Yugoslav Journal of Operations Research
21 (2011), Number 1, 79-92
DOI: 10.2298/YJOR1101079S

CONVEX POLYHEDRA WITH TRIANGULAR FACES AND CONE TRIANGULATION

Milica STOJANOVIĆ
Faculty of Organizational Sciences, Belgrade, Serbia
milicas@fon.bg.ac.rs
Milica VUČKOVIĆ
Faculty of Organizational Sciences, Belgrade,Serbia
milica@fon.bg.ac.rs

Received: December 2010 / Accepted: May 2011

Abstract: Considering the problem of the minimal triangulation for a given polyhedra (dividing polyhedra into tetrahedra) it is known that the cone triangulation provides the number of tetrahedra which is the smallest, or the closest to it. It is also shown that when we want to know whether the cone triangulation is the minimal one, it is necessary to find the order of all vertices, as well as the order of "separating circles". Here, we will give algorithms for testing the necessary condition for the cone triangulation if it is the minimal one. The algorithm for forming the cone triangulation will also be given.

Keywords: Triangulation of polyhedra, minimal triangulation, graph algorithms, abstract data type of graph.

MSC: *Primary*: 52C17, 68R10; *Secondary*: 52B05, 05C85, 68P05, 68Q65

1. INTRODUCTION

The division of any polygon with *n*-3 diagonals into *n*-2 triangles without gaps and overleaps is known. Such a division is called triangulation.

The generalization of this process to higher dimensions is also called triangulation. It divides a polyhedron (polytope) into tetrahedra (simplices). The problem of triangulation in higher dimensions is much more complicated. It is impossible to triangulate some nonconvex polyhedra [9] in three-dimensional space, and it is also proved that triangulations of the same polyhedron may lead to different numbers of tetrahedra [7], [10]. Considering the smallest and the largest number of tetrahedra in triangulation (the minimal and the maximal triangulation, respectively), the authors obtained values, which linearly, resp. squarely depend on the number of vertices. Some characteristics of triangulation in three-dimensional space are given by Chin, Fung, Wang [5], Develin [6] and Stojanović [11, 12].

This paper shall consider convex polyhedra with every 4 vertices noncoplanar, and all faces triangular. Furthermore, all considered triangulations are face to face. The number of edges from the same vertex will be called the order or degree of the vertex.

Algorithms given in this paper test the necessary conditions for the cone triangulation, when it is the minimal one. An algorithm for forming the cone triangulation is also given. All these algorithms are based on graph presentation of the given polyhedron. A similar method for the presentation of polyhedra is given in [1, 3], for example. We have implemented some of these algorithms in C# and we are currently working on the implementation of others.

It is known that establishing the minimal triangulation of convex polyhedra is an NP-hard problem [2, 5]. On the other hand, the algorithms for cone triangulation given in this paper are polynomial. The algorithms for testing the necessary conditions are polynomial as well. They give us the possibility to recognize the polyhedron for which is difficult to find minimal triangulation. Knowing the results of these algorithms in some of cases we consider improvements of cone triangulation, with the aim to get triangulation with small number of tetrahedra, close to number in the minimal one.

Theoretical results are summarized in Section 2. There are also introduced graphs and their applications to this problem. Abstract data type (ADT) of graph is presented [4, 8], including some elementary properties, and two main data structures for representing graphs are given. In Section 3 we will provide graph algorithms for testing conditions for establishing the degree of vertices and describe algorithm checking conditions for "separating circles". Algorithms for finding triangular faces and for acting cone triangulation are given as well. These algorithms work on the **graphs** of polyhedron representations. Section 4 contains the conclusions of the paper.

2. PRELIMINARIES

2.1 Cone triangulation. One of the triangulations, which gives a small number of tetrahedra, is the cone triangulation [10] described as follows.

One of the vertices is the common apex, which builds a tetrahedron with each triangular face of the polyhedron, except these containing apex.

