A Square Time Algorithm for Cyclic Edge Connectivity of Planar Graphs

Dingjun Lou

Department of Computer Science
Sun Yatsen University
Guangzhou 510275
People's Republic of China

Abstract

In this paper, we introduce an $O(n^2)$ time algorithm to determine the cyclic edge connectivity of a planar graph, where n is the order of the planar graph. This is the first correct square time algorithm for cyclic edge connectivity of planar graphs.

Keywords: Square time algorithm; cyclic edge connectivity; planar graph

1 Introduction and terminology

All graphs in this paper are finite, undirected and connected. Our algorithm only determines the cyclic edge connectivity of simple planar graphs. For the terminology and notation not defined in this paper, the reader is referred to [2].

Let $G = (V, E)$ be a graph, where V is the vertex set of G and E is the edge set of G. We denote |V| by $n(G)$ and |E| by $m(G)$. Let $S \subseteq V(G)$. The subgraph of G with vertex set S and edge set consisting of all edges in G with both ends in S is called the induced subgraph of G on S, denoted by $G[S]$. Let v be a vertex of G. We denote by $d(v)$ the degree of v in G.

A graph G is called a planar graph if it can be drawn in the plane so that its edges intersect only at their ends. Such a drawing of a planar graph G is called a planar embedding of G. A planar embedding of G partitions the rest of the plane into a number of connected regions. The closures of these regions are called the faces of G. We denote by $\phi(G)$ the number of faces of G. Given a planar embedding of G, we can define another graph G^* as

follows: corresponding to each face f of G there is a vertex f^* of G^*, and corresponding to each edge e of G there is an edge e^* of G^*; two vertices f^* and g^* are joined by the edge e^* in G^* if and only if their corresponding faces f and g are separated by the edge e in G. Then G^* is called the dual graph of G.

Let $C = v_1v_2\cdots v_kv_1$ be a cycle in a graph G. Then $|V(C)| = k$ is the length of C. A cycle of length k is called a k-cycle. A chord of C is an edge xy in G such that $x, y \in V(C)$ but $xy \notin E(C)$. A minimal cycle C of G is a cycle without chords. Let G be a connected planar graph and be a planar embedding, and C be a cycle in G. If there is a vertex of G in the inner region and outer region of C respectively, then C is called a separating cycle of G.

Let G be a connected graph. A cut vertex w (cut edge e) of G is a vertex (an edge) such that $G-w$ ($G-e$) is not connected. The biconnected components of G are those maximal subgraphs of G that have no cut vertex. The connectivity (edge connectivity) of G, denoted by $\kappa(G)$ ($\lambda(G)$) is the minimum number of vertices (edges) whose deletion disconnects G. If $G = K_n$, then $\kappa(G) = n - 1$. A cyclic edge cutset S of G is an edge cutset whose deletion disconnects G such that at least two of components of $G - S$ contain a cycle respectively. The cyclic edge connectivity, denoted by $c\lambda(G)$, is the cardinality of a minimum cyclic edge cutset of G. If no cyclic edge cutset exists in G, we say that $c\lambda(G) = \infty$.

By Lemma 2 in [8], deleting a pendant tree from a graph G does not alter the cyclic edge connectivity of G. Let G be a simple connected graph and let G' be obtained from G by deleting all pendant trees (G' is a single vertex if G is a tree). We call the graph G' the reduced form of graph G. By Lemma 3 in [8], we have $c\lambda(G') = c\lambda(G)$.

The concept of cyclic edge connectivity was introduced by Tait [14] in the proof of the Four Colour Theorem. Plummer [13] showed that the cyclic edge connectivity of 5-connected planar graphs is at most 13, but that of 4-connected planar graphs can be any natural number at least 4. So it is important to find out the cyclic edge connectivity of a planar graph fast. In [4] and [6], Holton, Lou and Plummer showed the relation between cyclic edge connectivity and n-extendable graphs. In a paper of Peroche [12], several sorts of connectivity, including cyclic edge connectivity, and their relation are studied. Nedela and Skoviera [11] introduced the concept of atom. If B is a cyclic edge cutset of size $c\lambda(G)$, then a subgraph P of G such that each edge in B has exactly one incident vertex in P is called a cyclic part. An atom is a cyclic part that is minimal under inclusion. In [3], Dvorak, Kara, Kral and Pangrac obtained the first efficient algorithm to determine the cyclic edge connectivity of cubic graphs. Lou and Wang [7] and [8] obtained the first efficient algorithm to determine the cyclic edge connectivity of k-regular graphs for $k \geq 3$ and an efficient algorithm to

determine whether a general graph has infinite cyclic edge connectivity.

In [9], Yuntin Lu and Xiu Lu claimed to obtain an $O(n^2)$ time algorithm to determine the cyclic edge connectivity of planar graphs. The idea of their algorithm is correct, but the time complexity analysis is wrong, which we shall explain in next section. In this paper, we shall introduce a correct algorithm to determine the cyclic edge connectivity of simple planar graphs, of which the time complexity is $O(n^2)$.

2 A simple algorithm for cyclic edge connectivity

First, we show a theorem which is the base of the following simple algorithm and our main algorithm.

Theorem 1: Let H" be a planar embedding of a planar graph H'. Then any minimal cyclic edge cutset of H' is an edge cutset S of H" such that $H" - S$ has two components C_1 and C_2 each of which has a face of H".
Proof. Let S be a minimal cyclic edge cutset of H'. Since S is a minimal cyclic edge cutset of H", $H" - S$ has a bipartition of the plane with two components C_1 and C_2 such that C_1 is on one side (region) and C_2 is on the other side (region), or C_1 is on outside (region) and C_2 is on inside (region), and S consists of only edges from vertices of C_1 to vertices of C_2, but not any edge with both ends in C_1 or C_2. Since C_1 has a cycle D and C_1 is the induced subgraph of H" on $V(C_1)$, the induced subgraph of H" on D and the vertices inside D remains in C_1, so C_1 has a face of H". By the same reason, C_2 also has a face of H". Notice that, in the case that C_1 is on outside (region), C_1 has the outer face of H". This completes the proof of Theorem 1. □

Then we give a simple algorithm to determine the cyclic edge connectivity of a planar graph.

Simple algorithm:
1. Use the algorithm in [5] to find a planar embedding G' of the input graph G and the set F of all faces of G';
2. IF, for any two faces C_1 and C_2 in F, $V(C_1) \cap V(C_2) \neq \emptyset$, THEN $c\lambda$(G) $= \infty$, and the algorithm exits;
3. $s \leftarrow |E(G)|$;
4. FOR any two faces C_1 and C_2 in F DO
   BEGIN

5. IF $V(C_1) \cap V(C_2) = \emptyset$ THEN
6. Construct a new graph H such that $V(H) = V(G) \cup \{x, y\}$, where $x, y \notin V(G)$, and $E(H)$ contains all edges in $E(G)$, and for each $u \in V(C_1)$, add $d(u) - 2$ multiple edges between x and u, and for each $v \in V(C_2)$, add $d(v) - 2$ multiple edges between y and v;
7. Use the algorithm in [10] to find the minimum edge cutset S_{xy} to separate x and y;
8. $s \leftarrow min\{s, |S_{xy}|\}$;
   END;
9. Then $c\lambda(G) = s$ and the algorithm exits.

