Fast 3-coloring Triangle-Free Planar Graphs*

Łukasz Kowalik†

Abstract

Although deciding whether the vertices of a planar graph can be colored with three colors is NP-hard, the widely known Grötzsch's theorem states that every triangle-free planar graph is 3-colorable. We show the first $o(n^2)$ algorithm for 3-coloring vertices of triangle-free planar graphs. The time complexity of the algorithm is $\mathcal{O}(n \log n)$.

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-time 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 $\mathcal{O}(n \log n)$.

In Section 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 such a way that it can be immediately transformed into an algorithm. In fact the proof can be treated as a description of the algorithm mixed with a proof of its correctness.

In Section 3 we discuss how to implement the algorithm efficiently. We describe details of non-trivial operations as well as data structures needed to

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

†Institute of Informatics, University of Warsaw, Banacha 2, 02-097 Warsaw, Poland. kowalik@mimuw.edu.pl. The research has been partially supported by grants from the Polish Ministry of Science and Higher Education, projects 4T11C04425 and N206 005 32/0807. A part of the research was done during the author's stay at BRICS, Aarhus University, Denmark.

perform these operations fast. The description and analysis of the most involved one, *Short Path Data Structure* (SPDS), is contained in Section 4. This data structure deserves a separate interest. In our paper with Maciej Kurowski [8] we presented a data structure built in linear time and enabling finding shortest paths between given pairs of vertices in constant time, provided that the distance between the vertices is bounded. This data structure works also in the dynamic environment and can be updated after adding or removing an edge in polylogarithmic time. In our coloring algorithm here 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 the identify operation, not described in [8].

1.1 Related work

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 conjecture that our techniques presented here with additional help of the general SPDS from [8] can be adapted to transform these proofs to $\mathcal{O}(n \operatorname{polylog} n)$ algorithms.

Also recently, Dvorak, Kral and Thomas [11] showed that for every surface Σ there exists a polynomial-time algorithm that given an input triangle-free graph G embedded in Σ correctly *determines* whether G is 3-colorable.

1.2 Terminology

We assume the reader is familiar with standard terminology and notation concerning graph theory and planar graphs in particular (see e.g. [12]). Let us recall here some notions that are not so widely used.

A *separating cycle* is a connected graph G is a cycle C such that removing all the vertices of C makes graph G disconnected.

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. Note that when there is a cutvertex incident with f the facial walk w is not a cycle. If the boundary of f is a cycle the walk is called a *facial cycle*. The length of walk w is denoted by $|w|$. We define the length of face f as the length of the corresponding facial walk, and we denote it by $|f|$.

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 $\operatorname{int} C$. Observe that $\operatorname{int} C$ is not necessarily a graph, while $C \cup \operatorname{int} C$ 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 the 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 the charge in the graph, without changing its total amount, in such a way that if the graph contained no desired facial cycle 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. Note that 3-vertices send negative charge, and after this operation the charge in every vertex is equal to 0. 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. Note that if a 5-face has two 2-vertices then it must be the outer face, since 2-vertices are nonadjacent and all are incident with f. It follows that 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 $|C|$ faces that share an edge with 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. □

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 [13]). 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 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 induction on $|V(G)|$. We assume that G has at least one uncolored vertex, for otherwise there is nothing left to do.

In what follows, we will need the following claim several times[1]:

Claim 1. *When G contains a separating cycle C', $4 \leq |C'| \leq 6$, then we can finish the proof by induction.*

Proof of the claim. Let $G_1 = G - \mathrm{int}(C')$. If C' has a chord in $\mathrm{int}(C')$, we additionally put it in G_1 (by planarity and absence of triangles C' has at most one chord in $\mathrm{int}(C')$). If C' is of length 6 and it has no chord in G, we put a chord of C' in G_1 between vertices at distance 3 in C' (so that we do not introduce a triangle). Now we use the induction hypothesis to get a 3-coloring of G_1. (If the vertices of the outer cycle C are colored we extend this coloring to G_1). The resulting 3-coloring of C' is also a proper 3-coloring of $G[V(C')]$, since G_1 contains all the chords of C'. Note that the coloring of C' is also safe, because when $|C'| = 6$, there is a chord of C' in G_1 between vertices at distance 3 in C'. It follows that we can use the induction to find a 3-coloring of

[1] Note that this is a claim, not a case. This is because we want this proof to correspond to an efficient algorithm. In the algorithm, one needs to verify which case applies very fast, i.e. in $O(\log n)$ time, while it is unclear how to find a separating cycle that fast. This claim corresponds to a procedure in the algorithm.

