

Fast 3-coloring Triangle-Free Planar Graphs*

Łukasz Kowalik[†]

Abstract

We show the first $o(n^2)$ algorithm for coloring vertices of triangle-free planar graphs using three colors. The time complexity of the algorithm is $\mathcal{O}(n \log n)$. Our approach can be also used to design $\mathcal{O}(n \text{ polylog } n)$ -time algorithms for two other similar coloring problems.

Keywords: graph algorithms, triangle-free planar graphs, Grötzsch's theorem, coloring, efficient algorithm

1 Introduction

The famous Four-Color Theorem says that every planar graph is vertex 4-colorable. The paper of Robertson et al. [2] describes an $\mathcal{O}(n^2)$ 4-coloring algorithm. This seems to be very hard to improve, since it would probably require a new proof of the 4-Color Theorem. On the other hand there are several linear 5-coloring algorithms (see e.g. [3]). Thus efficient coloring planar graphs using only three colors (if possible) seems to be the most interesting area still open for research. Although the general decision problem is NP-hard [4] the renowned Grötzsch's theorem [5] guarantees that every triangle-free planar graph is 3-colorable. It seems to be widely known that the simplest known proofs by Carsten Thomassen (see [6, 7]) can be easily transformed into $\mathcal{O}(n^2)$ algorithms. In this paper we improve this bound to $\mathcal{O}(n \log n)$.

*A preliminary version of this paper [1] was presented at ESA'2004

[†]Institute of Informatics, Warsaw University Banacha 2, 02-097 Warsaw, Poland. kowalik@mimuw.edu.pl. Phone: (+48 22) 55 44 422. Fax: (+48 22) 55 44 400. The research has been supported by KBN grant 4T11C04425. A part of the research was done during author's stay at BRICS, Aarhus University, Denmark.

In § 2 we present a new proof of Grötzsch’s theorem, based on a paper of Thomassen [6]. The proof is inductive and it corresponds to a recursive algorithm. The proof is written in a very special way thus it can be immediately transformed into the algorithm. In fact the proof can be treated as a description of the algorithm mixed with a proof of its correctness. In § 3 we discuss how to implement the algorithm efficiently. We describe details of non-trivial operations as well as data structures needed to perform these operations fast. The description and analysis of the most involved one, *Short Path Data Structure* (SPDS), is contained in § 4. This data structure deserves a separate interest. In the paper [8] we presented a data structure built in $\mathcal{O}(n)$ time and enabling finding shortest paths between given pairs of vertices in constant time, provided that the distance between the vertices is bounded. In our coloring algorithm we need to find paths only of length 1 and 2. Hence here we show a simplified version of the previous, general structure. The SPDS is described from the scratch in order to make this paper self-contained but also because we need to introduce some modifications and extensions, like insert and identify operations, not described in our previous paper [8]. We claim that techniques from the present paper can be extended to the general case making possible to use the general data structure in the fully dynamic environment. The description of this extension is beyond the scope of this paper and is intended to appear in the journal version of the paper [8].

Very recently, two new Grötzsch-like theorems appeared. The first one says that any planar graph without triangles at distance less than 4 and without 5-cycles is 3-colorable (see [9]). The other theorem states that planar graphs without cycles of length from 4 to 7 are 3-colorable (see [10]). We claim that our techniques can be adapted to transform these proofs to $\mathcal{O}(n \log^5 n)$ and $\mathcal{O}(n \log^7 n)$ algorithms respectively (a careful analysis would probably help to lower these bounds). However, we do not show it in the present paper, since it involves the general short path structure.

Terminology We assume the reader is familiar with standard terminology and notation concerning graph theory and planar graphs in particular (see e.g. [11]). Let us recall here some notions that are not so widely used. Let f be a face of a connected plane graph. A *facial walk* w corresponding to f is the shortest closed walk induced by all edges incident with f . If the boundary of f is a cycle the walk is called a *facial cycle*. The length of walk w is denoted by $|w|$. The length of face f is denoted by $|f|$ and equal $|w|$. Let C be a simple cycle in a plane graph G . The length of C will be denoted by $|C|$. The cycle C divides the plane into two disjoint open domains, D and E , such that D is homeomorphic

to an open disc. The set consisting of all vertices of G belonging to D and of all edges crossing this domain is denoted by $\text{int } C$. Observe that $\text{int } C$ is not necessarily a graph, while $C \cup \text{int } C$ is a subgraph of G . A k -path (k -cycle, k -face) refers to a path (cycle, face) of length k .

2 A Proof of Grötzsch's Theorem

In this section we give a new proof of Grötzsch's theorem. The proof is based on ideas of C. Thomassen [6]. The reason for writing the new proof is that the original one corresponds to an $\mathcal{O}(n \log^3 n)$ algorithm when we employ our algorithmic techniques presented in the following sections. In the algorithm corresponding to the proof presented below we don't need to search for paths of length 3 and 4, which reduces the time complexity to $\mathcal{O}(n \log n)$. We could also use the recent proof of Thomassen [7] but we suspect that the resulting algorithm would be more complicated and harder to describe. We will need a following lemma.

Lemma 2.1. *Let G be a biconnected plane graph with every inner face of length 5, and the outer face C of length $4 \leq |C| \leq 6$. Furthermore assume that every vertex not in $V(C)$ has degree at least 3 and that there is no pair of adjacent vertices of degree 2. Then G has a facial cycle C' such that $V(C') \cap V(C) = \emptyset$ and all the vertices of C' are of degree 3 except, possibly one of degree at most 5.*

Proof. We use the well-known discharging technique. We put a *charge* of $\deg_G(v) - 4$ on every vertex v of G . Moreover, each face q of G obtains a charge of $|q| - 4$. The outer face receives additional 7 units of charge. Let n, m, f denote the number of vertices, edges and faces of graph G , respectively and let V, F be the sets of vertices and faces of G , respectively. Using Euler's formula we can easily calculate the total charge on G :

$$\sum_{v \in V} (\deg_G(v) - 4) + \sum_{q \in F} (|q| - 4) + 7 = 2m - 4n + 2m - 4f + 7 = -1$$

Note that the total charge is negative. In the sequel we will show that we can redistribute charge in the graph, without changing its total amount, in such a way that if the graph contained no desired facial cycles then it would imply that the total charge is nonnegative, which is a contradiction.

We move the charge from vertices to faces in such a way that each vertex v sends $\frac{\deg_G(v) - 4}{\deg_G(v)}$ units of charge to every face incident with v . Let d_4 and d_5 denote the number of vertices of degree 4 and 5 in $V(C)$, respectively. Since

there are at most 3 vertices of degree 2 the outer face has got at least $|C| - 4 + 7 + 3 \cdot (-1) + 3 \cdot (-\frac{1}{3}) + d_4 \cdot \frac{1}{3} + d_5 \cdot (\frac{1}{3} + \frac{1}{5}) = |C| - 1 + d_4 \cdot \frac{1}{3} + d_5 \cdot \frac{8}{15}$ charge. All the faces have nonnegative charge except for, possibly, 5-faces with 4 vertices of degree 3 and one vertex of degree from 3 to 5 (charge at least $-\frac{2}{3}$) or 5-faces with a vertex of degree 2 (charge at least $-\frac{4}{3}$).

