CROATICA CHEMICA ACTA CCACAA **69** (3) 837–843 (1996)

ISSN-0011-1643
CCA–2345 *Original Scientific Paper*

Computing the Szeged Index

Janez Žerovnik

Institute of Mathematics, Physics and Mechanics, Department of Theoretical Computer Science, Jadranska 19, Ljubljana, Slovenia

and

Faculty of Mechanical Engineering, University of Maribor, Smetanova 17, Maribor, Slovenia

Received February 22, 1996; revised March 25, 1996; accepted April 9, 1996

We give an explicit algorithm for computing the Szeged index of a graph which runs in $O(mn)$ time, where n is the number of nodes and m is the number of edges.

INTRODUCTION

Chemists employ structural formulae in communicating information about molecules and their structure. *Structural* or *molecular graphs* are mathematical objects representing structural formulae. These objects encode impotant properties of the chemical structures.

A *topological index* is a numerical quantity derived in an unambiguous manner from the structural graph of a molecule. These indices are graph invariants, which usually reflect molecular size and shape[1] (for further examples of topological indexes see, for example, Refs. 2,3).

The Szeged index Sz is a recently proposed structural descriptor, based on the distances of vertices in a molecular graph.[4] As one can expect,[5] it is not unlike the Wiener index, the first[6] and still the most studied topological index.[7–13] For the reasons for introducing the Szeged index and for basic properties of Sz, see Ref. 4,5. Ref. 14 compares the Szeged index with the Wiener index. Since the two indices coincide on trees, there is a linear algorithm for computing the Szeged index of trees.[15] Recently, an algorithm was given for computing the Szeged index for benzenoid hydrocarbons.[16]

In this note, we give an explicit algorithm for computing the Szeged index of an arbitrary graph. The algorithm has a running time $O(mn)$, where n is the number of vertices and m is the number of edges of a graph. Space complexity of the algorithm is $O(mn)$. This is the same complexity as the complexity of the best known algorithm for computing the Winer index of arbitrary graphs[15] and hence we believe it is likely to be the best possible.

GRAPHS

We use the usual graph theoretical terminology (as, for instance, in Ref. 17). A *graph* G = (*V*,*E*) is a combinatorial object consisting of an arbitrary set $V = V(G)$ of *vertices* and a set $E = E(G)$ of unordered paris $\{x,y\} = xy$ of distinct vertices of G, called *edges*. A *simple path* from x to y is a sequence of distinct vertices $P = x_0, x_1, ..., x_l$ such that each pair x_i, x_{i+1} is connected by an edge and $x_0 = x$ and $x_l = y$. The *length* of the path is the number of edges $l(P) = l$. For any pair of vertices x, y, we define the *distance* $d(x,y)$ to be the length of the shortest path between x and y. If there is no (finite) path, we define $d(x,y) = \infty$. We will also write $d(u,v) = d_u(v) = d_v(u)$. A graph G is *connected if* $d(x,y) < \infty$ for any pair of vertices x, y. Here we will consider only connected graphs.

BREADTH FIRST SEARCH ALGORITHM

Breadth first search algorithm (shortly BFS) is a well known method for searching a graph (see, for example, Refs. 15, 18). We now recall briefly the idea of the algorithm. Starting with a given vertex v, a partial tree W of G is constructed. Whenever the current vertex has new neighbours, we add the neighbours to the tree W and put them in a so called FIFO (first in first out) queue. It can be shown that the order of visiting vertices has the property that the vertices of smaller distance from the starting vertex v appear before vertices which are more distant from v.

BFS algorithm scheme can be writtten as follows:

```
let queue = (v) and mark y as old
while queue is not empty do begin
   x := front(queue)
   if x is adjacent to new vertex y
      then add y at the tail of the queue and mark y as old
      else remove x from the queue
   endif
endwhile
```

We get the distances from the starting vertex v if we put $d(v) = 0$ initially, and for every new vertex y (adjacent to x), we put $d(y) = d(x) + 1$ when adding y to the tail of the queue.