By Euler's theorem, a polyhedron with n vertices has $2n$-4 faces if all of them are triangular. Therefore, the number of tetrahedra in triangulation is $2n$-10 at most, since, for $n \geq 12$, each polyhedron has at least one vertex of order 6 or more. Sleator, Tarjan and Thurston in [10] considered some cases of "bad" polyhedra, which need a large number of tetrahedra for triangulation. It is proved, using hyperbolic geometry that the minimal number of triangulating tetrahedra is close to $2n$-10. That value is tight for certain series of polyhedra, which exists for a sufficiently large n. Computer investigation of the equivalent problem of rotatory distance confirms, for $12 \leq n \leq 18$, that there are polyhedra with the smallest necessary number of tetrahedra equal to $2n$-10. This was the reason why the authors conjectured that the same statement is true for any $n \geq 12$. To prove this hypothesis, it would be enough to check those cases where the cone triangulation of polyhedra gives the smallest number of tetrahedra, and to show how to improve that in other cases. With this aim, in [10] the authors gave a polyhedron example which has vertices of great order and for which there exists a triangulation better than the cone one. They also gave advice on how to improve the method in this case and some similar cases. However, the polyhedra with vertices of great order give less than $2n$-10 tetrahedra in the cone triangulation, therefore, vertices of small order are considered in [11, 12]. The obtained results are as follows:

Theorem 1. *Let V be one of the vertices of a polyhedron P whose order is maximal. If the polyhedron P has a vertex of order 3 not connected with V, or a sequence of at least 2 vertices of order 4 connected with a chain, each of them not connected with V, then the cone triangulation of P with apex V will not give the smallest number of tetrahedra.*

Remark: When V is connected with a vertex at the end of the chain, the cone triangulation is not the minimal one whenever the chain contains at least 3 vertices.

Apart from the order of vertices, it is also necessary to consider the order of *separating circle*. Let us define the following:

- *A circle of p vertices* of the polyhedron P is a p-sided closed polygon $A_1, A_2, \dots , A_p$ where A_i $(i = 1, \dots , p)$ are different vertices of P and A_iA_{i+1} $(i = 1, \dots , p-1)$, A_pA_1 are edges of P.
- Let c be a circle on the polyhedron P, moreover M and N two vertices of P different from A_i. If all paths on P with end points M and N pass through some of the vertices A_i, then we say that a *separating circle c* separates M and N. We also say that M and N are on the different sides of c. If the circle c does not separate the vertices M and N, they are on the same side of the circle.

Theorem 2. *If a polyhedron contains a circle of p vertices, which separates vertices A and B of order v(A) and v(B), where* $v(A) \geq v(B) > p$, *then the cone triangulation with apex A is not the minimal one.*

The consequence of the theorems is that the candidates for the minimal triangulation with $2n$-10 tetrahedra are polyhedra with all vertices of order 5 or 6, occasionally some of order 4 which are not connected between themselves, and with separating circles of order six or more.

The condition for the order of circles is considered in [13]. Finding separating circles is performed in two steps.

First of all, it is necessary to find the series of neighbor circles in the following way: begin with a vertex of the polyhedron and take all its neighbor vertices. They are connected to form a circle – the first one in series. The new neighbor vertices of those in the first circle form the second circle. The third circle is formed from new neighbors of the vertices of the second circle, and so on. The process is finished when there are no unused neighbors of the vertices of the last circle. The last circle may be degenerate to a single vertex or a chain.

If all of the neighbor circles in this series (except may be the first and the last one) are of order six or more, then we are searching for a circle of smaller order.

It is shown in [13]] that if we require vertices to have order not greater than six, then the only possible case is a circle with vertices A_1, A_2, A_3 which lie on p and A_4, A_5 on q, while p, q are neighbor circles. Apart from that, the circle p has to be of order six and the new (*separate*) circle $A_1A_2A_3A_4A_5$ have to separate other vertices of p from those of q.

2.2 Graph representation. In this paper, the graph structure is used as a model for polyhedron representation [1]. Here, vertices and edges of a polyhedron are used as vertices and edges of the appropriate graph, while faces and solid are ignored. Graph provides a more natural and consistent approach for this class of algorithms.

Viewed abstractly, a graph $G = (V, E)$ consists of the set V of *vertices* and the set E of *edges* connecting the vertices in V. An edge $e = (u, v)$ is a pair of two vertices u and v. The vertices u and v are called *endpoints* of the edge (u, v).

