Computer Science Journal of Moldova, vol.**33**, no.2(98), 2025

A poor man's realization of Demoucron-Malgrange-Pertuiset algorithm

Constantin Ciubotaru

Abstract

An implementation of the Demoucron-Malgrange-Pertuiset (DMP) algorithm is proposed, based on specifying the notion of a segment (fragment, bridge) and developing an algorithm for calculating and updating the segments after each iteration using a depth-first search strategy (DFS). The algorithm also works for nonplanar undirected graphs by finally constructing a planar subgraph and displaying the list of segments that cannot be embedded, so as they generate edge intersections when drawing.

Keywords: biconnected graph, DMP algorithm, segment, face, segments/graphs embedding, segments/faces update.

1 Introduction

Drawing hierarchical structures (graphs, trees, schemes) is one of the most attractive ways to present information. Thanks to the geometric structures used in drawing, the range of applications of this subject is very wide: mathematics, computer science, social networks, databases, bioinformatics, linguistics, artificial intelligence, etc.

One and the same graph can be drawn in several ways. Some may be simpler, more comprehensible, having an attractive aesthetic appearance, others – more difficult to perceive, with an unsatisfying structure. All drawings from Figure 1.1 represent the same graph. Fig. 1.1(a) represents a variant drawn manually, the other variants being drawn semi-automatically: Fig. 1.1(b) represents a variant with random placement of vertices that we will call "spaghetti", Fig. 1.1(c) – a planar variant obtained automatically by the method of circular orbits,

doi:10.56415/csjm.v33.11

and the Fig. 1.1(d) – a variant based on the faces structure obtained by application of the *DMP* algorithm.

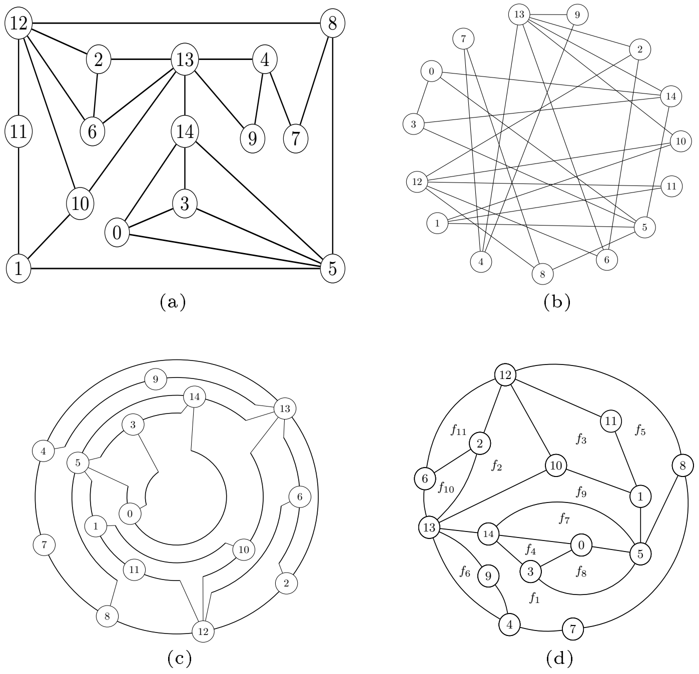


Figure 1.1. Four figures

Research in this area has led to the development of some practical methods of graphic representation that, in parallel, also tests graph planarity. Having multiple representations of the same graph, it is quite easy to choose the most suitable option, especially when having a lot of available appreciation criteria, usually of an aesthetic nature. For example, the graph must:

- be integrated into a determined, limited space,
- contain as few edge intersections as possible,
- avoid sharp bends,
- respect the proportions regarding the length of the edges and values

of the angles of incidence,
- favor the elements of symmetry, of concentration of vertices,
- use appropriate shapes for vertices,
- respect the orientation of the information flow (top to bottom or left to right), and others.

It is quite difficult to transmit these criteria to the computer. Here comes not only the problem of formalizing the criteria, but also the fact that some of them may be contradictory. In these cases, the compromise situations inevitably arise. Most planarity checking algorithms [1] are based on the application of two methods. The first method refers to the Kuratowski theorem (or the Wagner theorem).

Theorem 1.1. (Kuratowski, [2])
A finite graph is planar if and only if it does not contain a subgraph that is homeomorphic to K_5 (Fig. 1.2(a)) or $K_{3,3}$ (Fig. 1.2(b)).

Theorem 1.2. (Wagner, [3])
A finite graph is planar if and only if its minors include neither K_5 nor $K_{3,3}$.

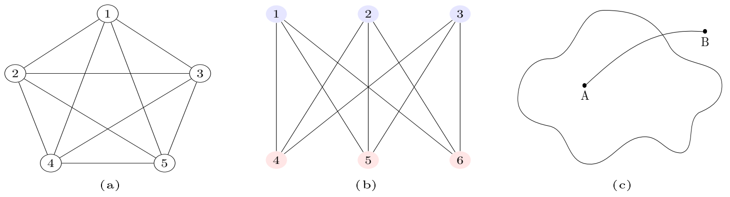


Figure 1.2. Kuratowski graphs and Jordan curve

The second method is based on the fundamental property of the closed Jordan curve, which states that no point inside the curve can be connected to any other point outside it without intersecting the curve (Fig. 1.2(c)).

The aim of this paper is not so much to test planarity, but to apply one of the methods of planarity testing to the automatic drawing of planar graphs, a second problem being no less complicated than testing planarity. Selecting that method will contribute to the achievement of the above-mentioned desires.

Thus, the Demoucron-Malgrange-Pertuiset algorithm (DMP) [4]–[8], based on the Jordan curve property, was selected, essentially modified, and implemented. This algorithm, in parallel to the planarity test, also constructs a partition of the graph (a set of faces) which is used for automatic graph drawing. In the work [9], this algorithm was called "the γ algorithm".

In the paper, figures obtained automatically by the circular orbit method based on the final results of the DMP algorithm are inserted. The implementation of the DMP algorithm requires specifying the notions (structures) used in the description of the algorithm, such as segment (fragment, bridge, component), face, *α-chain*, but also the choice of methods for constructing these structures. Most descriptions of this algorithm bypass the practical aspects related to the construction of these structures and the actual drawing. It is considered to expose the drawing scheme, assign coordinates to vertices, establish the shape of edges, calculate the values of incidence angles, and others. Moreover, some publications insert proofs for the correctness of the DMP algorithm that contain errors [10].

2 Preliminary notions

Traditionally, we will denote an *undirected graph* by $G = (V, E)$, where V is the set of vertices, E – the set of edges, and *adj(v)* – a list of vertices that are adjacent to v.

A sequence of vertices, in which every two consecutive vertices are adjacent, is called *path*: $(v_1\, v_2\, v_3 \ldots v_n)$ or $((v_1\, v_2)\, (v_2\, v_3) \ldots (v_{n-1}\, v_n))$ with the property $(v_i, v_{i+1}) \in E$ for $1 \leq i < n$. The length of a path is the number of edges on a path. We will denote the empty path by *nil* or (). The path is called *elementary* (*simple*) if all included vertices (edges) are distinct.

A path, in which the first vertex coincides with the last, is called *cycle*. The cycle is *elementary* if it consists only of distinct vertices, excluding the first and last. The minimum length of a cycle is 3 (for undirected graphs).

The graph G is called *connected* if there is a path between every pair of vertices. A vertex is called a *point of articulation* (*critical vertex*) if the subgraph, obtained by removing this vertex and its incident edges, is no longer connected.

The undirected graph G is called a *biconnected graph* if it has no articulation points. A *biconnected component* of a graph is a maximal biconnected subgraph with this property. Any connected graph can be decomposed into a set of bicomponents (also called blocks) that can be joined into a tree through the critical vertices.

A graph is *planar* if it can be drawn (embedded) in the plane, so that each vertex is represented by a distinct point, and each edge *(u v)* – by a Jordan curve, without self-intersections with the extremities u and v. All these curves do not mutually intersect except for the vertices incident to these curves. By $G_i = (V_i, E_i)$ we denote any subgraph of the graph $G = (V, E)$. A planar graph divides the plane into regions (bounded by edges), called *faces*. We denote the set of faces by $\mathscr{F}$.

