Ring Perception Using Breadth-First Search

John Figueras

399 Baker's Pond Road, Orleans, Massachusetts 02653

Received January 29, 1996[⊗]

Combining breadth-first search with new ideas for uncovering embedded rings in complex systems[1] yields a very fast routine for ring perception. With large structures, the new routine is orders of magnitude faster than depth-first ring detection, a result expected on the basis of recent work that establishes polynomial order for BFS.[2]

INTRODUCTION

Two recent publications describe new solutions to the problem of ring perception to find the Smallest Set of Smallest Rings (SSSR) using graph walking algorithms.[1,2] The essence of the first of these, an algorithm developed by Doucet and co-workers, is the following.

(1) Find and record the smallest ring that includes a given node N2 of degree 2. In Doucet's approach, the smallest ring is found through a succession of depth-first searches that converge to the smallest ring. Add the captured ring to the SSSR (Smallest Set of Smallest Rings).

(2) Remove the ring found in step 1 by deleting an N2 node from the ring (not necessarily the node selected in step 1) and—if it exists—the polymethylene chain containing the selected node.

(3) Repeat the preceding steps until all nodes of degree 2 have been eliminated.

Complications arise in selecting which N2 node in the perceived ring should be eliminated when several nonequivalent possibilities exist, as discussed later in this paper. The treatment of structures lacking N2 nodes requires special attention. Doucet's procedure is intriguing in its ability to uncover embedded rings, exemplified by the central ring in structure **13**. This structure poses a particular problem in a standard graph-walking algorithm because all vertices and edges are completely consumed after the corner triangles have been delineated, leaving no information to define the central ring. According to Doucet's procedure, the discovery of one triangle based on a corner node of degree 2 with concomitant removal of that node uncovers two-degree corners of the previously embedded three-membered ring and permits its discovery. The problem of embedded rings was encountered by Zamora[3] in his early work on a depth-first graph-walking routine applied to structure **5**, which yielded only two of the three six-membered rings. Zamora handled this problem by invoking additional edge-based searches. Structure **5** yields easily to Doucet's approach.

The second new approach to ring perception, published by Balducci and Pearlman,[2] is based on a breadth-first, message-passing search algorithm. The algorithm requires both an adjacency node list and a corresponding edge list. Attached to each node in the structure are two buffers, one to receive messages and the other to send messages. A "message" comprises the following: the initial node to which the message was attached (each node carries an initial message), the first edge traversed by the message, the last node from which the message was received, and a path record containing a history of all of the edges through which the message has been transmitted. The algorithm implements a cycle in which each *send* buffer transmits its messages to all of the neighboring *receive* buffers. Last node information in the message is used to prevent back-transmission of a message to its originating, neighboring node. After transmission, the path record and last node information in each message are updated. At this point, the path records attached to messages in the *receive* buffers at each atom and its nearest neighbors are examined for nodes or edges in common, which signal a ring closure. The component edges in the ring can be found by OR'ing together the colliding path records. The ring is checked for linear independence and then added to the SSSR. This cycle is repeated until the theoretical number (nullity) of rings has been found.

The Balducci algorithm offers several points of interest. The authors prove that their routine is of polynomial computational order, unlike a backtracting depth-first search, which is known to be of exponential computational order.[4] The basic breadth-first approach of this algorithm therefore must be of polynomial order. The implementation of dynamically updated path descriptions contained in each message is a nice solution to the problem of delineating rings. It has been recognized for some time that breadth-first search is a fast way to detect ring closures[5] and find the smallest ring, a fact confirmed recently by Qian,[6] but once a ring is found, it was necessary to trace back to the starting node in order to delineate the ring members. Because the Balducci algorithm repeatedly generates the same rings (for example, a ring-closing collision among messages will occur simultaneously at each of the six nodes in cyclohexane), an efficient test for linear independence must be included.