The *abstract data type* (ADT) is a mathematical model of a data structure that specifies a type of data stored, the operations supported on them, and the types of parameters of the operations. The ADT specifies *what* each operation does, but it does not describe the way it is done.

As an abstract data type, *a graph* is a positional container whose positions are its vertices and its edges. Hence, the graph ADT stores elements at either its edges or vertices (or both). A position in the graph is always defined relatively, that is, in terms of its neighbors.

To make the ways of storing elements abstract and unified, in various implementations of the graph, we introduce a concept of "a *position*" in the graph, which makes the intuitive notion of the "place" element formal, relative to others in the graph.

A position itself is an abstract data type that supports a simple *element*() method which returns the element that is stored at this position. We also use specialized iterators for vertices and edges. An iterator is an enumeration with traversal order which can be guaranteed in some way. In order to simplify the presentation, we denote a vertex position with v, and an edge position with e.

There are admittedly numerous methods for the graph ADT. However, some methods are unavoidable to a certain extent, since graphs are rich structures. We need different methods for accessing and updating some positions in a graph, as well as dealing with the relationships that can exist between these positions. We divide the graph methods into three main categories: general methods, accessor methods and methods for updating and modifying graphs. We do not discuss error conditions that may occur. In addition, we take into consideration only methods for dealing with undirected edges.

We begin by describing fundamental methods for the graph, which ignore the direction of the edges. Each of the following methods returns global information about a graph G:

numVertices()	Return the number of vertices in *G*
numEdges()	Return the number of edges in *G*
vertices():	Return an iterator of the vertices of *G*
edges()	Return an iterator of the edges of *G*

The following accessor methods take vertex and edge positions as arguments:

degree(*v*)	Return the degree of *v*
adjacentVertices(*v*)	Return an iterator of the vertices adjacent to *v*
incidentEdges(*v*)	Return an iterator of the edges incident upon *v*
opposite(*v*,*e*)	Return the endpoint of edge *e* distinct from *v*
areAdjacent(*v*,*w*)	Return whether vertices *v* and *w* are adjacent

We can also allow methods for updating which add or delete edges and vertices:

insertEdge(*v*,*w*)	Insert and return an undirected edge between vertices *v* and *w*
insertVertex(v)	Insert and return a new (isolated) numbering vertex *v* storing the object *o* at this position
removeVertex(*v*)	Remove vertex *v* and all its incident edges
removeEdge(*e*)	Remove edge *e*

In order to perform graph algorithms in a computer, we have to decide how to store a graph. There are several ways to realize the graph ADT with a concrete data structure. In this section, we discuss two popular approaches, usually referred to as the *adjacency list* structure and the *adjacency matrix* [4, 8].

There is a fundamental difference between the *adjacency list* and the *adjacency matrix*. The adjacency list structure only stores the edges actually present in the graph, while the adjacency matrix stores a placeholder for every pair of vertices (whether there is an edge between them or not). This difference implies that, for a graph G with n vertices and m edges, the edge list or adjacency list representation uses $O(n + m)$ space, whereas the adjacency matrix representation uses $O(n^2)$ space.

In modern object-oriented program languages, (such as C# and Java) the ADT can be expressed by an interface, which is simply a list of method declarations. The ADT is realized by a concrete data structure, which is modeled in object-oriented program languages by a *class*. A class defines the data stored and the operations supported by the objects which are instances of the class. Also, unlike interfaces, classes specify *how* the operations are performed. A Java class is said to *implement an interface* if its methods give life to all of those of the interface.

3. ALGORITHMS FOR CONE TRIANGULATION

3.1 The first two algorithms are the testing conditions of Theorem 1. Since priority queue is used, they give us the possibility of efficient implementation and significantly reduce running time. A pseudo-code for this first algorithm is given in the code fragment shown below. The input to the algorithm is an undirected graph *G* with *n* vertices. The output is a lists storing the incident edges on a vertex v_0, v_1, ..., v_n.

