ROMANIAN JOURNAL OF INFORMATION SCIENCE AND TECHNOLOGY
Volume **15**, Number 4, 2012, 344–354

Block Method for Convex Polygon Triangulation

Predrag S. STANIMIROVIĆ, Predrag V. KRTOLICA,
Muzafer H. SARAČEVIĆ, Sead H. MAŠOVIĆ

Faculty of Science and Mathematics, University of Niš
Višegradska 33, 18000 Niš, Serbia

E-mail: {pecko,krca}@pmf.ni.ac.rs,
{muzafers,sekinp}@gmail.com

Abstract. In this paper, the block method for convex polygon triangulation is presented. The method is based on the usage of the previously generated triangulations for polygon with smaller number of vertices. In the beginning, we establish some relations between the triangulations of the polygons having the consecutive number of vertices. Using these relations, we decompose the triangulation problem into subproblems which are themselves smaller instances of the starting problem. The recursion with memoization is used to avoid repeating the calculation of results for previously processed inputs. The corresponding algorithm is developed and implemented.

Key words: Convex polygon triangulation, Catalan numbers, Recursion, Memoization, Divide & Conquer algorithms.

1. Introduction and Preliminaries

Polygon triangulation is an important problem applicable in computer graphics. It is a basic primitive and preprocessing step for most nontrivial operations on polygons. Decompositions od two dimensional scenes are used, for *e.g.*, in contour filling, hit detection, clipping and windowing (see [1, 2, 5]).

Restricted on the convex case, triangulations are defined by non-intersecting internal diagonals of the polygon. Let P_n denote a polygon with n vertices. In this paper, we suggest a method to triangulate P_n using already produced blocks in the triangulation of a P_b, with $b < n$.

The total number T_n of n-gon triangulations is

$$T_n = C_{n-2} = \frac{1}{n-1}\binom{2n-4}{n-2} = \frac{(2n-4)!}{(n-1)!(n-2)!}, \quad n \geq 3. \tag{1}$$

Here, C_n represents the nth Catalan number (about the topic see *e.g.* [7]).

The set of all triangulations of the convex polygon P_n is denoted by $\mathcal{T}_n$. A diagonal connecting vertices i and j is denoted by $\delta_{i,j}$. An outer face edge can be considered as a diagonal, while non adjacent vertices are connected by an *internal diagonal*.

The general strategy used in our method is to decompose the problem into the smaller dependant subproblems. Each subproblem is solved only once and used many times avoiding unnecessary repetitions of calculation. The algorithm for deriving the set $\mathcal{T}_n$ from $\mathcal{T}_{n-1}$ is presented in [6]. We have been inspired by this algorithm to define our *block method* for convex polygon triangulation. Our method is aimed to accelerate the triangulation process of P_n using $\mathcal{T}_{n-1}$ as blocks of stored internal diagonals. More precisely, our algorithm generates the set $\mathcal{T}_n$ using all the previously generated triangulations $\mathcal{T}_b$, $b < n$. Here, the set $\mathcal{T}_b$ is used as many times as necessary as a block, *i.e.* it is repeated several times in $\mathcal{T}_n$.

Since $C_n = \dfrac{4n-2}{n+1}C_{n-1}$, it is not difficult to verify that the inequality $T_n > 2T_{n-1}$ is satisfied for all $n > 4$. Therefore, the number of triangulations $T_n = C_{n-2}$ can be recursively expressed as

$$T_n = 2T_{n-1} + \mathrm{rest}(R_n). \tag{2}$$

The following statement gives a formal background for our method.

Proposition 1.1. *Every triangulation from $\mathcal{T}_{n-1}$ appears as a starting part of exactly two triangulations form $\mathcal{T}_n$.*

Proof. If we consider an P_{n-1} and its edge $\delta_{1,n-1}$, then it is obvious that in $\mathcal{T}_{n-1}$ this edge belongs to the triangle $(1, i, n-1)$, $2 \leq i \leq n-2$. Let the point which is out of P_{n-1}, and labeled by n, be in the position to form (with nodes $1, \ldots, n-1$) a convex polygon. The diagonals $\delta_{1,i}$ and $\delta_{i,n-1}$ make vertices $2, \ldots, i-1, i+1, \ldots, n-2$ closed with respect to the vertex n. Also, the vertices $1, i, n-1$, and n form a quadrilateral. This quadrilateral can be triangulated in two ways, so each triangulation from $\mathcal{T}_{n-1}$ appears as a starting part in two triangulations from $\mathcal{T}_n$.

