Oracles for Bounded Length Shortest Paths in Planar Graphs

ŁUKASZ KOWALIK and MACIEJ KUROWSKI
Warsaw University, Warsaw, Poland

We present a new approach for answering short path queries in planar graphs. For any fixed constant k and a given unweighted planar graph $G = (V, E)$ one can build in $\mathcal{O}(|V|)$ time a data structure, which allows to check in $\mathcal{O}(1)$ time whether two given vertices are at distance at most k in G and if so a shortest path between them is returned. Graph G can be undirected as well as directed.

Our data structure works in fully dynamic environment. It can be updated in $\mathcal{O}(1)$ time after removing an edge or a vertex while updating after an edge insertion takes polylogarithmic amortized time. Besides deleting elements one can also disable ones for some time. It is motivated by a practical situation where nodes or links of a network may be temporarily out of service.

Our results can be easily generalized to other wide classes of graphs – for instance we can take any minor-closed family of graphs.

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Computations on discrete structures*; G.2.2 [**Mathematics of Computing**]: Discrete Mathematics—*Graph Algorithms*

General Terms: Algorithms, Theory

Additional Key Words and Phrases: Bounded length, dynamic environment, oracle, planar graph, shortest path

1. INTRODUCTION

Computing shortest paths in planar graphs is a fundamental problem with numerous practical applications – for an extensive survey see Ahuja et al. [1993], Ramalingam [1996]. It is also often used as a black box in other graph algorithms. In this case the complexity of computing the shortest paths has often a crucial influence on the overall complexity.

Recently the following problem attracts a lot of attention: for a given n-vertex planar graph, construct a data structure which allows to process quickly queries

A preliminary version of this paper, entitled "Short path queries in planar graphs in constant time" appeared in the *Proceedings of the 35th ACM Symposium on Theory of Computing (STOC'03)*. 2004, pp. 143–148.

The research has been supported by KBN grant 4T11C04425 and by a Marie Curie Fellowship of the European Community Programme Improving the Human Research Potential and the Socioeconomic Knowledge Base under contract number HPMT-CT-2000-00093.

Author's addresses: Ł. Kowalik, M. Kurowski, Institute of Informatics, Warsaw University, Banacha 2, 02-097 Warsaw, Poland, e-mail: {kowalik, kuros}@mimuw.edu.pl.

concerning shortest paths between vertices. Results in this area are often given in a form of a trade-off between the preprocessing space (or time) and the time needed to answer the shortest path query. For instance Djidjev [1996] showed a data structure which takes $\mathcal{O}(n^{4/3})$ space and supports queries in $\mathcal{O}(n^{1/3}\log n)$ time. When query time is constant or close to constant the data structure constructed in the preprocessing phase is called an *oracle*.

1.1 New Results

A Combinatorial Result. The core of our paper is a new combinatorial result on planar graphs describing the structure of bounded length shortest paths in these graphs. Roughly, we show that for any constant k we can orient edges of a planar graph, biorienting some of the edges, so that each vertex has bounded outdegree, and if in the original graph there is a path of length $l \leq k$ between vertices u and v then in the oriented graph there are two directed paths, first from u to some vertex x and the second from v to x such that the lengths of these two paths sum up to l. (See Theorem 5.9 for a precise formulation). We use this and similar results to design efficient oracles for bounded length shortest paths.

Shortest Paths. We study the following version of the shortest path problem. We fix a constant integer k. For an undirected unweighted planar graph G (without given plane embedding) we describe a preprocessing which takes time and space $\mathcal{O}(n)$. Then we provide a query algorithm which checks in $\mathcal{O}(1)$ time whether two given vertices are at distance at most k and if so it computes a corresponding shortest path. The query algorithm searches for paths using a data structure computed in the preprocessing phase. Our oracle can be updated in $\mathcal{O}(1)$ time after deleting any edge or any vertex of G. Moreover, updating the oracle after edge insertion takes polylogarithmic amortized time, namely $\mathcal{O}(\log^k n)$ time, thus it can work in the fully dynamic environment. Additionally, one can enable and disable edges and vertices of G in constant time. The query algorithm acts as the disabled elements were deleted from the graph. It models a real-life situation where nodes or links of some network may be out of service for some time.

