

Fast 3-coloring Triangle-Free Planar Graphs

Łukasz Kowalik*

Institute of Informatics, Warsaw University
Banacha 2, 02-097 Warsaw, Poland
kowalik@mimuw.edu.pl

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.

A remarkable ingredient of our algorithm is the data structure processing short path queries introduced recently in [9]. In this paper we show how to adapt it to the fully dynamic environment where edge insertions and deletions are allowed.

1 Introduction

The famous Four-Color Theorem says that every planar graph is vertex 4-colorable. The paper of Robertson et al. [10] 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. [4]). 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 [6] the renowned Grötzsch's Theorem [8] 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 [11, 12]) can be easily transformed into $\mathcal{O}(n^2)$ algorithms. In this paper we improve this bound to $\mathcal{O}(n \log n)$.

In § 2 we present a new proof of Grötzsch's Theorem, based on a paper of Thomassen [11]. 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 [9] 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

* 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.

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 [9]. 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 [9].

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 [2]). The other theorem states that planar graphs without cycles of length from 4 to 7 are 3-colorable (see [1]). 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. [13]). 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 [11]. 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 [12] but we suspect that the resulting algorithm would be more complicated and harder to describe. We will need a following lemma that can be easily proved using discharging technique. Due to space limitations we omit the proof.

Lemma 1. *Let G be a biconnected plane graph with every inner face of length at least 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.*

Instead of proving Grötzsch's Theorem it will be easier for us to show the following more general result. (Let us note that it follows also from Theorem 5.3 in [7]). A 3-co-

loring 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 1. *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. 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.

Case 2. G has a vertex x joined to two or three colored vertices. 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 . As Case 2 is excluded, x_2 and x_1 have no colored neighbors, except for x_3 and z respectively. Then $G[V(C) \cup \{z, x_1, x_2\}]$ 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 $\deg_G(x_2) \geq 3$. By the induction hypothesis, the 3-coloring of C can be extended 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_1v or uy_1y_2v , $y_1 \notin V(C')$. Then $y_2 \notin V(C')$, 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 or a multiple edge. Thus it suffices to apply induction to the resulting graph. Observe that when C was colored and $|C| = 6$ then the coloring of the outer cycle remains safe.

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 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 $G - C'$.

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' . Then there is a 4-cycle $x_iy_ix_jy_jx_i$ 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 $i \neq j$. 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 . 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).

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 $y_1x_1x_2y_2$. 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 if $|C'| = 6$ 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 by our algorithm. Then we discuss how to use recursion efficiently and how to implement some 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 adjacent to uv . A face is called *right face adjacent 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 adjacent to* (u, v) . Each face f is stored as the corresponding facial walk, i. e. a list of pointers to successive adjacency lists elements corresponding to the edges of the walk. For each such element e corresponding to neighbor v of vertex u , face f is the right face adjacent 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 1, 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 $V(A)$ 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 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 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 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 succeeding element and x_2 to the preceding element in $A(u)$. Similarly, we assign x_3 to the succeeding element and x_4 to the preceding element in $A(v)$. 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. *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 $\text{Identify}(u,v)$ operation, the total time spent on performing failed Path3 operations is linear in the number of edge deletions caused by identifying vertices. By Lemma 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 $4 \cdot \deg_H(v)$. Note that since v will be colored in graph H 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 (an edge xv added after deleting xu during $\text{Identify}(u,v)$ is not counted as a new one here). Thus the time used by successful operations is $\mathcal{O}(n)$. \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. [5]) for finding a right face adjacent to a given edge. The amortized time of the search is $\mathcal{O}(\log^* n)$. 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 (proof is omitted due to space limitations):

Lemma 4. *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.*

Corollary 1. *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 2. *Graph G_2 can be $\mathcal{O}(1)$ -oriented.*

Corollary 1 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 $P(e)$.

To build the shortcut graph data structure for a given graph G_1 , namely graphs \vec{G}_1 and \vec{G}_2 , we start from creating two graphs containing the same vertices as G_1 but no edges. Then for every edge uv from G_1 we add uv to SPDS using insertion algorithm described in the following section. In § 4.3 we show that it takes only $\mathcal{O}(|V(G_1)|)$ time.