Lus' algorithm [9] is similar to the above simple algorithm, but at step 1, they try to find all minimal cycles, and hence is more complicated than the above algorithm.

However, the time complexity of the simple algorithm is not $O(n^2)$. By the Euler's Formula, $\phi = m - n + 2$, for a planar graph, m can be at most $3n - 6$, so ϕ is $O(n)$. The FOR loop at step 4 is for the combination of any two faces in F, so it is excecuted $O(n^2)$ times, and the algorithm at step 7 takes $O(mn)$ time. So the whole algorithm takes $O(n^3m)$ time. For planar graphs, $m \leq 3n - 6$ and m is $O(n)$. So the time complexity of the simple algorithm is $O(n^4)$.

3 The main algorithm

Now we introduce our main algorithm to determine the cyclic edge connectivity of a simple planar graph H. At steps 1 and 2, we deal with the cases that $c\lambda(H) = \infty$ and $c\lambda(H) = 1$. After step 3, by Theorem 2 in next section, we claim that when $\lambda(H) \geq 2$, the length c of a shortest separating cycle in the dual graph G of the planar embedding H" of H is the cyclic edge connectivity of H" and hence is $c\lambda(H)$. This is the main idea of our algorithm.

We use pseudo PASCAL to describe the algorithm. For the meaning of data structure in the algorithm, the reader is referred to the theorems of correctness proof of the algorithms in next section.

Main algorithm
Input: A simple planar graph H.
Output: The cyclic edge connectivity $c\lambda(H)$ of H.
BEGIN
1. Use the algorithm in [8] to determine whether the cyclic edge connectivity of H is infinite. If it is, $c\lambda(H) = \infty$ and the algorithm exits. Otherwise, delete all pendent trees of H to get reduced form H' of H;

2. Use the DFS algorithm in [1] to find all biconnected components of H', if one of the components is an edge, then $\lambda(H') = 1$, and then $c\lambda(H) = 1$ and the algorithm exits (Since H' does not have a pendent tree, any cut edge is a cyclic edge cutset of H', and hence it is a cyclic edge cutset of size 1 in H);

/* If the algorithm reaches here, H' is 2-edge-connected */

3. Use the algorithm in [5] to determine whether H' is planar. If it is, find a planar embedding H" of H' , and return all faces $F[1] \sim F[\phi(H")]$, each $F[f]$ is a circular linked list of vertices on the boundary of face f in clockwise order;

4. Use procedure BuildAdjacencyList to build the dual graph G of H" and to build the adjacency list $AL[v]$ for every vertex v of G such that $AL[v]$ is a bidirectional linked list, and all vertices adjacent to v are listed in $AL[v]$ in the same order as they appear when we go around v clockwise in the planar embedding of G, which is the dual graph of H". Notice that if G has multiple edges, for every edge uv, u appears once in $AL[v]$. That is, u may appear multiple times in $AL[v]$;

5. Use procedure Separating_2_Cycle to determine whether G has a separating 2-cycle, if it returns true, then $c\lambda(H) = 2$ and the algorithm exits; Otherwise, we go to step 6;

6. Use procedure DeleteMultipleEdges to delete all consecutive multiple edges incident with v in $AL[v]$ for every vertex v of G such that for consecutive multiple copies of each edge uv in $AL[v]$, we keep only one copy of uv. Notice that after step 5, G has no separating 2-cycle, so step 6 deletes all multiple edges and keeps only one copy;

/* If the algorithm reaches here, G is a simple graph and every separating cycle of G has length at least 3 */

7. $c \leftarrow n(G)$; /* Assign initial value to c */

8. Use procedure SeparatingCycle to find the length c of a shortest separating cycle in G, then $c\lambda(H) = c$ and the algorithm exits;