Then the outer face sends equal charge of $1 - \frac{1}{|C|} \geq \frac{3}{4}$ units to each of the edges in $E(C)$. Next each of these edges resends its charge to the other incident face (distinct from C). Observe that now all the faces having a common edge with the outer face have positive charge: faces with a vertex of degree 2 have got at least $-\frac{4}{3} + 2 \cdot \frac{3}{4} = \frac{1}{6}$ units of charge, while the other faces end up with $\geq -\frac{2}{3} + \frac{3}{4} = \frac{1}{12}$ units of charge.

Note that if there is now a face q of negative charge sharing a vertex x with C then $\deg x \in \{4, 5\}$ and the other vertices of q are not in C . The charge of each such face becomes nonnegative after moving $\frac{1}{3}$ of charge from the outer face when $\deg x = 4$ or $\frac{2}{15}$ of charge when $\deg x = 5$. As we move at most $d_4 \cdot \frac{1}{3} + d_5 \cdot 2 \cdot \frac{2}{15}$ units of charge from the outer face it also ends up with nonnegative charge. Now we see that if the desired face does not exist the total charge in graph is nonnegative, which is a contradiction. \square

Instead of proving Grötzsch's theorem it will be easier for us to show the following more general result. (Although we obtained it independently, let us note that it follows also from Theorem 5.3 in [12]). A 3-coloring of a cycle C is called *safe* if $|C| < 6$ or the sequence of successive colors on the cycle is neither $(1, 2, 3, 1, 2, 3)$ nor $(3, 2, 1, 3, 2, 1)$.

Theorem 2.2. *Any connected triangle-free plane graph G is 3-colorable. Moreover, if the boundary of the outer face of G is a cycle C of length at most 6 then any safe 3-coloring of $G[V(C)]$ can be extended to a 3-coloring of G .*

Proof. Either all vertices of graph G are uncolored or all vertices incident with the outer face are colored while all the other vertices of G are uncolored. In the first situation we will show that there is a 3-coloring of G . We assume that the latter situation can appear only if the outer face is bounded by a cycle of length at most 6. Moreover we assume that the coloring of C is safe and the induced graph $G[V(C)]$ is properly colored. Then our goal is to show that this coloring can be extended to a 3-coloring of the whole graph G .

The proof is by the induction on $|V(G)|$. We assume that G has at least one uncolored vertex, for otherwise there is nothing left to do. We are going to consider several cases. After each case we assume that none of the previously considered cases applies to G .

Case 1. G has an uncolored vertex x of degree at most 2. Then we can remove x and easily complete the proof by induction. If x is a cutvertex the induction is applied to each of the connected components of the resulting graph.

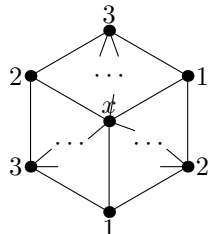


Fig. 1: Case 2.

Case 2. G has an uncolored vertex x joined to two or three colored vertices (see Fig. 1). Note that if x has 3 colored neighbors then $|C| = 6$ and the neighbors cannot have 3 different colors, as the coloring of C is safe. Thus we extend the 3-coloring of C to a 3-coloring of $G[V(C) \cup \{x\}]$. Note that for every facial cycle of $G[V(C) \cup \{x\}]$ the resulting 3-coloring is safe. Then we can apply induction to each face of $G[V(C) \cup \{x\}]$.

Case 3. C is colored and has a chord. We proceed similarly as in Case 2.

Case 4. G has a facial walk $C' = x_1x_2 \cdots x_kx_1$ such that $k \geq 6$ and at least 1 vertex of C' is uncolored. As Case 2 is excluded we can assume that x_1, x_2 are uncolored.

Case 4a. Assume that x_3 is colored and x_1 has a colored neighbor z (see Fig. 2). As Case 2 is excluded, x_2 and x_1 have no colored neighbors, except for x_3

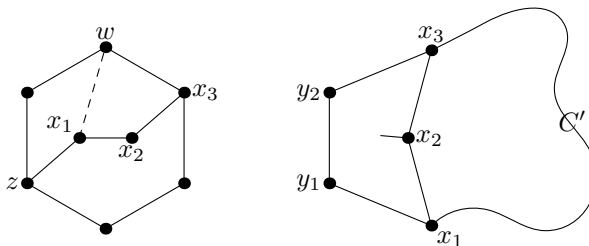


Fig. 2: Cases 4a (left) and 4b (right).

and z respectively. Then $G[V(C) \cup \{x_1, x_2\}]$ has precisely two inner faces, each of length at least 4. Let C_1 and C_2 denote the facial cycles corresponding to these faces and let $|C_1| \leq |C_2|$. We see that $|C_1| \leq 6$, because otherwise $|C| \geq 7$ and

C is not colored. We can assume that C_1 is separating for otherwise $C_1 = C'$, $|C_1| = 6$, $|C_2| = 6$ and C_2 is separating. Let $G' = G - \text{int}(C_1)$. Observe that if cycle C_1 is of length 6 we can add an edge to G' joining x_1 and the vertex w at distance 3 from x_1 in C_1 without creating a triangle. Then we can apply the induction hypothesis to G' . Note that the resulting 3-coloring of C_1 is safe because when $|C_1| = 6$ vertices x_1 and w are adjacent in G' . Moreover, as C_1 is chordless in G , it is also a 3-coloring of $G[V(C_1)]$ so we can use induction and extend the coloring to $C_1 \cup \text{int}(C_1)$.

Case 4b. There is a path $x_1y_1y_2x_3$ or $x_1y_1x_3$ distinct from $x_1x_2x_3$. Since G does not contain a triangle, $x_2 \notin \{y_1, y_2\}$. Let C'' be the cycle $x_3x_2x_1y_1y_2x_3$ or $x_3x_2x_1y_1x_3$ respectively. Let $G_1 = G - \text{int}(C'')$. Then $|V(G_1)| < |V(G)|$ because $\deg_G(x_2) \geq 3$. Using the induction hypothesis we get a 3-coloring of G_1 . (If the vertices of C are colored we extend this coloring to a 3-coloring of G_1 .) The resulting 3-coloring of C'' is also a safe 3-coloring of $G[V(C'')]$, since $|C''| \leq 5$ and C'' is chordless. Thus we can use induction to find a 3-coloring of $C'' \cup \text{int}(C'')$.

Case 4c. Since Cases 4a and 4b are excluded, we can identify x_1 and x_3 without creating a chord in C or a triangle in the resulting graph G' . Hence we can apply the induction hypothesis to G' .

Case 5. G has a facial cycle C' of length 4. Furthermore if C is colored assume that $C' \neq C$. Observe that C' has two opposite vertices u, v such that one of them is not colored and identifying u with v does not create an edge joining two colored vertices. For otherwise, there is a triangle in G or either of cases 2, 3 occurs.