Note that there are some other triangulations in $\mathcal{T}_n$ which should be derived in other way. □

Memoization is a useful and important tool for solving heavy recursive computations (cf. [3]). The triangulation problem is, by its nature, the recursive and time consuming job. The underlying idea is to speed up recursive algorithms by avoiding recomputations caused by the overlapping subproblems.

Instead of having an entry in the table for the solution of each subproblem, in this case we have stored previously generated triangulations $\mathcal{T}_{n-1}$. As it is explained above, in the significant number of cases, a triangulation from $\mathcal{T}_{n-1}$ is an initial part of two triangulations in $\mathcal{T}_n$. Here it is not necessary to proceed with recursion further

(down to the triangle), but, in the memoization style, we just use what we already have and, suitably complementing it, we get two new triangulations. Moreover, in the case of traditional recursion, generating of $\mathcal{T}_{n-1}$ will occur twice in independent calculation subtrees. Now, this repetition is avoided.

This general idea is illustrated in Fig. 1, where the transformation process from a P_5 triangulation in two corresponding P_6 triangulations is presented. In part (a) we see that the diagonals $\delta_{2,4}$ and $\delta_{2,5}$ make all vertices closed except vertices 1, 2, 5, and 6 which form a quadrilateral. The parts (b) and (c) show two ways to triangulate a quadrilateral which give two P_6 triangulations having the P_5 triangulation as a starting block.

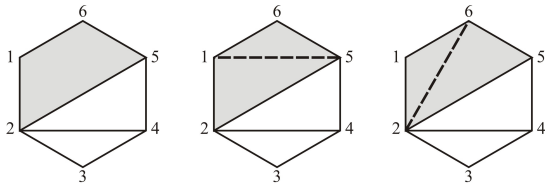


Fig. 1. Transformation from a P_5 triangulation into the corresponding P_6 triangulations.

A vertex i is *closed* by an internal diagonal with respect to the vertex j if vertex i can not be an ending point of internal diagonal $\delta_{i,j}$ because it implies the diagonal crossing (recall that a triangulation is made by non-intersecting internal diagonals). We simply say that i is the *closed* vertex if i is closed with respect to all other vertices in the polygon. The closed vertex has the degree 2 and it is alternatively called an *ear*.

The algorithm presented in Section 2 is compared with algorithm developed by Hurtado and Noy and presented in [6]. For this reason, we restate here this algorithm.

Algorithm 1 Hurtado algorithm

Require: Positive integer n and the set $\mathcal{T}_{n-1}$ of P_{n-1} triangulations. Each triangulation is described as a structure containing $2n-5$ vertex pairs presenting P_{n-1} diagonals (here *diagonals* means both internal diagonals and outer face edges).

1: Check the structure containing $2n-5$ vertex pairs looking for pairs $(i_k, n-1)$, $i_k \in \{1, 2, \ldots, n-2\}$, $2 \leq k \leq n-2$, i.e. diagonals incident to vertex $n-1$. The positions of these indices i_k within the structure describing a triangulation should be stored in the array.
2: For every i_k perform the transformation $(i_l, n-1) \rightarrow (i_l, n)$, $i_l < i_k$, $0 \leq l \leq n-3$.
3: Insert new pairs (i_k, n) and $(n-1, n)$ into the structure.
4: Take next i_k, if any, and go to Step (2).
5: Continue the above procedure with next $(n-1)$-gon triangulation (*i.e.* structure with $2n-5$ vertex pairs) if any. Otherwise halt.

In the Section 2 we present the algorithm for the block method, while the execution times for both algorithms are given in Section 3. Some interesting parts of the code are presented in Appendix.

2. Algorithm for block method

Similarly to the definition of the edge length in [9], we define a distance between two polygon vertices.

Definition 2.1. The distance between two integers i and j, where $i, j \in \{1, \ldots, m\}$, is defined as

$$d(i,j) = d(j,i) = \min\{|i-j|, m-|i-j|\}.$$

In the procedure used for finding and eliminating closed vertices, we should start from an ear. As a triangulation has at least two ears, and, in the worst case one ear can be a vertex n, then we always have at least one ear among the rest of the vertices.