The Girth. The *girth* of a graph G is the length of the shortest cycle in G. For any fixed constant k our approach can be used to construct a linear time algorithm which verifies whether a given graph has girth at most k and if so it computes the corresponding cycle.

Given Length Paths and Cycles. We also show a constant time algorithm for another natural type of queries. For two given vertices u and v and an integer $t \leq k$ the query returns a simple u,v-path of length exactly t or reports that there is no such a path in graph G. As we can put $u = v$ it can be also used to find a given length cycle containing u and further to find a cycle of given length in planar graph in linear time.

Generalizations. We would like to emphasize that we consider undirected unweighted input graphs merely for the sake of simplicity. We also decided to focus on the family of planar graphs because it is widely recognized and particularly important in applications. Nevertheless our results can be further generalized. Our

oracle can be easily extended to service directed graphs and some weighted graphs (weights are positive integers). Moreover, the family of planar graphs can be replaced by any class $\mathcal{C}$ of graphs satisfying the following three conditions:

(i) Graphs from $\mathcal{C}$ are sparse, i.e. there exists a constant c such that for every $G \in \mathcal{C}$, $|E(G)| \leq c|V(G)|$,

(ii) $\mathcal{C}$ is closed under taking subgraphs,

(iii) If a graph G from $\mathcal{C}$ is a subdivision of a graph H, then $H \in \mathcal{C}$.

We show that every class of graphs closed under taking minors satisfies the above conditions. It follows that for arbitrary constant number g the class of graphs embeddable in a surface of genus g satisfies our conditions. Another example of a class closed under taking minors is the class the graphs of bounded tree-width.

1.2 Related Work

Shortest Paths Oracles. There is a number of results concerning general distance queries in weighted planar digraphs. Lipton and Tarjan [1979], Frederickson [1987], Arkati et al. [1996] and Djidjev [1996] have shown that after preprocessing that uses space s the distance queries can be answered in $\mathcal{O}(n^2/s)$ time. For space $s \in [n^{4/3}, n^{3/2}]$ Djidjev [1996] improved query time to $\mathcal{O}(n/\sqrt{s}\log n)$. Further improvements were presented by Chen and Xu [2000] but still it is not known whether it is possible to achieve near-linear preprocessing space/time with near-constant query time.

The situation gets better when we consider approximate distances. Mikkel Thorup [2001] presented an oracle answering $(1+\varepsilon)$-approximate distance queries in planar graphs. For undirected graphs it takes $\mathcal{O}(\varepsilon^{-1}n\log n)$ space and the queries are processed in $\mathcal{O}(\varepsilon^{-1})$ time. For digraphs his oracle has $\mathcal{O}(\varepsilon^{-1}n\log n\log\Delta)$ space and $\mathcal{O}(\log\log\Delta+\varepsilon^{-1})$ query time, where Δ is the largest finite distance in the graph and weights of edges are non-negative integers. This oracle can be used to process queries of our type. In order to get exact answer for vertices at distance at most k it suffices to put $\varepsilon = k^{-1}$. For every constant k the result of Thorup yields an oracle for undirected planar graphs with $\mathcal{O}(n\log n)$ oracle space and constant query time. For planar digraphs the bounds for oracle space and query time increase to $\mathcal{O}(n\log n\log\Delta)$ and $\mathcal{O}(\log\log\Delta)$ respectively. Similar results, but only for undirected planar graphs, were discovered independently by P. Klein [2002].

The shortest path problem addressed in our paper was considered previously by David Eppstein [1999]. He applied methods developed for subgraph isomorphism problem. His approach uses also a linear preprocessing but the queries are answered in $\mathcal{O}(\log n)$ time. The algorithm of Eppstein combines methods of tree decomposition with clustering techniques of Frederickson [1997]. In the same paper, Eppstein presents another algorithm, based on tree decomposition technique, which uses $\mathcal{O}(n\log n)$ time preprocessing and the queries are processed in $\mathcal{O}(1)$ time. Both algorithms of Eppstein need a plane embedding of the input graph.