3 Demoucron-Malgrange-Pertuiset algorithm

Any subgraph G_i of a planar graph G can be obtained from the drawn image of the graph G by removing some number of edges and vertices. Thus, any subgraph of a planar graph is also planar. If G_i is a subgraph of the graph G, then the vertices of the subgraph G_i are called *contact vertices*. The reverse action is also possible: building a new planar subgraph by adding to G some number of edges and vertices from $G \setminus G_i$. This is the idea of the *DMP* algorithm. When describing the algorithm, the notions of *fragment* and *α–path* play an important role. The definition of the *fragment* notion used in the papers [4],[5],[9] is inserted below.

Definition 3.1. (Fragment)
If G_i is a subgraph of the graph G, then the *fragment* of G with respect to G_i is:

- an edge that does not belong to G_i but has the extremities in G_i or
- a connected component of $G \setminus G_i$ along with all edges and vertices connecting it to G_i.

Definition 3.2. ***α-path***
The *α-path* is defined as any simple path l which belongs to a fragment of G with respect to G_i and has contact vertices as extremities.

Below is inserted the pseudocode of DMP algorithm [4]–[7],[9](Algorithm 3.1).

Algorithm 3.1. ***Pseudocode of DMP algorithm***

```
1. Let $G = (V, E)$ be a biconnected undirected graph.
2. An arbitrary simple cycle of the graph $G$ is selected and embedded
   (drawn) obtaining a planar graph (subgraph of the graph $G$) denoted
   by $G_c=(V_c, E_c)$.
3. if $G_c=G$ then return "The graph is planar".
4. The set of all fragments $\mathscr{S}$ of $G$ with respect to $G_c$ is built.
5. if $\mathscr{S}=\varnothing$ then return "The graph is planar".
6. For all fragments $s$ of $\mathscr{S}$ build the set of all admissible faces for $s$,
   $\mathscr{F}(s)$.
7. if $\exists$ $s$ with $\mathscr{F}(s)=\varnothing$ then return "The graph is nonplanar".
8. if $\exists$ $s$ with $|\mathscr{F}(s)|=1$, $\mathscr{F}(s)=\{f\}$, then let $l$ be an arbitrary $\alpha$-path of
   the fragment $s$ else let $s$ be an arbitrary fragment for which $|\mathscr{F}(s)|>1$,
   $f$ an arbitrary face from $\mathscr{F}(s)$, and $l$ – an arbitrary $\alpha$–path of the frag-
   ment $s$.
9. $l$ is embedded in $f$ and $\mathscr{F}$ is modified. $G_c := G_c \cup l$.
10. goto 4.
```

Observations and proposals regarding implementation of the algorithm

1. From the description of the *DMP* algorithm, the importance of checking the graph for connectedness and biconnectivity is observed. If the graph is connected, then the critical points can be highlighted, and the set of bicomponents can be constructed, which can be joined in a tree through the critical points. The biconnected graph will contain only one bicomponent, the graph itself. It is also known that a graph is planar if and only if all its bicomponents are planar [11]. Thus, without imposing serious restrictions, we will consider the problem of checking planarity and graph drawing only for biconnected graphs.
2. For the implementation of the algorithm, additional explanations

of notions "fragment (segment)", "H-fragment", and "bridge" are needed.
3. It is necessary to mention (or add) the methods that select the initial cycle and build the sets $\mathscr{S}$, $\mathscr{F}$, $\mathscr{F}(s)$.
4. Most publications ignore the importance of practical aspects that accomplish the stages of the algorithm, especially the aspects concerning graph drawing: the drawing scheme, assignment of coordinates for vertices, choice of edge shapes, incidence angle values, and others.
5. It is easy to see that at any iteration of the algorithm, only one α–path is drawn, the fragments being used only to prove the correctness of the algorithm. As an α–path is at the same time a fragment (segment), we can operate on the implementation of the algorithm only with α–path.

As follows, we present an implementation of the *DMP* algorithm specifying the notions used and applying the simplest, most comprehensible methods in order to achieve the decisive phases, possibly decreasing the efficiency, bypassing the methods considered more effective, but sometimes difficult to understand and implement.

4 Segments building

Let us first specify the notion of *segment* which will be used frequently along the way.

Definition 4.1. Segment
The notion ***segment*** of $G = (V, E)$ with respect to the subgraph $G_c = (V_c, E_c)$ is defined as:
- the edge $e = (u, v)$, $e \notin E_c$, $u, v \in V_c$ (*segment–edge*), or
- the simple path $(e_1 e_2 \dots e_n) = ((u_1 v_1) \dots, (u_{n-1} v_{n-1})(u_n v_n))$, $e_1, e_2, \dots, e_n \notin E_c$, $u_1, v_n \in V_c$, (u_1, v_n are contact vertices), $\{v_1, u_2, v_2, \dots, u_{n-1}, v_{n-1}, u_n\} \cap V_c = \varnothing$ (*segment–path*).

The following notations will be used:
– ***NonContactEdges*** – the set of edges in E not involved in E_c, $NonContactEdges = E \setminus E_c$;

– ***SegEdges*** –the set of segments–edges from *NonContactEdges*,
$SegEdges = \{(u\,v) | (u\,v) \in NonContactEdges, (v \in V_c) \wedge (u \in V_c)\}$;
– ***SegPaths*** – the set of segments–path;
– $\mathscr{S}$ – the set of all possible segments, $\mathscr{S} = SegEdges \cup SegPaths$;
– ***InitSegPath*** – the set of edges $(u\,v)$ from *NonContactEdges* with $u \in V_c$ or $v \in V_c$ (start/end of segment), used in building the set *SegPaths*, $InitSegPath = \{(u\ v) | (u\ v) \in NonContactEdges, (v \in V_c) \vee (u \in V_c)\}$.

It should be noted that throughout the exposure and during performing the operations, it will be taken into account that $(u\ v) = (v\ u)$, the edge $(u\ v)$, if necessary, can also be interpreted as a chain $((u\ v))$, also “palindrome” chains will be considered equal: $((v_0\,v_1)(v_1\,v_2)(v_2\,v_3)(v_3\,v_4)) = ((v_4\,v_3)(v_3\,v_2)(v_2\,v_1)(v_1\,v_0))$.

The segments building algorithm ***Segment building*** includes the following stages:

i. Building sets: *NonContactEdges*, *SegEdges*, *InitSegPath*;
ii. Calling the ***add–edge*** function that builds the set of segments path, *SegPaths*;
iii. Building the set of all segments, $\mathscr{S}$.

The pseudocode of the algorithm is inserted below, Algorithm 4.1. Choosing the cycle of maximum length contributes to shrinking the number of iterations, which also depends on the number of edges included in E_c. After the last iteration $E_c = E$, thus, it is important to include at each iteration as many edges as possible in the updated E_c.

The recursive function ***add–edge*** represents a DFS achievement [12] over the set *NonContactEdges*, and builds segments–chains by choosing matching edges. The function uses the global variable V_c and is called with two parameters: ***Path*** and ***Remainder***. The function pseudocode is inserted in Algorithm 4.2.