For this purpose we make a list of ordered pairs of the form

$$L = \{(1,1), (2,2), \ldots, (n,n)\}. \tag{3}$$

After the elimination of $n-l$ pairs the list L becomes

$$L = \{(s, i_s),\ s = 1, \ldots, l\},\ 4 \le l \le n,\ i_l = n. \tag{4}$$

The values $i_s,\ s = 1, \ldots, l$ are the vertex marks, while values $1, \ldots, l$ represent the relative vertex positions in the list L.

Algorithm 2 Pair elimination

Require: List L of the form (4) and vertices i_p and i_q, where $d(p,q) = 2$.

1: Remove from the list L the pair placed between the pairs (p, i_p) and (q, i_q) in circular manner.
2: Decrease by one the first pair members in the pairs following the eliminated one.

Further, we check the first $n-4$ columns in the table for $\mathcal{T}_n$ looking for an ear. The ear is recognized when, for diagonal δ_{i_p,i_q}, we have $d(p,q) = 2$. Then we eliminate the corresponding list element and decrease relative positions of the pairs following the eliminated pair.

Algorithm 3 Form a qaudrilateral

Require: List L of the form (3), integer n and array of $n-4$ diagonals (*i.e.* a row in the table for $\mathcal{T}_n$).

1: Find a diagonal δ_{i_p,i_q} where $d(p,q) = 2$ in the list L.
2: Call Algorithm 2 for parameters i_p and i_q.
3: Repeat Steps 1–2 $n-4$ times.

After $n-4$ pair eliminations, we have four vertices in the list forming a quadrilateral which can be triangulated in two ways.

Algorithm 4 Algorithm for block method

Require: An integer n and $\mathcal{T}_b$ with $row_b = C_{n-3}$ rows and $col_b = n-4$ columns)

1: Create an empty table for $\mathcal{T}_n$ with $row_n = C_{n-2}$ rows and $col_n = n-3$ columns.

2: Fill the table for $\mathcal{T}_n$ by the triangulations from $\mathcal{T}_b$ duplicating each row from $\mathcal{T}_b$.

3: Fill the rest of entered blocks (the last column in the first $2row_b$ rows) in the following way.

`for` $(i = 1; i <= 2row_b; i+ = 2)$
{
Make a list L of the form (3).
Call Algorithm 3 with row i from table for $\mathcal{T}_n$ as a parameter.
From the remaining four vertices in list L make diagonal δ_{i_1,i_3} and place it in the last column of the row i and diagonal δ_{i_2,i_4} and place it in the last column of the row $i+1$.
}

4: Fill the rest of the table for $\mathcal{T}_n$ containing $T_n - 2T_b$ rows.

4.1 Filling the first $n-4$ columns in the last $row_n - 2row_b$ rows.
$i = 2 * row_b + 1$;
Make the list L of the form (3).
Eliminate the vertices adjacent to n calling Algorithm 2 for parameters 1 and $n-1$.
Fill the current table row i by diagonals $\delta_{2,n}, \delta_{3,n}, \ldots, \delta_{n-2,n}$.
The first $n-4$ columns in the rest $row_n - 2row_b - 1$ rows should be filled by diagonals with the last vertex n, while the first vertices are combinations of the $(n-4)$th class in the set $\{2, 3, \ldots, n-2\}$. The number of these combinations is $\binom{n-3}{n-4} = n-3$.

4.2 Filling the last column in the last $(row_n - 2row_b)$ rows.
`for` $(i = 2row_b + 2; i <= row_n; i++)$
{
Make the list L of the form (3).
Call Algorithm 3 with row i from table for $\mathcal{T}_n$ as a parameter.
From the remaining four vertices in list L make diagonal δ_{i_1,i_3} and place it in the last column of the row i.
}

Example 2.1. As an example, we will present the procedure to derive $\mathcal{T}_6$ on the base of $\mathcal{T}_5$. As we have already presented, Algorithm 4 consists of four steps. In the first step, an empty table with the appropriate number of columns and rows is created. In the second step, the table is filled with $\mathcal{T}_b$ in the duplicate. In the third and fourth step, the remaining empty parts of the table are filled in (Fig. 2). In Step (1) an empty table is created. Step (2) consists of copying the already derived blocks. In Step (3) we fill the last column where blocks are copied. Step (4.1) fills the first two columns in the rest of the rows and Step (4.2) completes the last column.