$G_2 = C' \cup \mathrm{int}(C')$. Together with the coloring of G_1 it gives the desired coloring of G, which ends the proof of Claim 1.

Now we are going to consider several cases. In each case we assume that the situations described in the previously considered cases are excluded.

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.

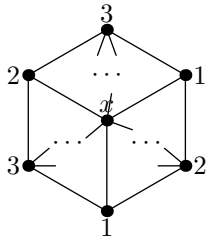

Fig. 1: Case 2.

Case 2. G has an uncolored vertex x joined to two or three colored vertices (see Fig. 1). Observe that $x \notin C$. Moreover, if x has three colored neighbors then $|C| = 6$ and the neighbors cannot have three different colors, as the coloring of C is safe. Hence we extend the 3-coloring of $G[V(C)]$ 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 $G[V(C) \cup \{x\}]$, i.e. when a face of $G[V(C) \cup \{x\}]$ has a facial cycle C', we apply the induction hypothesis to the subgraph of G defined by $C' \cup \mathrm{int}(C')$.

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

Case 4. G has a facial walk $C' = x_1 x_2 \cdots x_k x_1$ such that $k \geq 6$ and at least one vertex of C' is uncolored, say x_1. As Case 2 is excluded we can assume that x_2 or x_k is uncolored, w.l.o.g. assume x_2 is 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

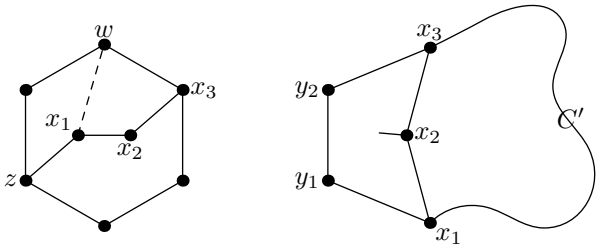

Fig. 2: Cases 4a (left) and 4b (right).

x_3 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. W.l.o.g. we can assume that C_1 is separating in G for otherwise $C_1 = C'$, $|C_1| = 6$, $|C_2| = 6$ and C_2 is separating. Hence we can apply Claim 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. Then C'' is separating because $\deg_G(x_2) \geq 3$, so we can use Claim 1.

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 $y_2 \notin V(C')$,

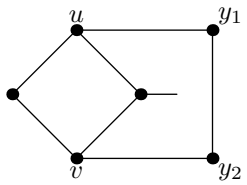

Fig. 3: Case 5a.

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 5, so we use Claim 1 again.

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 does not change, so it remains safe. Hence it suffices to apply induction to the resulting graph.

Case 6 (main reduction). Note that since Case 3 is exluded, every inner face is incident with at least one uncolored vertex. Hence, since cases 4 and 5 are excluded and G is triangle-free, every inner face of G has length 5. By Case 4, the outer face C is of length at most 6.

Now we claim G is biconnected. If v is a cutvertex in G then v is uncolored and there is a facial walk w that visits v at least two times. Since G contains no 1-vertices by Case 1 and G is triangle-free, w has length at least 8, which is a contradiction, because Case 4 is excluded. Hence indeed G is biconnected.

Moreover we observe that 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 the vertices x_i, $1 \leq i \leq 5$, 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 $G - C'$.

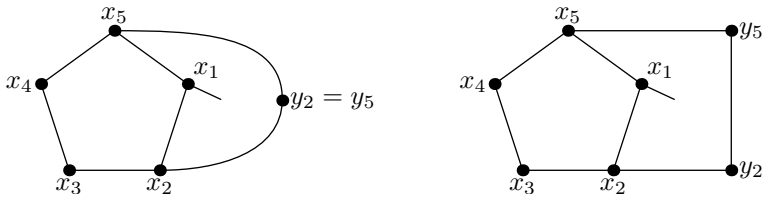

Fig. 4: Cases 6a (left) and 6b (right)

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 use Claim 1.

Case 6b. $y_iy_j \in E(G)$ for some $i \neq j$ (see Fig. 4). Then there is a separating cycle of length 4 or 5 in G and we use Claim 1.