Shortest Paths Avoiding Failure Elements. The paper of Demetrescu and Thorup [2002] describes an oracle answering the queries of the form: "What is the length of a shortest path from vertex x to vertex y avoiding a failed link (u, v) and what edge leaving x should be used to get on such a shortest path?". The

oracle works for general graphs, takes space $\mathcal{O}(n^2 \log n)$ and queries are answered in $\mathcal{O}(\log n)$. Thus, for integer edge weights, the shortest path of length k can be computed using at most k queries in $\mathcal{O}(k \log n)$ time. The structure allows only *one* failure edge. Thus, as soon as a failure in a graph is noticed the oracle returns shortest paths avoiding it but then the structure has to be computed from the scratch in the background.

Finding Given Length Cycles and Computing the Girth. The problem of finding cycles of specified lengths in a planar graphs attracted many researchers. There was a number of linear algorithms for finding cycles of particular lengths from 3 to 6, see Papadimitriou and Yannakakis [1981], Chiba and Nishizeki [1985], Chrobak and Eppstein [1991], Richards [1986], Kowalik [2003]. Alon et al. [1997] designed an algorithm for finding cycles of arbitrary fixed length working in $\mathcal{O}(n)$ expected time or $\mathcal{O}(n \log n)$ worst case time. Finally, Eppstein [1999] presented a linear algorithm for finding any fixed pattern in a planar graph. His results can be also used to compute the girth of a bounded value in $\mathcal{O}(n)$ time. If the girth is not bounded the problem seems to be harder – the best known algorithm is due to H. Djidjev [2000] and runs in $\mathcal{O}(n^{5/4} \log n)$ time.

2. BASIC NOTIONS AND NOTATION

We start with some basic definitions. A plane graph is a planar graph $G = (V, E)$ together with its fixed planar embedding. A *path* is a non-empty graph $P = (V, E)$ with vertices $V = \{x_0, \dots x_t\}$ and edges $E = \{x_0x_1, x_1x_2, \dots, x_{t-1}x_t\}$ where the x_i are all distinct. The vertices $x_1, \dots x_{t-1}$ are called *inner* vertices of P. We also say that P *joins* vertices x_0, x_k and we call path P an x_0, x_k-path. When $x_0 = x_t$ such a path is called a *cycle*. A *walk* in a graph G is a nonempty alternating sequence $v_0e_0v_1e_1 \dots e_{t-1}v_t$ of vertices and edges in G such that $e_i = v_iv_{i+1}$ for all $i = 0, \dots t-1$. Note that in a walk the same vertices may occur many times. The vertices $v_1, \dots v_{t-1}$ are called *inner* vertices of the walk (observe that v_0 or v_t can be inner when walk visits them many times). If all the vertices of a walk are distinct, it corresponds to a path in G. Sometimes, instead of path we will say *simple path* to emphasize that the vertices are visited only once.

A multigraph in which edges are assigned directions is called a *directed graph* (or *digraph* in short). Then (u, v) denotes an edge that leaves u and enters v. A *directed path* (or *dipath* in short) is a is a non-empty digraph $P = (V, E)$ with vertices $V = \{x_0, \dots x_t\}$ and edges $E = \{(x_0, x_1), (x_1, x_2), \dots, (x_{t-1}, x_t)\}$ where the x_i are all distinct. A directed walk is also defined similarly as in undirected case but now each edge e_i leaves v_i and enters v_{i+1}.

For a vertex v in a digraph G we define the *outdegree* as the number of edges leaving v and the *indegree* as the number of edges entering v. We denote these values by $\text{indeg}_G(v)$ and $\text{outdeg}_G(v)$ respectively. A digraph is *d-oriented* when the outdegree of each vertex is bounded by d. We say that a graph is $\mathcal{O}(1)$-oriented when it is d-oriented for some $d = \mathcal{O}(1)$. Let G be a undirected graph and let H be a directed graph. Assume that H can be obtained from G by replacing each edge uv of G by either (u, v) or (v, u) or the pair (u, v) and (v, u). We say that digraph H is a *biorientation* of graph G. When a biorientation does not contain a 2-cycle we call it an *orientation*. Let H be an orientation of G. When H is d-oriented

we call it *d-orientation of* G and we say that G *can be d-oriented*. $\overrightarrow{G}$ will denote certain orientation of a graph G. In our proofs we will need the following simple fact bounding the number of colors needed to color a d-oriented graph.