END.

```
PROCEDURE BuildAdjacencyList;
BEGIN
   /* Assign initial values */
1. FOR u←1 TO n(H") DO
2. FOR v←1 TO n(H") DO
   BEGIN
3. A[u,v].FirstVertex←0;
4. A[u,v].SecondVertex←0;
5. A[u,v].EdgeNumber←0;
   END;
6. EdgeNum←0;
```

```
7. FOR face←1 TO φ(H") DO
   BEGIN
8. u←the first vertex of F[face];
9. head←u;
10. v←0; /* 0 is not any vertex */
11. WHILE v≠head DO
   BEGIN
12. v←the next vertex of F[face];
13. IF A[u,v].FirstVertex=0 THEN
   BEGIN
14. A[u,v].FirstVertex←face;
15. EdgeNum←EdgeNum+1;
16. A[u,v].EdgeNumber←EdgeNum;
17. A[v,u].FirstVertex←A[u,v].FirstVertex;
18. A[v,u].EdgeNumber←A[u,v].EdgeNumber;
   END
19. ELSE
   BEGIN
20. A[u,v].SecondVertex←face;
21. A[v,u].SecondVertex←A[u,v].SecondVertex;
   END;
22. u←v;
   END; /* WHILE */
   END; /* FOR */
23. FOR face←1 TO φ(H") DO
24. AL[face] is set to be an empty list;
25. FOR face←1 TO φ(H") DO
   BEGIN
26. u←first vertex of F[face];
27. head←u;
28. v←0;
29. WHILE v≠head DO
   BEGIN
30. v←the next vertex of F[face];
31. Set up a new record CV;
32. IF u=head THEN
33. CV.MARK←1 /* It is the first element of AL[face] */
34. ELSE
35. CV.MARK←0; /* It is one of the other elements of AL[face] */
36. IF A[u,v].FirstVertex=face THEN
   BEGIN
37. CV.VERTEX←A[u,v].SecondVertex;
38. CV.EDGENUM←A[u,v].EdgeNumber;
```

```
    END
39. ELSE IF A[u,v].SecondVertex=face THEN
    BEGIN
40. CV.VERTEX←A[u,v].FirstVertex;
42. CV.EDGENUM←A[u,v].EdgeNumber;
    END
42. ELSE ERROR;
43. IF face is not the outer face of H" THEN
44.  Add CV to the tail of adjacency list AL[face] to make AL[face] a
bidirectional linked list
45. ELSE Add CV to the head of adjacency list AL[face] to make AL[face]
a bidirectional linked list;
46. u←v;
    END; /* WHILE */
47. IF face is the outer face of H" THEN
48.  Let head pointer AL[face] point to the last element of the adjacency
list, of which the MARK field is 1;
    END; /* FOR*/
END; /* PROCEDURE */

PROCEDURE Separating_2_Cycle;
BEGIN
1. u←1;
2. foundcycle←false;
3. WHILE u ≤ n(G) AND NOT foundcycle DO
    BEGIN
4. FOR each vertex w in V(G) DO Visited[w]←false;
5. head←MARK field of the first vertex in AL[u];
6. FVERTEX←VERTEX field of the first vertex in AL[u];
7. w.MARK←MARK field of the next vertex in AL[u];
8. w.VERTEX←VERTEX field of the above vertex in AL[u];
9. Visited[FVERTEX]←true;
10. 2_Cycle←false;
11.  WHILE w.MARK≠head AND NOT foundcycle AND NOT 2_Cycle
DO
12. IF Visited[w.VERTEX] THEN
    BEGIN
13. z←VERTEX field of the vertex prior to w in AL[u];
14. IF z≠w.VERTEX THEN
    BEGIN
15. /* Let w denote the current element w in AL[u]; */
16. 2_Cycle←true;
17. x.MARK←MARK field of the vertex next to w in AL[u];
```

```
18. x.VERTEX←VERTEX field of the above vertex in AL[u];
19. WHILE x.MARK≠head AND NOT foundcycle DO
20. IF x.VERTEX≠w.VERTEX THEN
21. foundcycle←true
22. ELSE
    BEGIN
23. x.MARK←MARK field of the next vertex in AL[u];
24. x.VERTEX←VERTEX field of the above vertex in AL[u];
    END; /* WHILE */
    END /* IF z≠w.VERTEX */
25. ELSE /* z=w.VERTEX */
    BEGIN
26. w.MARK←MARK field of the next vertex of w in AL[u];
27. w.VERTEX←VERTEX field of the above vertex in AL[u];
    END;
    END /* IF Visited[w.VERTEX] */
28. ELSE /* NOT Visited[w.VERTEX] */
    BEGIN
29. Visited[w.VERTEX]←true;
30. w.MARK←MARK field of the next vertex in AL[u];
31. w.VERTEX←VERTEX field of the above vertex in AL[u];
    END; /* WHILE */
32. IF 2_Cycle AND NOT foundcycle THEN
    BEGIN
33. Redgenum←EDGENUM field of w in AL[u];
34. x.VERTEX←VERTEX field of the vertex prior to w in AL[u];
35. WHILE x.VERTEX≠w.VERTEX DO
36. x.VERTEX←VERTEX field of the prior vertex in AL[u];
37. Ledgenum←EDGENUM field of x in AL[u];
38. x.MARK←MARK field of the vertex x in AL[u];
39. WHILE x.MARK≠head AND NOT foundcycle DO
40. IF x.VERTEX≠w.VERTEX THEN
41. foundcycle←true
42. ELSE
    BEGIN
43. x.MARK←MARK field of the prior vertex in AL[u];
44. x.VERTEX←VERTEX field of the above vertex in AL[u];
    END; /* WHILE */
45. IF x.MARK=head AND x.VERTEX≠w.VERTEX THEN
46. foundcycle←true;
47. IF NOT foundcycle THEN
    BEGIN
48. v←w.VERTEX;
```

```
49. x.EDGENUM←EDGENUM field of the first vertex in AL[v];
50. x.VERTEX←VERTEX field of the above vertex in AL[v];
51. WHILE x.EDGENUM≠Redgenum DO
    BEGIN
52. x.EDGENUM←EDGENUM field of the prior vertex in AL[v];
53. x.VERTEX←VERTEX field of the above vertex in AL[v];
    END;
54. IF x.VERTEX=u THEN
    BEGIN
55. x.VERTEX←VERTEX field of the prior vertex in AL[v];
56. x.EDGENUM←EDGENUM field of the above vertex in AL[v];
57. WHILE x.EDGENUM≠Ledgenum AND NOT foundcycle DO
58. IF x.VERTEX≠u THEN
59. foundcycle←true
60. ELSE
    BEGIN
61. x.VERTEX←VERTEX field of the prior vertex in AL[v];
62. x.EDGENUM←EDGENUM field of the above vertex in AL[v];
    END; /* WHILE */
    END /* IF x.VERTEX=u */
63. ELSE ERROR;
    END; /* IF NOT foundcycle */
    END; /* IF 2_cycle AND NOT foundcycle */
64. u←u+1;
    END; /* WHILE */
65. IF foundcycle THEN RETURN(true)
66. ELSE RETURN(false);
END; /* PROCEDURE */

PROCEDURE DeleteMultipleEdges;
BEGIN
1. FOR each vertex w in V(G) DO
    BEGIN
2. head←MARK field of the first vertex in AL[w];
3. prior←VERTEX field of the first vertex in AL[w];
4. v.MARK←MARK field of the next vertex in AL[w];
5. v.VERTEX←VERTEX field of the above vertex in AL[w];
6. WHILE v.MARK≠head DO
7. IF v.VERTEX=prior THEN
    BEGIN
8. Delete the vertex v from AL[w];
9. v.MARK←MARK field of the next vertex in AL[w];
10. v.VERTEX←VERTEX field of the above vertex in AL[w];
```

```
    END
11. ELSE
    BEGIN
12. prior←v.VERTEX;
13. v.MARK←MARK field of the next vertex in AL[w];
14. v.VERTEX←VERTEX field of the above vertex in AL[w];
    END; /* WHILE */
    /* Now v.MARK=head */
15. mark←MARK field of the vertex prior to v in AL[w];
16. IF mark≠head AND v.VERTEX=prior THEN
17. delete the vertex prior to v in AL[w];
    END; /* FOR */
END; /* PROCEDURE */

PROCEDURE SeparatingCycle;
BEGIN
1. FOR each vertex w in V(G) DO
    /* For each vertex w in G, use the Breadth First Search strategy to find
a shortest separating cycle containing w, then find a shortest separating
cycle C in G among these cycles, and return c = |V(C)| */
    BEGIN
    /* Assign the initial values to variables */
2. FOR each vertex v in V(G) DO
    BEGIN
3. Ancestor[v]←0;
4. Level[v]←0;
5. Count[v]←0;
6. Father[v]←0;
7. Lbranch[v]←false;
8. Rbranch[v]←false;
9. Visited[v]←false;
10. Newbranch[v]←false;
11. AL[v] points to the first vertex of the adjacency list of v of which the
MARK field is 1 ;
    END;
12. Set Queue empty;
    /* Process w first, and we shall build the BFS tree with root w at the
bottom and the tree above*/
13. counts←1;
14. Visited[w]←true;
15. Count[w]←counts;
16. Level[w]←0;
17. i←0; /* i marks the ith child (from left to right) of the root w */
```