Case 5a. There is a path uy_1y_2v , $y_1 \notin V(C')$ (see Fig. 3). Then $y_2 \notin V(C')$,

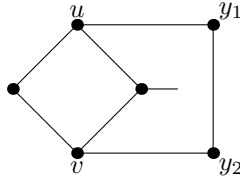


Fig. 3: Case 5a.

for otherwise there is a triangle in G . Then the path together with one of the two u, v -paths contained in C' creates a separating cycle C'' of length 4 or 5 respectively. Then we can apply induction as in Case 4b.

Case 5b. Since Case 5a is excluded, we can identify u and v without creating a triangle. Observe that when C was colored and $|C| = 6$ then the coloring of

the outer cycle remains safe. Hence it suffices to apply induction to the resulting graph.

Case 6 (main reduction). Observe that since inner faces of G have length 5 and G is triangle-free, G is biconnected. Moreover there is no pair of adjacent vertices of degree 2, for otherwise the 5-face containing this pair contains either a vertex joined to two colored vertices or a chord of C (Case 2 or 3 respectively). Hence by Lemma 2.1 there is a face $C' = x_1x_2x_3x_4x_5x_1$ in G such that $\deg(x_1) = \deg(x_2) = \deg(x_3) = \deg(x_4) = 3$, $\deg(x_5) \leq 5$ and $V(C) \cap V(C') = \emptyset$. Then vertices x_i are uncolored. Let y_i be the neighbor of x_i in $G - C'$, for $i = 1, 2, 3, 4$. Moreover, let y_5, \dots, y_m be the neighbors of x_5 in $G - C'$.

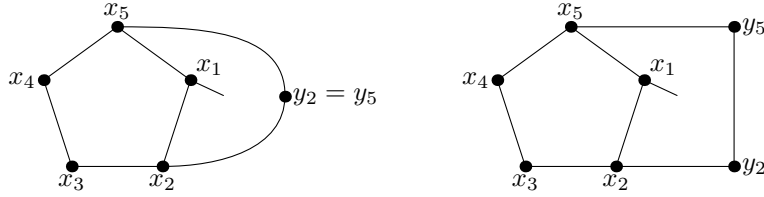


Fig. 4: Cases 6a (left) and 6b (right)

Case 6a. $y_i = y_j$ for $i \neq j$. Since G is triangle-free x_i and x_j are at distance 2 in C' (see Fig. 4). Then there is a 4-cycle $x_iy_ix_jzx_i$ in G . As Case 5 is excluded the cycle is separating and we proceed as in Case 4a.

Case 6b. $y_iy_j \in E(G)$ for $i \neq j$ (see Fig. 4). Then there is a separating cycle of length 4 or 5 in G and we proceed as in Case 4a again.

Case 6c. There are three distinct vertices $x_i, x_j, x_k \subset \{x_1, \dots, x_5\}$ such that each has a colored neighbor. Then at least one of cycles in $G[V(C) \cup \{x_i, x_j, x_k\}]$ is a separating cycle in G of length from 4 to 6. Then we can proceed exactly as in Case 4a. By symmetry we can assume that y_1 or y_2 is not colored (i.e. we change denotations of vertices x_1, \dots, x_4 and y_1, \dots, y_4 , if needed).

Case 6d. y_i, y_j and z are colored and z is a neighbor of y_k for distinct i, j, k . Moreover, y_i and z have the same color and x_i is adjacent with x_k (see Fig. 5). Again one can see that at least one of cycles in $G[V(C) \cup V(C') \cup \{y_k\}]$ is a separating cycle in G of length from 4 to 6 so we can proceed as in Case 4a.

Case 6e. y_2 and a neighbor of y_1 have the same color (or y_1 and a neighbor of y_2 have the same color). As Case 6d is excluded y_3 and y_4 are uncolored. By symmetry we can assume that identifying y_1 and y_2 does not introduce an edge with both ends of the same color (i.e. we change denotations of vertices x_1, \dots, x_4 and y_1, \dots, y_4 , if needed).

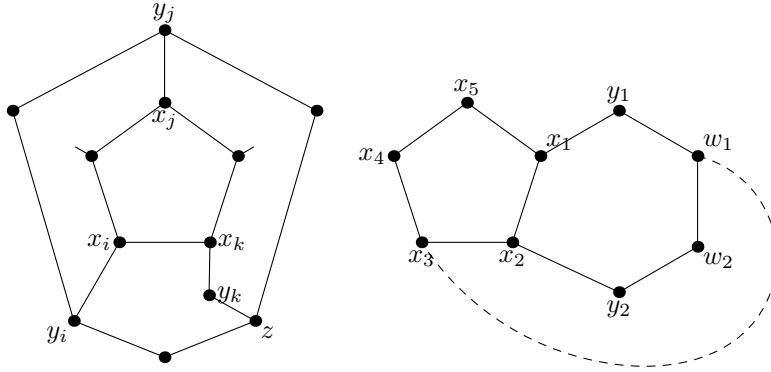


Fig. 5: Cases 6d (left) and 6g (right).

Case 6f. y_3 is colored and x_5 has a colored neighbor. As Cases 6c and 6d are excluded y_2 is uncolored and y_4 has no neighbor with the same color as y_3 . By symmetry we can assume that identifying y_1 with y_2 and y_3 with x_5 does not introduce an edge with both ends of the same color.

Case 6g. There is a path $y_1w_1w_2y_2$ distinct from $y_1x_1x_2y_2$ (see Fig. 5). Then we consider a cycle $C' = y_1w_1w_2y_2x_2x_1y_1$. As G is triangle-free, $\deg x_1 = \deg x_2 = 3$ and $y_1y_2 \notin E(G)$ cycle C' has no chord. Case 4 is excluded so C' is a separating cycle and we can proceed similarly as in Case 4a. The only difference is that this time we try to join x_2 and w_1 to assure that a safe coloring of C' is obtained. We cannot do it when there is a 2-path in $G - \text{int}(C')$ between the vertices we want to join because it would create a triangle. Since $\deg x_2 = 3$ this 2-path is $x_2x_3w_1$. But then 5-cycle $x_2x_3w_1y_1x_1x_2$ is separating and we proceed as in Case 4a.

Case 6h. There is a path $x_5y_kwy_3$ for some $k \in \{5, \dots, m\}$. Then we consider a cycle $C' = y_kx_5x_4x_3y_3wy_k$ and proceed similarly as in Case 6g.

Case 6i. Let G' be the graph obtained from G by deleting x_1, x_2, x_3, x_4 and identifying x_5 with y_3 and y_1 with y_2 . Since we excluded cases 6c–6h, G' is triangle-free and the graph induced by colored vertices of G' is properly 3-colored. Thus we can use induction to get a 3-coloring c of G' . We then extend c to a 3-coloring of G . As Case 6b is excluded, the (partial) coloring inherited from G' is proper. If $c(y_1) = c(x_5)$ then we color x_4, x_3, x_2, x_1 , in that order, always using a free color. If $c(y_1) \neq c(x_5)$ we put $c(x_2) = c(x_5)$ and color x_4, x_3, x_1 in that order. This completes the proof. \square

3 An Algorithm

The proof of Grötzsch's theorem presented in § 2 can be treated as a scheme of an algorithm. As the proof is inductive, the most natural approach suggests that the algorithm should be recursive. In this section we describe how to implement efficiently the recursive algorithm arising from the proof. In particular we need to explain how to recognize successive cases and how to perform relevant reductions efficiently. We start from describing data structures used in our algorithm. Then we discuss how to use recursion efficiently and how to implement non-trivial operations of the algorithm. Throughout the paper G refers to the graph given in the input of our recursive algorithm and n denotes the number of its vertices.