LEMMA 2.1. *Every d-oriented graph can be vertex* $(2d+1)$*-colored in linear time.*

PROOF. Let G be an arbitrary d-oriented graph. We use induction on the number of vertices. Since $\sum_v \text{indeg}(v) = \sum_v \text{outdeg}(v)$ there exists in G a vertex v such that $\text{indeg}(v) \leq \text{outdeg}(v)$. Thus $\deg(v) \leq 2d$. We can color inductively $G - \{v\}$ and assign to v a free color. □

3. KEY IDEAS

It is widely known that for any simple planar graph G we can find its 5-orientation $\overrightarrow{G_1}$ (i.e. orientation with outdegrees at most 5) using the following simple algorithm. While G is not empty pick a vertex v of degree at most 5 (it always exists), orient every incident edge $vw \in G$ as (v, w) in $\overrightarrow{G_1}$ and remove v from G. Then $xy \in E(G)$ iff $(x, y) \in E(\overrightarrow{G_1})$ or $(y, x) \in E(\overrightarrow{G_1})$. Thus after a linear preprocessing we can process the queries of the form:"Are vertices x and y adjacent?" in $\mathcal{O}(1)$ time.

We generalize this result to much wider class of queries, i.e. we can check in $\mathcal{O}(1)$ time whether given vertices are at distance at most k and if so we can compute a shortest path between them.

Now we focus on the case $k = 2$ which reveals some key ideas of our approach. The general case is considered in the latter sections. Let v, w be vertices at distance 2 in G. Consider a path p of length 2 between v and w. Edges of p are oriented in $\overrightarrow{G_1}$ in one of three possible ways excluding the symmetrical one (see Fig. 1).

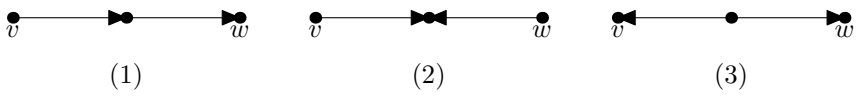

Fig. 1. Possible cases of orienting edges of a 2-path.

As one can see the cases (1) and (2) are easy to detect in $\mathcal{O}(1)$ time. The main obstacle is to detect the last case because the indegrees of v and w can be arbitrarily large. In order to resolve this problem we extend G by adding edges of weight 2 between each pair of vertices v and w which match the third case. The graph compound of edges with weight 2 is denoted by G_2. Formally $vw \in G_2$ iff there exists a vertex $x \in V(\overrightarrow{G_1})$ such that $(x, v) \in E(\overrightarrow{G_1})$ and $(x, w) \in E(\overrightarrow{G_1})$. The vertex x is said to *support* the edge vw (See Fig. 2). Since $\overrightarrow{G_1}$ has bounded outdegree every vertex supports $\mathcal{O}(1)$ edges in G_2. Consequently G_2 can be constructed in linear time. All we need is to verify adjacency in G_2 in constant time. Although G_2 is not necessarily a planar graph we show how to compute a decomposition of G_2 into $\mathcal{O}(1)$ planar graphs.

To this end we start with 5-coloring of the vertices of G and then we partition graph G_2 into 5 subgraphs. Each of them contains edges supported by vertices from the same color class. Observe that for each vertex v there are at most $\binom{5}{2}$ edges in

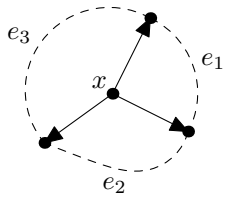

Fig. 2. Edges $e_1, e_2, e_3 \in E(G_2)$ are supported by v.

G_2 supported by v. Each of the 5 subgraphs is further partitioned into $\binom{5}{2}$ plane graphs in such a way that none of them contains two edges supported by the same vertex. Hence we partitioned G_2 into $5 \cdot \binom{5}{2} = \mathcal{O}(1)$ graphs. Moreover each of these graphs is a *subdrawing* of G: it can be embedded in the plane in such a way that its embedding is completely contained in the embedding of G. In another words their edges are embedded like the corresponding 2-paths in G. Moreover, the way we partitioned the edges guarantees that each of the resulting $\mathcal{O}(1)$ graphs contains no pair of crossing edges. It follows that these graphs are planar.