The new breadth-first algorithm that is the subject of this paper borrows ideas from the two methods discussed above. It uses a modified form of the Doucet node-elimination procedure to uncover embedded rings and accelerate ring perception. It avoids exponential computation order by means of a queue-based breadth-first search (BFS) and uses Balducci's solution to the problem of delineating paths in BFS by associating an individual path record with each node. Proliferation of messages, which adds substantial overhead to the Balducci algorithm, is avoided through the use of the queue and Doucet-style node selection and node elimination. For purposes of comparison with the new routine, versions of Balducci's message-passing algorithm and Doucet's depth-

[⊗] Abstract published in *Advance ACS Abstracts,* August 1, 1996.

```
atomProps = record
                atomName : string[6];
                ct,bt : array[1..6] of byte;
                degree : byte
            end;
```

Figure 1. Data structure for description of atom properties. ct carries the connection table and bt carries the bond table.

first, ring-finding algorithm were written in a common language (Borland Pascal, version 7.0 for DOS) and compiled and run on a common machine (a 486 PC running at 66 mhz).

IMPLEMENTATIONS

The three algorithms share common data structures. Each atom is represented by a Pascal record of type **atomProps**. Figure 1 shows an abbreviated form of the record used in these programs. A molecular structure is represented as an array, **S**, of atomProps. The connection table is contained in the fields, **ct**, of the array **S**. For example, **S[6].ct[3]** contains the atom index for the third connection to atom 6. Pascal sets are used wherever bit strings are required for representation of sets. The user-defined Pascal type **atomSet** is defined in the program as the Pascal type *set of 1..191*, allowing a maximum molecular size of 191 atoms.

BALDUCCI'S ALGORITHM

Implementation of the message-passing algorithm was quite straightforward because of excellent published documentation.[2] The original procedure was written in C; the present version in Pascal required only a few changes. The path description (called **BEEP** in the original work for Binary Edge Encoded Path) contained in the message record originally implemented in C as a bit string is implemented here in Pascal as a set, to which the same kinds of boolean operations can be applied; e.g., two paths, represented as sets, that are known to have collided can be OR'ed together to extract the ring in a variable **ringSet** of type **atomSet**.

An important part of the message-passing algorithm is a check against the current SSSR for the linear independence of each ring generated, which can be carried out by XOR'ing the newly generated ring with the SSSR. Because no XOR operator was available in Borland Pascal, the following equivalent process was used. The currently known Smallest Set of Smallest Rings is assumed to be contained in an array of **atomsets**, **SSSR[i]**. A working **atomSet**, **temp**, is initialized to the new **ringSet** to be tested for linear independence, and the value of **temp** is found by the following iterative computation (**sum** is defined as an atomSet):

```
for all elements i := 1 to n of the SSSR array do

                sum := temp+SSSR[i];

                temp := sum-ringSet*SSSR[i];
```

Linear independence is established if the final value of **temp** is a non-null set.

DOUCET'S RING REDUCTION ALGORITHMS[1]

In this approach, atoms of degree 2 (N2 nodes) play a central role. Structures without N2 nodes require special treatment, considered below. Assuming N2 nodes are present, an N2 node is selected, and the smallest ring containing it is found. The procedure employed by Doucet et al. for finding this smallest ring is a backtracking depth first search (DFS) starting from the selected node. After the ring has been found, the selected N2 node and the chain of N2 nodes in which it is embedded (if such a chain exists) is erased. As the result of node elimination elimination, new N2 nodes may emerge; these are added to the list of N2 nodes remaining to be treated. The process is iterated until all N2 nodes have been used. Basically, two procedures are required.

(1) **getRing(n,ringSet,ringSize)**, which returns in **ringSet** a list (in implementation, a Pascal *set*) of nodes that comprise the smallest ring containing the node **n**.

(2) **trim(n)**, which removes the node **n** from the connection table. **trim(n)** does not actually eject the node from the connection table but removes from the table all connections to node **n**, reducing its degree to zero. This avoids renumbering the connection table and consequent bookkeeping problems. Eliminated nodes are distinguishable from current nodes on the basis of node degree.

The basic algorithm comprises the following loop:

```
Repeat until all N2 nodes have been used:
   (1) Select a node N2.
   (2) Apply getRing to find the smallest ring containing
       N2. Record the ring in the SSSR.
   (3) Select an N2 node in the just-discovered ring (not
       necessarily the starting N2) and erase the ring by
       removing N2 and the chain of N2-type nodes
       containing it.
```