ALGORITHM FOR THE SZEGED INDEX

Before giving the algorithm let us recall the definition of the Szeged index.

$$Sz(G) = \sum_{e \in E(G)} n_1(e|G)\, n_2(e|G)$$

where the sum runs over all edges of G and the numbers $n_1(e|G)$ and $n_2(e|G)$ are cardinalities of the sets $N_1(e|G)$ and $N_2(e|G)$. $N_1(e|G)$ is the set of vertices of G which are closer to u than to v, where u and v are the endpoints of e. More formally, $i \in N_1(e|G)$ if $d(i,u) < d(i,v)$, *i.e.* if $d_u(i) - d_v(i) < 0$. Similarly, $i \in N_2(e|G)$ if $d_u(i) - d_v(i) > 0$. The contribution of edge e to the Szeged index is thus a product of the number of positive entries times the number of negative entries of the vector $d_u(i) - d_v(i)$.

1. for all $v \in V$ compute $d_v(i)$ using BFS
2. $Sz := 0$
   for all $e = uv \in E$ do
      $n_1 := 0$ $n_2 := 0$
      for all $i \in V$ do
         if $d_u(i) - d_v(i) < 0$ then $n_1 := n_1 + 1$
         else if $d_u(i) - d_v(i) > 0$ then $n_2 := n_2 + 1$
         endif
      endfor
      $Sz := Sz + n_1 * n_2$
   endfor

Time complexity of the algorithm is $O(mn)$, where n is the number of vertices and m is the number of edges. At step 1, we have a call of BFS for every vertex as a starting vertex, hence $O(nm)$. At step 2, the outer for loop goes over all edges, and the inner loop goes over all vertices, again $O(mn)$. Note that this is the same as the time complexity of computing the Wiener index.[15]

Note that the algorithm is highly parallel. There are many possible models of parallel computation. For this short discussion, we choose a simple model, where a polynomial number of processors is assumed available.[18] An algorithm in this model is considered to be efficient if its time complexity is a polynomial function of the logarithm of the size of the input. Szeged index can be computed in parallel time $O(\log^2 n)$ as follows:

1. compute the distances $d_v(i)$ for all v and all i
2. for all edges $e = uv$ do in parallel
   for all i do in parallel $diff_e(i) := d_u(i) - d_v(i)$ endfor
   $n_{1,e}$:= the number of positive entries of vector $diff_e$
   $n_{2,e}$:= the number of negative entries of vector $diff_e$
   Sz := sum all $n_{1,e} * n_{2,e}$
endfor

By a method avoiding BFS, the distance matrix can be computed in $O(\log^2 n)$ time using $O(n^3/\log n)$ processors, for details we refer to Refs. 19, 20. Step 2 is even more easy to parallelize, since computation for each edge is independent. Furthermore, the inner loop can be seen as first computing a difference of two vectors, which is an independent task for each vector entry, and then counting the number of positive and negative signs in the resulting vector, which can be completed in $O(\log n)$ time. Summing up the contributions due to different edges gives another factor of $O(\log m) = O(\log n)$. The overall time complexity of step 2 is therefore $O(\log^2 n)$ on $O(mn)$ processors.

While the above considerations are rather theoretical, it should be noted that the algorithm uses operations on vectors and hence admits high speed-ups also on vector parallel processors, which may be of more practical importance on the currently available parallel computer equipment.

EXAMPLES

The implementation of the algorithm given below was tested on graphs for which formulas are known. The test examples included some polyacenes, L_h, h = 1, 2, ..., 24 and coronere/circumcoronere graphs H_n. In both cases, the results were consistent with the formulas of Gutman and Klavžar.[16]