4.2 Inserting and Deleting Edges

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 . While removing is easy, after adding an edge we may need to reorient some edges to leave the graph $\mathcal{O}(1)$ -oriented. We use the approach of G. Brodal and R. Fagerberg [3]. Assume that H is an arbitrary graph of arboricity a . Let $\Delta = 6a - 1$ and assume that we want to maintain a Δ -orientation of G , denoted by \vec{H} . They consider the following routine for inserting an edge uv . First add (u, v) to \vec{H} . If $\text{outdeg } u = \Delta + 1$, repeatedly a node w with out-degree larger than Δ is picked, and the orientation of all the edges outgoing from w is changed. Deleting an edge does not need any reorientations. We will refer to these routines as `Insert` (u, v) and `Delete` (u, v) respectively and we will use them for inserting and deleting edges in \vec{G}_1 and \vec{G}_2 . Observe that the out-degrees in \vec{G}_1 will be bounded by 17 since it has arboricity 3 as a planar graph. Subsequently Corollary 1 guarantees that \vec{G}_2 has bounded arboricity and hence it will be also $\mathcal{O}(1)$ -oriented.

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 . Since e has at most $d = \mathcal{O}(1)$ children, the following proposition holds:

Proposition 2. *After deletion of an edge the SPDS can be updated in $\mathcal{O}(1)$ time.*

Updating the SPDS After an Insertion To perform insertion we first call `Insert` in graph \vec{G}_1 . Whenever any edge in \vec{G}_1 changes its orientation we act as if it was deleted and update \vec{G}_2 as described above. Moreover, when any edge (u, v) appears in \vec{G}_1 , both after `Insert` (u, v) and after reorienting (v, u) , we add an edge vw to G_2 for each edge (u, w) present in \vec{G}_1 .

4.3 Time Complexity of Updating the SPDS

There are four operations performed by our coloring algorithm on an input graph. First two are insertions and `Identify` operations. The third one is deleting a vertex of degree at most 3 and will be denoted as `DeleteVertex`. The last operation is `Cut` described in § 3.2. These four operations will be called *meta-operations*. Each of them will be

performed using a sequence of deletions and insertions. In this section we show that for the particular sequence of meta-operations 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 [3]. Their proof remains valid even for the modified formulation presented below.

Lemma 5. *Given an arboricity a preserving sequence σ of edge insertions and deletions on an initially empty graph, let H_i be the graph after the i -th operation.*

Let $\vec{H}_1, \vec{H}_2, \dots, \vec{H}_{|\sigma|}$ be any sequence of (2a)-orientations with at most r edge reorientations in total and let $\vec{F}_1, \vec{F}_2, \dots, \vec{F}_{|\sigma|}$ be the Δ -orientations of $H_1, H_2, \dots, H_{|\sigma|}$ appearing when Insert and Delete algorithms are used to perform σ . Let k be the number of edges uv such that $(u, v) \in \vec{F}_i$, $(v, u) \in \vec{H}_i$ and the i -th operation in σ is insertion of uv .

Then the Insert and Delete algorithms perform at most $3(k + r)$ edge reorientations in total on the sequence σ .

Lemma 6. *The total number of reorientations in \vec{G}_1 used by Insert and Delete algorithms to perform a sequence of k meta-operations leading to the empty graph is $\mathcal{O}(k)$.*

Proof. Let us denote the sequence of meta-operations by σ . Let G^i be the graph G_1 after the i -th meta-operation. We will construct a sequence of 6-orientations $\mathcal{G} = \vec{G}^1, \vec{G}^2, \dots, \vec{G}^k$. Observe that \vec{G}^k has no edges thus it is 6-oriented. We will describe \vec{G}^i using \vec{G}^{i+1} . If σ_i is an insertion of edge uv , \vec{G}^i is obtained from \vec{G}^{i+1} by deleting uv . If $\sigma_i = \text{Identify}(u, v)$ 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 . Thus $\text{outdeg}_{\vec{G}^i} u \leq \text{outdeg}_{\vec{G}^i} x$ and $\text{outdeg}_{\vec{G}^i} v \leq \text{outdeg}_{\vec{G}^i} x$.