When describing the function, the notation *Remainder(Path)* will be used that means the rest of the edges from *NonContactEdges* after removing all edges of the path *Path*, *Remainder* (nil) = *NonContact-Edges*;

Algorithm 4.1. ***Segment building***

```
    Segments building algorithm

0. $SegEdges := \varnothing$; $SegPaths := \varnothing$; $InitSegPaths := \varnothing$;
   $NonContactEdges := E \setminus E_c$;
1. for all $e = (u\ v) \in NonContactEdges$ do
       if $u \in V_c$ then if $v \in V_c$ then $SegEdges := SegEdges \cup \{(u\ v)\}$;
                                  else $InitSegPath := InitSegPath \cup (v\ u)$;
                 else if $v \in V_c$ then $InitSegPath := InitSegPath \cup (u\ v)$;
2. end for all $e$
3. $NonContactEdges := NonContactEdges \setminus SegEdges$;
4. for all $e = (u\ v) \in InitSegPath$ do
       call add–edge($(u\ v)$, $NonContactEdges \setminus \{(u\ v), (v\ u)\}$);
5. end for all $e$
       /* the set of segments–paths $SegPaths$ is build */
6. $\mathscr{S} := SegEdges \cup SegPaths$;
7. end
```

Algorithm 4.2. ***Function "add–edge"***

```
      add–edge(Path,Remainder)
/* $Path = ((v\ v_1)(v_1\ v_2) \ldots (v_{n-1}\ v_n))$ */

1. for all $r = (w\ z) \in Remainder$ do
2.     $rv := (z\ w)$; $p_1 := r \| path$; $p_2 := rv \| path$;
3.     if $((v = z) \wedge (p_1$ – is a simple path$)$ then
          if $w \in V_c$ then $SegPath := SegPath \cup \{p_1\}$;
                   else call add–edge ($r \|$Path,Remainder$\setminus \{r\}$);
                                              else
       if $((v = w) \wedge (p_2$ – is a simple path$)$ then
          if $z \in V_c$ then $SegPath := SegPath \cup \{p_2\}$;
                   else call add–edge ($rv \|$Path,Remainder$\setminus \{rv\}$);
4. end for all $r$;
5. return $SegPath$;
6. end
```

Theorem 4.1. (Segment buiding)
Algorithm 4.1 correctly builds the set of all segments of the graph $G = (V, E)$ with respect to the subgraph $G_c = (V_c, E_c)$.

Proof.
1) At step 5, traversing all edges from *NonContactEdges*, all segments–edges (the set of *SegEdges*) and the set of all end-edges of segments–path (*InitSegPath*) are built.
2) Segment–paths are built by calling the ***add–edge*** function. By construction, all final paths will be segment–paths. By performing a DFS traversal over the set *NonContactEdges*, the function will exhaustively examine all edges in *NonContactEdges* by matching them to chain extensions. Each edge from *NonContactEdges* will be included in at least one segment–path. Otherwise, the graph G would not be biconnected. ■

Example 4.1. Graph G=(V,E) from Fig. 1.1

```
V = {(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14},
E = {(13 14),(11 12),(10 13),(10 12),(9 13),(8 12),(7 8),(6 13),(6 12),
        (5 14),(5 8),(4 13),(4 9),(4 7),(3 14),(3 5),(2 13),(2 12),(2 6),
        (1 11),(1 10),(1 5),(0 14),(0 5),(0 3)}
A fundamental cycle of maximum length is chosen, which will constitut
the contact nodes, for example:
Vc = (5 1 10 12 2 6 13 4 7 8 5),
Ec = {(5 1),(1 10),(10 12),(12 2),(2 6),(6 13),(13 4),(4 7),(7 8) (8 5)}.
NonContactEdges = {(0 3),(0 5),(0 14),(1 11),(2 13),(3 5),(3 14),(4 9),
        (5 14),(6 12),(8 12),(9 13),(10 13),(11 12),(13 14)})
Segments–edges:
SegEdges = {(10 13),(8 12),(6 12),(2 13)}
Non-contact edges after removing segments–edges::
NonContactEdges={(13 14),(11 12),(9 13),(5 14),(4 9),(3 14),(3 5),
        (1 11),(0 14),(0 5),(0 3)}
InitSegPath = {(14 13),(11 12),(9 13),(14 5),(9 4),(3 5),(11 1),(0 5)}
The segments-paths builded:
SegPath = {((13 14)(14 5)),((13 9)(9 4)),((13 14)(14 0)(0 3)(3 5)),
        ((13 14)(14 3)(3 5)),((12 11)(11 1)),((13 14)(14 3)(3 0)(0 5)),
        ((13 14)(14 0)(0 5))}
Segments–paths represented by nodes:
SegPath = {(13 14 5),(13 9 4),(13 14 0 3 5),(13 14 3 5),(12 11 1),
        (13 14 3 0 5),(13 14 0 5)}
```

Path	**Active Edges**	**NewPath**	**Property**
((14 13))	(0 14) (5 14) (3 14)	((0 14)(14 13)) ((5 14)(14 13)) ((3 14)(14 13))	active path α–path active path
((14 5))	(0 14) (13 14)) (3 14)	((0 14)(14 5)) ((13 14)(14 5)) ((3 14)(14 5))	active path palindrome active path
((9 13))	(4 9)	((4 9)(9 13))	α–path
((9 4))	(9 13)	((13 9)(9 4))	palindrome
((3 5))	(0 3) (3 14)	((0 3)(3 5)) ((14 3)(3 5))	active path active path
((11 12))	(1 11)	((1 11)(11 12))	α–path
((11 1))	(12 11)	((12 11)(11 1))	palindrome
((0 5))	(0 3) (0 14)	((3 0)(0 5)) ((14 0)(0 5))	active path active path
((0 14)(14 13))	(0 3) (0 5)	((3 0)(0 14)(14 13)) ((5 0)(0 14)(14 13))	active path α–path
((3 14)(14 13))	(0 3) (3 5)	((0 3)(3 14)(14 13)) ((5 3)(3 14)(14 13))	active path α–path
((0 14)(14 5))	(0 3)	((3 0)(0 14)(14 5))	deadlock
((3 14)(14 5))	(0 3)	((0 3)(3 14)(14 5))	deadlock
((0 3)(3 5))	((0 14))	((14 0)(0 3)(3 5))	active path
((14 3)(3 5))	(0 14) (13 14)	((0 14)(14 3)(3 5) ((13 14)(14 3)(3 5))	deadlock palindrome
((3 0)(0 5))	(3 14)	((14 3)(3 0)(0 5))	active path
((14 0)(0 5))	(3 14) (13 14)	((3 14)(14 0)(0 5)) ((13 14)(14 0)(0 5))	deadlock palindrome
((3 0)(0 14)(14 13))	(3 5)	((5 3)(3 0)(0 14)(14 13))	α–path
((0 3)(3 14)(14 13))	((0 5)	((5 0)(0 3)(3 14)(14 13))	α–path
((14 0)(0 3)(3 5))	(13 14)	(13 14)(14 0)(0 3)(3 5))	palindrome
((14 3)(3 0)(0 5))	(13 14)	(13 14)(14 3)(3 0)(0 5))	palindrome

$\mathscr{S} = \mathit{SegEdges} \cup \mathit{SegmentsPaths} = \{(10\ 13), (8\ 12), (6\ 12), (2\ 13), (5\ 14\ 13), (4\ 9\ 13), (1\ 11\ 12), (5\ 0\ 14\ 13), (5\ 3\ 14\ 13), (5\ 3\ 0\ 14\ 13), (5\ 0\ 3\ 14\ 13)\}$

Figure 4.1. Final add–edge results for Example 4.1

Path represents a simple path $((v_0\ v_1)(v_1\ v_2)\dots(v_{n-1}\ v_n))$ with the properties:
(1) $\{(v_0\ v_1),(v_1\ v_2),\dots,(v_{n-1}\ v_n)\}\subseteq$*NonContactEdges*,
(2) $v_n \in V_c$,
(3) $\{(v_0\ v_1),(v_1\ v_2),\dots,(v_{n-1}\ v_n)\}\cup$ *Remainder*=*NonContactEdges.*
Initial $n=1$, *Path*=$((v_0\ v_1))$, *Remainder* =
NonContactEdges$\setminus\{(v_0\ v_1)\}$.

The execution of the algorithm for the graph drawn in Fig. 1.1 is presented in Fig. 4.1. Final results of the ***add–edge*** calls are shown in Fig. 4.1. The edges that can cause the extension of the *Path* have been noted by *Active edges.*

5 Segments embedding

Segments embedding (the pseudocode of the "***Segments embedding***" algorithm is inserted in Algorithm 5.1) is an iterative process, which at each iteration embeds (draws) one segment. At the same time, the planarity of the graph is checked and the sets $\mathscr{F}$, $\mathscr{S}$, V_c, and E_c are modified. Since each iteration builds a new face, a counter ***iter*** will be used to keep track of the faces and iterations. Thus, each new face will be included in the set $\mathscr{F}$ with the order number ***iter***. The process lasts as long as the set $\mathscr{S}$ is not empty. Finally, if the graph is planar, it will be drawn completely; otherwise, a planar subgraph will be drawn, and the set of non–embeddable segments (which generate edge intersections) will be printed.