Note that the algorithm uses a priority queue ADT *Q* to store the vertices in *G* with their degrees as keys. In each iteration of the outer **for** loop, we put the current vertex and its degree into priority queue *Q*. This ADT allows us to later extract vertices from *Q* in nondecreasing order by their degrees.

Therefore, at each iteration of the **while** loop, algorithm extracts the vertex v_i with the smallest degree (call to the priority queue *removeMin* method) and finds its incident edges (call to graph *incidentEdges* method). Also, the algorithm puts each found edge incident to the current vertex v_i into list L_i (call to list *InsertLast* method).

```
Algorithm1:Finding the order for each  vertex of a graph G
and all incident edges for it
      Input:  A undirected graph G with n vertices
      Output: The lists L0,L1,... Ln store the incident
edges on a vertex v0, v1 and so on

Let Q be an initially empty priority queue;
for each vertex v in G.vertices()  {
       Q.insertItem (G.degree(v), v);  // insert a vertex v
with degree of v as key into Q
}
i = 0;
while Q is not empty
{      Let Li be an initially empty list;
       d = Q.minKey();        //  returns the smallest key
in Q
       v = Q.removeMin();      // removes from Q and
returns vertex v with  the smallest key
       writeLine ( “vertex: {0}   degree: {1} “,v, d);
       writeLine(“incident edges on vertex”);
       for each edge e in G.incidentEdges(v)
      {      w = G.opposite (v,e);         // return the
endpoint of edge e distinct from v
            write (“ ( {0}, {1} ) “,v, w );
           Li.insertLast((v,w)); // insert incident edge
(v,w) on vertex v into Li  as last element
      }
     i++;
} return L0, L1, …, Ln;
```

Figure 1. *Pseudo-code for finding the degrees of all vertices and their incident edges*

Example 1.

The undirected graph *G* with six vertices, shown below, is used as algorithm’s input.

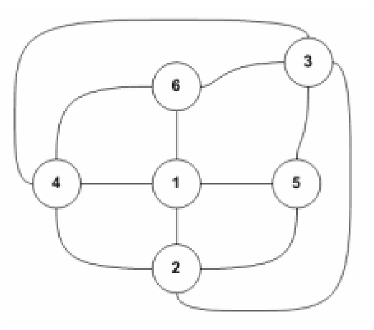

Figure 2. *The graph – input to the algorithm*

The algorithm generates following output:

```
Vertex: 5   Degree: 3
Incident edges on vertex:
(5,1) (5,2) (5,3)

Vertex: 6  Degree: 3
Incident edges on vertex:
(6,1) (6,3) (6,4)

Vertex: 1  Degree: 4
Incident edges on vertex:
(1,2) (1,4) (1,5) (1,6)

Vertex: 2  Degree: 4
      .
      .
```

3.2 In the second algorithm, we consider the problem of finding connected vertices in an undirected graph *G* where each of the vertices has the same order. In the case of Theorem 1, order *d*=4 is of interest, but algorithm is more general. The pseudo-code description of the algorithm is given in Figure 3.

Input to the algorithm is the undirected graph *G* with *n* vertices and specified degree *d.* The algorithm forms, as output, list *L* that contains all connected vertices in *G* where each of them has the same order.

In each iteration of the outer **for** loop, we start with a new vertex *v* and check its degree. If the degree of the vertex *v* is equal to a specified order *d*, then we find all the vertices adjacent to vertex *v*, (call to graph *adjacentVertices* method). In fact, in each iteration of the inner **loop** we check the degree for each vertex *u* found, (*u is adjacent to v*), in order to determine if the connected edge (*v*,*u*) should be added to the list *L*.

```
Algorithm2:For finding connected vertices of the same order
      Input:   A graph G and specified degree d
      Output:  List L
```

```
Let L be an initially empty list;
for each vertex v in G.vertices()
{     if ( G.degree(v) == d) then
      for each vertex u in G.adjacentVertices(v)
      {     if (G.degree(u) == d) then
            L.insertLast ((v,u));   // insert edge (v,u)
into L
      }
}
if  L is not empty
{     writeLine (" specified order{0}", d);
      writeLine(" connected vertices: ");
      for each edge (v,w) in L.elements() {
            write (" ( {0}, {1} ) ",v, w );
      }
      writeLine();
  }  return L
```

Figure 3. *Pseudo-code for finding connected vertices of the same order*

Example 2.