3.1 Data Structures

Input Graph, Adjacency Lists. W.l.o.g. we can assume that the input graph is connected, for otherwise the algorithm is executed separately in each connected component. Moreover, the input graph is given in the form of adjacency lists. We also assume that there is given a planar embedding of the graph, i.e. neighbors of each vertex appear in the relevant adjacency list in the clockwise order given by the embedding.

Faces and Face Queues. Observe that using a planar embedding stored in adjacency lists we can easily compute the faces of the input graph. As the graph is connected each face corresponds to a certain facial walk. For every edge uv there are at most two faces incident to uv . A face is called *right face incident to (u, v)* when v succeeds u in the sequence of successive vertices of the facial walk corresponding to the face given in the clockwise order. Otherwise the face is called *left face incident to (u, v)* . Each face f is stored as the corresponding facial walk, i.e. a list of pointers to adjacency lists elements corresponding to the successive edges of the walk. For each such element e corresponding to neighbor v of vertex u , face f is the right face incident to (u, v) . Additionally, e stores a pointer to f . Each face stores also its length, i.e. the length of the corresponding facial walk.

We will also use three queues $Q_4, Q_5, Q_{\geq 6}$ storing faces of length 4, 5, and ≥ 6 respectively, satisfying conditions described in cases 5, 6, 4 of the proof of Theorem 2.2, respectively.

Low Degree Vertices Queue. In order to recognize Case 1 fast we maintain a queue storing the vertices of degree at most 2.

Short Path Data Structure (SPDS) In order to search efficiently for 2-paths joining a given pair of vertices we maintain the Short Path Data Structure described in § 4.

3.2 Recursion

Note that in the recursive algorithm induced by the proof given in § 2 we need to split G into two parts. Then each of the parts is processed separately by successive recursive calls. By splitting the graph we mean splitting the adjacency lists and all the other data structures described in the previous section. As the worst-case depth of the recursion is $\Theta(n)$ the naïve approach would involve $\Theta(n^2)$ total time spent on splitting the graph. Instead, before splitting the information on G our algorithm finds the smaller of the two parts. It can be easily done using two DFS calls run in parallel in each of the parts, i.e. each time we find a new vertex in one part, we suspend the search in this part and continue searching in the another. Such an approach finds the smaller part A in linear time with respect to the size of A . The other part will be denoted by B . Then we can easily split adjacency lists, face queues and low degree vertices query. The vertices of the separating cycle (or path) are copied and the copies are added to A . Note that there are at most 6 such vertices. Next, we delete all the vertices of $V(A) \setminus V(B)$ from the SPDS. We will refer to this operation as CUT. As a result of CUT we obtain an SPDS for B . A Short Path Data Structure for A is computed from the scratch. In § 4 we show that deletion of an edge from the SPDS takes $\mathcal{O}(1)$ time and the new SPDS can be built in $\mathcal{O}(|A|)$ time. Thus the splitting is performed in $\mathcal{O}(|V(A)|)$ worst-case time.

Proposition 3.1. *The total time spent by the algorithm on splitting data structures before recursive calls is $\mathcal{O}(n \log n)$*

Proof. We can assume that each time we split the graph into two parts – the possible split into three ones described in Case 2 is treated as two successive splits. Let us call the vertices of the separating cycle (or path) as *outer vertices* and the remaining ones from the smaller part A as *inner vertices*. The total time spent on splitting data structures is linear with the total number of inner and outer vertices. As there are $\mathcal{O}(n)$ splits, and each split involves at most 6 outer vertices the total number of outer vertices to be considered is $\mathcal{O}(n)$. Moreover, as during each split of a k -vertex graph there are at most $\lfloor \frac{k}{2} \rfloor$ inner vertices each vertex of the input graph becomes an inner vertex at most $\log n$ times. Hence the total number of inner vertices is $\mathcal{O}(n \log n)$. \square

3.3 Non-trivial Operations

Identifying Vertices In this section we describe how our algorithm updates the data structures described in § 3.1 during the operation of identifying a pair of vertices u, v . Identifying two vertices can be performed using deletions and insertions. More precisely, the operation $\text{IDENTIFY}(u, v)$ is executed using the following algorithm. First it compares degrees of u and v in graph G . Assume that $\deg_G(u) \leq \deg_G(v)$. Then for each neighbor x of u we delete edge ux from G and add a new edge vx , unless it is already present in the graph.

Lemma 3.2. *The total number of pairs of delete/insert operations performed by IDENTIFY algorithm is bounded by $\mathcal{O}(n \log n)$.*

Proof. For now, assume that there are no other edge deletions performed by our algorithm, except for those involved with identifying vertices. The operation of deleting edge ux and adding vx during $\text{IDENTIFY}(u, v)$ will be called *moving edge ux* . We see that each edge of the input graph can be moved at most $\lceil \log n \rceil$ times, for otherwise there would appear a vertex of degree $> n$. Subsequently, there are $\mathcal{O}(n \log n)$ pairs of delete/insert operations performed during IDENTIFY operation. It is clear that this number does not increase when we consider additional deletions. \square

As we always identify a pair of vertices in the same facial walk it is straightforward to update adjacency lists. Lemma 3.2 shows that we need $\mathcal{O}(n \log n)$ time in total for these updates including updating the information about the faces incident to x and y . Each of affected faces is then placed in appropriate face queue (if one has to be changed). In § 4 we show how to update the Short Path Data Structure efficiently after IDENTIFY. To sum up, identifying vertices takes $\mathcal{O}(n \log n)$ time including updating data structures.

Finding Short Paths In our algorithm we need to find paths of length 1, 2 or 3 between given pairs of vertices. Observe that there are $\mathcal{O}(n)$ such queries during an execution of the whole algorithm. As we show in § 4 paths of length 1 or 2 can be found in $\mathcal{O}(1)$ time using the Short Path Data Structure. It remains to focus on paths of length 3.

Let us describe an algorithm PATH3 that will be used to find a 3-path between a pair of vertices u, v , if there is any. We can assume that we are given a path p of length 2 (cases 4b and 5a) or of length 3 (Case 6g) joining u and v . W.l.o.g. we assume that $\deg(u) \leq \deg(v)$. Let $A(u), A(v)$ denote the adjacency lists of u and v , respectively. We start from assigning variables x_1 and x_2 to the element of $A(u)$ corresponding to the edge of p incident with u . Similarly, we

assign x_3 and x_4 to the element of $A(v)$ corresponding to the edge of p incident with v . Then we start a loop. We assign x_1 to the preceding element and x_2 to the succeeding element in $A(u)$. Similarly, we assign x_3 to the preceding element and x_4 to the succeeding element in $A(v)$ (see Fig. 6).