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. Assume w.l.o.g. that x_i, x_j and x_k appear in this order around C'. Since Case 6b is excluded, the three pairs y_i and y_j, y_j and y_k, y_k and y_i are connected in the outer cycle C by paths of length at least 2. Since $|C| \leq 6$ these paths have all length exactly 2. Now observe that at least two of the vertices x_i, x_j, x_k are at distance 2 in C', say x_i and x_j. Then let C'' be the cycle consisting of the 2-path between x_i and x_j in C', edges x_iy_i and x_jy_j, and the 2-path between y_i and y_j in C. We see that $|C''| = 6$ and C'' is separating in G (as Case 4 is excluded), so we can apply Claim 1.

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).

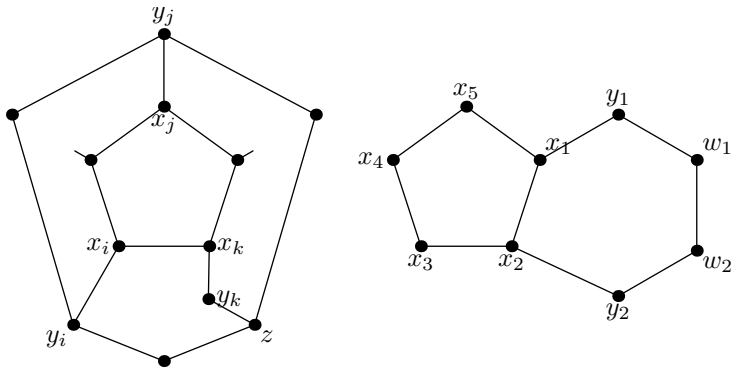

Fig. 5: Cases 6d (left) and 6h (right).

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). Consider the cycle C'' consisting of the path $y_i x_i x_k y_k z$ and of the path between y_i and z in C. Since y_i and z have the same color they are at distance at least 2 in the outer cycle C. It follows that $|C''| \geq 6$. Since Case 4 is excluded, C'' is separating and we can use Claim 1.

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. Then we swap the denotations of x_1 and x_4, y_1 and y_4, x_2 and x_3, y_2 and y_3. Note that then Case 6e does not apply any more and still y_1 or y_2 is uncolored (actually then both of them are uncolored).

Case 6f. y_3 is colored and x_5 has a colored neighbor. As Case 6c is excluded, y_1, y_2 and y_4 are uncolored. As 6d is excluded y_4 has no neighbor with the same color as y_3. Then we swap the denotations of x_1 and x_4, y_1 and y_4, x_2 and x_3, y_2 and y_3. Note that then neither Case 6e nor Case 6f applies and y_1 or y_2 is still uncolored.

Observe that after excluding Cases 6e and 6f one can identify y_1 with y_2 and x_5 with y_3 without introducing an edge with both ends of the same color. We are going to identify these pairs (additionally removing vertices x_1, x_2, x_3 and x_4), but earlier we need to exclude the situations when identifying introduces a triangle.

Case 6g. There is a path $x_5 y_k w y_3$ for some $k \in \{5, \ldots m\}$, $w \in V(G)$. Then we consider the 6-cycle $C' = y_k x_5 x_4 x_3 y_3 w y_k$. Since Case 4 is excluded, C' is separating and we use Claim 1.

Case 6h. There is a path $y_1 w_1 w_2 y_2$ distinct from $y_1 x_1 x_2 y_2$ (see Fig. 5). Then we consider the 6-cycle $C' = y_1 w_1 w_2 y_2 x_2 x_1 y_1$. Since Case 4 is excluded, C' is separating and we use Claim 1.

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. A planar embedding of G' is inherited from that of G. In both of these pairs at most one vertex is colored and the new vertex inherits its color from this vertex. Since we excluded cases 6e and 6f, the graph induced by the colored vertices of G' is properly 3-colored. Since we excluded cases 6g and 6h, G' is triangle-free. Hence we can use the 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. □

3 The Algorithm

The proof of Grötzsch's theorem presented in Section 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

3.1.1 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.

3.1.2 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 the *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 the *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.

3.1.3 Low Degree Vertices Queue

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

3.1.4 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 Section 4.