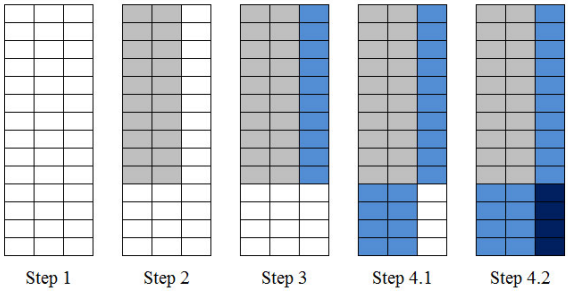
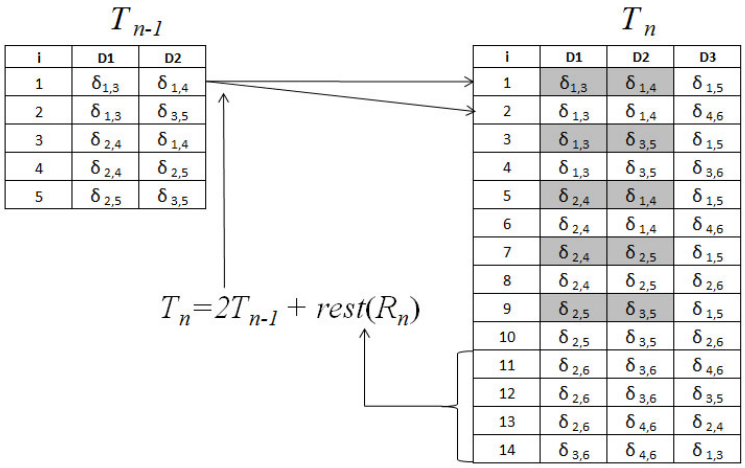


Fig. 2. Filling the table (step by step).

The process of $\mathcal{T}_6$ generation, based on a given $\mathcal{T}_5$, is presented in Fig. 3.



i	D1	D2
1	$\delta_{1,3}$	$\delta_{1,4}$
2	$\delta_{1,3}$	$\delta_{3,5}$
3	$\delta_{2,4}$	$\delta_{1,4}$
4	$\delta_{2,4}$	$\delta_{2,5}$
5	$\delta_{2,5}$	$\delta_{3,5}$

i	D1	D2	D3
1	$\delta_{1,3}$	$\delta_{1,4}$	$\delta_{1,5}$
2	$\delta_{1,3}$	$\delta_{1,4}$	$\delta_{4,6}$
3	$\delta_{1,3}$	$\delta_{3,5}$	$\delta_{1,5}$
4	$\delta_{1,3}$	$\delta_{3,5}$	$\delta_{3,6}$
5	$\delta_{2,4}$	$\delta_{1,4}$	$\delta_{1,5}$
6	$\delta_{2,4}$	$\delta_{1,4}$	$\delta_{4,6}$
7	$\delta_{2,4}$	$\delta_{2,5}$	$\delta_{1,5}$
8	$\delta_{2,4}$	$\delta_{2,5}$	$\delta_{2,6}$
9	$\delta_{2,5}$	$\delta_{3,5}$	$\delta_{1,5}$
10	$\delta_{2,5}$	$\delta_{3,5}$	$\delta_{2,6}$
11	$\delta_{2,6}$	$\delta_{3,6}$	$\delta_{4,6}$
12	$\delta_{2,6}$	$\delta_{3,6}$	$\delta_{3,5}$
13	$\delta_{2,6}$	$\delta_{4,6}$	$\delta_{2,4}$
14	$\delta_{3,6}$	$\delta_{4,6}$	$\delta_{1,3}$

Fig. 3. Generated triangulation for $n = 6$.