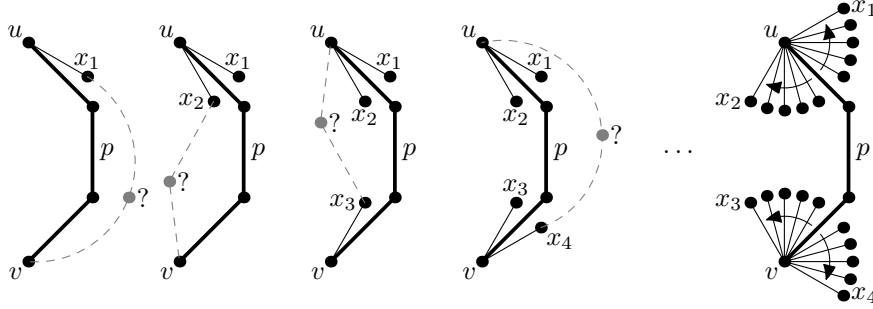


Fig. 6: Order of queries in algorithm PATH3.

Then we use the Short Path Data Structure to search for paths of length 2 between x_1 and v , x_2 and v , x_3 and u , x_4 and u . If a path is found, the algorithm stops, otherwise we repeat the loop. If no 3-path exists at all, the loop stops when all the neighbors of u are checked.

Lemma 3.3. *The total time spent on performing PATH3 algorithm is $\mathcal{O}(n \log n)$.*

Proof. We can divide these operations into two groups. The operation is called *successful* if there exists a 3-path joining u and v , distinct from p when $|p|=3$, and *failed* in the other case. Recall that when there is no such 3-path, vertices u and v are identified. As the time complexity of a single execution of PATH3 algorithm is $\mathcal{O}(\deg u)$ and all the edges incident with u are deleted during IDENTIFY(u, v) operation, the total time spent on performing failed PATH3 operations is linear in the number of edge deletions caused by identifying vertices. By Lemma 3.2 there are $\mathcal{O}(n \log n)$ such edge deletions.

Now it remains to estimate the time used by successful operations. Let us consider one of them. Let C' be the separating cycle compound of the path p and the 3-path that was found. Let H be the graph $C' \cup \text{int}(C')$. Recall that one of vertices u, v is not colored, say v . Then the number of queries sent to the SPDS is at most $4 \cdot \deg_H(v)$. Observe that v will be colored in graph H , just before the recursive call for graph H . Hence the total number of queries asked during executions of successful PATH3 operations is at most 8 times larger than the total number of edges appearing in G (to each edge we assign 8 queries, 4

queries to each of the ends). The number of such edges, including these added during IDENTIFY operation, is bounded by $\mathcal{O}(n \log n)$. Thus the time used by the successful operations is $\mathcal{O}(n \log n)$. \square

Removing a Vertex of Degree at Most 3 In cases 1 and 6i we remove vertices of degrees 1, 2 or 3. There are at most $\mathcal{O}(n)$ such operations in total. As the degrees are bounded the total time spent on updating the Short Path Data Structure and adjacency lists is $\mathcal{O}(n)$. We also need to update information about incident faces. We may need to join two or three of them. It is easy to update the facial walk of the resulting face in $\mathcal{O}(1)$ time by joining the walks of the faces incident to the deleted vertex. The problem is that edges contained in the walk corresponding to the new face store pointers to two different faces. Thus we use the well-known Find and Union algorithm (see e.g. [13]) for finding a right face incident to a given edge and for joining faces. The amortized time of the search is $\mathcal{O}(\alpha(n))$, where $\alpha(n)$ is the inverse of the Ackerman's function. As the total number of all these searches is $\mathcal{O}(n)$ it does not increase the overall time complexity of our coloring algorithm.

Before deleting an edge we additionally need to check whether it is a bridge. The check can be easily done by verifying whether the edge has the right face equal to its left face. If so, after deleting the edge the face is split into two faces in $\mathcal{O}(1)$ time and we process each of the connected components recursively.

Searching for Faces To search for the faces described in the cases 5, 6, 4 of the proof from § 2 we use queues $Q_4, Q_5, Q_{\geq 6}$, respectively. The queues are initialized in $\mathcal{O}(n)$ time and the searches are performed in $\mathcal{O}(1)$ time.

Additional Remarks To recognize cases 2, 3, 4a, 6c–6f efficiently it suffices to pass down the outer cycle in the recursion, when the cycle is colored. Then it takes only $\mathcal{O}(1)$ time to recognize each case (in some cases we use Short Path Data Structure) since there are at most 6 colored vertices.

4 Short Path Data Structure

In this section we describe the *Short Path Data Structure* (SPDS) which can be built in linear time and enables finding shortest paths of length at most 2 in planar graphs in $\mathcal{O}(1)$ time. Moreover, we show here how to update the structure after deleting an edge, adding an edge and after identifying a pair of vertices. Then we analyze the total time needed for updates of the SPDS during the particular sequence of operations appearing in our 3-coloring algorithm. The time turns out to be bounded by $\mathcal{O}(n \log n)$.

4.1 The Structure and Processing the Queries

The Short Path Data Structure consists of two elements, denoted as \vec{G}_1 and \vec{G}_2 . We will describe them after introducing some basic notions.

A directed graph is said to be k -oriented if its every vertex has the out-degree at most k . If one can orient edges of an undirected graph H obtaining k -oriented graph H' we say that H can be k -oriented. In particular, when $k = \mathcal{O}(1)$ we will say that H' is $\mathcal{O}(1)$ -oriented and H can be $\mathcal{O}(1)$ -oriented. \vec{H} will denote certain orientation of a graph H . The *arboricity* of a graph H is the minimal number of forests needed to cover all the edges of H . Observe that a graph with arboricity a can be a -oriented.

Graph \vec{G}_1 and Adjacency Queries. In this section G_1 denotes a planar graph for which we build a SPDS (recall from § 3.2 that it is not only the input graph). It is widely known that planar graphs have arboricity at most 3. Thus G_1 can be $\mathcal{O}(1)$ -oriented. Let \vec{G}_1 denote such an orientation of G_1 . Then $xy \in E(G_1)$ iff $(x, y) \in E(\vec{G}_1)$ or $(y, x) \in E(\vec{G}_1)$. Thus, providing that we can maintain bounded out-degrees in \vec{G}_1 during our coloring algorithm, we can process in $\mathcal{O}(1)$ time the queries of the form: “Are vertices x and y adjacent?”.