As discussed in further detail below, proper node selection in step (3) is critical because in some cases different selection produces a different—and possibly incorrect—SSSR. Structure **8** presents an easily visualized case; if node **a** is selected as the starting N2 node, the five-membered ring (the smallest ring) containing **a** is recorded. Now this ring can be removed either by deleting node **a** or by deleting node **b** (and its adjacent N2 node). Erasing node **a** leaves a residual seven-membered ring and yields finally an (incorrect) SSSR = [57]. On the other hand, erasing node **b** leaves a residual six-membered ring, resulting in (correct) SSSR = [56]. A special procedure, **checkNodes**, determines which N2 node in **ringSet** should be eliminated as follows. A trial N2 node in **RingSet** is eliminated. For each of the remaining chains in **RingSet** (there may be more than one, or none), the size of the smallest ring incorporating the chain (which need be found only for a single N2 node in each chain) is determined, after which the original N2 node is replaced. This is done for each chain of N2 nodes in **RingSet**. That chain of N2 nodes in **RingSet** giving the smallest ring is selected for elimination. Applying this procedure to structure **8**, the five-membered ring associated with node **a** is first removed by erasing node **a**. **GetRing** is applied to node **b** and finds a *seven-membered* ring as the smallest ring including node **b**. Now node **a** is replaced, and the five-membered ring associated with node **a** is removed again, this time by erasing node **b**. **GetRing** is applied to node **a** and finds a *six-membered* ring as the smallest ring including node **a**. Therefore, to guarantee retrieval of smallest rings, node **b** (and adjacent N2 nodes), which leaves a residual six-membered ring, should be selected for removal in preference to node **a**, which leaves a residual seven-membered ring.

Unfortunately, cases can be found in which this scheme for node selection fails; structure **11** is such a case. If node **a** in structure **11** is selected, there are no N2 nodes other than **a** in its smallest ring, so node **a** will be removed and the final SSSR will be [558]. Even if an N2 node from the six-membered ring is selected, the operation of the procedure **checkNodes** discussed above (created to handle one type of selection problem) will cause selection and removal of node **a**. Only in the circumstance that one of the rightmost N2 nodes in structure **11** is first removed would **checkNodes** give the correct SSSR = [556]; in other words, the outcome depends upon the way in which the structure is numbered, which will determine the sequence in which N2 nodes are selected for elimination. Consequently, structures such as **12** containing problem node arrangements may or may not give correct SSSR's; in the case of structure **12**, initiation of N2 node clearance outside the seven-membered ring containing problem node **a** will give a correct SSSR, and initiation of N2 node clearance in the seven-membered ring (which always results in the removal of node **a** because of the operation of **checkNodes**) will give an incorrect SSSR. Assuming random numbering, the starting probability of obtaining a correct SSSR in this case is 0.43. Doucet[7] proposes to handle this problem by treating the disposition of an N2 node placed between two N3 nodes as a special case. Unfortunately, one can derive structures from **11** that will fail as special cases and fail in **checkNodes** as well. A fundamental solution to the problem of node selection is considered in the section describing the new BFS approach.

If no N2 nodes are present (see structures **14**−**17**), a slightly different approach is used by Doucet et al. A node of degree 3 (an N3 node) is selected, and **getRing** is applied to find **ringSet**, the smallest ring containing the node; the ring is added to the SSSR. We will refer to this as "the source ring". An N3 node in the source ring is temporarily erased, which reduces by one the degree of its attached nodes. This N3 node is provisionally marked for elimination. **getRing** is applied to each of the resulting N2 nodes, and the size of the smallest ring in the N2 node set is noted. The N3 node in the source ring is restored. The process is repeated for the next N3 node in the source ring; if the size of the smallest ring attached to any of the members new N2 set is less than that in a previous set, the N3 node currently selected from the source ring replaces the one provisionally marked for elimination. This process continues until all N3 nodes in the source ring have been processed. Finally, the marked N3 node is erased, thus removing the source ring and generating a set of N2 nodes which serve as foci for further ring perception. In this study of the Doucet algorithm, treatment of N3 nodes similar to that just described is used, except that *edge* removal rather than *node* removal is the basis for cracking open the ring assembly. The procedure **checkEdges** that implements this operation is described in detail below.

THE NEW BFS APPROACH

Two major modifications were made in Doucet's approach in developing a new ring perception routine. The first modification is replacement of the DFS search in the **getRing** module (which returns the smallest ring at a specified node) with its BFS equivalent. The second modification takes care of the problems involved in selection of N2 nodes for elimination.