18. CL←1; /* CL is the current level to process in the BFS tree */
19. Put Lhead at the tail of Queue;
    /* Lhead is a symbol to mark the head of one level of the BFS tree */
20. FOR each vertex v in AL[w] DO
    BEGIN /* Put the vertices adjacent to w into Queue */
21. i←i+1;
22. Ancestor[v]←i;
    /* We set the ancestor of the ith child v of w and all v's descendants to be i */
23. Level[v]←CL;
24. Father[v]←w;
25. counts←counts+1;
26. Count[v]←counts;
27. Visited[v]←true;
28. Put v to the tail of Queue;
    END;
29. Put Ltail to the tail of Queue;
    /* Ltail is a symbol to mark the tail of one level of the BFS tree */
30. foundcycle←false;
31. Newbranch[w]←true;
32. exits←false;
33. WHILE c $\geq$ 2CL+2 AND NOT foundcycle AND NOT exits AND i $\geq$ 2 DO
    BEGIN
    /* Process the whole BFS tree rooted at w and the initial value of c is n(G) at step 7 in the main algorithm*/
34. Take Lhead out of Queue;
35. Put it to the tail of Queue;
36. IF the first element in Queue is Ltail THEN exits←true
37. ELSE
    BEGIN
38. head←the first vertex of Queue;
    /* It is not taken out of Queue */
39. $c' \leftarrow c$;
    END;
40. WHILE the first element of Queue is not Ltail AND NOT ($c' \leq 2CL+1$) DO
    BEGIN /* Process one level of the BFS tree */
41. Take one vertex v out from the head of Queue;
42. Search AL[v] until we find Father[v] in AL[v];
43. u←the next vertex of Father[v] in AL[v];
44. WHILE u is not Father[v] DO
    BEGIN /* Process each vertex u in AL[v] */

45. IF NOT Visited[u] THEN
    BEGIN
46. Ancestor[u]←Ancestor[v];
47. Level[u]←Level[v]+1;
48. counts←counts+1;
49. Count[u]←counts;
50. Father[u]←v;
51. Visited[u]←true;
52. Newbranch[v]←true;
53. IF the vertex x prior to u in AL[v] is Father[v] OR (Level[x] = Level[v] AND Count[x] = Count[v]−1) THEN
54. Lbranch[u]←Lbranch[v];
55. IF the next vertex x of u in AL[v] is Father[v] OR (Level[x] = Level[Father[v]] AND Count[x] = Count[Father[v]]+1) THEN
56. Rbranch[u]←Rbranch[v];
57. Put u to the tail of Queue;
    END
58. ELSE /* Visited[u]=true */
    BEGIN
59. IF head=v AND the vertex prior to Ltail in Queue is u THEN
60. IF (the vertex x prior to u in AL[v] is Father[v] OR (x is Father[u] AND the vertex prior to x in AL[v] is Father[v])) AND the next vertex of v in AL[u] is Father[u] AND NOT Lbranch[v] AND NOT Rbranch[u] THEN
61. $c' \leftarrow c'$ /* we do nothing */
62. ELSE IF Count[u]=Count[v]+1 AND NOT Rbranch[v] AND NOT Lbranch[u] AND (the next vertex x of u in AL[v] is Father[v] OR (x is Father[u] AND the next vertex of x in AL[v] is Father[v])) AND the vertex prior to v in AL[u] is Father[u] THEN
63. $c' \leftarrow c'$ /* we do nothing */
64. ELSE
    BEGIN
65. foundcycle←true;
66. $c' \leftarrow \min\{c', 2CL + 1\}$;
67. IF Ancestor[u]≠Ancestor[v] THEN
68. $c \leftarrow \min\{c, c'\}$;
    END
69. ELSE IF Count[u]=Count[v]+1 AND Level[u]=Level[v] THEN
70. IF NOT Rbranch[v] AND NOT Lbranch[u] AND (the next vertex x of u in AL[v] is Father[v] OR (x is Father[u] AND the next vertex of x in AL[v] is Father[v])) AND the vertex prior to v in AL[u] is Father[u] THEN
71. $c' \leftarrow c'$ /* we do nothing */
72. ELSE
    BEGIN

73. foundcycle←true;
74. $c' \leftarrow \min\{c', 2CL+1\}$;
75. IF Ancestor[u]≠Ancestor[v] THEN
76. $c \leftarrow \min\{c, c'\}$;
    END
77. ELSE IF Count[u] > Count[v]+1 AND Level[u]=Level[v] THEN
    BEGIN
78. foundcycle←true;
79. $c' \leftarrow \min\{c', 2CL+1\}$;
80. IF Ancestor[u]≠Ancestor[v] THEN
81. $c \leftarrow \min\{c, c'\}$;
    END
82. ELSE IF Count[u]=Count[v]+1 AND Level[u]=Level[v]+1 THEN
    /* v is the last vertex in Level[v] and u is the first vertex in Level[v]+1 */
83. IF NOT Lbranch[u] AND NOT Rbranch[v] AND (the next vertex x of u in AL[v] is Father[v] OR (x is Father[u] AND the next vertex of x in AL[v] is Father[v])) AND the vertex prior to v in AL[u] is Father[u] THEN
84. $c' \leftarrow c'$ /* we do nothing */
85. ELSE IF Count[u]=counts AND Count[Father[u]]=Count[v]−1 AND NOT Rbranch[u] AND NOT Lbranch[v] AND (the vertex x prior to u in AL[v] is Father[v] OR (x is Father[u] AND the vertex prior to x in AL[v] is Father[v])) AND the next vertex of v in AL[u] is Father[u] THEN
86. $c' \leftarrow c'$ /* we do nothing */
87. ELSE
    BEGIN
88. foundcycle←true;
89. $c' \leftarrow \min\{c', 2CL+2\}$;
90. IF Ancestor[u]≠Ancestor[v] THEN
91. $c \leftarrow \min\{c, c'\}$;
    END
92. ELSE IF Count[u]=counts AND Count[Father[u]]=Count[v]−1 AND Level[u]=Level[v]+1 THEN
93. IF NOT Rbranch[u] AND NOT Lbranch[v] AND (the vertex x prior to u in AL[v] is Father[v] OR (x is Father[u] AND the vertex prior to x in AL[v] is Father[v])) AND the next vertex of v in AL[u] is Father[u] THEN
94. $c' \leftarrow c'$ /* we do nothing */
95. ELSE
    BEGIN
96. foundcycle←true;
97. $c' \leftarrow \min\{c', 2CL+2\}$;
98. IF Ancestor[u]≠Ancestor[v] THEN
99. $c \leftarrow \min\{c, c'\}$;

```
   END
100. ELSE IF Level[u]=Level[v]+1 THEN
   BEGIN
101. foundcycle←true;
102. c' ← min{c', 2CL + 2};
103. IF Ancestor[u]≠Ancestor[v] THEN
104. c ← min{c, c'};
   END
105. ELSE c' ← c' /* we do nothing because Count[u] < Count[v] and the
edge uv has been processed when we process AL[u] */
   END; /* ELSE Visited[u] = true */
106. IF c' ≤ 2CL + 1 THEN u←Father[v]
107. ELSE u←the next vertex in AL[v];
   END; /* WHILE at step 44*/
108. IF NOT Newbranch[v] THEN
   BEGIN
109. x←the first element of Queue; /* x is not taken out of Queue */
110. IF x is not Ltail THEN Lbranch[x]←true;
111. y←the element at the tail of Queue; /* y is not taken out of Queue
*/
112. IF y is not Lhead THEN Rbranch[y]←true;
   /* y is the last vertex of Level[v]+1 at the time */
   END;
   END; /* WHILE at step 40*/
113. IF NOT foundcycle AND c > 2CL + 2 THEN
   BEGIN
114. CL←CL+1;
115. Take out Ltail from the head of Queue;
116. Put Ltail to the tail of Queue;
   END;
   END; /* WHILE at step 33*/
   END; /* FOR */
END; /* PROCEDURE */
```

4 Correctness and time complexity

In this section, we shall show correctness of the algorithms. Theorem 2 is the basis of the main algorithm, while Theorems 3, 4, 5 and 6 show correctness of procedures BuildAdjacencyList, Separating_2_Cycle, DeleteMultipleEdges and SeparatingCycle. Theorem 7 shows that the time com-

plexity of the main algorithm is $O(n^2)$.