Graph \vec{G}_2 . Let G_2 be a graph with the same vertex set as G_1 . Moreover, edge vw is in G_2 iff there exists vertex $x \in V(\vec{G}_1)$ such that $(x, v) \in E(\vec{G}_1)$ and $(x, w) \in E(\vec{G}_1)$. Vertex x is said to *support* edge vw . Since \vec{G}_1 has bounded out-degree every vertex supports $\mathcal{O}(1)$ edges in G_2 . Hence G_2 is of linear size. The following lemma states even more:

Lemma 4.1. *Let \vec{G}_1 be a directed planar graph with out-degree bounded by d . Let G_2 be an undirected graph with $V(G_2) = V(\vec{G}_1)$ and $E(G_2) = \{vw : (x, v) \in E(\vec{G}_1) \text{ and } (x, w) \in E(\vec{G}_1)\}$. Then G_2 is a union of at most $4 \cdot \binom{d}{2}$ planar graphs.*

Proof. It is well known that every planar graph is 4-colorable. Hence let us take an arbitrary 4-coloring of \vec{G}_1 . Subsequently we can partition edges of G_2 into $4 \cdot \binom{d}{2}$ graphs in such a way that if two edges belong to the same graph they are supported by two different vertices of the same color. Then it is easy to show that each of the $4 \cdot \binom{d}{2}$ graphs has a plane embedding: we consider an arbitrary plane embedding \mathcal{E} of \vec{G}_1 , we draw the vertices of G_2 in the same points as in \mathcal{E} , while the embedding of every edge in G_2 is equal to the embedding of the corresponding path in \vec{G}_1 . \square

Corollary 4.2. *If the out-degree in graph \vec{G}_1 is bounded by d then graph G_2 has arboricity bounded by $12 \cdot \binom{d}{2}$.*

Corollary 4.3. *Graph G_2 can be $\mathcal{O}(1)$ -oriented.*

Corollary 4.2 follows immediately since the arboricity of a planar graph is at most 3. By \vec{G}_2 we will denote an $\mathcal{O}(1)$ -orientation of G_2 . Let e be an edge in \vec{G}_2 equal (v, w) or (w, v) . Let x be a vertex that supports e and let $e_1 = (x, v)$ and $e_2 = (x, w)$, $e_1, e_2 \in E(\vec{G}_1)$. We say that edges e_1 and e_2 are *parents* of e and e is a *child* of e_1 and e_2 . We say that a pair $\{e_1, e_2\}$ is a *couple of parents* of e . Notice that each edge can have more than one couple of parents. We additionally store the following information:

- for each $e \in E(\vec{G}_2)$ a list $P(e)$ of all pairs $\{e_1, e_2\}$ such that e is a common child of e_1 and e_2 ,
- for each $e \in E(\vec{G}_1)$ a list $C(e)$ of pairs (c, p) where $c \in E(\vec{G}_2)$ is a common child of e and certain edge f and p is a pointer to $\{e, f\}$ in the list $P(c)$.

Queries About 2-paths It is easy to see that when \vec{G}_1 and \vec{G}_2 are $\mathcal{O}(1)$ -oriented we can find a path uxv of length 2 joining a pair of given distinct vertices u, v in $\mathcal{O}(1)$ time as follows:

- (i) check whether there is an oriented path uxv or vxu in \vec{G}_1 ,
- (ii) check whether there is a vertex x such that $(u, x), (v, x) \in E(\vec{G}_1)$,
- (iii) check whether there is an edge $e = (u, v)$ or $e = (v, u)$ in \vec{G}_2 . If so, pick any of its couples of parents $\{(x, u), (x, v)\}$ stored in $P(e)$.

4.2 Inserting and Deleting Edges

Algorithm 1 Maintaining D -orientation of graph H

```

1: procedure REORIENT( $w$ )
2:    $S \leftarrow \{w\}$ 
3:   while  $S \neq \emptyset$  do
4:      $x \leftarrow \text{POP}(S)$ 
5:     for all  $(x, y) \in E(\vec{H})$  do
6:       Change orientation of edge  $(x, y)$  to  $(y, x)$ .
7:       if  $\text{outdeg}(y) = D + 1$  then
8:         PUSH( $S, y$ )

```

Maintaining Bounded Out-degrees in \vec{G}_1 and \vec{G}_2 . As graph G_1 is dynamically changing we will need to add and remove edges from \vec{G}_1 and \vec{G}_2 .

Assume that H is an arbitrary graph. Assume that we want to maintain a D -orientation of H , denoted by \vec{H} . While removing is easy, after adding an edge there may appear a vertex of outdegree larger than D . Then we need to reorient some edges to leave the graph D -oriented. We use the approach of G. Brodal and R. Fagerberg [14]. As long as graph \vec{H} contains a vertex of outdegree larger than D we pick such a vertex x and we change orientation of all the edges leaving x . We will denote this routine as $\text{REORIENT}(w)$, where w is the initial vertex of degree larger than D (see Alg. 1). We will use algorithm REORIENT for maintaining bounded orientation in \vec{G}_1 and \vec{G}_2 .

Updating the SPDS After a Deletion Note that after deleting an edge from \vec{G}_1 we need to find out which edges in \vec{G}_2 should be deleted, if any. Assume that e is an edge of \vec{G}_1 and it is going to be deleted. For each pair $(c, p) \in C(e)$ we have to perform the following operations: remove the pair $\{e, f\}$ referenced by pointer p from list $P(c)$, remove the pair (c, p) from the list $C(f)$. If list $P(c)$ becomes empty we delete edge c from G_2 . We will refer to this routine as $\text{DELETESHORTCUT}(e)$. Since e has at most $d = \mathcal{O}(1)$ children, the following proposition holds:

Proposition 4.4. *After deletion of an edge the SPDS can be updated in $\mathcal{O}(1)$ time using DELETESHORTCUT routine.*

Updating the SPDS After an Insertion To update the SPDS after adding an edge uv to G_1 we start from adding (u, v) to \vec{G}_1 . If then $\text{outdeg}(u) = D + 1$ we perform $\text{REORIENT}(u)$. Whenever any edge e in \vec{G}_1 changes its orientation we act as if it was deleted and we perform $\text{DELETESHORTCUT}(e)$. Moreover, when any edge (u, v) appears in \vec{G}_1 , both after $\text{INSERT}(u, v)$ and after reorienting (v, u) , we add an edge (v, w) to \vec{G}_2 for each edge (u, w) present in \vec{G}_1 . When we add an edge to \vec{G}_2 we also use REORIENT if needed. We will refer to this routine as $\text{INSERTSHORTCUT}(uv)$.

Building the SPDS One could build the SPDS by adding successive edges of the input graph, each time updating the structure like in the previous paragraph. However, this does not give a linear-time algorithm. To get linear time we should first build graph \vec{G}_1 and then \vec{G}_2 . More precisely, we start from creating two graphs with the same vertices as G_1 but no edges. Then for every edge uv of G_1 we add (u, v) to \vec{G}_1 and perform REORIENT if needed. After adding all edges we build graph \vec{G}_2 . To this end, for each pair of edges $(x, u), (x, v)$ in graph \vec{G}_1 we add (u, v) to \vec{G}_2 and perform REORIENT if needed. In the following we will show that these two steps take only $\mathcal{O}(|V(G_1)|)$ time.

Updating the SPDS After Identifying Vertices Now we describe a routine IDENTIFYSHORTCUT(u, v) for updating the SPDS after identifying u and v . W.l.o.g. we assume that $\deg_{G_1}(u) \leq \deg_{G_1}(v)$. We start from identifying u and v in $\overrightarrow{G_1}$. More precisely, for each edge (x, u) we find the relevant element of vertex x adjacency list and replace u by v^1 . We also join adjacency lists of u and v and store the resulting list in v . Clearly it takes $\mathcal{O}(\deg_{G_1})$ time. As a result it may happen that $\text{outdeg}_{\overrightarrow{G_1}}(v)$ is too large. Then we perform REORIENT(v). Similarly we identify u and v in $\overrightarrow{G_2}$. Finally, for each pair of edges $(v, x), (v, y) \in E(\overrightarrow{G_1})$ we add xy to G_2 unless it is already present in G_2 .