If $\sigma_i = \text{DeleteVertex}$ we simply add a vertex of degree at most 3 to \vec{G}^i in such a way that the added edges leave the new vertex. Its out-degree is 3 and the out-degrees of other vertices do not change. Finally we consider the case when σ_i is Cut operation. Let A be the graph induced by the vertices removed from G^i . As A is planar it can be 3-oriented. The remaining edges, joining a vertex from A , say x , with a vertex from G^{i+1} , say y are oriented from x to y . Observe that a vertex in A can be adjacent with at most 3 vertices in G^{i+1} , since there are no triangles. Thus \vec{G}^{i+1} is 6-oriented. Description of the sequence \mathcal{G} of orientations is finished now. Observe that there is no single reorientation in this sequence.

Now let us compare the sequence \mathcal{G} with the sequence \mathcal{F} of 17-orientations appearing when Insert and Delete algorithms are used to perform σ . Consider insertions involved with a single Identify operation. Then at most $6 + 17 = \mathcal{O}(1)$ inserted edges obtain different orientations in \mathcal{G} and \mathcal{F} . Hence the total number of such insertions involved with identifying vertices is $\mathcal{O}(k)$. Operations DeleteVertex and Cut do not use insertions and trivially there are at most k ordinary insertions in σ . Thus by Lemma 5 the Insert and Delete algorithms perform $\mathcal{O}(k)$ reorientations in \vec{G}_1 to perform σ . \square

Lemma 7 (proof omitted). *Let σ be a sequence of $k + l$ meta-operations performed on an initially empty planar graph $G_1 = (V, E)$. Moreover, let the first k operations*

in σ be insertions. Then the total number of reorientations in $\overrightarrow{G_2}$ used by Insert and Delete algorithms to perform σ is $\mathcal{O}(l \log |V|)$.

Corollary 3. For any planar graph G_1 the Short Path Data Structure can be constructed in $\mathcal{O}(|V(G_1)|)$ time.

Corollary 4. Let n be the number of vertices of the graph on the input of the coloring algorithm. The total time needed to perform all meta-operations executed by the coloring algorithm, including updating the Short Path Data Structure, is $\mathcal{O}(n \log n)$.

Proof. By Lemma 2 all Identify operations cause $\mathcal{O}(n \log n)$ pairs of deletions/insertions. By Proposition 1 all Cut operations cause $\mathcal{O}(n \log n)$ deletions. There is $\mathcal{O}(n)$ meta-operations performed. Hence, by Lemmas 6 and 7 the total number of reorientations needed by insert operations is $\mathcal{O}(n \log n)$. It ends the proof. \square

Let us also note that using techniques from proof of Lemma 7 one can show that update of SPDS after an insertion in *arbitrary* sequence of insert/delete operations takes amortized $\mathcal{O}(\log^2 n)$ time. The approach presented here can be extended to the general version of SPDS. It needs $\mathcal{O}(1)$ time to find a shortest path between vertices at distance bounded by a constant k and updates after an insertion in $\mathcal{O}(\log^k n)$ amortized time.

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. 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.
2. 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.
3. G. S. Brodal and R. Fagerberg. Dynamic representations of sparse graphs. In *Proc. 6th Int. Workshop on Algorithms and Data Structures*, volume 1663 of *LNCS*, pages 342–351. 1999.
4. N. Chiba, T. Nishizeki, and N. Saito. A linear algorithm for five-coloring a planar graph. *J. Algorithms*, 2:317–327, 1981.
5. T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT, 2001.
6. M. R. Garey and D. S. Johnson. Some simplified NP-complete graph problems. *Theoretical Computer Science*, 1(3):237–267, February 1976.
7. J. Gimbel and C. Thomassen. Coloring graphs with fixed genus and girth. *Transactions of the AMS*, 349(11):4555–4564, November 1997.
8. H. Grötzsch. Ein dreifarbensatz für dreikreisfreie netze auf der kugel. Technical report, Wiss. Z. Martin Luther Univ. Halle Wittenberg, Math.-Nat. Reihe 8, 1959.
9. Ł. Kowalik and M. Kurowski. Shortest path queries in planar graphs in constant time. In *Proc. 35th Symposium Theory of Computing*, pages 143–148. ACM, June 2003.
10. N. Robertson, D. P. Sanders, P. Seymour, and R. Thomas. Efficiently four-coloring planar graphs. In *Proc. 28th Symposium on Theory of Computing*, pages 571–575. ACM, 1996.
11. C. 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.
12. C. Thomassen. A short list color proof of Grötzsch's theorem. *Journal of Combinatorial Theory, Series B*, 88:189–192, 2003.
13. D. West. *Introduction to Graph Theory*. Prentice Hall, 1996.