The basic problem accompanying node elimination is loss of information. In consideration of structure **11**, we note that after recording the information that node **a** is a member of the (smallest) five-membered ring, subsequent removal of **a** destroys the six-membered ring required later for correct ring perception. The solution to this problem is to forego node elimination until *all* N2 nodes have been assigned to smallest rings, then eliminate the current set of N2 nodes *en bloc*. This obviates the need for the procedure **checkNodes** and treatment of special cases that fail in **checkNodes**, giving a simpler and faster program. The new program requires the following modules:

(1) **getRing(n,ringSet,ringSize)**... a breadth-first routine that returns in **ringSet** (a Pascal set) the smallest ring containing the node **n**.

(2) **trim(n)**...removes from the connection table all connections to node **n**, leaving that node with degree 0.

(3) **checkEdges(n,ringSet)**... selects an optimum edge for elimination in structures without N2 nodes. Each edge in **ringSet** is selected in turn for trial, and the edge is restored after its trial is finished. The larger of the two rings that includes the end nodes of the trial edge is noted. Final selection for edge elimination is that edge that is incident on the smallest of the collection of these largest rings. After an edge is removed, new N2 nodes will appear to serve as foci for further ring perception.

With these procedures in hand, the following pseudocode algorithm for ring perception can be stated. **trimSet** is the set of nodes of degree one that have been removed by **trim**. Ultimately all nodes are reduced to degree one (by bond cleavages) and removed; the algorithm terminates when **trimSet** covers the whole structure. **nodesN2** is the current set of N2 nodes.

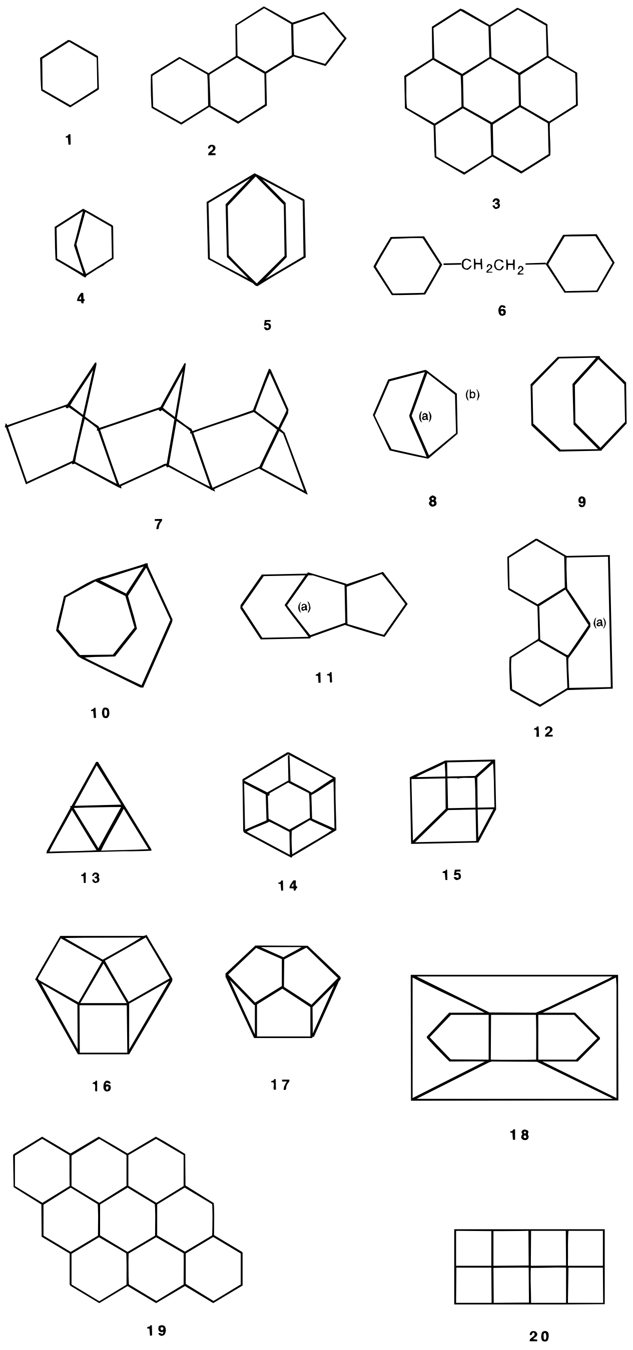
```
fullSet := [1..OKatoms];  {All atoms in fullSet}
trimSet := [];  {initialize to the empty set}
repeat
  Add nodes of degree 0 to trimSet
  Add nodes of degree N2 in fullSet-trimSet to nodesN2
  Find a node init in fullSet-trimSet having minimum degree
  If minimum degree=1 then trim(init) and add init to trimSet
  else if minimum degree =2 then
  begin
    for each node i in nodesN2
    begin
      getRing(i,ringSet,ringSize);
      if ringSize>0 then
         check SSSR for a duplicate of ringSet
         if no duplicate is found add ringSet to the SSSR
     end
     for each chain of N2 nodes, isolate one N2 node and break
     one bond
  end
  else if minimum degree=3 then
  begin
    getRing(init,ringSet,ringSize)
    if ringSize>0 then save ringSet in SSSR
    checkEdges(init,ringSet)
  end
until fullSet=trimSet
```