The input to the algorithm is the graph shown in Figure 2 (see previous section). This algorithm formes following output:

```
Specified order: 4
connected vertices:
(1,2) (1,4) (2,1) (2,3) (2,4)
(3,2) (3,4) (4,1) (4,2) (4,3)
```

3.3 The conditions of Theorem 2 are tested in algorithms given in [13]. Here, only short descriptions and examples are presented to complete the investigation. In testing these conditions, one algorithm forms neighbor circles, while the other one forms inserted circles.

The input to the algorithm for forming neighbor circles is an undirected graph G with n vertices and a specific starting vertex $s \in G$. This algorithm forms circles C_0, $C_1,\ldots, C_m$ and lists L_0, $L_1,\ldots,L_{m-1}$. Each C_i represents neighbor circle. Each list L_i contains the edges which connect pairs (u,v) of vertices, where $u \in C_i$ and $v \in C_{i+1}$.

1. We begin with initializing the graph (circle) C_0, so it contains a specific vertex s.
2. In each iteration, the algorithm forms a new graph (circle) C_{i+1} and a new list L_i. The process is repeated until the graph G becomes empty.
3. For each vertex $w \in C_i$ we find its incident edges in graph G. Algorithm for each discovered edge (w,q)
   i. insert SITS endpoint q to graph C_{i+1}
   ii. inserts edge (w,q) to the list L_i
   iii. removes edge (w,q) from the graph G.

4. Algorithm for each vertex $w \in C_{i+1}$ finds all vertices in G, adjacent to vertex w; for each found vertex $v \in G$ adjacent to w, we check if v is in C_{i+1}. If vertex $v \in C_{i+1}$, then edge (w,v) is inserted in C_{i+1}.
5. When C_{i+1} is formed, all edges in C_{i+1} are removed from G.

Example:

An undirected graph G with 14 vertices, shown below, and vertex B_5, are used as algorithm input.

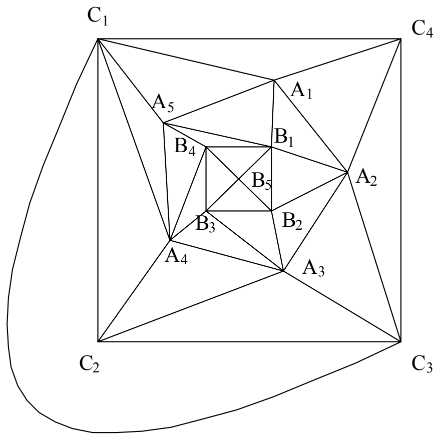

Figure 4. *Input to the algorithm*

The algorithm has generated the following output:

```
formed neighbor circles:
    ordinal number of circle:  0      degree of circle: 1
    vertices of circle:  B5
    edges of  circle:
    edges that connect two adjacent circles  C0  and C1:
       (B5,B1) (B5,B2) (B5,B3) (B5,B4)

    ordinal number of circle:  1      degree of circle:  4
      vertices of circle:  B1, B2, B3, B4,
      edges of  circle:   (B1,B2) (B2,B3) (B3,B4) (B4,B1)
   edges that connect two adjacent circles C1 and C2:
      (B1,A1) (B1,A2) (B1,A5) (B2,A2) (B2,A3) (B3,A3)
      (B3,A4) (B4,A4) (B4,A5)
    ordinal number of circle:  2      degree of circle: 5
      . . . . . . . .
    ordinal number of circle:  3      degree of circle: 4
      . . . . . . . .
```

The input to the algorithm for forming inserted circles are graphs C_0, C_1,…,C_m and lists L_0, L_1,…,L_{m-1}, which are formed by the precedent algorithm. Each graph C_i represents a neighbor circle. Each list L_i stores the edges which connect the pairs (u,v) of vertices, where $u \in C_i$ and $v \in C_{i+1}$.