Initially, the process starts with the subgraph $G_c=(V_c,E_c)$, two faces – internal face $f_1=V_c$, external face $f_0=V_c$, *iter*=1, and the set of segments $\mathscr{S}$, built by Algorithm 4.1. It should be mentioned that f_0 will denote the external face throughout the execution of the algorithm. The algorithm ***Segments embedding*** calls the algorithm ***Segments updating*** and updates the set of segments $\mathscr{S}$.

For all segments $s\in\mathscr{S}$, we denote by:
$\mathscr{F}(s)$ – the set of valid faces for s,

Algorithm 5.1. ***Segments embedding***

```
0.  Input values: $G_c$=($V_c$, $E_c$), $\mathscr{F}$={$f_0$, $f_1$}, $f_0$=$V_c$, $f_1$=$V_c$, $\mathscr{S}$=SEG.
1.  do while $\mathscr{S} \neq \varnothing$
2.  for all $s \in \mathscr{S}$ do $\mathscr{F}(s) := \{f| \ f \in \mathscr{F}, f$ is valid for $s\}$;end for all;
3.  $\mathscr{S}_1$:={$s$ | $|\mathscr{F}(s)| = 1$}; $\mathscr{S}_2$:={$s$ | $|\mathscr{F}(s)| \geq 2$}; $\mathscr{N}_p$:={$s$ | $|\mathscr{F}(s)|$=0};
4.  if ($\mathscr{S}_1$=$\varnothing$) $\wedge$ ($\mathscr{S}_1 = \varnothing$) $\wedge$ $\mathscr{N}_p \neq \varnothing$ then
      return ('The graph is not planar." , $\mathscr{F}$ , $\mathscr{N}_p$);
5.  if ($\mathscr{S}_1$=$\varnothing$) then $s = \{s| \ |s| = \max_{\mathscr{S}_2} |s|\}$ else $s = \{s| \ |s| = \max_{\mathscr{F}_1} |s|\}$;
          /* The longest admissible segment is chosen */
      $\mathscr{S}$:=$\mathscr{S} \setminus \{s\}$;
6.  if ($\mathscr{S}_1 \neq \varnothing$) then $\mathscr{F}(s) = \{f_i\}$ else $s \in \mathscr{S}_2$, $|\mathscr{F}(s)| \geq 2$, $f_i \in \mathscr{F}(s)$;
7.  $nf$:=$nf$+1;
8.  if $f_i = f_0$ then ($fout$, $fin$):=split–face–out($f_0$, $s$); $f_0$:=$fout$;
      $f_{nf}$:=$fin$; $\mathscr{F}$ := $\mathscr{F}$ $\cup$\{$f_{nf}$\};else
      ($fin1$, $fin2$):= call split–face–in($f_i$, $s$); $f_i$:=$fin1$;
      $f_{nf}$:=$fin2$; $\mathscr{F}$:=$\mathscr{F}$ $\cup$\{$f_{nf}$\};
9.  seg–update($s$, $\mathscr{S}$);
10. end do while
11. return "The graph is planar.", $\mathscr{F}$)
```

$\mathscr{S}_1$ – the set of all segments that have only one valid face,
$\mathscr{S}_2$ – the set of all segments that have 2 or more valid faces,
$\mathscr{N}_p$ – the set of all segments, for which there are no valid faces (segments that violate planarity).

The actual embedding is done by functions ***split–face–in(**f_i, s**)*** and ***split–face–out***(f_0, s), presented in Fig. 5.1.

If s=($v_0 v_1 \ldots v_{n-1} v_n$) then, in the case of the function ***split–face–in(**f_i, s**)***, the segment s will be drawn inside the face f_i, joining the contact vertex v_0 with the contact vertex v_n, {v_0, v_n}$\in f_i$, by a Jordan curve without self-intersections passing through all other vertices $v_1 \ldots v_{n-1}$ placed inside the face f_i. As a result, the face f_i will be divided into two faces – *fin1, fin2*, f_i:=*fin1*, f_{iter}:=*fin2*, (Fig. 5.1(a)), $\mathscr{F}$:=$\mathscr{F}$ $\cup$ {f_{iter}}.

In the case of the function ***split–face–out***(f_0, s), we proceed analogously, the Jordan curve being drawn outside the face f_0. Two faces will be obtained, one internal and one external, and two variants are possible (Fig. 5.1(b),(c)). The algorithm optionally chooses one of the variants. For example, the internal face with a shorter length.

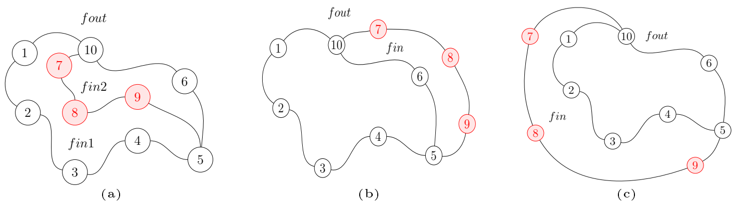


Figure 5.1. Face split schema

6 Segments update

Segments embedding generates new faces and new contact vertices. Thus, the need to modify the set of segments inevitably arises. Due to the fact that only two new faces appear at each iteration, only the segments that have certain tangents to these faces will undergo modification. More precisely, the segments inside which new contact vertices appeared must be modified.

Algorithm 6.1. ***Segments update***

```
        seg–update(s, $\mathscr{S}$)
1. Input values: $\mathscr{S}$, $s=(v_0\, v_1\, \ldots\, v_{n-1}\, v_n)$, $newcn=\{v_1, v_2, \ldots, v_{n-1}\}$.
2. if $newcn=\varnothing$ then return ($\mathscr{S}$)
3. $V_c := V_c \cup newcn$, $E_c := E_c \cup \{(v_0\, v_1), (v_1, v_2), \ldots, (v_{n-1}, v_n)\}$.
4. for all $s \in \mathscr{S}$ do
5. for all $v_c \in newcn$ do
     if $s=(v_0\, v_1\, \ldots\, v_i\, v_c\, v_{i+1}\, \ldots\, v_{n-1}\, v_n)$ then
     $\mathscr{S} := (\mathscr{S} \setminus \{s\}) \cup \{(v_0\, v_1\, \ldots\, v_i\, v_c), (v_c\, v_{i+1}\, \ldots\, v_{n-1}\, v_n)\}$
6. end for all $v_c$;
7. end for all $s$;
8. simplify $\mathscr{S}$: remove duplicates, segments–edges included in $E_c$,
     palindromes.
9. return($\mathscr{S}$)
```

Obviously, the set of valid faces for these segments will also be changed. The idea behind the change is simple. Any segment containing at least one new contact vertex will be split into two new segments. For example, if the segment $s=(v_0\, v_1\, \ldots v_i\, v_c\, v_{i+1}\, \ldots\, v_{n-1}\, v_n)$, where v_0 and v_n are old contact vertices and v_c – new contact vertex, then s will split into two new segments: $s_1 = (v_0\, v_1\, \ldots\, v_i\, v_c)$ and

$s_2=(v_c\, v_{i+1}\, \dots\, v_{n-1}\, v_n)$. This algorithm will be called after each embedding, which generates new contact vertices. We denote by *newcn* the set of new contact vertices. If $newcn=\varnothing$, then new segments will not be generated. The pseudocode of the algorithm is presented in Algorithm 6.1.

Below, the results of applying the algorithms ***"Segments embedding"*** and ***"Segment update"*** for Example 4.1 are inserted in Fugure 6.1. The final result for this example is inserted in Fig. 7.1.