The algorithm gives priority to removal of nodes of degree 1 (which are added to **trimSet**) and automatically removes substituents at ring systems. If the minimum node degree is 2, the **getRing** procedure is called for each node in **nodesN2**, and the rings returned in **ringSet** are saved in the SSSR when they are not duplicates. In the actual program, not all N2 nodes need be placed in **nodesN2:** one N2 node from a chain of such nodes can be selected. In naphthalene, for example, only two N2 nodes are required for complete ring retrieval. A check for ring duplication is essential because in some instances it is impossible to avoid selecting N2 nodes that return the same ring (the *peri* atoms in anthracene are an example). After ring retrieval one bond is broken at each N2 node in the set **NodesN2**, which reduces the degree of these nodes to unity. Subsequent processing loops will clean out these one-degree nodes, which are added to **trimSet**. Removal of one degree nodes will create new ones when chains are processed; thus, the algorithm contains an implicit mechanism for clearing substituents prior to ring perception.

If the minimum node degree is 3, the **checkEdges** procedure locates a suitable edge in **ringSet** and removes it, reducing end nodes of the edge to degree 2. These nodes subsequently become centers for smallest ring retrieval and elimination in subsequent loop cycles. Processing is complete when all nodes have joined the **trimSet**.

THE DEPTH-FIRST SEARCH ALGORITHM (DFS)

The neighbors of a selected starting node, **n**, are placed in an **atomSet** named **goal**. An array, **map**, and a pointer, **mptr**, to the current element in **map** are established; **map[1]** is initialized to **n** (the starting node), and **mptr** is initialized to unity. Node **n** is marked "visited". A variable **cost** is attached to each **map** entry. This variable is the length of the path traversed from **n**. For **map[1]**, **cost** has the value *zero*. Two subprocedures implement the search. The procedure **candidate** returns an unvisited node attached to the last node in **map**; if it cannot find such a node, it returns a node value of zero, otherwise the **map** pointer, **mptr**, is incremented, the found node is stored in the new **map** element, the node is marked as *visited* and the value of **cost** associated with the previous **map** element is incremented and attached to the new **map** element. If the search reaches a deadend (procedure **candidate** returns a node value *zero*), a **backtrack** procedure reverses the direction of search and moves back through **map** (decrementing **mptr** and unmarking visited nodes) until it finds a **map** node that still has candidates attached. If such a node is found, the search will move forward again. If it cannot find such a node, the algorithm terminates. In this way, the program is forced to explore all branches in the structure. If the node stored in the last element in the **map** array is in the **goal** set, ring closure has been encountered. If the cost attached to the colliding node is less than the current minimum cost (a variable, **minCost**, originally assigned the largest possible integer value, maxInt), the nodes in **map** array, which constitute the ring nodes, are stored in **ringSet**, and **minCost** is set to the cost attached to the colliding node. The search continues until all nodes have been explored. Because of backtracking, this algorithm is of exponential computational order (NP complete). Searching is abbreviated somewhat by the fact that once a value has been established for **minCost**, exploration of any other path for which the current **cost** reaches **minCost** can be discontinued and



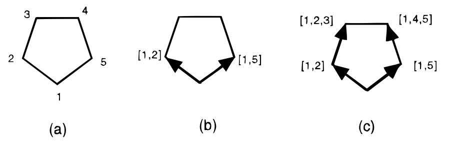


Figure 2. Illustration of breadth-first search and generation of path sets for a simple cycle. Valid ring closure occurs when path sets on adjacent atoms (3 and 4) have one element in common.

backtracking will occur. This heuristic for improving search efficiency is known as "branch and bound".