1. Algorithm begins with process traverses circles (graphs) C_0, C_1, … ,C_m by considering the order of each graph.
2. If algorithm finds a graph C_i of order six, then
   - for each vertex $v \in C_i$

     2.1 finds its two adjacency vertices x and y in C_i

     2.2 adds vertices v, x, y to temporary queue Q

     2.3 **calls** algorithm, for finding two vertices w and q in circle C_{i-1} neighbor to C_i, such that vertex w connects with vertex $x \in C_i$, as vertex q connects with vertex $y \in C_i$. Vertex w also *has to* be adjacent to vertex q, and v has to not be adjacent to w or q. Algorithm returns queue D that contains five vertices of the inserted circle, which is formed.

     2.4 **calls** algorithm for forming inserted circle. The input in this algorithm is queue D and a new empty graph P_k. The algorithm output formed an inserted circle P_k of order five.

   2.5 Furthermore, the main algorithm repeats the above steps 2.3 and 2.4 for circle C_{i+1} neighbor to C_i.
3. The above process repeats while circles C_i of order six exist.
4. When the process is terminated, the algorithm returns inserted circles P_0, P_1, …, P_k.

Example 3.

Input to this algorithm is the polyhedron shown in Figure 5, with neighbor circles A_{01}; A_{11}, A_{12}, A_{13}, A_{14}, A_{15}, A_{16}; A_{21}, A_{22}, A_{23}, A_{24}, A_{25}, A_{26}; A_{31} and edges that connect two adjacent circles: $A_{01}A_{11}$, $A_{01}A_{12}$, $A_{01}A_{13}$, $A_{01}A_{14}$, $A_{01}A_{15}$, $A_{01}A_{16}$; $A_{11}A_{21}$, $A_{12}A_{21}$, $A_{12}A_{22}$, $A_{12}A_{23}$, $A_{13}A_{23}$, $A_{13}A_{24}$, A_{14} A_{24}, A_{14} A_{25}, $A_{14}A_{26}$, $A_{15}A_{26}$, $A_{16}A_{26}$, $A_{16}A_{21}$; $A_{21}A_{31}$, $A_{22}A_{31}$, $A_{23}A_{31}$, $A_{24}A_{31}$, $A_{25}A_{31}$, $A_{26}A_{31}$.

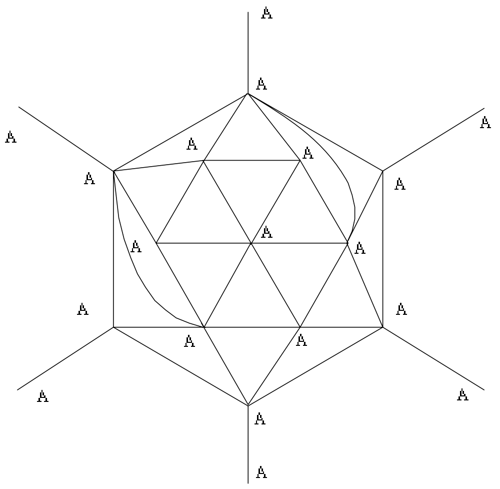

Figure 5. *Input to the algorithm*

The algorithm generates the following output:

```
ordinal number of inserted circle:   0
edges of inserted  circle:
(A13,A14)  (A13,A12)  (A14,A26)  (A12,A21)  (A26,A21)
```

3.4 The first step in doing cone triangulation is finding triangular faces of graph, representing given polyhedron. This is necessary, since the data on faces are omitted in this representation.

The input to the algorithm is an undirected graph *G* with *n* vertices. The algorithm forms list *T,* as the output, which contains triangular faces. The pseudo-code description of the main algorithm is given in Figure 6.

In each iteration of the outer **for** loop, the algorithm starts with a new vertex *u* and finds its incident edges (call to graph *incidentEdges* method). Also, for each found edge (*u*, *v*) incident to the current vertex *u*, the algorithm finds all the vertices adjacent to vertex *u* (call to graph *adjacentVertices* method).