7 Modified DMP algorithm

The modified *DMP* algorithm (*MDMP*) is obtained by assembling all the algorithms presented above, including the biconnectivity verification. Below, the pseudocode of the algorithm (Algorithm 7.1) is inserted.

Algorithm 7.1. *Modified DMP algorithm*

```
1. Let $G = (V, E)$ be an arbitrary undirected graph. The algorithm exposed in [13] is applied for checking biconnectivity and building the fundamental set of cycles.
2. If the graph is biconnected, a fundamental cycle of maximum length is chosen, subgraph of the graph $G$ is denoted by $G_c=(V_c, E_c)$. Otherwise return "The graph is not biconnected".
3. if $G_c=G$ then return "The graph is planar".
4. Initial assignments:
   $\mathscr{F}:=\{f_0, f_1\}$; $f_0:=V_c$; $f_1:=V_c$; $\mathscr{S}:=\varnothing$; $\mathscr{N}_p:=\varnothing$; $iter:=1$;
5. The "Segments building" algorithm is called, which builds
   the set of all segments $\mathscr{S}$.
6. do while $\mathscr{S} \neq \varnothing$
7.     $iter:=iter+1$;
8.     for all $s \in \mathscr{S}$ do $\mathscr{F}(s):=\{f \mid f \in \mathscr{F}, f$ is valid for $s\}$; end for all.
9.     The "Segments embedding" algorithm is called, which checks the
       non-planarity of the graph, embeds a segment, modifies the set of
       faces $\mathscr{F}$, and updates the set of segments $\mathscr{S}$ by calling the algo-
       rithm Segments update.
10. end do while
11. return ("The graph is planar", $\mathscr{F}$).
```

– ***initially***, until the ***"Segment embedding"*** algorithm is applied, we have:
V_c={5,1,10,12,2,6,13,4,7,8,5},
E_c={(5 1),(1 10),(10 12),(12 2),(2 6),(6 13),(13 4),(4 7),(7 8),(8 5)},
$\mathscr{F}=\{f_0,f_1\}$= {(5 1 10 12 2 6 13 4 7 8 5), (5 1 10 12 2 6 13 4 7 8 5)},
$\mathscr{S}$={(13 14 3 0 5) (13 14 0 3 5), (13 14 0 5),(13 14 3 5),(12 11 1), (13 9 4), (13 14 5), (2 13),(6 12),(8 12),(10 13)}.

– ***segments embedding***

At the first embedding, ***split-face-in***(s, f_1) will be executed, where
s=(13 14 3 0 5), f_1=(5 1 10 12 2 6 13 4 7 8 5), *iter*=2. Two new faces are obtained: *fin1*=(5 0 3 14 13 4 7 8 5), *fin2*=(13 6 2 12 10 1 5 0 3 14 13). Face *fin1* will replace f_1, and the *fin2* face will be added to $\mathscr{F}$ as f_2.
Thus, we get $\mathscr{F}=\{f_0,f_1,f_2\}$= {(5 1 10 12 2 6 13 4 7 8 5),(5 0 3 14 13 4 7 8 5), (13 6 2 12 10 1 5 0 3 14 13)}, *newcn*={14 3 0}, changes for V_c and E_c:
V_c={5,1,10,12,2,6,13,4,7,8,5,14,3,0},
E_c={(5 1),(1 10),(10 12),(12 2),(2 6),(6 13),(13 4),(4 7),(7 8), (8 5),(13 14), (14 3),(3 0),(0 5)}.

– ***segments update***

$\mathscr{S}$={(13 14 0 3 5),(13 14 0 5),(13 14 3 5),(12 11 1), (13 9 4), (13 14 5),(2 13), (6 12),(8 12)(10 13)},
s=(13 14 3 0 5), *newcn*={14,3,0}.

– ***segments splitting***

For v_c=14:
(13 14 0 3 5) will be splitted into: (13 14) and (14 0 3 5),
(13 14 0 5) will be splitted into: (13 14) and (14 0 5),
(13 14 3 5) will be splitted into: (13 14) and (14 3 5),
(13 14 5) will be splitted into: (13 14) and (14 5).

We get the updated set of segments
$\mathscr{S}$={((13 14),(14 0 3 5),(14 0 5),(14 3 5),(12 11 1),(13 9 4),(14 5),(2 13),(6 12), (8 12),(10 13)} (duplicate segments have been removed).

For v_c=3:
(14 0 3 5) will be splitted into: (14 0 3) and (3 5),
(14 3 5) will be splitted into: (14 3) and (3 5).

We get the updated set of segments
$\mathscr{S}$={(13 14),(14 0 3),(3 5),(14 0 5),(14 3),(12 11 1),(13 9 4),(14 5),(2 13),(6 12), (8 12),(10 13)} (duplicate segments have been removed).

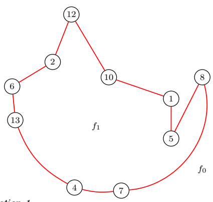
For v_c=0:
(14 0 3) will be splitted into: (14 0) and (0 3),
(14 0 5) will be splitted into: (14 0) and (0 5).

We get the updated set of segments
$\mathscr{S}$={(13 14),(14 0),(0 3),(3 5),(0 5),(14 3),(12 11 1),(13 9 4), (14 5),(2 13),(6 12), (8 12),(10 13)} (duplicate segments have been removed).

Finally, after elimination of palindromes and edges that have been included in the updated set E_c (for this example (13 14),(14 3),(0 3),(0 5)), we get the final set of updated segments
$\mathscr{S}$= {(10 13),(8 12),(6 12),(2 13),(14 5),(13 9 4),(12 11 1),(3 5),(14 0)}

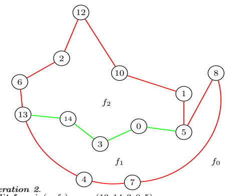
Note that segments–edges won't be able to generate new segments, because will be ignored calls to the ***"Segments update"*** algorithm.

Figure 6.1. Segments embedding and updatting for Example 4.1



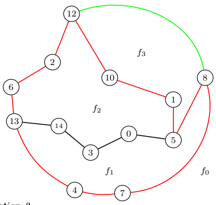
Iteration 1.
V_c =(5 1 10 12 2 6 13 4 7 8 5)=f_0=f_1,
$\mathscr{S}$={(13 14 3 0 5),(13 14 0 3 5),(13 14 0 5),(13 14 3 5),
(12 11 1),(13 9 4),(13 14 5),(2 13),(6 12),(8 12),(10 13))},
$\mathscr{F}$={f_0, f_1}

(a)



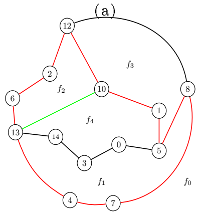
Iteration 2.
split-face-in(s, f_1), s = (13 14 3 0 5),
f_1=(5 1 10 12 2 6 13 4 7 8 5), $\mathscr{S}$={(13 14 3 0 5),
(13 14 0 3 5),(13 14 0 5),(13 14 3 5),(12 11 1),
(13 9 4),(13 14 5),(2 13), (6 12),(8 12),(10 13))},
$\mathscr{F}$={f_0, f_1, f_2}

(b)



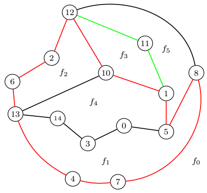
Iteration 3.
split-face-out(s, f_0), s = (8 12),
f_0=(8 5 1 10 12 2 6 13 4 7), $\mathscr{S}$={(8 12),(12 11 1),
(13 9 4),(14 0),(3 5),(14 5),(2 13),(6 12),(10 13))},
$\mathscr{F}$={f_0, f_1, f_2, f_3}

(c)



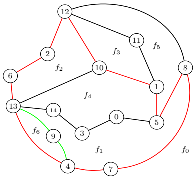
Iteration 4.
split-face-in(s, f_1), s = (10 13),
f_1=(5 1 10 12 2 6 13 14 3 0 5), $\mathscr{S}$={(10 13)(12 11 1)
(13 9 4)(14 0)(3 5)(14 5)(2 13)(6 12)}, $\mathscr{F}$={f_0, f_1, f_2, f_3, f_4}

