force algorithm, we have yet to find a graph for which the algorithm runs an exponential number of steps. In fact, in all tests carried out in the previous section, we find the number of strings produced to be a constant independent of the number of atoms. Furthermore, when canonizing atomic signatures, as the number of atoms increases, we find our algorithm to be substantially faster than Nauty. The worst-case scenario that we have identified so far is with projective planes,[16] where four colors have to be assigned, thus generating at most $O(n^4)$ signature strings. Graphs derived from projective planes are not necessarily relevant to chemistry but are notoriously hard to canonize. They can efficiently be dealt with in the latest version of Nauty only after having been preprocessed with a special-purpose subroutine that partitions the vertices.

The algorithm presented in this paper was coded in standard C language. The code reads Accelrys's Cerius2 Polygraf and Insight formats,[17] Tripos's PDB and PFS formats,[18] and MDL's SDF format.[19] Additionally, the code can read general graphs in Nauty format.[5] The code outputs a signature file containing the molecular signature using the notations given in Table 1. An option is also available to canonically relabel the input format. The code handles covalently bonded compounds that may be composed of several connected components. The code does not distinguish stereoisomers.

ACKNOWLEDGMENT

Funding for this work was provided by the Math, Information and Computer Science program of the U.S. Department of Energy and Sandia National Laboratories under Grant No. DE-AC04-76DP00789.

REFERENCES AND NOTES

(1) Faulon, J.-L.; Visco, J.; D. P.; Pophale, R. S. The Signature Molecular Descriptor. 1. Extended Valence Sequences and Topological Indices. *J. Chem. Inf. Comput. Sci.* **2003**, *43*, 707−721.
(2) Faulon, J.-L.; Churchwell, C. J.; Visco, J. The Signature Molecular Descriptor. 2. Enumerating Molcules from their Extended Valence Sequences. *J. Chem. Inf. Comput. Sci.* **2003**, *43*, 722−734.
(3) Churchwell, C. J.; Rintoul, M. D.; Martin, S.; Visco, D.; Kotu, A.; Larson, R. S.; Sillerud, L. O.; Brown, D. C.; Faulon, J. L. The Signature Molecular Descriptor. 3. Inverse Quantitative Structure−Activity Relationship of ICAM-1 Inhibitory Peptides. *J. Mol. Graphics Modell.*, in press.
(4) McKay, B. D. Practical Graph Isomorphism. *Congr. Numerantium* **1981**, *30*, 45−87.
(5) McKay, B. D. *Nauty User's Guide*, Version 2.2 (beta 6); 2003 (http://cs.anu.edu.au/people/bdm/nauty/).
(6) Faulon, J.-L. Isomorphism, Automorphism Partitioning, and Canonical Labeling Can Be Solved in Polynomial Time for Molecular Graphs. *J. Chem Inf. Comput. Sci.* **1998**, *38*, 432−444.
(7) Luks, E. M. Isomorphism of Graphs of Bounded Valence Can Be Tested in Polynomial Time. *J. Comput. Syst. Sci.* **1982**, *25*, 42−65.

(8) Wong, H.-W.; Li, X.; Swihart, M. T.; Broadbelt, L. J. Encoding Polycyclic Si-Containing Molecules for Determining Species Uniqueness in Automated Mechanism Generation. *J. Chem. Inf. Comput. Sci.* **2003**, *43*, 735−742.
(9) Freemantle, M. Unique Labels for Compounds. *Chem. Eng. News* **2002**, *80*, 33−35.
(10) Hopcroft, J. E.; Tarjan, R. E. Isomorphism of Planar Graphs. In *Complexity of Computer Computations*; Miller, R. E., Thatcher, J. W., Eds.; Plenum Press: New York, 1972; pp 131−150.
(11) http://dtp.nci.nih.gov/docs/3d_database/structural_information/structural_data.html.
(12) Rucker, C.; Rucker, G.; Meringer, M. Exploring the Limits of Graph Invariant- and Spectrum-Based Discrimination of (Sub)structures. *J. Chem. Inf. Comput. Sci.* **2002**, *42*, 640−650.
(13) Balaban, A. T. Highly Discriminating Distance-Based Topological Index. *Chem. Phys. Lett.* **1982**, *89*, 399−404.
(14) Jeong, H.; Tombor, B.; Albert, R.; Oltval, Z. N.; Barabasi, A. L. The Large-Scale Organization of Metabolic Networks. *Nature* **2000**, *407*, 651−654.
(15) Maslov, S.; Sneppen, K. Specificity and Stability in Topology of Protein Networks. *Science* **2002**, *296*, 910−913.
(16) Colbourn, C. J.; Dinitz, J. H. *The CRC Handbook of Combinatorial Designs*; CRC Press: Boca Raton, FL, 1996.
(17) *Cerius2 Software*; Accelrys: San Diego, CA, 2003.
(18) *Sybyl Software*; Tripos: St. Louis, MO, 2003.
(19) MDL Information Systems Inc.: San Leandro, CA, 2003.

CI0341823