THE BREADTH-FIRST SEARCH ALGORITHM (BFS)

Source code for the BFS algorithm is given in the Appendix. Before considering details of implementation, a simple, pictorial example will make clear the basis for the algorithm. We wish to find the smallest ring in Figure 2a that includes node 1. Attached to each node **i** in the structure

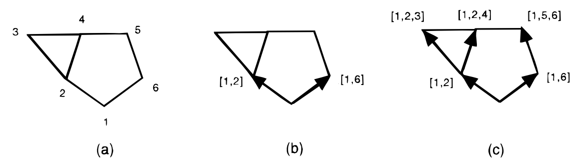


Figure 3. Illustration of breadth-first search for a five-membered ring in the presence of an interfering three-membered ring. The path sets on adjacent atoms 3 and 4 have more than one element in common; this does not represent a valid ring closure.

is a variable, **path[i]** (a Pascal set), which will contain the nodes in the path leading from the starting node (node 1 in this example) to node **i**.

We start the computation by assigning initial values to the paths associated with the nodes attached to the starting node 1; these are path[2] = [1,2] and path[5] = [1,5] (Figure 2b); all other paths are empty, initialized to the empty set, []. Starting from node 2, we move to attached node 3, but before doing so, we examine the path for node 3 to determine whether or not it is empty. If it is not empty, a closed path has been found, with a possible ring closure. In this case, path[3] is empty, so there is no closure yet. We proceed to assign a value, path[3] = path[2] + [3] = [1,2,3]. In the same way, in breadth-wise fashion, we extend the path from node 5 to node 4 (Figure 2c), after testing for closure; path-[4] = path[5] + [4] = [1,4,5].

When we consider extending the path from node 3 to node 4, we find that path[4] is now non-null. Path collision has occurred with possible ring closure. To test for *valid* ring closure, we compute the intersection path[3]*path[4] = [1,2,3]*[1,4,5], which has the value [1]. *A valid ring closure must have a singleton path intersection.* With valid ring closure, the set of ring nodes is obtained as the sum (union) of the two colliding paths, path[3] + path[4] = [1..5].

The singleton path intersection requirement for ring closure allows breadth-first search to operate successfully in the presence of interfering internal branches and in the presence of connecting acyclic methylene chains. An example of a structure with an interfering branch appears in Figure 3a. We are to start the search from node 1. The problem here is that the routine must select the five-membered ring, not the six-membered envelope. The first stage of BFS gives path[2] = [1,2] and path[6] = [1,6] (Figure 3b) after intersection tests. In the next stage, Figure 2c, because of branching at node 2, three paths are evaluated: path[5] = [1,5,6]; path[3] = [1,2,3]; path[4] = [1,2,4], after intersection tests. Now we attempt to extend BFS from node 3 to node-[4] and note that path[4] is now non-null; i.e., a collision has occurred. We compute the intersection for path[3] and path[4] = [1,2,3]*[1,2,4] = [1,2], which is not a singleton intersection, and which therefore is not a valid ring closure. In consequence, the path from node 3 to node 4 is ignored, and subsequent BFS proceeds normally through node **4** to give ultimately closure to the five-membered ring [1,2,4,5,6]. Implementation of the breadth-first routine uses a queue, a standard data structure for BFS.[8] Elements are added to the back of a queue and retrieved from the front, i.e., on a first-in, first out (FIFO) basis. The queue is initialized by placing on it the nodes $\mathbf{m_1}$,$\mathbf{m_2}$,... attached to the given root node **n**. Each node placed on the queue carries with it its source node and a path. In pseudo code, the operation of BFS can be described as follows:

Chart 1

```
procedure getRing(rootNode : integer; var ringSet : atomSet;
                  var ringSize : byte);
type qPtr = ^queue;
     queue = record
               atomIndex,source : byte;
               next : qPtr
             end;
var
    i,j,frontNode,m,k : integer;
    fPtr,bPtr,tmpPtr : qPtr;
    intersection : atomSet;

begin
  for i := 1 to OKatoms do path[i] := [];
 {Ringset returns the smallest ring that contains rootNode}
  ringSize := 0; ringSet := [];
  new(fptr); fptr^.next := NIL; bptr := fptr;
 {Initialize the queue with nodes attached to rootNode}
  with S[rootNode] do for j := 1 to degree do if
  S[ct[j]].degree>0 then
  begin
    if j>1 then
    begin
      new(bptr^.next);
     bptr := bptr^.next;
     bptr^.next := NIL
   end;
   bptr^.source := rootNode;
   bptr^.atomIndex := ct[j];
{Initialize paths for the new queue elements}
   path[ct[j]] := [rootNode]+[ct[j]];
 end;
 while true and (fptr<>NIL) do
 begin
{get data from the front pointer}
   frontNode := fptr^.atomIndex;
   with S[frontNode] do for j := 1 to degree do
   begin
{Examine nodes m attached to frontNode. Avoid frontNode's source.}
     m := ct[j];
     if (S[m].degree>0) and (m<>fptr^.source) then
     begin
{Collision occurs when attached node m's path is no longer empty}
       if path[m]<>[] then
       if intersection<>[] then
       begin
         intersection := path[frontNode]*path[m];
{Valid collision only if intersection is a singleton}
         k := 0;
         repeat inc(k) until k in intersection;
         if intersection-[k]=[] then   {Singleton check}
         begin
{Return data to the calling program}
           ringSet := path[frontNode]+path[ct[j]] + [theNode];
           for i := 1 to OKatoms do if i in ringSet then
              inc(ringSize);
{Clean out pointers: memory management}
           repeat
             tmpPtr := fptr^.next;
             dispose(fptr);
             fptr := tmpPtr
           until tmpPtr=NIL;
           exit
         end
       end
       else begin
{Update the path m.}
         path[m] := [m]+path[m]+path[frontNode];
{Put attached node m onto the back of the queue.}
         new(bptr^.next);
         bptr := bptr^.next;
         with bptr^ do
         begin
           next := NIL;
           source := frontNode;
           atomIndex := ct[j]
         end  {with bptr^ do..}
       end  {intersection=[]}
     end  {if S[m].degree>0 ...}
   end;   {for j := 1 to degree...}
 {Release the front pointer}
   if fptr<>NIL then
   begin
     tmpPtr := fptr^.next;
     dispose(fptr);
    fptr := tmpPtr
   end
  end {while true do..}
end;
```

1. Remove node **frontNode** and its **source** from the front of the queue.
2. For each node **m** attached to **frontNode**, and not equal to source:

   If path[m] is null, compute path[m] = path[frontNode] +[m] and put node **m** (with its source, **frontNode**) on the back of the queue.

   If path[m] is not null then

   1) Compute the intersection path[frontNode]*path[m].

   2) If the intersection is a singleton, compute the ring set = path[m]+path[frontNode] and exit.
3. Return to step 1.

Table 1[a]

structure	no. atoms	run time, s DFS	BBFS	BFS
1	6	0.006	0.006	0.005
2	17	0.022	0.066	<0.001
3	24	0.038	0.149	0.016
4	7	0.006	0.011	0.005
5	10	0.017	0.027	0.005
6	14	0.011	0.043	0.011
7	18	0.044	0.094	0.016
8	8	0.005	0.022	0.005
9	10	0.011	0.038	0.006
10	10	0.011	0.038	0.011
11	11	0.011[b]	0.038	0.005
12	15	0.028	0.087	0.011
13	6	0.006	0.006	<0.001
14	12	0.044	0.050	0.011
15	8	0.005	0.022	0.005
16	9	0.027	0.011	0.017
17	10	0.033	0.028	0.011
18	14	0.044	0.050	0.011
six-ring arrays in echelon; cf. structure **19**				
2×2	16	0.016	0.066	0.005
3×3	30	0.088	0.236	0.027
4×4	48	0.440	0.599	0.050
5×5	70	3.180	1.279	0.143
6×6	96	147.0	2.439	0.159
four-ring rectangular arrays; cf. structure **20**				
5×10	66	53.88	0.736	0.143
6×10	77	155.4	1.028	0.159
7×10	88	435.4	1.373	0.203
10×10	121		2.714	0.373

[a] DFS, Doucet's method;[1] BBFS, Balducci's method;[2] BFS, new method described in this paper. [b] Incorrect SSSR 557. See text.