Theorem 2: In the main algorithm, after step 3, the length c of a shortest separating cycle C in the dual graph G of H" is the cyclic edge connectivity $c\lambda(H")$ of H".

Proof. First, we prove that, for any separating cycle $C = v_1v_2\cdots v_cv_1$ in G, there is a cyclic edge cutset S corresponding to C in H" with $c = |V(C)| = |S|$.

Since C is a separating cycle, $G - V(C)$ has two components D_1 and D_2 with D_1 inside C and D_2 outside C. Let u be a vertex of D_1. As $u \neq v_i$ $(i = 1, 2, \cdots, c)$, the face f of H" corresponding to u has its boundary inside C. Suppose not. Then f has one boundary edge e_1 going out C by crossing the edge v_jv_{j+1} of C. But f surrounds u, so f has another edge e_2 going back to inside of C. Hence v_j or v_{j+1} is surrounded by f, by the definition of dual graph, f surrounds only one vertex of G, hence $u = v_j$ or $u = v_{j+1}$, which is a contradiction. So the face f of H" corresponding to u has its boundary inside C. By the same reason, D_2 has a vertex v and the face f' of H" corresponding to v has its boundary outside C. Now $S = \{e_i | e_i \in E(H")$ and e_i crosses v_iv_{i+1}, $i = 1, 2, \cdots, c-1$, and e_c crosses $v_cv_1\}$ is an edge cutset of H" such that $H" - S$ has two components each of which has a cycle (face). So S is a cyclic edge cutset of H" corresponding to C with $c = |V(C)| = |S|$.

Then we prove that , for any minimal cyclic edge cutset S of H", there is a separating cycle C of G corresponding to S with $c = |V(C)| = |S|$. Let $S = \{e_1, e_2, \cdots, e_c\}$ be a minimal cyclic edge cutset of H". By Theorem 1, we know that $H" - S$ has two components C_1 and C_2 each of which has a face. Without loss of generality, assume that in the planar embedding H", $e_1, e_2, \cdots, e_c$ appear in turn. Since, after step 3 in the main algorithm, H" is a 2-edge connected simple graph, each e_i will not appear on the boundary of only one face. So each e_i lies on the common boundary of two faces f_{i-1} and f_i. Notice that, between e_c and e_1, there is the outer face of H" if $H" - S$ has two components on two sides or there is an inner face of H" if $H" - S$ has two components on inside and outside respectively. Let v_i be the vertex of G corresponding to f_i, $i = 1, 2, \cdots, c-1$ and v_c corresponds to the face between e_c and e_1. Now, if $G[\{v_1, v_2, \cdots, v_c\}]$ is a cycle C, then C is a separating cycle in G since $H" - S$ has two components each of which has a face by Theorem 1, so $G - V(C)$ has two components inside C and outside C respectively, each of which has at least one vertex, and each e_i in S crosses $v_{i-1}v_i$ of C, $i = 1, 2, \cdots, c$, where v_0=v_c corresponds to the face between e_c and e_1 in H". So $c = |V(C)| = |S|$. If $G[\{v_1, v_2, \cdots, v_c\}]$ is not a cycle, by the construction of $G[\{v_1, v_2, \cdots, v_c\}]$, it is a closed trail, then $G[\{v_1, v_2, \cdots, v_c\}]$ has a subgraph to be a separating cycle C' of G. By the argument before, C' corresponds to a cyclic edge cutset $S' \subset S$ in

H", which contradicts the minimality of S. So, for any minimal cyclic edge cutset S of H", there is a separating cycle C of G corresponding to S with $c = |V(C)| = |S|$.

By the above argument, the length c of a shortest separating cycle C of G corresponds to a minimum cyclic edge cutset S of H" with $c = |V(C)| = |S|$. Hence Theorem 2 is proved. □

Theorem 3: Procedure BuildAdjacencyList builds the dual graph G of H" and builds the adjacency list $AL[v]$ for every vertex v of G such that $AL[v]$ is a bidirectional linked list, and all vertices adjacent to v are listed in $AL[v]$ in the same order as they appear when we go around v clockwise in the planar embedding of G, which is the dual graph of H".

Proof. Procedure BuildAdjacencyList input all faces f numbered from 1 to $\phi(H")$, where $\phi(H")$ is the number of faces in H". For each face f, $F[f]$ is a circular linked list of the vertices on the boundary of face f in clockwise order.

$A[1 \cdot\cdot n(H"), 1 \cdot\cdot n(H")]$ is a 2 dimensional array of records. For each edge uv of H", $A[u, v]$ (and $A[v, u]$) records the edge of G crossing uv , where the fields FirstVertex and SecondVertex record the two faces of H" beside the edge uv (the two end vertices of the edge crossing uv in G). Different edges of G have different numbers in the field EdgeNumber, which can distinguish different edges, especially multiple edges.

In BuildAdjacencyList, steps 1−5 assign initial value to each array element $A[u, v]$. In steps 7−22, for each $face$, we search the boundary $F[face]$. For each edge uv on the boundary of the $face$ in H", if it is the first time to process uv, then the $face$ is the first end vertex (FirstVertex) of the edge crossing uv in G, otherwise the $face$ is the second end vertex (SecondVertex) of the edge crossing uv in G, and each edge in G is assigned a different number to EdgeNumber field. So we complete array A.

In steps 23−24, for each $face$ (vertex of G), we set up an empty initial adjacency list $AL[face]$.

In steps 25−46, for each $face$ of H" (vertex of G), we search the boundary $F[face]$ clockwise. For each edge uv on the boundary, we put the other end vertex than the $face$ into $AL[face]$ according to the edge recorded in $A[u, v]$ (or $A[v, u]$). So we list all vertices adjacent to (the end vertices of all edges incident with) the $face$ in G in $AL[face]$ clockwise. Notice that, if the $face$ is the outer face of H", since $F[face]$ is in clockwise order, but when we go around the vertex $face$ clockwise in G, the vertices in $F[face]$ will appear in reverse order. So, in steps 43−45, if the $face$ is the outer face of H", we build up $AL[face]$ in reverse order, otherwise in clockwise order. Steps 47−48 adjust the head pointer $AL[face]$ to the proper first element of the adjacency list of the vertex $face$ in G, of which the MARK

field is 1. Notice that, for each vertex v in G, the MARK field of the first element in $AL[v]$ is 1 and the MARK fields of the other elements in $AL[v]$ are 0.