Lemma 4.5. *Performing IDENTIFYSHORTCUT(u, v) routine takes $\mathcal{O}(\deg_{G_1}(u) + r)$ time, where r is the number of reorientations performed by REORIENT algorithm.*

Proof. Let D_1 denote the bound on outdegrees in $\overrightarrow{G_1}$, $D_1 = \mathcal{O}(1)$. It is straightforward to see that the time is $\mathcal{O}(\deg_{G_1}(u) + \deg_{G_2}(u) + r)$. However, since any edge $uz \in G_2$ corresponds to a pair $(x, u), (x, z)$ in $\overrightarrow{G_1}$ it follows that $\deg_{G_2}(u) \leq D_1 \cdot \deg_{G_1}(u) = \mathcal{O}(\deg_{G_1}(u))$. \square

4.3 Time Complexity of Updating the SPDS

In this section we show that for the particular sequence of updates appearing in our coloring algorithm the total time needed for updating the SPDS is $\mathcal{O}(n \log n)$. The following lemma is a slight generalization of Lemma 1 from the paper [14]. Their proof remains valid even for the modified formulation presented below.

Lemma 4.6. *Let σ be a sequence of edge insertions/deletions and vertices identify operations performed on some graph. Assume that after inserting edges and identifying vertices we use algorithm REORIENT to maintain a D -orientation of this graph. Let H_0 be the initial graph and let $\overrightarrow{H_0}$ be its initial orientation. Assume that $\overrightarrow{H_0}$ is δ -oriented, for some δ such that $D \geq 2\delta$. Let H_i be the graph after i -th operation of sequence σ and let k denote the number of insertions in σ .*

If there exists a sequence $\overrightarrow{H_1}, \overrightarrow{H_2}, \dots, \overrightarrow{H_{|\sigma|}}$ of δ -orientations with at most r edge reorientations in total then algorithm REORIENT performs at most

$$(k + r) \frac{D + 1}{D + 1 - 2\delta}$$

¹ It takes only $\mathcal{O}(1)$ time, provided that with each vertex a we store pointers to adjacency lists elements corresponding to edges entering a .

edge reorientations in total on the sequence σ .

Lemma 4.7. *For any graph H with arboricity a one can build its $(3a - 1)$ -orientation \vec{H} in $\mathcal{O}(|V(H)| + |E(H)|)$ time adding successive edges to \vec{H} , each time using algorithm REORIENT to keep outdegrees bounded.*

Proof. We will describe a sequence of a -orientations needed in Lemma 4.6. $\vec{H}_{|\sigma|}$ is an arbitrary a -orientation of $H_{|\sigma|}$. For each $i = 1, \dots, |\sigma| - 1$ we get \vec{H}_i from \vec{H}_{i+1} by removing relevant edge. Clearly there is no single reorientation in this sequence. It suffices to apply Lemma 4.6 to finish the proof. We get that the total number of reorientations is bounded by $(|E(H)| + 0) \frac{3a-1+1}{3a-1+1-2a} = 3|E(H)|$. \square

Corollary 4.8. *For any planar graph G_1 the Short Path Data Structure can be constructed in $\mathcal{O}(|V(G_1)|)$ time.*

Lemma 4.9. *Let σ be the sequence of insert, delete and identify operations performed in graph G_1 by the 3-coloring algorithm, not including the initial inserts performed to build the SPDS. Let k be the number of insertions in σ . Then the total number of reorientations in \vec{G}_1 used by REORIENT algorithm is $\mathcal{O}(k)$.*

Proof. Let G^i be the graph G_1 after the i -th operation. We will construct a sequence of 6-orientations $\mathcal{G} = \vec{G}^1, \vec{G}^2, \dots, \vec{G}^k$. It follows from the proof of theorem 2.2 that the last graph in this sequence has no uncolored vertices. Hence it contains at most 6 vertices and it is trivial to find its 6-orientation. For each $i = 1, \dots, t - 1$ we will describe \vec{G}^i using \vec{G}^{i+1} .

If σ_{i+1} is an insertion of edge uv , \vec{G}^i is obtained from \vec{G}^{i+1} by deleting uv .

If $\sigma_{i+1} = \text{IDENTIFY}(u, v)$ the edges incident with the identified vertex x in \vec{G}^{i+1} are partitioned into two groups in \vec{G}^i without changing their orientation. (Recall that u and v are not adjacent in \vec{G}^i .) Clearly, $\text{outdeg}_{\vec{G}^i} u \leq \text{outdeg}_{\vec{G}^{i+1}} x$ and $\text{outdeg}_{\vec{G}^i} v \leq \text{outdeg}_{\vec{G}^{i+1}} x$.

Now assume that σ_i is an edge deletion. First let us consider the case when it is not a deletion involved with CUT operation. Recall from the proof of theorem 2.2 that such a deletion is caused by removing a vertex of degree at most 3. Denote this vertex by x . Hence to get \vec{G}^i it suffices to copy \vec{G}^{i+1} and add an edge leaving x .

Finally let us consider a sequence of edge deletions caused by CUT operation. Let G^i and G^j denote the graph before and after CUT operation, respectively. All edges of G^i that are present in G^j inherit their orientations. Now we will describe how to orient the remaining ones. Let A be the graph induced by the

vertices removed from G^i . As A is planar there is its 3-orientation \vec{A} . The remaining edges, joining a vertex from A , say x , with a vertex from G^j , say y , are oriented from x to y . Observe that a vertex in A can be adjacent with at most 3 vertices in G^j , since there are no triangles (see Fig 7). Clearly, the

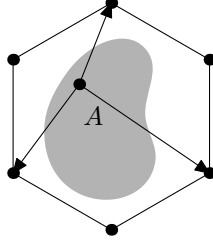


Fig. 7: Vertices of graph A are adjacent to at most 3 vertices of G^j .

resulting graph \vec{G}^i is 6-oriented. For each $t = i, \dots, j-2$ graph \vec{G}^{t+1} is obtained from \vec{G}^t by deleting successive edge.