For each vertex *w* found, (*w* adjacent to the vertex *u*), the algorithm checks if the vertices *w* and *v* are adjacent, in order to determine if the new triangular (*u*, *v*, *w*) should be added to the list *T*, (call to list *InsertLast* method). Finally, algorithm processed edge (*u*,*v*), removes from the graph *G*.

When the process is finished, the algorithm returns the formed list *T,* which contains the found triangular faces.

Algorithm3:For finding triangular faces

```
      Input:   A undirected graph G with n vertices
      Output:  List T

Let T be an initially empty list;

for each vertex u  in G.vertices()  {
      for each edge e in G.incidentEdges(u)
            {          v = G.opposite (u,e);          //
return the endpoint of edge e distinct from u
                  for each vertex w  in G.adjacentVertex(u)
{
                  if  G.areAdjacent(v,w) then
                              T.InsertLast((u,v,w)); //
insert a new triple (u, v, w) into list T
             }

            G.removeEdge((u,v)); //remove edge (u,v) from G
      }
  }
writeLine("formed triangular faces");
for each triple (u,v,w)  in T.elements() {
            write (" T ( {0}, {1}, {2}  ) " u,v,w );
} return T
```

Figure 6. *Pseudo-code for forming triangular faces*

Example 4.

An undirected graph G shown in figure 4 is used as algorithm input. We may assume that vertices are given in the following order: A_1, A_2, A_3, A_4, A_5, B_1, B_2, B_3, B_4, B_5, C_1, C_2, C_3, C_4 and edgese in corresponding: A_1A_2, A_1A_5, A_1B_1, A_1C_1, A_1C_4, A_2A_3, A_2B_1, A_2B_2, A_2C_3, A_2C_4, A_3A_4, A_3B_2, A_3B_3, A_3C_2, A_3C_3, A_4A_5, A_4B_3, A_4B_4, A_4C_1, A_4C_2, A_5B_1, A_5B_4, A_5C_1, B_1B_2, B_1B_4, B_1B_5, B_2B_3, B_2B_5, B_3B_4, B_3B_5, B_4B_5, C_1C_2, C_1C_3, C_1C_4, C_2C_3, C_3C_4. Then, in the output, triangular faces are in the following order:
$T(A_1, A_2, B_1)$, $T(A_1, A_2, C_4)$, $T(A_1, A_5, B_1)$, $T(A_1, A_5, C_1)$, $T(A_1, C_1, C_4)$, $T(A_2, A_3, B_2)$, $T(A_2, A_3, C_3)$, $T(A_2, B_1, B_2)$, $T(A_2, C_3, C_4)$, $T(A_3, A_4, B_3)$, $T(A_3, B_4, C_2)$, $T(A_3, B_2, B_3)$, $T(A_3, C_2, C_3)$, $T(A_4, A_5, B_4)$, $T(A_4, A_5, C_1)$, $T(A_4, B_3, B_4)$, $T(A_4, C_1, C_2)$, $T(A_5, B_1, B_4)$, $T(B_1, B_2, B_5)$, $T(B_1, B_4, B_5)$, $T(B_2, B_3, B_5)$, $T(B_3, B_4, B_5)$, $T(C_1, C_2, C_3)$, $T(C_1, C_3, C_4)$.

3.5 The final algorithm serves for the purpose of making cone triangulation.

The inputs to the algorithm are the list T, which is formed by the previous algorithm for finding triangular faces, and a particular vertex q, which represents the apex of the triangulation. The algorithm forms list S as output, *that* contains the found tetrahedra. The pseudo-code description of the main algorithm is given in Figure 7.

In each iteration of the first **for** loop, the algorithm checks if the triangular face from the list T contains a particular apex q. In case the triangular face does not contain a particular apex q, the apex q is added as a fourth vertex, so the tetrahedron formed this way is added to the list S.

```
Algorithm4:For cone triangulation
      Input:  List T and particular vertex q
      Output: List S

Let S be an initially empty list;

for each triple (u,v,w) in T.elements() {
       if not ( (u == q) or (v == q) or (w == q) )   then
      S.InsertLast((q,u,v,w)); // Insert a new tetrahedron
(q,u,v,w) into list S
}
writeLine("formed tetrahedra");
for each tetrahedron (q,u,v,w)  in S.elements() {

            write (" S ( {0}, {1}, {2}, {3} ) " q,u,v,w );
} return S
```

Figure 7. *Pseudo-code for cone triangulation*

Example 5.