(d)



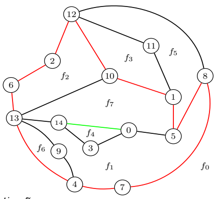
Iteration 5.
split-face-in(s, f_3), s = (12 11 1),
f_3=(8 12 10 1 5 8) $\mathscr{S}$={(12 11 1),(13 9 4),(14 0),(3 5),
(14 5),(2 13),(6 12)}, $\mathscr{F}$={$f_0, f_1, f_2, f_3, f_4, f_5$}

(e)



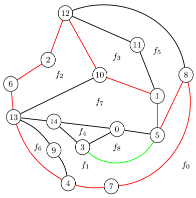
Iteration 6.
split-face-in(s, f_3), s = (13 9 4),
f_2=(13 14 3 0 5 8 7 4 13), $\mathscr{S}$={(13 9 4),(14 0),(3 5),(14 5),
(2 13),(6 12)}, $\mathscr{F}$={$f_0, f_1, f_2, f_3, f_4, f_5, f_6$}

(f)



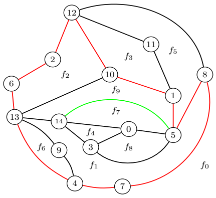
Iteration 7.
split-face-in(s, f_3), s = (14 0),
f_4=(13 14 3 0 5 1 10 13),
$\mathscr{S}$={(14 0),(3 5),(14 5),(2 13),(6 12)},
$\mathscr{F}$={$f_0, f_1, f_2, f_3, f_4, f_5, f_6, f_7$}

(g)



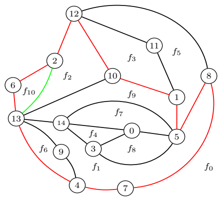
Iteration 8.
split-face-in(s, f_1), s = (3 5),
f_1=(13 9 4 7 8 5 0 3 14 13), $\mathscr{S}$={(3 5),(14 5),(2 13),(6 12)},
$\mathscr{F}$={$f_0, f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8$}

(h)



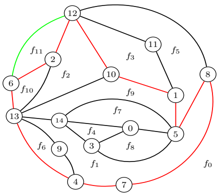
Iteration 9.
split-face-in(s, f_7), s = (14 5),
f_7=(0 5 1 10 13 14 0), $\mathscr{S}$={(14 5),(2 13),(6 12)},
$\mathscr{F}$={$f_0, f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8, f_9$}

(i)



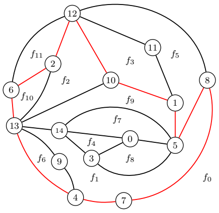
Iteration 10.
split-face-in(s, f_2), s = (2 13),
f_2=(10 13 6 2 12 10), $\mathscr{S}$={(2 13),(6 12)},
$\mathscr{F}$={$f_0, f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8, f_9, f_{10}$}

(j)



Iteration 11.
split-face-out(s, f_0), s = (6 12),
f_0=(12 2 6 13 4 7 8 12), $\mathscr{S}$={(6 12)},
$\mathscr{F}$={$f_0, f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8, f_9, f_{10}, f_{11}$}

(k)



The graph is planar.
$\mathscr{S}$={}, $\mathscr{F}$={$f_0, f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8, f_9, f_{10}, f_{11}$}

(l)

Figure 7.1. Final result for Example 4.1

8 Examples. The case of non-planar graph

Example 8.1. (*[9], p.177)*

We will apply the elaborated algorithm for the example from [9] (Fig. 8.1(a)).

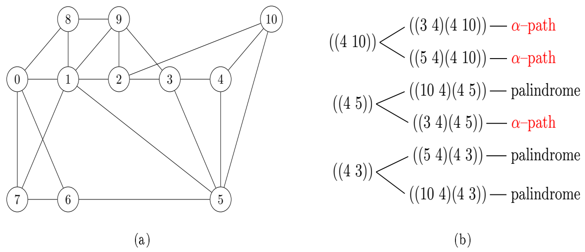
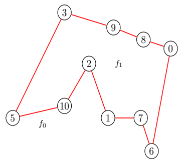


Figure 8.1. Segments embedding for Example 8.1

Initially, we select the fundamental cycle of maximal length V_c=(0 8 9 3 5 10 2 1 7 6 0), Fig. 8.1(a), and the initial faces f_0=f_1=V_c, $\mathscr{F}$={f_0, f_1}. As a result of applying the algorithm ***Segments building***, we will get *SegmentEdges*={(0 1)(0 7)(1 5)(1 8)(1 9) (2 3)(2 9)(5 6)} and *InitSegPath*={(4 3)(4 5)(4 10)}. After applying the function ***Add–edge***, we obtain *SegmentPath*={(3 4 5)(5 4 10)(3 4 10)}. The calling scheme of ***Add–edge*** is presented in Fig. 8.1b.

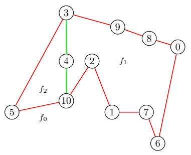
The drawing steps are shown in Fig. 8.2. It should be noted that the presence of a large number of segments–edges actually speeds up the drawing process, because these segments do not generate new contact vertices, thus ignoring calls to the expensive algorithm ***"Segments update"***. The red edges will show the initial cycle, the green ones – the segment currently drawn, and the black ones – all other edges. It should be noted that all coordinates of the vertices were defined manually. An automatically drawn variant applying the method of circular orbits is presented in Fig. 8.2(l).

Example 8.2. (*[9], p.182)*



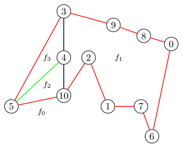
Iteration 1. V_c=(0 8 9 3 5 10 2 1 7 6 0), $\mathscr{F}$={f_0, f_1}, $\mathscr{S}$= {(3 4 10),(5 4 10),(3 4 5), (5 6),(2 9),(2 3),(1 9),(1 8),(1 5),(0 7),(0 1)}.

(a)



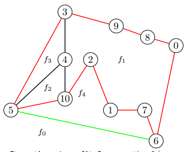
Iteration 2. split-face-in(s, f_1), s=(3 4 10), f(1)=(0 8 9 3 5 10 2 1 7 6 0), $\mathscr{S}$={(4 5),(5 6),(1 5),(2 9),(2 3),(1 9), (1 8),(0 7),(0 1)}, $\mathscr{F}$={f_0, f_1, f_2}

(b)



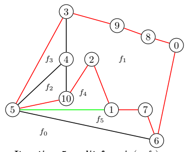
Iteration 3. split-face-in(s, f_2), s = (4 5), f_2=(10 5 3 4 10), $\mathscr{S}$={(5 6),(1 5),(2 9),(2 3), (1 9),(1 8),(0 7),(0 1)}, $\mathscr{F}$={f_0, f_1, f_2, f_3}

(c)



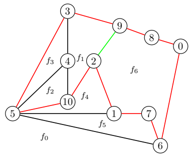
Iteration 4. split-face-out(s, f_0), s = (5 6), f(0)=(0 8 9 3 5 10 2 1 7 6 0), $\mathscr{S}$= {(1 5),(2 9),(2 3),(1 9),(1 8),(0 7),(0 1)}, $\mathscr{F}$={f_0, f_1, f_2, f_3, f_4}

(d)



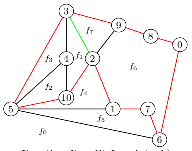
Iteration 5. split-face-in(s, f_4), s = (1 5), f_4=(5 6 7 1 2 10 5), $\mathscr{S}$= {(2 9),(2 3),(1 9),(1 8),(0 7),(0 1)}, $\mathscr{F}$={$f_0, f_1, f_2, f_3, f_4, f_5$}

(e)



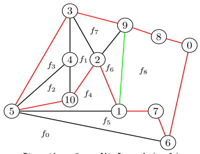
Iteration 6. split-face-in(s, f_1), s = (2 9), f(1)=(3 4 10 2 1 7 6 0 8 9 3), $\mathscr{S}$= {(2 3),(1 9),(1 8),(0 7),(0 1)}, $\mathscr{F}$={$f_0, f_1, f_2, f_3, f_4, f_5, f_6$}