Upon the creation of the empty table and its filling by $\mathcal{T}_5$ in the duplicate (Steps (1) and (2)), in Step (3) we fill the last column of the first 10 rows. For each of 5 odd rows ($T_5 = 5$ and even rows in first two rows contain a copy of the previous odd indexed row) we should make a list $L = \{(1,1),(2,2),(3,3),(4,4),(5,5),(6,6)\}$. Then, we try to find a diagonal whose ending points have the distance 2. Such a diagonal in the first row is $\delta_{1,3}$. So, we eliminate a vertex between them, i.e. the vertex 2, decreasing the first pair element by 1 in the pairs following the eliminated one. In this way, our list becomes $L = \{(1,1),(2,3),(3,4),(4,5),(5,6)\}$. Now, the ending points of the diagonal $\delta_{1,4}$ also have the distance 2. The list transforms to $L = \{(1,1),(2,4),(3,5),(4,6)\}$. We have eliminated $n - 4 = 2$ closed vertices, and within the list there are four remaining vertices 1, 4, 5, and 6. They form a quadrilateral which can be triangulated in two ways having $\delta_{1,5}$ or $\delta_{4,6}$ as a diagonal. So, in the last column of the first row we store $\delta_{1,5}$ and in the last column of the second row we store $\delta_{4,6}$.

We fill the first 10 rows in a similar way.

In Step (4) we fill the rest of the table. In this part any already present diagonal (from the first 10 rows) can not be repeated. The first 10 rows have 5 rows with diagonals not ending in the vertex 6 and 5 rows with exactly one diagonal ending in the vertex 6. Therefore, in the rest of the table we must have rows with exactly three diagonals ending in vertex 6 (there is exactly one such a row) or rows with exactly two diagonals ending in 6.

After we have made a list L and eliminated vertices 1 and 5 (adjacent to 6), our list is $L = \{(1,2),(2,3),(3,4),(4,6)\}$. There is one possibility with three diagonals ending in 6, $\delta_{2,6}$, $\delta_{3,6}$, and $\delta_{4,6}$. Further, there are three combinations with exactly two diagonals with ending point in 6, namely $\delta_{2,6}$, $\delta_{3,6}$; $\delta_{2,6}$, $\delta_{4,6}$; $\delta_{3,6}$, $\delta_{4,6}$. This is done in step (4.1). The step (4.2) completes the work. Again we make the list $L = \{(1,1),(2,2),(3,3),(4,4),(5,5),(6,6)\}$ and checking the non-completed rows we make vertex elimination as in Step (3). The diagonal $\delta_{2,6}$ forces the elimination of vertex 1 transforming a list in $L = \{(1,2),(2,3),(3,4),(4,5),(5,6)\}$. After this, the diagonal $\delta_{3,6}$ also has the ending points on the distance 2 and implies the elimination of the vertex 2. The list becomes $L = \{(1,3),(2,4),(3,5),(4,6)\}$. We need a diagonal not ending in 6, so we choose $\delta_{3,5}$ and store it in the last column. This step, repeated for the remaining rows, completes the algorithm.

3. Experimental Results

In [8] we investigate which programming language (Java, C++ and Python) is most suitable to implement algorithms for a convex polygon triangulation. Based on comparative analysis, Java programming language gave the best results in terms of speed of generating triangulations. For implementation of Block method we used NetBeans IDE environment for Java.

Table 1. The execution times for two algorithms (in seconds)

n	**Number of triangulations**	**Hurtado**	**Block Method**	**Speedup**
5	5	0.25	0.16	1.56
6	14	0.34	0.26	1.31
7	42	0.43	0.34	1.26
8	132	0.49	0.41	1.20
9	429	0.67	0.46	1.46
10	1,430	1.18	0.54	2.19
11	4,862	3.81	0.85	4.48
12	16,796	12.46	1.32	9.44
13	58,786	50.51	4.40	11.48
14	208,012	119.05	24.13	4.93
15	742,900	318.63	85.30	3.74
16	2,674,440	/	529.30	

The execution times for both algorithms are presented in Table 1. The table column “Speedup” shows the quotient of values contained in the previous two columns.

The testing is performed in NetBeans testing module "Profile Main Project / CPU Analyze Performanse" in configuration*: *CPU - Intel(R) Core(TM)2Duo CPU, T7700, 2.40 GHz, L2 Cache 4 MB (On-Die,ATC,Full-Speed), RAM Memory - 2 Gb, Graphic card - NVIDIA GeForce 8600M GS.*

The algorithm produces the triangulations of an n-gon using already known triangulations of an $(n-1)$-gon. The number of triangulations growths rapidly with n and these triangulations are stored in a file. For any larger n, all $(n-1)$-gon triangulations can not be in the same time in memory. It is not important is it for $n = 15, 16$, or 20 (memories are getting larger in time) - the fact is: from some n we have to include time needed for reading/writing data form disk. This fact implies significant increase of the execution time. But it does happen in the same way with Hurtado's algorithm witch we try to accelerate. Experiment with larger values of n and inclusion of I/O in execution time will make the performance difference of two algorithms blurry. For $n > 16$, in Table 1, the RAM memory is exhausted. This problem can be overcome applying the so-called *virtual memory paging file* (VMPF).