Each of the $\mathcal{O}(1)$ planar graphs that sum up to G_2 can be 5-oriented. Hence we obtain an $\mathcal{O}(1)$-orientation of G_2 and adjacency queries in G_2 can be processed in exactly the same way as in G. It follows that the case (3) from Fig. 1 can be detected in $\mathcal{O}(1)$ time.

The approach described above can be extended to any fixed natural number k. We start from creating $\overrightarrow{G_1}$, an $\mathcal{O}(1)$-orientation of G. Then we build successively undirected graphs: $G_2, G_3, \ldots, G_k$ and their orientations $\overrightarrow{G_2}, \overrightarrow{G_3}, \ldots, \overrightarrow{G_k}$. All the graphs have the same vertex set as the initial graph G. For any $t > 1$ an edge uv belongs to G_t iff there exists $x \in V(G)$ such that $(x, u) \in E(\overrightarrow{G_i})$ and $(x, v) \in E(\overrightarrow{G_j})$ for some i, j satisfying $i + j = t$. Thus, every edge in graph G_t corresponds to certain walk of length t in G.

A union of graphs $\overrightarrow{G_i}$ is called *a shortcut graph* and denoted by $\overrightarrow{S_k(G)}$. The edges in G_t have weights equal to t. The weight of a path in $\overrightarrow{S_k(G)}$ is the sum of weights of its edges. The following properties allow us to use the shortcut graph as a short paths oracle:

(i) Each graph G_i is a union of a bounded number of plane graphs and therefore it can be $\mathcal{O}(1)$-oriented. Notice that after such an orientation only a bounded number of vertices is reachable from any vertex v by paths of weight at most k in the shortcut graph.

(ii) Let v and w be vertices at distance $l \leq k$ in G. Then in the shortcut graph there are two dipaths $v \rightsquigarrow x$ and $w \rightsquigarrow x$ such that the sum of their weights is l.

Observe that by property (i) we can easily find the paths described in (ii) in $\mathcal{O}(1)$ time. For an example of a shortcut graph see Fig. 3. The input graph is a path of length 12 and $k = 7$. Vertices v and w are at distance 7, dipaths described in (ii) are bold.

It turns out that any shortcut graph can be significantly simplified. Once we build the shortcut graph we can transform it simply to a biorientation of the input

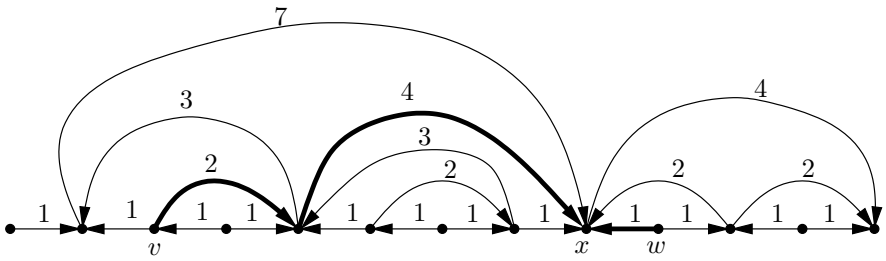

Fig. 3. An example of a Shortcut Graph.

graph G. The biorientation has similar properties as the shortcut graph and can be also used for finding shortest paths of bounded length with the same performance as before. However, the biorientation cannot be maintained in the dynamic environment.

Organization of the Paper The rest of this paper is organized as follows. In Section 4 we define operation $\odot$ which is used in latter sections to describe formally the construction of the shortcut graph. Then we show properties of the introduced operation (see Corollary 4.3) that easily imply the constraint (i) above. Section 5 contains detailed description and time complexity analysis of the shortcut graph construction algorithm as well as the shortest path query algorithm. We also show that constraint (ii) holds (see Theorem 5.3) which provides correctness of the query algorithm. Next in Section 5.4 we show the transformation of the shortcut graph to the biorientation mentioned above which constitutes the most important combinatorial result of this paper.