If in the previous example, with graph *G* given in figure 4, the particular vertex is A_2, then the output is a list of tetrahedra given in the following order: $S(A_1, A_5, B_1, A_2)$, $S(A_1, A_5, C_1, A_2)$, $S(A_1, C_1, C_4, A_2)$,$S(A_3, A_4, B_3, A_2)$,$S(A_3, B_4, C_2, A_2)$, $S(A_3, B_2, B_3, A_2)$, $S(A_3, C_2, C_3, A_2)$, $S(A_4, A_5, B_4, A_2)$, $S(A_4, A_5, C_1, A_2)$,$S(A_4, B_3, B_4, A_2)$, $S(A_4, C_1, C_2, A_2)$, $S(A_5, B_1, B_4, A_2)$, $S(B_1, B_2, B_5, A_2)$, $S(B_1, B_4, B_5, A_2)$, $S(B_2, B_3, B_5, A_2)$, $S(B_3, B_4, B_5, A_2)$, $S(C_1, C_2, C_3, A_2)$, $S(C_1, C_3, C_4, A_2)$.

4. CONCLUSIONS

In the paper are given algorithms for cone triangulation which are polynomial. That is significant, since finding minimal triangulation of convex polyhedra is an NP-hard problem and cone triangulation provides close number of tetrahedra. The algorithms for testing necessary conditions, given in this paper, are polynomial as well.

Knowing the results of algorithms for necessary conditions along with the results given in [13], we can consider improvements of cone triangulation. The proposed approach can also be extended for the purpose of solving similar geometric problems.

REFERENCES

[1] Baumgart, B. G., "A polyhedron representation for computer vision", in: *Proceedings of the 1975 National Computer Conference, AFIPS Conference Proceedings,* vol. 44. AFIPS Press, Reston, Va., 1975.

[2] Below, A., Loera, J.A., and Gebert, J.R., "The complexity of finding small triangulations of convex 3-polytopes", *arXiv:math/*0012177v1.

[3] Brinkmann, G., and McKay, B.D., "Fast generation of planar graphs", *MATCH Commun. Math. Comput. Chem.*, 58 (2) (2007) 323-357.

[4] Goodrich M., and Tamassia R., *Data Structures and Algorithms in Java*, Second Edition. John Wiley & Sons, 2001.

[5] Chin, F.Y.L., Fung, S.P.Y., and Wang, C.A., "Approximation for minimum triangulations of simplicial convex 3-polytopes", *Discrete Comput. Geom.*, 26 (4) (2001) 499-511.

[6] Develin, Mike, "Maximal triangulations of a regular prism", *J. Comb. Theory, Ser.A,* 106 (1) (2004) 159-164.

[7] Edelsbrunner, H., Preparata, F.P., and West, D.B., "Tetrahedrizing point sets in three dimensions", *J. Simbolic Computation*, 10 (1990) 335-347.

[8] McConnell J., *Analysis of Algorithms: An Active Learning Approach*, Jones and Bartlett Publishers, 2001.

[9] Ruppert, J., and Seidel, R., "On the difficulty of triangulating three – dimensional nonconvex polyhedra", *Discrete Comput. Geom.*, 7 (1992) 227-253.

[10] Sleator, D.D., Tarjan, R.E., and Thurston, W.P., "Rotatory distance, triangulations, and hyperbolic geometry", *J. of the Am. Math. Soc.*, 1 (3) 1988.

[11] Stojanović, M., "Algorithms for triangulating polyhedra with a small number of tetrahedral", *Mat. Vesnik,* 57 (2005) 1-9.

[12] Stojanović, M., "Triangulations of some cases of polyhedra with a small number of tetrahedral", *Krag. J. Math.*, 31 (2008) 85-93.

[13] Stojanović, M., and Vučković, M., "Algorithms for investigating optimality of cone triangulation for a polyhedron", *Krag. J. Math.*, 30 (2007) 327-342.