Thus we have described a sequence of 6-orientations $\vec{G}^1, \dots, \vec{G}^{|\sigma|}$. Observe that there is no single edge reorientation in this sequence. Let \vec{G}^0 be the initial orientation of graph G_1 . By Lemma 4.7 \vec{G}^0 is 8-oriented. Now we consider the sequence $\vec{G}^0, \dots, \vec{G}^t$. Orientations \vec{G}^0 and \vec{G}^1 can differ very much. In order to avoid this situation we modify orientations $\vec{G}^1, \dots, \vec{G}^t$. For each $i = 1, \dots, t$ each edge from graph G^i which is also present in G^0 is oriented exactly like in \vec{G}^0 . Thus we obtain a sequence of $(6+8)$ -orientations $\vec{G}^0, \dots, \vec{G}^t$. Clearly this sequence contains no edge reorientations. Now we use Lemma 4.6, putting $\delta = 14$ and $D = 3\delta - 1$. It implies that the total number of edge reorientations performed by algorithm REORIENT in our sequence of operations is bounded by $\mathcal{O}(k)$. \square

Lemma 4.10 (see Lemma 2 in [14]). *Let $H = (V, E)$ be a graph with arboricity at most a and let \vec{H} be an orientation of H . If $u \in V(\vec{H})$ has out-degree at least $2a$ then there exists a vertex v with out-degree smaller than $2a$ such that there is a directed path from u to v of length $\lceil \log_2 |V| \rceil$ in \vec{H} .*

Lemma 4.11. *Let H be an arbitrary n -vertex graph of arboricity a and let \vec{H} be its initial δ -orientation, $\delta \geq 2a$. For arbitrary sequence of p edge insertions, q edge deletions and r identify operations and such that the arboricity of the graph after each operation does not exceed a algorithm REORIENT maintains $(3\delta - 1)$ -orientation, performing $\mathcal{O}((p + r\delta) \log n)$ edge reorientations.*

Proof. For each $i = 1, \dots, p+q+r$ let H_i denote the graph after the i -operation and let $H_0 = H$, $\vec{H}_0 = \vec{H}$. We will describe a sequence of δ -orientations needed in Lemma 4.6.

For each $i = 1, \dots, p+q+r$ we get \vec{H}_i from \vec{H}_{i-1} as follows. If the i -th operation is delete we simply remove relevant edge from \vec{H}_{i-1} . If the i -th operation is an insertion of an edge uv , we add edge (u, v) to \vec{H}_{i-1} . Then if the outdegree of u is larger than δ we pick the path described in Lemma 4.10 and revert all edges in this path. Clearly the resulting graph is δ -oriented and we choose it as \vec{H}_i . Finally, if the i -th operation is identifying vertices u and v we start from identifying u and v in graph \vec{H}_{i-1} . Let x be the new vertex. In the resulting graph $\text{outdeg}(x) \leq 2\delta$. Then if $\text{outdeg}(x) > \delta$ we apply Lemma 4.10 $\text{outdeg}(x) - \delta$ times each time reverting edges of the relevant path. The resulting graph is δ -oriented and we choose it as \vec{H}_i . The description of the sequence of orientations is finished now. The total number of reorientations in this sequence is bounded by $(p+r\delta)\lceil \log n \rceil$. It suffices to apply Lemma 4.6 to finish the proof. \square

Corollary 4.12. *The total number of reorientations in \vec{G}_2 , performed by algorithm REORIENT during a sequence of operations containing t edge insertions and identify operations, performed on an n -vertex graph G_1 , is $\mathcal{O}(t \log n)$.*

Proof. Graph G_2 is n -vertex graph of bounded arboricity. Hence \vec{G}_2 is initially $\mathcal{O}(1)$ -oriented. Lemma 4.9 implies that there are $\mathcal{O}(t)$ insertions in G_2 caused by insertions and identify operations in \vec{G}_1 . Apart from that there can be at most t identifyings in G_2 . Then it follows from Lemma 4.11 that the total number of reorientations in \vec{G}_2 is bounded by $\mathcal{O}(t \log n)$. \square

The following theorem follows immediately from lemmas 4.5, 3.2, 4.9 and 3.2.

Theorem 4.13. *Let n be the number of vertices of the graph on the input of the coloring algorithm. The total time spent by our 3-coloring algorithm at updating the Short Path Data Structures is $\mathcal{O}(n \log n)$.*

5 Conclusions and Further Research

In this paper we showed a new algorithm for 3-coloring triangle-free planar graphs. Our algorithm is almost linear, i.e. its time complexity is $\mathcal{O}(n \log n)$. It raises a natural question about a linear-time algorithm for this problem. We suppose that this situation is similar to 4-coloring planar graphs. It is known

how to 4-color planar graphs in $\mathcal{O}(n^2)$ time [2] and it seems that to get a faster algorithm we need a significantly new proof of the four color theorem. Analogously, we suppose that we need a really new proof of Grötzsch's theorem to design a linear-time algorithm for 3-coloring triangle-free planar graphs. Recently, Carsten Thomassen published a new proof of the theorem saying that each planar graph with no triangles and 4-cycles is 3-colorable [7]. It seems that basing on this proof one can design a complicated but linear-time algorithm for coloring such graphs. Nevertheless analogous proof for triangle-free planar graphs is not yet known [15].

Acknowledgments I would like to thank Krzysztof Diks and anonymous referees for reading this paper carefully and helpful comments. Thanks go also to Maciej Kurowski for many interesting discussions in Århus, not only these on Grötzsch's theorem.

References

- [1] Łukasz Kowalik. Fast 3-coloring triangle-free planar graphs. In Susanne Albers and Tomasz Radzik, editors, *Proc. 12th Annual European Symposium on Algorithms (ESA 2004)*, volume 3221 of *Lecture Notes in Computer Science*, pages 436–447. Springer-Verlag, 2004.
- [2] Neil Robertson, Daniel P. Sanders, Paul Seymour, and Robin Thomas. Efficiently four-coloring planar graphs. In *Proc. 28th Symposium on Theory of Computing*, pages 571–575. ACM, 1996.
- [3] N. Chiba, T. Nishizeki, and N.Saito. A linear algorithm for five-coloring a planar graph. *J. Algorithms*, 2:317–327, 1981.
- [4] M. R. Garey and D. S. Johnson. Some simplified NP-complete graph problems. *Theoretical Computer Science*, 1(3):237–267, February 1976.
- [5] H. Grötzsch. Ein dreifarbensatz für dreikreisfreie netze auf der kugel. Technical report, *Wiss. Z. Martin Luther Univ. Halle Wittenberg, Math.-Nat. Reihe 8*, 1959.
- [6] Carsten Thomassen. Grötzsch's 3-color theorem and its counterparts for the torus and the projective plane. *Journal of Combinatorial Theory, Series B*, 62:268–279, 1994.
- [7] Carsten Thomassen. A short list color proof of Grötzsch's theorem. *Journal of Combinatorial Theory, Series B*, 88:189–192, 2003.

- [8] Łukasz Kowalik and Maciej Kurowski. Shortest path queries in planar graphs in constant time. In *Proc. 35th Symposium Theory of Computing*, pages 143–148. ACM, June 2003.
- [9] O. V. Borodin and A. Raspaud. A sufficient condition for planar graphs to be 3-colorable. *Journal of Combinatorial Theory, Series B*, 88:17–27, 2003.
- [10] O. V. Borodin, A. N. Glebov, A. Raspaud, and M. R. Salavatipour. Planar graphs without cycles of length from 4 to 7 are 3-colorable. 2003. Submitted to *J. of Comb. Th. B*.
- [11] Douglas West. *Introduction to Graph Theory*. Prentice Hall, 1996.
- [12] John Gimbel and Carsten Thomassen. Coloring graphs with fixed genus and girth. *Transactions of the AMS*, 349(11):4555–4564, November 1997.
- [13] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT, 2001.
- [14] Gerth Stølting Brodal and Rolf Fagerberg. Dynamic representations of sparse graphs. In *Proc. 6th Int. Workshop on Algorithms and Data Structures*, volume 1663 of *LNCS*, pages 342–351. 1999.
- [15] Carsten Thomassen. private communication, 2004.