3.2 Recursion

Note that in the recursive algorithm induced by the proof given is Section 2 we need to split G into two parts. Then each of the parts is processed separately by a recursive call. 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 queue. 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 scratch. In Section 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 parts described in Case 2 is treated as two successive splits. Let us call the vertices of the separating cycle (or path) *outer vertices* and the remaining vertices from the smaller part A are called *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 $\mathcal{O}(n \log n)$. □

3.3 Non-trivial Operations

3.3.1 Identifying Vertices

In this section we describe how our algorithm updates the data structures described in Section 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 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 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 when we consider also edge deletions, any identify operation moves at most as many edges as when the deletions were ignored (although then a single edge can be moved even $\Omega(n)$ times). Hence, even with deletions allowed, the total number of delete/insert operations performed during IDENTIFY operation is $\mathcal{O}(n \log n)$. □

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 the affected faces is then placed in the appropriate face queue (if one has to be changed). In Section 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.

3.3.2 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 Section 4 paths of length 1 (edges) 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. Finding paths of length 3 is used in (recognizing) the cases 4b, 5a and 6h. (Note that recognizing case 6g can be done by checking at most $m \leq 5$ paths of length 2.) We can assume that we are given a path p of length 2 (cases 4b and 5a) or of length 3 (Case 6h) 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 by 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).

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.

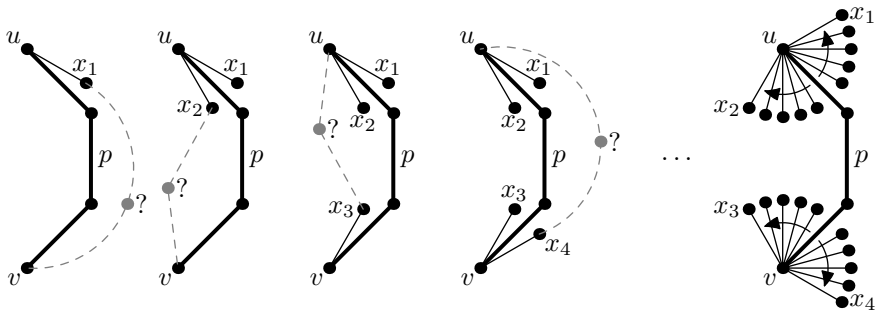

Fig. 6: Order of queries in algorithm PATH3.

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. Operation PATH3(u,v,p) 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 (when the condition of Case 4b does not hold, Case 4c applies and the relevant vertices are identified; similar situation appears in Case 5a and Case 6h). 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 which is the union of the path p and the 3-path that was found. Let H be the graph $C' \cup \mathrm{int}(C')$. 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 $\mathcal{O}(n \log n)$. □

3.3.3 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. To deal with it we use the well-known Union-Find algorithm (see e.g. [14]) for finding the 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.

3.3.4 Searching for Faces

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

3.3.5 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. This total 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 $\overrightarrow{G_1}$ and $\overrightarrow{G_2}$. We will describe them after introducing some basic notions.

A directed graph is said to be *k-oriented* if its every vertex has 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. $\overrightarrow{H}$ will denote a 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.

4.1.1 Graph $\vec{G_1}$ and Adjacency Queries

In this section G_1 denotes a planar graph for which we build a SPDS (recall from Section 3.2 that it is not always 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 $\overrightarrow{G_1}$ denote such an orientation of G_1. Then $xy \in E(G_1)$ iff $(x,y) \in E(\overrightarrow{G_1})$ or $(y,x) \in E(\overrightarrow{G_1})$. Hence, providing that we can maintain bounded out-degrees in $\overrightarrow{G_1}$ during our coloring algorithm, we can process in $\mathcal{O}(1)$ time queries of the form: "Are vertices x and y adjacent?".

4.1.2 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 a vertex $x \in V(\overrightarrow{G_1})$ such that $(x,v) \in E(\overrightarrow{G_1})$ and $(x,w) \in E(\overrightarrow{G_1})$. Vertex x is said to *support* edge vw. Since $\overrightarrow{G_1}$ 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 $\overrightarrow{G_1}$ be a directed planar graph with out-degree bounded by d. Let G_2 be an undirected graph with $V(G_2) = V(\overrightarrow{G_1})$ and $E(G_2) = \{vw : (x,v) \in E(\overrightarrow{G_1})$ and $(x,w) \in E(\overrightarrow{G_1})\}$. Then G_2 is the 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 $\overrightarrow{G_1}$. 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 $\overrightarrow{G_1}$, 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 $\overrightarrow{G_1}$. □