We conclude with the listing of a Pascal program in which only the procedure get graph, which should provide an input graph as adjacency list, is omitted.

```
    program SZEGED(input,output);
(*
This program calculates the Szeged index of a graph. Graph is given by
adjacency lists of its vertices.
*)
  const size = 100; max..degree = 10;
  type vector = array [1..max degree] of integer;
     vector_of_vectors = array[1..size] of vector;
  var Degree: vector;                    (*VERTEX DEGREES*)
```

```
        Adj_list: vector_of_vectors;          (*ADJACENCY LISTS OF THE GRAPH*)
        Distance: array[1..size] of vector; (*DISTANCES*)
        n: integer;                           (*NUMBER OF VERTICES*)
        Sz: longint;
        u, v, i, j, dif : integer;
        n1, n2 : integer;
        Visited: array[1..size] of boolean;
(* Data structure FIFO queue *)
        queue: vector;
        First,Last: integer;
procedure reset_queue;
    begin First:=1; Last:=0; end;
procedure append_queue(x: integer);
    begin queue[Last +1] := x; Last := Last + 1; end;
procedure get_queue(var x: integer);
    begin x:= queue[First]; First := First + 1; end;
function empty_queue:boolean;
    begin if First>Last then empty_queue:= true else empty_queue:= false end;
(* Breadth first search *)
procedure BFS(Start_vertex:integer);
var i:integer; neighbor: integer;
begin
    for i:=1 to n do Visited[i]:= false;
    reset_queue;
    Distance[Start_vertex][Start_vertex]:= 0; Visited[Start_vertex] := true;
    for i:=1 to Degree[Start_vertex] do begin
        neighbor:= Adj_list[Start_vertex][i];
        append_queue(neighbor); Visited[neighbor] := true;
        Distance[Start_vertex][neighbor] := 1;
    end;
    while (not(empty_queue)) do begin
        get_queue(u); Visited[u]:= true;
        for i:=1 to Degree[u] do begin
            neighbor:= Adj_list[u][i];
            if not(Visited[neighbor]) then begin
                append_queue(neighbor); Visited[neighbor]:= true;
                Distance[Start_vertex][neighbor] := Distance[Start_vertex][u] + 1;
            end;
        end;
    end;
end(* BFS *)
```

```
(* Main *)
begin
   get_graph; (*procedure which reads or generates a graph*)
   for v:=1 to n do BFS(v);
   Sz := 0;
   for v:=1 to n do for j:=1 to Degree[v] do begin
      u:= Adj_list[v][j];
      if u>v then begin
         n1:=0; n2:=0;
         for i:= to n do begin
            diff := Distance[v][i] – Distance [u][i];
            if diff<0 then n1:= n1+1;
            if diff>0 then n2:= n2+1
         end;
      Sz := Sz + n1 * n2;
      end;
   end;
   writeln( 'Szeged index =', Sz:0);
end.
```

REFERENCES

1. D. H. Rouvray, *Sci. Am.* (Sept.1986) 40–47.
2. A. T. Balaban, I. Motoc, D. Bonchev, and O. Mekenyan, in: *Topic in Current Chemistry* **114** No.21, Springer, Berlin/Heidelberg 1983, 21–33
3. I. Fabič-Petrač, B. Jerman-Blažič, and V. Batagelj, *J. Math. Chem.* **8** (1991) 121–134.
4. I. Gutman, *Graph Theory Notes, New York,* **27** (1994) 9–15.
5. P. V. Khadikar, N. V. Deshpande, P. O. Kale, A. Dobrynin, I. Gutman, and G. Dömötör, *J. Chem. Inf. Comput. Sci.* **35** (1995) 547–550.
6. H. Wiener, *J. Am. Chem. Soc.* **69** (1947) 2636–2638.
7. S. S. Tratch, M. I. Stankevitch, and N. S. Zefirov, *J. Comp. Chem.* **11** (1990) 889–908.
8. A. Graovac and T. Pisanski, *J. Math. Chem.* **8** (1991) 53–62.
9. I. Lukovits, *Int. J. Quantum Chem.: Quantum Biol. Symp.* **19** (1992) 217–223.
10. T. Pisanski and J. Žerovnik, *J. Chem. Inf. Comput. Sci.* **34** (1994) 395–397.
11. I. Lukovits, Croat. *Chem. Acta* **68** (1995) 99–103.
12. M. Juvan and B. Mohar, *J. Chem. Inf. Comput. Sci.* **35** (1995) 217–219.
13. S. Nikolić, N. Trinajstić, and Z. Mihalić, *Croat. Chem. Acta* **68** (1995) 105–129.
14. S. Klavžar, A. Rajapaxi, and I. Gutman, *Appl. Math. Lett.* **9** (1996) 45–49.
15. B. Mohar and T. Pisanski, *J. Math. Chem.* **2** (1988) 267–277.
16. I. Gutman and S. Klavžar, *J. Chem. Inf. Comput. Sci.* **35** (1995) 1011–1–14.
17. N. Trinajstić, *Chemical Graph Theory*, CRC Press, Boca Raton, FL., 1992.
18. N. L. Biggs, *Discrete Mathematics*, Claredon Press, Oxford, 1989.

19. A. Gibbons and W. Rytter, *Efficient Parallel Algorithms*, Cambridge University Press, Cambridge, 1988.
20. J. Žerovnik, *Lecture Notes in Computer Science* **591** (1992) 359–368.

SAŽETAK

Računanje Szeged indeksa

Dan je algoritam kompleksnosti $O(mn)$ za računanje Szeged indeksa proizvoljnog grafa gdje n označava broj čvorova a m broj grana grafa.