In Section 6 we show how to upgrade the shortcut graph to allow constant-time updates after deletions of edges or vertices. We also describe how to update the shortcut graph in amortized $\mathcal{O}(\log^k n)$ time after insertion of an edge. Section 7 shows flexibility of our structure; we show that it can be extended to service directed or weighted graphs and that it can process other natural query types. Eventually in Section 8 we discuss which classes of graphs can replace planar graphs in our results.

4. SUBDRAWINGS AND OPERATION $\odot$

Recall that graph F is a *subdivision* of graph H if F can be obtained from H by subdividing some of the edges, that is, by replacing the edges by paths. We say that F is a *t-subdivision* of H if F can be obtained from H by replacing some edges of H by paths of length at least 1 and at most t. H is a *subdrawing* of G if a certain subdivision of H is a subgraph of G. Similarly, H is a *t-subdrawing* of G if certain t-subdivision of H is a subgraph of G. If edge $e \in E(H)$ is replaced by a path p in G we say that e *corresponds* to p. In this case we denote $e = e_H(p)$ and $p = p_G(e)$.

Example Figure 4 shows a 5-subdrawing H of the 4×4 grid graph G. Edge $xy \in E(H)$ corresponds to 4-path $xabcy$ in grid G. We can write $xy = e_H(xabcy)$ and $xabcy = p_G(xy)$. Observe that if G is a plane graph and H is its subdrawing every edge of H can be drawn in the plane as a union of drawings of edges of G

obtaining a plane embedding of H.

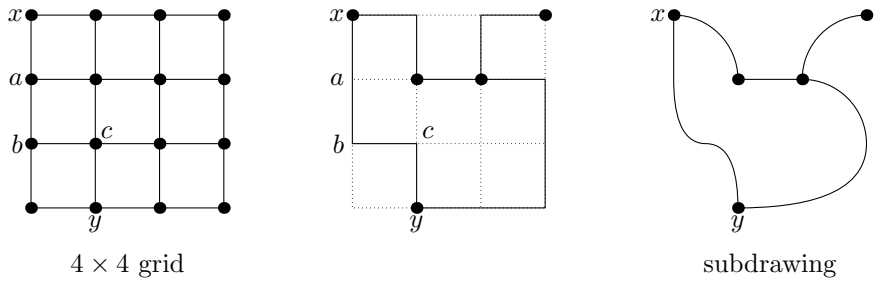

Fig. 4. An exemplary subdrawing of the 4×4 grid graph.

Let G and H be undirected graphs. If H can be decomposed into s t-subdrawings of G then such a decomposition is called (G, t)-*representation of* H *of thickness* s. When H is directed and each of the s subdrawings is d-oriented we say that the representation is d-oriented.

Let F and H be a pair of directed subdrawings of a graph G. We define a relation on the set of edges of F. We say that an edge $e_1 = (u, v) \in E(F)$ *points with* H to a different edge $e_2 = (x, y) \in E(F)$ when there exists an edge $h = (u, w) \in E(H)$ such that vertices of $p_G(h)$ and inner vertices of $p_G(e_2)$ intersect, i.e. $V(p_G(h)) \cap (V(p_G(e_2)) - \{x, y\}) \neq \emptyset$ (see Fig. 5).

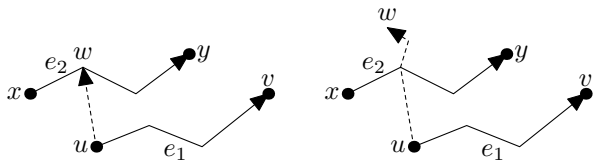

Fig. 5. Edge e_1 points to edge e_2.

The following lemma binds the notions of the subdrawing and the pointing relation.