Corollary 4.2. *If the out-degree in graph $\overrightarrow{G_1}$ 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 $\overrightarrow{G_2}$ we will denote an $\mathcal{O}(1)$-orientation of G_2. Let e be an edge in $\overrightarrow{G_2}$ with ends v and w. Let x be a vertex that supports e and let $e_1 = (x,v)$ and $e_2 = (x,w)$, $e_1, e_2 \in E(\overrightarrow{G_1})$. 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 $e \in E(\overrightarrow{G_2})$ 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 $e \in E(\overrightarrow{G_1})$ a list $C(e)$ of pairs (c, p) where $c \in E(\overrightarrow{G_2})$ is a common child of e and a certain edge f and p is a pointer to $\{e, f\}$ in the list $P(c)$.

4.1.3 Queries About 2-paths

It is easy to see that when $\overrightarrow{G_1}$ and $\overrightarrow{G_2}$ 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 $\overrightarrow{G_1}$,

(ii) check whether there is a vertex x such that $(u, x), (v, x) \in E(\overrightarrow{G_1})$,

(iii) check whether there is an edge $e = (u, v)$ or $e = (v, u)$ in $\overrightarrow{G_2}$. 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 ← {w}
3:     while S ≠ ∅ do
4:         x ← POP(S)
5:         for all (x, y) ∈ E(H⃗) do
6:             Change the orientation of edge (x, y) to (y, x).
7:             if outdeg(y) = D + 1 then
8:                 PUSH(S, y)
```

4.2.1 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 $\overrightarrow{G_1}$ and $\overrightarrow{G_2}$. Assume that H is an arbitrary graph. Assume that we want to maintain a D-orientation of H, denoted by $\overrightarrow{H}$. 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 Brodal and Fagerberg [15]. As long as graph $\overrightarrow{H}$ 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 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 $\overrightarrow{G_1}$ and $\overrightarrow{G_2}$.

4.2.2 Updating the SPDS After a Deletion

Note that after deleting an edge from $\overrightarrow{G_1}$ we need to find out which edges in $\overrightarrow{G_2}$ should be deleted, if any. Assume that e is an edge of $\overrightarrow{G_1}$ 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 DeleteShortcut(e). Since both e and f have 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.*

4.2.3 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 $\overrightarrow{G_1}$. If then $\mathrm{outdeg}(u) = D+1$ we perform Reorient(u). Whenever any edge e in $\overrightarrow{G_1}$ changes its orientation we act as if it was deleted and we perform DeleteShortcut(e). Moreover, when any edge (u,v) appears in $\overrightarrow{G_1}$, both after Insert(u,v) and after reorienting (v,u), we add an edge (v,w) to $\overrightarrow{G_2}$ for each edge (u,w) present in $\overrightarrow{G_1}$. When we add an edge to $\overrightarrow{G_2}$ we also use Reorient if needed. We will refer to this routine as InsertShortcut(uv).

4.2.4 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 $\overrightarrow{G_1}$ and then $\overrightarrow{G_2}$. 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 $\overrightarrow{G_1}$ and perform Reorient if needed. After adding all edges we build graph $\overrightarrow{G_2}$. To this end, for each pair of edges $(x,u),(x,v)$ in graph $\overrightarrow{G_1}$ we add (u,v) to $\overrightarrow{G_2}$ and perform Reorient if needed. In what follows we will show that these two steps take only $\mathcal{O}(|V(G_1)|)$ time.

4.2.5 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[2]. We also join adjacency lists of u and v and store the resulting list in v. Clearly it takes $\mathcal{O}(\deg_{G_1} u)$ time. As a result it may happen that $\mathrm{outdeg}_{\overrightarrow{G_1}}(v)$ 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.

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

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 it 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 $\deg_{G_2}(u) \leq D_1 \cdot \deg_{G_1}(u) = \mathcal{O}(\deg_{G_1}(u))$. □

4.3 The Time Complexity of Building and 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 [15]. Their proof remains valid even for the modified formulation presented below.