(f)



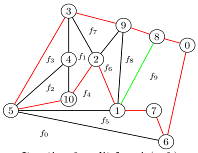
Iteration 7. split-face-in(s, f_1), s = (2 3), f_1=(2 9 3 4 10 2), $\mathscr{S}$= {(1 9),(1 8),(0 7),(0 1)}, $\mathscr{F}$={$f_0, f_1, f_2, f_3, f_4, f_5, f_6, f_7$}

(g)



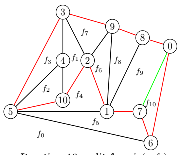
Iteration 8. split-face-in(s, f_6), s = (1 9), f_6=(9 8 0 6 7 1 2 9), $\mathscr{S}$= {(1 8),(0 7),(0 1)}, $\mathscr{F}$={$f_0, f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8$}

(h)



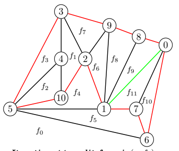
Iteration 9. split-face-in(s, f_8), s = (1 8), f_8=(1 7 6 0 8 9 1)), $\mathscr{S}$= {(0 7),(0 1)}, $\mathscr{F}$={$f_0, f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8, f_9$}

(i)



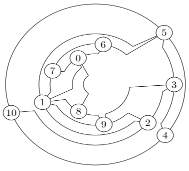
Iteration 10. split-face-in(s, f_9), s = (0 7), f_9=(8 0 6 7 1 8), $\mathscr{S}$={(0 1)}, $\mathscr{F}$={$f_0, f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8, f_9, f_{10}$}

(j)



Iteration 11. split-face-in(s, f_9), s = (0 1), f_9=(0 7 1 8 0) , $\mathscr{S}$={}, $\mathscr{F}$={$f_0, f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8, f_9, f_{10}, f_{11}$}

(k)



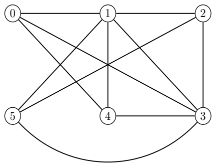
Automatically drawn by applying the method of circular orbits

(l)

Figure 8.2. Segments embedding, the example from [9], p.177

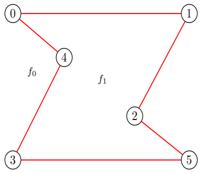
The second example inserts the solution for the graph presented in [9], p. 182. For this example, choosing the maximum length cycle led to the construction of the set of segments $\mathscr{S}$, which contains only segments–edges. This fact essentially simplifies drawing the graph, because the need for recalculation of the set $\mathscr{S}$ disappears, which ignores calls of the algorithm ***Segments update*** (building new segments/fragments).

The solution of the example is laid out step by step in Fig. 8.3.



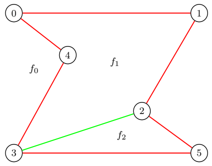
Initial graph, $G = (V\ E)$, V=(0 1 2 3 4 5), E={(0 1)(0 3)(0 4) (1 2)(1 3)(1 4)(1 5)(2 3) (2 5)(3 4)(3 5)}

(a)



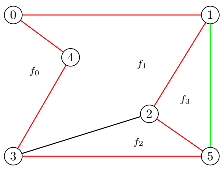
Iteration 1. V_c=(0 1 2 5 3 4 0), $\mathscr{F}$={f_0, f_1}, $\mathscr{S}$={(0 3)(1 3)(1 4)(1 5)(2 3)}

(b)



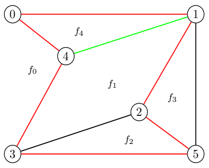
Iteration 2. split-face-in(s, f_1), s=(2 3), $f(1)$=(0 1 2 5 3 4 0), $\mathscr{S}$={(0 3)(1 3)(1 4)(1 5)}, $\mathscr{F}$={f_0, f_1, f_2}

(c)



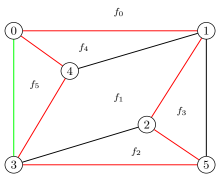
Iteration 3. split-face-out(s, f_0), s=(1 5), $f(0)$=(0 1 2 5 3 4 0), $\mathscr{S}$= {(0 3)(1 3)(1 4)}, $\mathscr{F}$={f_0, f_1, f_2, f_3}

(d)



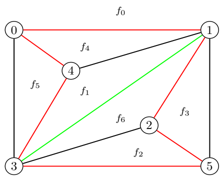
Iteration 4. split-face-in(s, f_1), s=(1 4), $f(1)$=(2 3 4 0 1 2), $\mathscr{S}$= {(0 3)(1 3)}, $\mathscr{F}$={f_0, f_1, f_2, f_3, f_4}

(e)



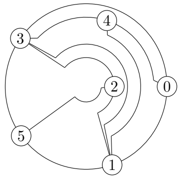
Iteration 5. split-face-out(s, f_0), s=(0 3), $f(0)$=(1 5 3 4 0 1), $\mathscr{S}$= {(1 3)}, $\mathscr{F}$={$f_0, f_1, f_2, f_3, f_4, f_5$}

(f)



Iteration 6. split-face-in(s, f_1), s=(1 3), $f(1)$=(1 4 3 2 1), $\mathscr{S}$=∅, $\mathscr{F}$={$f_0, f_1, f_2, f_3, f_4, f_5, f_6$}

(g)



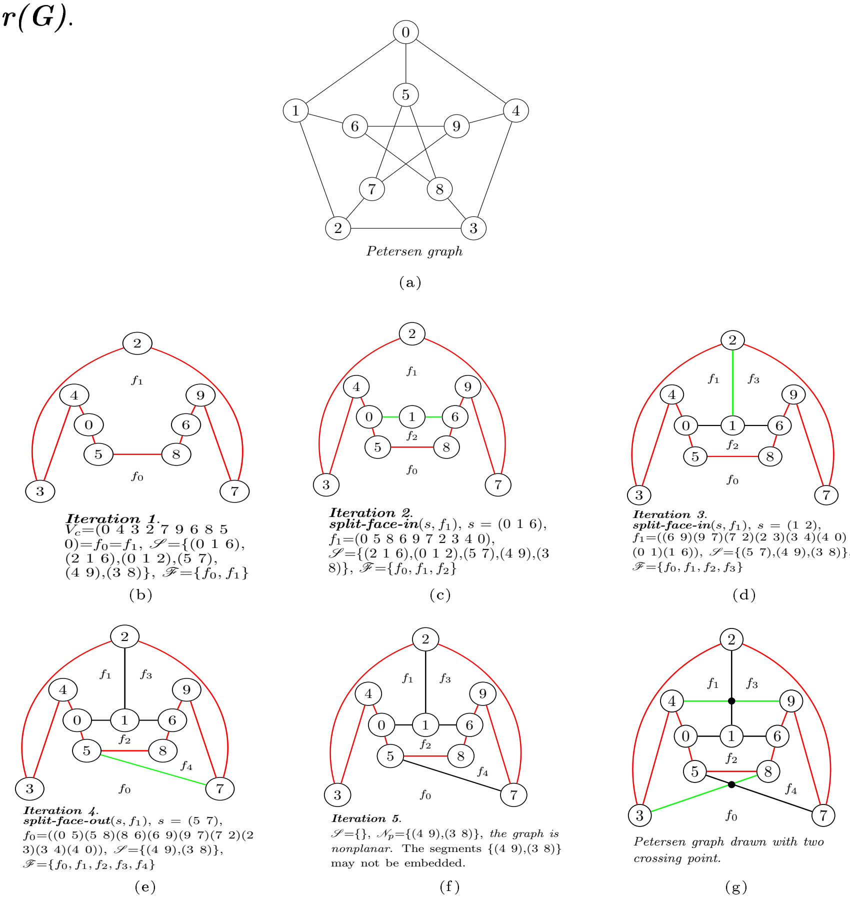
Automatically drawn by applying the method of circular orbits

(h)

Figure 8.3. Segments embedding, the example from [9], p.182

Example 8.3. *Petersen graph*

The next example concerns the Petersen graph 8.4a, which is not planar, as is known. The developed algorithm generates a planar subgraph, included in Fig. 8.4f, and will return the list of segments that could not be embedded: (3 8) and (4 9). In Fig. 8.4(g), we insert a variant of the Peterson graph with two intersection points, which has the minimal number of intersections for this graph, noted in the theory of graphs by ***cr(G)***.