So we complete the proof of Theorem 3. □

Theorem 4: Procedure Separating_2_Cycle succeeds to find a separating 2-cycle and returns true if G has such a cycle; otherwise it returns false.

Proof. In the procedure, we simulate the planar embedding of every vertex u and its neighbours in G to find a separating 2-cycle, where u takes the numbers of all vertices in G from 1 to $n(G)$, foundcycle is true if we find a separating 2-cycle in G; otherwise foundcycle is false. The 2_cycle is true if we find a 2-cycle which surrounds at least one vertex but we have not found vertices outside it yet; 2_cycle is false if we do not find such a cycle. In the WHILE loop at step 3, we search every vertex u in G until we find a separating 2-cycle or we have searched all vertices in G but we have not found any separating 2-cycle. In the WHILE loop at step 11, we search the adjacency list $AL[u]$ until we find a separating 2-cycle or we find a 2-cycle which surrounds a vertex or we have searched up the whole $AL[u]$ and find no such a 2-cycle. At step 12, if we find that w is already visited, it means that we find that the end of current edge uw is the end of another edge uw formerly processed. Then we find a 2-cycle. If the prior vertex z of the current w is not w, then the 2-cycle surrounds at least one vertex z. Then we search $AL[u]$ to see if there is a vertex $x(\neq w)$ adjacent to u after the current edge uw. If there is such an x, then we have found a separating 2-cycle with x ouside it. If, after the current edge uw, we have not found any edge ux with $x \neq w$, we come to step 32. We mark the edge uw right edge with u at the bottom and w at the top. Then we search $AL[u]$ reversely to find the edge uw we first encounter, where w is previously visited, and mark this edge left edge. Notice that, between the left edge and the right edge, there is at least a vertex $x(\neq w)$ adjacent to u which is surrounded by the 2-cycle. Then in the WHILE loop at step 39 we search for a vertex $x(\neq w)$ adjacent to u before (on the left of) the left edge in $AL[u]$. If there is such a vertex x, then we find a separating 2-cycle. If we have not found a separating 2-cycle, let v be the w at step 48. Then we search $AL[v]$ anticlockwise. First, we find the right edge in $AL[v]$ in the WHILE loop of steps 51−53. Then we search $AL[v]$ anticlockwise until we find a vertex $x(\neq u)$ adjacent to v or we find the left edge. If we find a vertex $x(\neq u)$, then x is outside the 2-cycle that we found before, so the 2-cycle is a separating 2-cycle. If we have not found such an x until we find the left edge, then the search of $AL[u]$ does not find any separating 2-cycle, and u takes another vertex at step 64 and we go back step 3. At steps 65 and 66, if, in the whole procedure, we have found a separating 2-cycle, then

it returns true; otherwise it returns false.

So we complete the proof of Theorem 4. □

Theorem 5: Procedure DeleteMultipleEdges deletes all consecutive multiple edges incident with v in $AL[v]$ for every vertex v of G such that for consecutive multiple copies of each edge uv in $AL[v]$, we keep only one copy of uv.

Proof. In the FOR loop at step 1, for each vertex w in G, we search $AL[w]$ clockwise. At step 7, for each current vertex v in $AL[w]$, if v is equal to the prior vertex in $AL[w]$, then we find a consecutive multiple edge, and we delete this vertex v from $AL[w]$. In steps 15−17, if the last vertex in $AL[w]$ is equal to the first vertex in $AL[w]$, then we find a consecutive multiple edge, and we delete the last vertex in $AL[w]$. For consecutive multiple copies of each edge wv, we keep only one copy of wv.

Notice that, after step 5 of the main algorithm, G has no separating 2-cycle, so procedure DeleteMultipleEdges deletes all multiple edges incident with w for each vertex w in G.

Notice that for multiple edges uv, the only one copy of uv remained in $AL[u]$ may not have the same EDGENUM (edge number) as the only copy of uv remained in $AL[v]$. But it does not matter. In the following steps of the main algorithm, we are only concerned about whether there is such an edge uv but not which copy of the multiple edges. □

Theorem 6: Procedure SeparatingCycle succeeds to find a minimum separating cycle C in G and returns its length $c = |V(C)|$, and hence $c\lambda(H) = c$ at step 8 in the main algorithm.

Proof. In the procedure, for each vertex w, we build a BFS tree T rooted at w (at the bottom) to simulate the planar embedding of graph G and to find a shortest separating cycle containing w, then we find a shortest separating cycle C of G among these cycles, and return $c = |V(C)|$.

We have several arrays, for a nonnegative integer i, Ancestor[v]=i means that the vertex v is a decendant of the ith (i $\geq$ 1) child (from left to right in clockwise orientation) of the root w in the BFS tree T; Level[v]=i means that v is at the ith level of T; Count[v]=i (i $\geq$ 1) means that v is the ith visited vertex; Father[v]=i means that vertex i is the father of vertex v in T; Lbranch[v]=true means that, in the BFS tree T (root w is at the bottom and T is above), there is a branch on the left of v, but it does not grow up a new branch at higher level; Rbranch[v]=true means that, in the BFS tree T, there is a branch on the right of v, but it does not grow up a new branch at higher level; Visited[v]=true means that the vertex v has been visited; Newbranch[v]=true means that, in the BFS tree T, when we visit vertex v, it grows up a new branch. We have a Queue to help the breadth

first search. The value of variable CL is the number of current level to process in the BFS tree T; foundcycle=true means that we have found a separating cycle in T; exits=true means that the current level to process in T has no vertex, that is, we have processed all vertices in G, so we shall exit the loop.

In the FOR loop at step 1, for each vertex w, we build the BFS tree T rooted at w to find a shortest separating cycle containing w, then find a shortest separating cycle C among these cycles for all w and return $c = |V(C)|$. By Theorem 2, c is $c\lambda(H")$ and hence is $c\lambda(H)$ since $c\lambda(H) = c\lambda(H') = c\lambda(H")$.

For each $w \in V(G)$, we do the following. In steps 2–12, we assign initial values to the arrays and variables. In steps 13–16, we visit the root w and w is at the 0th level of the BFS tree T. (Notice that the root w is at the bottom and the tree T is above). At step 17, the variable i marks the ith child of w, the ith child and all its descendants in T have Ancestor[v]=i . At step 18, CL=1, now we shall process the 1st level of T. We put the head mark Lhead of a level to the Queue. In steps 20–28, we visit every child of w in T and put them in the Queue. Then we put the tail mark Ltail of a level to the Queue. Notice that c is the length of a shortest separating cycle that we have currently found. Its initial value is $n(G)$.