Roughly, this lemma says that if we want to maintain a bounded outdegree orientation of a bounded arboricity dynamic graph, and we know that it *can be done* using r edge reorientations, then algorithm REORIENT performs only $O(k+r)$ edge reorientations, where k is the number of insertions.

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 the* i*th operation in sequence* σ *and let* k *denote the number of insertions in* σ*.*

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

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

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* $\overrightarrow{H}$ *in* $\mathcal{O}(|V(H)| + |E(H)|)$ *time by adding successive edges to* $\overrightarrow{H}$*, each time using algorithm* REORIENT *to keep outdegrees bounded.*

Proof. We will describe a sequence of a-orientations needed in Lemma 4.6. $\overrightarrow{H_{|\sigma|}}$ is an arbitrary a-orientation of $H_{|\sigma|}$. For each $i = 1, \ldots, |\sigma| - 1$ we get $\overrightarrow{H_i}$ from $\overrightarrow{H_{i+1}}$ by removing the relevant edge. Clearly there is not a single reorientation in this sequence. It suffices to apply Lemma 4.6 to finish the proof. We get that the total number of reorientations performed by REORIENT during all insertions is bounded by $(|E(H)| + 0)\frac{3a-1+1}{3a-1+1-2a} = 3|E(H)|$. □

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

Proof. As it was mentioned in Section 4.2.4 we first build $\overrightarrow{G_1}$, which takes $O(|E(G_1)|)$ time by Lemma 4.7, which can be bounded by $O(|V(G_1)|)$, since G_1 is planar. Then we build $\overrightarrow{G_2}$, also using Lemma 4.7. Note that since $\overrightarrow{G_1}$ is $\mathcal{O}(1)$-oriented, $\overrightarrow{G_2}$ has $O(|V(G_1)|)$ edges, hence the corollary follows. □

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 $\overrightarrow{G_1}$ used by* Reorient *algorithm is $\mathcal{O}(k)$.*

Proof. Let G^i be the graph G_1 after the ith operation. We will construct a sequence of 6-orientations $\mathcal{G} = \overrightarrow{G^1}, \overrightarrow{G^2}, \ldots, \overrightarrow{G^{|\sigma|}}$. The last graph, $G^{|\sigma|}$, is the graph at the bottom level of the recursion (the SPDS is not needed when the algorithm returns from recursive calls, so it is not updated any more). It follows from the proof of Theorem 2.2 that $G^{|\sigma|}$ has no uncolored vertices. Hence it contains at most 6 vertices and it is trivial to find its 6-orientation. For each $i = 1, \ldots, t-1$ we will describe $\overrightarrow{G^i}$ using $\overrightarrow{G^{i+1}}$.

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

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

Now assume that σ_i is an edge deletion. First let us consider the case when it is not a deletion involved with a Cut operation. Recall from the proof of Theorem 2.2 (cases 1 and 6i) that such a deletion is caused by removing a vertex of degree at most 3. Denote this vertex by x. Hence to get $\overrightarrow{G^i}$ it suffices to copy $\overrightarrow{G^{i+1}}$ and add an edge leaving x.

Finally let us consider a sequence of edge deletions caused by a 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 exists its 3-orientation $\overrightarrow{A}$. 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 and A is bounded by a cycle in G^j of length at most 6 (see Fig 7). Clearly, the resulting graph $\overrightarrow{G^i}$ is 6-oriented. For each $t = i, \ldots, j-2$ graph $\overrightarrow{G^{t+1}}$ is obtained from $\overrightarrow{G^t}$ by deleting a successive edge.

Thus we have described a sequence of 6-orientations $\overrightarrow{G^1}, \ldots, \overrightarrow{G^{|\sigma|}}$. Observe that there is not a single edge reorientation in this sequence. Let $\overrightarrow{G^0}$ be the

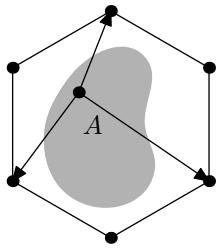

Fig. 7: Vertices of graph A are adjacent to at most 3 vertices of G^j.

initial orientation of graph G_1. By Lemma 4.7 $\overrightarrow{G^0}$ is 8-oriented. Now we consider the sequence $\overrightarrow{G^0}, \ldots, \overrightarrow{G^{|\sigma|}}$. Orientations $\overrightarrow{G^0}$ and $\overrightarrow{G^1}$ can differ very much. In order to avoid this situation we modify the orientations $\overrightarrow{G^1}, \ldots, \overrightarrow{G^{|\sigma|}}$. For each $i = 1, \ldots, |\sigma|$ each edge from graph G^i which is also present in G^0 is oriented exactly like in $\overrightarrow{G^0}$. Thus we obtain a sequence of (6+8)-orientations $\overrightarrow{G^0}, \ldots, \overrightarrow{G^{|\sigma|}}$. 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 $\mathcal{O}(k)$. □