The better performances of the *Block* method with respect to the *Hurtado* algorithm can also be confirmed by using the so-called performance profile, introduced in [4]. The underlying metric is defined by the CPU time spanned for the construction of all triangulations for the cases $n = \{5, \ldots, 15\}$. Following the notations given in the paper [4] we have that the number of solvers is $n_s = 2$ (*Block* and *Hurtado*) and the number of numerical experiments is $n_p = 11$. By $t_{p,s}$ we denote the number of iterations required for solving the problem p by the solver s. The quantity

$$r_{p,s} = \frac{t_{p,s}}{\min\{t_{p,s} : s \in \{Hurtado, Block\}\}}$$

is called the performance ratio. The performance of the solver s is defined by the following cumulative distribution function

$$\rho_s(\tau) = \frac{1}{n_p}\text{size}\{p \in \mathcal{P} : r_{p,s} \leq \tau\},\ s \in \{Hurtado, Block\}$$

where $\tau \in \mathbb{R}$ and $\mathcal{P}$ represents the set of problems.

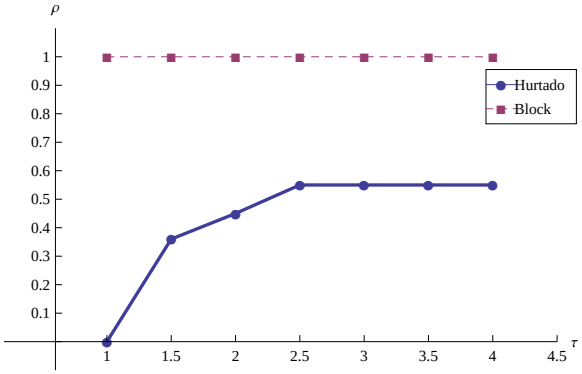


Fig. 4. Performance profile regarding the CPU times arranged in Table 1.

Figure 4 shows data arranged in Table 1.

It is clear from Figure 4 that the *Block* method shows better performances compared to *Hurtado* method. Namely, the probability of being the optimal solver is in favor of the *Block* algorithm.

4. Conclusion

We have developed the algorithm for convex polygon triangulation which uses the already made triangulations of a polygon with one vertex less. The similar approach is used in [6], so we have compared these two algorithms. Our block method uses the triangulations $\mathcal{T}_{n-1}$ as whole blocks in $\mathcal{T}_n$ making the respectable part of the job done quickly. The achieved significant speedup is a consequence of this approach.

Appendix: Application and Implementation Details

The applications **Hurtado-Noy order** and **Block method** are implemented in programming language Java in NetBeans environment. The application input is an integer n, i.e. the number of vertices for the polygon which is about to be triangulated. Firstly, it is checked if there already exists the set $\mathcal{T}_{n-1}$ stored in a file.

Both applications are implemented using the following classes: `Triangulation`, `App`, `Node`, `LeafNode`, `Point`, and `PostScriptWriter`. The difference between them is in the main class `App` containing executive method `main()`. In the application for Hurtado algorithm, the class `App` calls its own method `Hurtado()` while the Block method application calls the method `Block()`.

The method `Block()` corresponds to Algorithm 4. This method can be divided in four parts. The first one takes triangulations $\mathcal{T}_{n-1}$ (Step 1). After this, $\mathcal{T}_{n-1}$ are copied (Step 2). In Step 3, the last column is filled with an additional diagonal. This is the job of the method `contains()` which is a member of the class `Node`. The method corresponds to Algorithm 3 and it is aimed to find the non-closed vertices forming a quadrilateral. The method `contains()` calls the method `elimination()` corresponding to Algorithm 2. The last part finds the rest of triangulations (Step 4.1). Within this part, the method `contains()` is called again to complete Step 4.2.