In the WHILE loop at step 33, we build the whole BFS tree T rooted at w until we find a separating cycle (foundcycle = true), or we have visited all vertices of G (exits = true), or the length c of the shortest separating cycle that we have currently found is less than or equal to 2CL+1. Notice that, when we process the current level CL of T, the shortest separating cycle C containing w that we can find has length $|V(C)| \geq 2CL+1$. So we exit the loop when $c \leq 2CL+1$. Also notice that now i is the number of the children of w, if $i \leq 1$, then we cannot find a separating cycle containing w, so the WHILE loop at step 33 will not be executed. At step 36, if the new level of T is empty, then exits = true; otherwise the variable head records the first vertex of the current level of T, and $c' = c$ (initial value).

The WHILE loop at step 40 process one level (the current level) of T until we find the tail mark Ltail of a level in the Queue or we have found a separating cycle of length less than or equal to $c' \leq 2CL+1$. As we mentioned before, when we process the current level CL of T, the shortest separating cycle C containing w that we can find has length $|V(C)| \geq 2CL+1$, so we exit the loop if $c' \leq 2CL+1$. In steps 42, 43 and in the whole procedure, we try to simulate the planar embedding of G when we build the BFS tree T and notice that AL[v] is in clockwise order.

In the WHILE loop at step 44, we visit all vertices u adjacent to the current vertex v at level CL by searching forward AL[v] clockwise starting from the next vertex of Father[v] in AL[v] and ending at Father[v]. At steps 106, 107, if we have found a separating cycle of length less than or equal

to $c' \leq 2CL + 1$, then we exit the loop at step 44 immediately; otherwise, we search forward AL[v].

In steps 45–57, when we search AL[v], if we find a vertex u adjacent to v which is not visited before, then we build a new branch vu on the tree T and put u in the Queue. In steps 53–56, if u is the left most new branch of v, then Lbranch[u] remembers the branch on the left of v; if u is the right most new branch of v, then Rbranch[u] remembers the branch on the right of v. In other cases that u does not remember the left branch or the right branch of v, we have already found a separating cycle. If we find a vertex u adjacent to v which has been visited before at step 58, then we have the following cases to study.

At step 59, if v is the first vertex and u is the last vertex at the current level CL, then we find a cycle C; if there is no vertex outside C at step 60, then C is not a separating cycle and we do nothing; if there is no vertex inside C at step 62, then C is not a separating cycle and we do nothing; otherwise C has vertices both inside and outside it at step 64, then C is a separating cycle. Notice that at steps 64, 72, 77, 87, 95 and 100 when we find a separating cycle C, C may not contain the root w, then c' is assigned $\min\{c', 2CL+1\}$ or $\min\{c', 2CL+2\}$, where 2CL+1 or 2CL+2 is the length of the currently found separating cycle if it contains w. But the length of C is always less than or equal to c'. If Ancestor[u] $\neq$ Ancestor[v] at steps 67, 75, 80, 90, 98 and 103, then u and v are descendants of different children of w, so C contains w. Then c is modified to be c' if $c' < c$. Notice that if C does not contain w but C is a shortest separating cycle in G, then it must contain another vertex w', when we build the BFS tree rooted at w', we shall find C at last and c will be the length of the shortest separating cycle C in G, and notice that a separating cycle C containing w may not be found when we build the BFS tree rooted at w, but it will be found at last when we build a BFS tree rooted at another vertex w' which is also contained in C, or we have found a separating cycle shorter than C. Also notice that c' has initial value c at step 39, then in the loop at step 40, whenever we find a separating cycle of length less than or equal to c' $\leq 2CL + 1$, we shall stop processing the current level of T, and because foundcycle = true or $c \leq 2CL + 1$, the loop at step 33 will also exit.

Now we continue our case study. At step 69, v is adjacent to the next vertex u at the same level CL and we have a cycle C. If above cases does not happen, then there must be a vertex outside C. If there is no vertex inside C at step 70, then we do nothing; otherwise C is a separating cycle at step 72 and we do the same process as that at step 64. At step 77, C has vertices both inside and outside it, then C is a separating cycle, and we do the same process as that at step 64. At step 82, v is the last vertex at Level[v] and u is the first vertex at Level[v]+1. Then we have a cycle C. If there is no vertex outside C at step 83, then C is not a separating cycle

and we do nothing; If there is no vertex inside C at step 85, then C is not a separating cycle either; otherwise C is a separating cycle at step 87 and we do the similar process to that at step 64. At step 92, u is Father[u]'s right most child and v is Father[u]'s next vertex at the same level and we have a cycle C. If the last case does not happen, then there must be a vertex outside C. If there is no vertex inside C at step 93, then C is not a separating cycle and we do nothing, otherwise, at step 95, C is a separating cycle and we do the similar process to that at step 64. At step 100, C has vertices both inside and outside it, so C is a separating cycle and we do the similar process to that at step 64. In the other cases at step 105, we do nothing because now Count[u] < Count[v] and the edge uv has been processed when we process AL[u].

At step 108, after processing AL[v], if v does not grow up a new branch in the BFS tree T, then we set Lbranch[x] = true for the x on the right of v and we set Rbranch[y] = true for the y at Level[v]+1 on the left of v. (Notice that the vertex at Level[v] on the left of v has been processed). So v is a branch on the left of x and on the right of y. In the cases that v is not a branch on the left of x or on the right of y, we have already found a separating cycle.

At step 113, after we search the current level CL of T, if we have not found a separating cycle in searching the level of T and the length c of the shortest separating cycle that we already found before is greater than 2CL+2 (i. e. $c \geq 2(CL+1)+1$), then the value of CL increases by 1, and we shall search the higher level of T.

Hence we complete the proof of Theorem 6. □

Theorem 7: The time complexity of the main algorithm is $O(n^2)$, where $n = |V(H)|$.

Proof. Now we analyze the time complexity of the main algorithm.

According to [8], step 1 needs $O(|V||E|)$ time. Notice that H is a planar graph, $|E| \leq 3|V| - 6$, so $O(|V||E|)=O(|V|^2)= O(n^2)$.

According to [1], the DFS algorithm to find all biconnected components of H' needs $O(|E(H')|)$ time. But $|E(H')| \leq |E(H)|$ and by the above argument, $O(|E(H)|) = O(|V(H)|)$. So step 2 needs $O(|V(H)|) = O(n)$ time.

According to [5], it takes $O(|V(H')|)$ time to find the planar embedding $H"$ of H'. But $|V(H')| \leq |V(H)|$. So step 3 needs $O(n)$ time.

Now let $n(H") = |V(H")|$, $m(H") = |E(H")|$, ϕ = the number of faces of $H"$. We have $|V(G)| = n(G) = \phi$, $|E(G)| = m(G) = m(H")$ and $n(H") \leq n$.