Lemma 4.10 (see Lemma 2 in [15])**.** *Let $H = (V, E)$ be a graph with arboricity at most a and let $\overrightarrow{H}$ be an orientation of H. If $u \in V(\overrightarrow{H})$ 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 $\overrightarrow{H}$.*

Lemma 4.11. *Let H be an arbitrary n-vertex graph of arboricity a and let $\overrightarrow{H}$ be its initial δ-orientation, $\delta \geq 2a$. For an 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 a $(3\delta - 1)$-orientation, performing $\mathcal{O}((p + r\delta) \log n)$ edge reorientations.*

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

For each $i = 1, \ldots, p + q + r$ we get $\overrightarrow{H_i}$ from $\overrightarrow{H_{i-1}}$ as follows. If the i-th operation is delete we simply remove the relevant edge from $\overrightarrow{H_{i-1}}$. If the i-th operation is an insertion of an edge uv, we add edge (u, v) to $\overrightarrow{H_{i-1}}$. Then if the outdegree of u is larger than δ we pick the path described in Lemma 4.10 and reverse all edges in this path. Clearly the resulting graph is δ-oriented and we choose it as $\overrightarrow{H_i}$. Finally, if the i-th operation is identifying vertices u and v we start from identifying u and v in graph $\overrightarrow{H_{i-1}}$. 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 reversing the edges of the relevant path. The

resulting graph is δ-oriented and we choose it as $\overrightarrow{H_i}$. 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. □

Corollary 4.12. *The total number of reorientations in $\overrightarrow{G_2}$, 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 $\overrightarrow{G_2}$ 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 $\overrightarrow{G_1}$. 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 $\overrightarrow{G_2}$ is bounded by $\mathcal{O}(t \log n)$. □

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 in updating the Short Path Data Structures is $\mathcal{O}(n \log n)$.*

5 Conclusions and Final Remarks

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. Very recently, such an algorithm was presented by Dvořák, Kawarabayashi and Thomas [16].

Acknowledgments I would like to thank Krzysztof Diks and anonymous referees for reading this paper carefully and numerous helpful comments and suggestions. 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] Norishige Chiba, Takao Nishizeki, and Nobuji Saito. A linear algorithm for five-coloring a planar graph. *J. Algorithms*, 2:317–327, 1981.

[4] Michael R. Garey and David S. Johnson. Some simplified NP-complete graph problems. *Theoretical Computer Science*, 1(3):237–267, February 1976.

[5] Herbert Grötzsch. Ein dreifarbensatz fur dreikreisfreie netze auf der kuzel. 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. Oracles for bounded-length shortest paths in planar graphs. *ACM Trans. Algorithms*, 2(3):335–363, 2006.

[9] Oleg V. Borodin and André Raspaud. A sufficient condition for planar graphs to be 3-colorable. *Journal of Combinatorial Theory, Series B*, 88:17–27, 2003.

[10] Oleg V. Borodin, Alexei N. Glebov, André Raspaud, and Mohammad R. Salavatipour. Planar graphs without cycles of length from 4 to 7 are 3-colorable. *J. Comb. Theory, Ser. B*, 93(2):303–311, 2005.

[11] Zdenek Dvorak, Daniel Král, and Robin Thomas. Coloring triangle-free graphs on surfaces. In *Algorithms and Computation, 18th International Symposium, ISAAC 2007, Sendai, Japan, December 17-19, 2007, Proceedings*, volume 4835 of *LNCS*, pages 2–4, 2007.

[12] Douglas West. *Introduction to Graph Theory*. Prentice Hall, 1996.

[13] John Gimbel and Carsten Thomassen. Coloring graphs with fixed genus and girth. *Transactions of the AMS*, 349(11):4555–4564, November 1997.

[14] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT, 2001.

[15] 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.

[16] Z. Dvořák, K. Kawarabayashi, R. Thomas, *Three-coloring triangle-free planar graphs in linear time*, SODA '09: Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms, 2009, 1176–1182.