Pascal source code that implements this algorithm appears in Chart 1. The queue is implemented as a linked list of records, each record containing a node and its source.

RESULTS AND DISCUSSION

Computations were made in parallel with all three algorithms using a test suite of structures containing embedded rings, nodes of degree 3 only, and acyclic nodes. To test the dependence of run time on problem size, various networks of six-membered rings arranged in echelon as in structure **19** and rectangular arrays of four-membered rings as in structure **20** were tested. Results are shown in Table 1. BBFS is the Balducci BFS message-passing algorithm, BFS is the new BFS routine; and DFS is the backtracking depth-first routine. All runs gave the correct number of rings in the SSSR except structure **11**, which failed in DFS for reasons previously discussed. Structure **18**, taken from a paper by Elk,[9] is of interest. Elk points out, quoting Dyson, that as drawn, a casual deduction of the SSSR would yield two 7-rings, two 5-rings, and three 4-rings. The structure can be redrawn to display two 5-rings and five 4-rings, the ring population found by each of the ring perception algorithms.

Runs were made on a 66 Mhz Micron 486 Personal Computer. Ten passes through each algorithm were made for each structure (except for DFS applied to structures with more than 30 atoms) to improve timing precision by averaging. Differences between the three algorithms were small for small structures, although the new BFS routine was fastest in most cases. With larger structures, the BFS algorithm is roughly six times faster than the Balducci (BBFS) routine, reflecting the overhead burden of the Balducci routine. Evidence of exponential computational order for the DFS algorithm appears in Table 1; for the 7×10 array of four-membered rings, the BFS algorithm is over 2000 times as fast as the DFS algorithm. With a 10×10 four-membered ring array, BFS returned results in 0.373 seconds, but after half an hour, the DFS routine still had not returned results. Such behavior is not unknown for DFS algorithms; Qian[6] reports CPU times for DFS ring perception of over 30 min on a mainframe computer in certain cases having many deep paths. Results parallelling those found with four-membered ring arrays were obtained with six-membered ring arrays.

APPENDIX

In Chart 1 is a Pascal source code listing for the breadth-first search algorithm for finding the smallest ring that includes given node, **rootNode**. The variable **S** containing the chemical structure is defined globally as an array[1..191] of **atomProps** (defined in Figure 1); each element of **S** contains a description of an atom in the structure. **OKatoms**, also a global variable, is the number of atoms in the structure.

REFERENCES AND NOTES

(1) Fan, T. F.; Panaye, A.; Doucet, J.-P.; Barbu, A. Ring Perception. A New Algorithm for Directly Finding the Smallest Set of Smallest Rings from a Connection Table. *J. Chem. Inf. Comput. Sci.* **1993**, *33*, 657−662.
(2) Balducci, R.; Pearlman, R. S. Efficient and Exact Solution of the Ring Perception Problem. *J. Chem. Inf. Comput. Sci.* **1994**, *34*, 822−831.
(3) Zamora, A. An Algorithm for Finding the Smallest Set of Smallest Rings. *J. Chem. Inf. Comput. Sci.* **1976**, *16*, 40−43.
(4) Garey, M. R.; Johnson, D. S. Computers and Intractability. Freeman: 1979.
(5) Downs, G. M.; Gillett, V. J.; Holliday, J. D.; Lynch, M. F. Review of Ring Perception Algorithms for Chemical Graphs. *J. Chem. Inf. Comput. Sci.* **1989**, *29*, 172−187.
(6) Qian, C.; Fisanick, W.; Hartzler, S. W.; Chapman, S. W. Enhanced Algorithm for Finding the Smallest Set of Smallest Rings. *J. Chem. Inf. Comput. Sci.* **1990**, *30*, 105−110.
(7) Doucet, J. P. private communication.
(8) Horowitz, E.; Sahni, S. Fundamentals of Data Structures in Pascal. Computer Science Press: 1984.
(9) Elk, S. B. Derivation of the Principle of Smallest Set of Smallest Rings from Euler's Polyhedron Equation and a Simplified Technique for Finding This set. *J. Chem. Inf. Comput. Sci.* **1984**, *24*, 203−205.

CI960013P