Iteration 1. V_c=(0 4 3 2 7 9 6 8 5 0)=f_0=f_1, $\mathscr{S}$={(0 1 6), (2 1 6),(0 1 2),(5 7), (4 9),(3 8)}, $\mathscr{F}=\{f_0, f_1\}$

Iteration 2. ***split-face-in***(s, f_1), s = (0 1 6), f_1=(0 5 8 6 9 7 2 3 4 0), $\mathscr{S}$={(2 1 6),(0 1 2),(5 7),(4 9),(3 8)}, $\mathscr{F}=\{f_0, f_1, f_2\}$

Iteration 3. ***split-face-in***(s, f_1), s = (1 2), f_1=((6 9)(9 7)(7 2)(2 3)(3 4)(4 0) (0 1)(1 6)), $\mathscr{S}$={(5 7),(4 9),(3 8)}, $\mathscr{F}=\{f_0, f_1, f_2, f_3\}$

Iteration 4. ***split-face-out***(s, f_1), s = (5 7), f_0=((0 5)(5 8)(8 6)(6 9)(9 7)(7 2)(2 3)(3 4)(4 0)), $\mathscr{S}$={(4 9),(3 8)}, $\mathscr{F}=\{f_0, f_1, f_2, f_3, f_4\}$

Iteration 5. $\mathscr{S}$={}, $\mathscr{N}_p$={(4 9),(3 8)}, *the graph is nonplanar.* The segments {(4 9),(3 8)} may not be embedded.

Petersen graph drawn with two crossing point.

Figure 8.4. Segments embedding for Petersen graph

Example 8.4. *Tutte graph*

We will apply the algorithm to check the planarity of the Tutte graph [13], [14] presented in Fig. 8.5a, which contains 46 vertices and 69 edges. The algorithm will build the fundamental set of cycles (24 cycles) and select a cycle of maximal length equal to 27. As a result of the verification, 25 faces will be obtained. The variant drawn by the method of circular orbits is presented in Fig. 8.5(b).

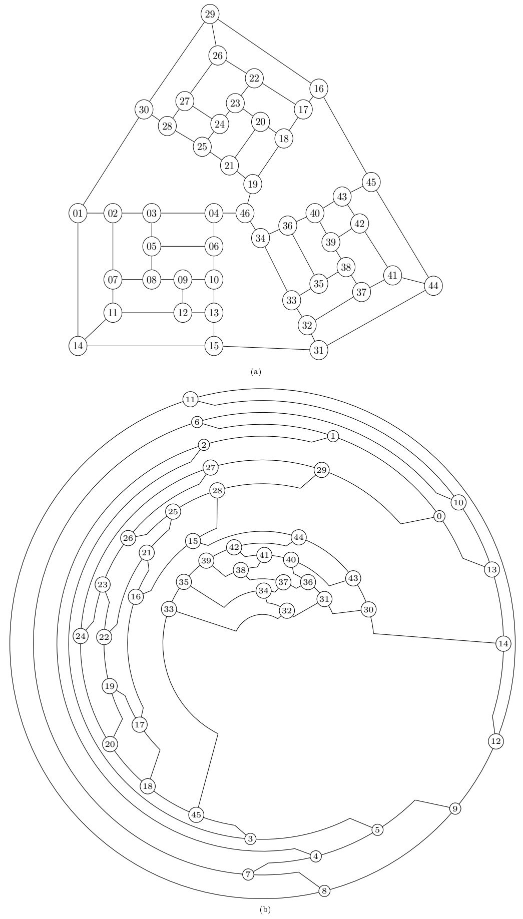


Figure 8.5. Tutte graph

9 Conclusions

The paper presents a complete and detailed realization of the DMP algorithm with all necessary explanations, the exposition in detail (pseudocodes, programs) of the algorithms, and many examples for testing. Namely, this moment was influenced by the title of the article "poor man's realization". The algorithm can be used for testing planarity of graphs, generation of planar graphs, and elaboration of automatic graph drawing methods. It is easy to develop an algorithm, which for the set of constructed faces $\mathscr{F}$ and any segment from $\mathscr{N}_p$, will allow us to find the minimum number of intersections needed to draw this segment. This algorithm, of course, does not solve the crossing number problem, which is known to be NP-complete [15], but can be used to build a plausible drawing version; for example, the version of the Petersen graph, presented in Fig. 8.4(g).

Acknowledgments. The project SIBIA 011301 has supported part of the research for this paper.

References

[1] *Handbook of Graph Drawing and Visualization*, R. Tamassia, Ed. Chapman & Hall, 2013, 862 p. DOI: `https://doi.org/10.1201/b15385`.

[2] K. Kuratowski, "On the problem of skew curves in topology," *Fund. Math.*, vol. 15, no. 1, pp. 271–283, 1930. (in French)

[3] K. Wagner, "On a property of plane complexes," *Math. Ann.*, vol. 114, pp. 570–590, 1937, DOI: 10.1007/BF01594196. (in German)

[4] G. Demoucron, Y. Malgrange, R. Pertuiset, "Planar Graphs: Recognition and Construction of Topological Planar Representations," *Revue Française de Rech. Oper.*, vol. 8, pp. 33–47, 1964. (in French)

[5] A. Kohnert, "Algorithm of Demoucron, Malgrange, Pertuiset," 2004, [Online]. Available: `http://www.mathe2.uni-bayreuth.de/EWS/demoucron.pdf`.

[6] A. Gibbons, *Algorithmic Graph Theory*, Cambridge: Cambridge University Press, 1985, 259p.

[7] J.A. Bondy and U. S. R. Murty, *Graph Theory with Applications*, New York, USA: Elsevier Science Publishing Co., Inc., 1976, 270 p.

[8] D. B. West, *Introduction to Graph Theory*, Subsequent ed., Pearson College Div, 2000, 588 p.

[9] V. A. Yemelichev, O. I. Melnikov, V. I. Sarvanov, R. I. Tyshkevich, *Lectures on graph theory*, Nauka, 1990, 382 p. (in Russian). (English translation: O. Melnikov, R. Tyshkevich, V. Yemelichev, V. Sarvanov, *Lectures on graph theory*, Mannheim-Leipzig-Wein-Zurich: Wissenschaftsverlag, 1994, 361 p.)

[10] W. J. Myrvold and W. Kocay, "Errors in graph embedding algorithms," *J. Comput. Syst. Sci.*, vol. 77, no. 2, 2011, pp. 430–438.

[11] C. Berge, *Graph theory and its applications*, Paris: DUNOD, 1958, 275 p. (in French)

[12] C. Ciubotaru, "The modified deep first search algorithm: functional implementation", *Computer Science Journal of Moldova*, vol. 33, no.1(97), pp. 129–140, 2025.

[13] "Tutte graph," Wikipedia. [Online]. Available: `https://en.wikipedia.org/wiki/Tutte_graph`

[14] "Tutte's graph," Wolfram MathWorld. [Online]. Available: `https://mathworld.wolfram.com/TuttesGraph.html`

[15] M. R. Garey and D. S. Johnson, *Computers and intractability. A Guide to the Theory of NP-Completeness,* (Series of Books in the Mathematical Sciences), 1st ed., San Francisco, Calif.: W. H. Freeman and Co.(ed.), 1979, 340 p.

Constantin Ciubotaru

Received March 06, 2025
Revised June 04, 2025
Accepted June 17, 2025

ORCID: `https://orcid.org/0009-0005-8896-0966`
Moldova State University,
Vladimir Andrunachievici Institute of Mathematics and Computer Science
E–mails: `constantin.ciubotaru@math.usm.md`, `chebotar@gmail.com`