Java source code for Block method - (corresponds to Algorithm 4)

```
public Vector<Node> Block (int CatNum) throws IOException {

//step 1

Vector<Vector> bl = new Vector<Vector>();
this.openFile("baze/[T"+(CatNum+1)+" BAZA].jdb");
bl.add(new Vector<LeafNode>());
bl.get(0).add(new LeafNode());

//step 2

int lim=2; lim=CatNum;
```

```
for (int n=0; n <=CatNum; n++) {
        Vector<Node> L = new Vector();
        bl.add(L);
    for (int row = 0; row < bl.get(n).size(); row++) {
            String as="row"+"#"+(row+1);
              if(n==lim) System.out.println("row "+(row+1));
                  Node t = (Node)bl.get(n).get(row);
                  Node s = new Node(new LeafNode(),t.copy());
                  L.add(s);
//step 3

int remainingBranch= n-2*n1;
for (int k = 0; k < t.remainingBranch();  k++) {
          s = t.copy();
          Node r = s;
          L.contains(r);
          Vector<Node> R = new Vector();
          r.equals(R);

//step 4

for (int i=0; i < k; i++){

//step  4.1
          s = s.getLeft(); L.lastElement();
          s.setLast(new Node(new LeafNode(), s.getLeft()));
//step  4.2
          L.contains(r);
          L.add(r); }
         }
    }
return bl.get(CatNum);
}
```

Java source code for method `contains()`-(corresponds to Algorithm 3)

```
   public int contains() {
    int a=this.contains()-this.left.contains();
    int b=this.right.contains()-(this.contains()-this.left.contains());
        for (i=0; i<=(n-4); i++){
              if((a-b)==2) {
             L.elimination();}
           }
           return L.remainingBranch();
        }
```

Java source code for method `elimination()`-(corresponds to Algorithm 2)

```
public int elimination(){
           int t=this.right.elimination()-this.left.elimination();
           return t; }
```

Java source code for method `Hurtado()`

```
public Vector<Node> Hurtado(int CatNum) {
    Vector<Vector> H = new Vector<Vector>();
    H.add(new Vector<LeafNode>());
    H.get(0).add(new LeafNode());

        for (int n=0; n <= CatNum; n++) {
            Vector<Node> level = new Vector();

        for (int i= 0; i < H.get(n).size(); i++) {
            Node t = (Node)H.get(n).get(i);
            Node s = new Node(new LeafNode(), t.copy());
            level.add(s);

        for (int k = 0; k < t.leftBranch(); k++) {
            s = t.copy();
            Node l = s;

        for (int i=0; i < k; i++)
            s = s.getLeft();
            s.setLeft(new Node(new LeafNode(), s.getLeft()));
            level.add(l); }
                }
            H.add(level);
        }

return H.get(CatNum);
}
```

References

[1] Chazelle B., *Triangulation a Simple Polygon in Linear Time*, Discrete Computational Geometry, Vol. **6**, pp. 485-524, 1991.

[2] Chazelle B., Palios L., *Decomposition Algorithms in Geometry*, Algebraic Geometry and its Application (ed. Ch. L. Bajaj), Springer-Verlag, Vol. **27**, pp. 419–447, 1994.

[3] Cormen T. H., Leiserson C.E., Rivest R. L., Stein C., *Introduction to Algorithms, Second Edition*, The MIT Press, 2001.

[4] Dolan E. D., Moré J.J., *Benchmarking optimization software with performance profiles*, Mathematical Programming. Vol. **91**, pp. 201–213, 2002.

[5] Garey M. R., Johnson D. S., Preparata F.P., Tarjan R.E., *Triangulating a simple polygon*, Information Processing Letters, Vol. **7**, pp. 175–180, 1978.

[6] Hurtado F., Noy M., *Graph of Triangulations of a Convex Polygon and Tree of Triangulations*, Computational Geometry, Vol. **13**, pp. 179–188, 1999.

[7] Koshy T., *Catalan Numbers with Applications*, Oxford University Press, New York, 2009.

[8] Saračević M., Stanimirović P.S., Mašović S., Biševac E., *Implementation of the convex polygon triangulation algorithm*, Facta Universitatis, series: Mathematics and Informatics, Vol. **27**, pp. 213–228, 2012.

[9] Sen-Gupta S., Mukhopadhyaya K., Bhattacharya B. B., Sinha B. P., *Geometric Classification of Triangulations and Their Enumeration in a Convex Polygon*, Computers and Mathematics with Applications, Vol.**27**, pp. 99–115, 1994.