In procedure BuildAdjacencyList, steps 1–5 take $O(n(H")^2) = O(n^2)$ time. The FOR loop at step 7 is executed ϕ times. The WHILE loop at

step 11 takes at most O(d($face$)) time, where d($face$) is the degree of the $face$, it is also the degree of the vertex $face$ in G. So the whole FOR loop at step 7 takes O($\sum$d($face$)) = O(2m(G)) = O(m(G)) = O(m(H")) time. But m(H") $\leq$ 3n(H")$-$6, so O(m(H")) = O(n(H")) = O(n). The FOR loop at step 23 takes O(ϕ) = O(n) time. The FOR loop at step 25 is executed ϕ = O(n(G)) times and the WHILE loop at step 29 takes O(d($face$)) time. By the same argument as above, the whole FOR loop at step 25 takes O(m(G)) = O(m(H")) = O(n) time. Steps 47, 48 take O(1) time. Hence the whole procedure at step 4 of the main algorithm takes O(n^2) time.

In procedure Separating_2_Cycle, the WHILE loop at step 3 is executed at most O(n(G)) = O(ϕ) times. By the Euler's Formula, ϕ = m(H")$-$n(H")+2, but m(H") $\leq$ 3n(H")$-$6, so ϕ = O(n(H")) = O(n). The FOR loop at step 4 takes O(n(G)) = O(n) time. Steps 5$-$10 take O(1) time. The WHILE loop at step 11 searches the adjacency list AL[u], the WHILE loop at step 19 completes the search, the other parts take O(1) time. So the whole loop at step 11 takes O(d(u)) time. The WHILE loop at step 35 takes at most O(d(u)) time. The WHILE loop at steps 39$-$44 also searches AL[u], it takes at most O(d(u)) time. The WHILE loop at step 51 searches AL[v], it takes at most O(d(v)) time and the WHILE loop at step 57 also searches AL[v], it takes at most O(d(v)) time. The other parts take O(1) time. So the body of the WHILE loop at step 3 takes at most O(max$\{n, d(u) + d(v)\}$) time, where v is a vertex adjacent to u by multiple edges. In the WHILE loop at step 3, u takes all vertices of G until we find a separating 2-cycle. Notice that AL[v] will not be searched more than two times in the whole procedure otherwise we shall find a separating 2-cycle and the loop will exit. So the loop at step 3 takes O(max$\{n(G)^2$, $2\sum_{v \in V(G)} d(v)\}$) = O(max$\{n(G)^2$, $4m(G)\}$) = O(max$\{n(G)^2$, $n(G)\}$) = O($n(G)^2$) = O(n^2) time since G is a planar graph and O($m(G)$) = O($n(G)$). Hence the whole procedure at step 5 in the main algorithm needs O(n^2) time.

In the procedure DeleteMultipleEdges, for each vertex w in V(G), we search AL[w]. So it takes O($\sum_{w \in V(G)} d(w)$) = O($m(G)$) = O(n) time at step 6 in the main algorithm.

Step 7 in the main algorithm takes O(1) time.

In the procedure SeparatingCycle, the FOR loop at step 1 is executed O($n(G)$) = O(n) times. The FOR loop at step 2 needs O($n(G)$) = O(n) time. The FOR loop at step 20 takes O(d(w)) time. The WHILE loop at step 33 builds the BFS tree T rooted at w, the WHILE loop at step 40 processes one level of T, and for each vertex v at the current level, the WHILE loop at step 44 searches AL[v]. So the whole loop at step 33 does a BFS search and needs O($m(G)$) = O(n) time. What we only need to

explain is that, at steps 59, 60 and 62, only when v is the first vertex and u is the last vertex at the current level, we need to search AL[u] besides AL[v]; at steps 69 and 70, only when Count[u] = Count[v]+1 and Level[u] = Level[v], we need to search AL[u] besides AL[v]; at steps 82, 83 and 85, only when Count[u] = Count[v]+1 and Level[u] = Level[v]+1, we need to search AL[u] besides AL[v]; at steps 92 and 93, only when Count[u] = counts, Count[Father[u]] = Count[v]−1, Level[u] = Level[v]+1, we need to search AL[u] besides AL[v]. These AL[u] will not be searched more than once besides that time at the loop of step 44 when v = u. But it may happen that when we search AL[v], we need also search AL[u] with u on the left of v and Level[u] = Level[v]+1 and also search AL[w] with w on the right of v and Level[w] = Level[v] and Count[w] = Count[v]+1. So the WHILE loop at step 44 needs at most O(d(v)+d(u)+d(w)) time for different u, v and w. For each vertex v in G, d(v) is used at most 3 times. Then the whole loop at step 33 needs at most $O(3\sum_{v\in V(G)} d(v)) = O(6m(G)) = O(m(G)) = O(n)$ time. To sum up, the whole procedure at step 8 in the main algorithm needs $O(m(G)n(G)) = O(n(G)^2) = O(n^2)$ time.

Hence the whole main algorithm needs $O(n^2)$ time. □

Remark: The space that the main algorithm needs is also $O(n^2)$ since we use the array A[$1\cdots n(H")$, $1\cdots n(H")$] in the procedure BuildAdjacencyList. The other data structures need only $O(n)$ space.

References

[1] A. V. Aho, J. E. Hopcroft and J. D. Ullman, The design and analysis of computer algorithms, Addison-Wesley Press, Reading, Massachusetts, (1976).

[2] J. A. Bondy and U. S. R. Murty, Graph theory with applications, MacMillan Press, London, (1976).

[3] Z. Dvorak, J. Kara, D. Kral and O. Pangrac, An algorithm for cyclic edge connectivity of cubic graphs, SWAT 2004, Lecture Notes in Computer Science 3111(2004), 236–247.

[4] D. A. Holton, Dingjun Lou and M. D. Plummer, On the 2-extendability of planar graphs, Discrete Math. 96(1991), 81–99.

[5] J. E. Hopcroft and R. E. Tarjan, Efficient planarity testing, J. ACM 21:4(1974), 549–569.

[6] Dingjun Lou and D. A. Holton, Lower bound of cyclic edge connectivity for n-extendability of regular graphs, Discrete Math. 112(1993), 139–150.

[7] Dingjun Lou and Wei Wang, An efficient algorithm for cyclic edge connectivity of regular graphs, Ars Combinatoria 77(2005), 311–318.

[8] Dingjun Lou and Wei Wang, Characterization of graphs with infinite cyclic edge connectivity, Discrete Math. 308(2008), 2094–2103.

[9] Yuntin Lu and Xiu Lu, An efficient algorithm for cyclic edge connectivity of planar graphs, Proc. 2009 Asia-Pacific Conference on Information Processing, 2009, 193–198.

[10] H. Nagamochi and T. Ibaraki, Computing edge connectivity in multigraphs and capacitated graphs, SIAM Journal on Discrete Mathematics 5(1992), 54–66.

[11] R. Nedela and M. Skoviera, Atoms of cyclic connectivity in cubic graphs, Math. Slovaca 45(1995), 481–499.

[12] B. Peroche, On several sorts of connectivity, Discrete Math. 46(1983), 267–277.

[13] M. D. Plummer, On the cyclic connectivity of planar graphs, in: Y. Alavi, D. R. Lick and A. T. White eds. Graph Theory and Applications, Springer-Verlag, Berlin (1972), 235–242.

[14] R. G. Tait, Remarks on the colouring of maps, Proc. Soc. Edinburg 10(1880), 501–503.