LEMMA 4.1. *Let G be an arbitrary graph. Let A and B be 1-oriented digraphs such that A is subdrawing of G and B is b-subdrawing of G. Let e be an edge in graph A. Then e points with B to at most b edges in A.*

PROOF. Let $e = (u, v)$. Since B is 1-oriented there is at most one edge in B leaving u. If there is no such edge we have nothing to prove since then e does not point with B to any edge. Otherwise, let h be the edge leaving u. Observe that no two paths corresponding to some two edges in A share a common inner vertex. Hence each of at most $b+1$ vertices of path $p_G(h)$ is an inner vertex of at most one path corresponding to an edge in A. Moreover u is not an inner vertex of the path corresponding to an edge in A. Thus e points with B to at most b edges in A. □

Now we define a binary operation $\odot$. Its arguments are two digraphs G, H while the result is an undirected simple graph with vertex set $V(G) \cup V(H)$. Two distinct vertices u and v are adjacent in $G \odot H$ iff there exists a vertex x such that $(x, u) \in E(G)$ and $(x, v) \in E(H)$. We say that an edge uv is *supported by* x. Observe that one edge can be supported by many vertices.

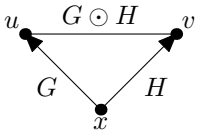

Fig. 6. An edge uv from graph $G \odot H$ is supported by vertex x.

We will show a useful fact concerning the introduced operation.

THEOREM 4.2. *Let G be an arbitrary graph. Let A and B be 1-oriented digraphs. Moreover let A and B be a-subdrawing and b-subdrawing of G respectively. Then graph $A \odot B$ has a $(G, a+b)$-representation of thickness $5 \cdot (2a+1)(2b+1)$. If a and b are bounded, this representation can be computed in linear time.*

PROOF. From Lemma 4.1 each edge of A points with B to at most b other edges. In another words the set of edges of A and the pointing relation define a b-oriented digraph. By Lemma 2.1 we can $(2b+1)$-color the edges of A in such a way that if one edge points to another then they are given different colors. Hence we can partition A into $2b+1$ graphs:

$$A = A_1 \cup A_2 \cup \ldots \cup A_{2b} \cup A_{2b+1},$$

each formed by the edges from the same color class. Similarly we can divide graph B into $2a+1$ graphs:

$$B = B_1 \cup B_2 \cup \ldots \cup B_{2a} \cup B_{2a+1}.$$

The above decomposition guarantees that in each graph A_i (respectively B_j) there is no edge which points to another edge with B (respectively A). In order to compute $(G, a+b)$-representation of $A \odot B$ we use the following formula:

$$A \odot B = \bigcup_{\substack{1 \le i \le 2b+1 \\ 1 \le j \le 2a+1}} A_i \odot B_j.$$

Now it suffices to show that each component $A_i \odot B_j$ is a union of at most five $(a+b)$ - subdrawings of G. To this end we need one more decomposition. Consider graph $A_i \cup B_j$, for arbitrary i, j. Observe that it is 2-oriented. Applying Lemma 2.1 we get a vertex 5-coloring of graph $A_i \cup B_j$. The vertices of the same color are neither connected in A_i nor in B_j.

Let c be any of the 5 colors used to color $A_i \cup B_j$. Denote by $A_{i,c}$ graph A_i restricted to the edges leaving the vertices colored c. Similarly we denote by $B_{j,c}$ graph B_j restricted to the edges leaving the vertices colored c. Obviously the

following equality holds:

$$A_i \odot B_j = \bigcup_{1 \le c \le 5} A_{i,c} \odot B_{j,c}.$$

Now it suffices to show that for every i, j, c graph $R_{i,j,c} = A_{i,c} \odot B_{j,c}$ is a $(a+b)$-subdrawing of G. To this end for each edge of graph $R_{i,j,c}$ we need to assign a corresponding path. Let vw be any edge of $R_{i,j,c}$. According to the definition of the operation $\odot$ there exists a vertex x such that (x, v) is an edge in $A_{i,c}$ and (x, w) is an edge in $B_{j,c}$. If there is more than one such vertex we choose an arbitrary one. We will build path $p_G(vw)$ using some edges from $p_G(vx)$ and $p_G(xw)$. Let $w_G(vw)$ be the walk that begins in v, then visits successive edges and vertices of path $p_G(vx)$ including vertex x and finally visits successive edges and vertices of path $p_G(xw)$ with vertex w in the very end. Observe that $w_G(vw)$ is not necessarily a path, since $p_G(xv)$ and $p_G(xw)$ can intersect in many vertices. Instead, as $p_G(vw)$ we take the shortest path between v and w in graph $p_G(xv) \cup p_G(xw)$. Nevertheless, we will need the notion of walk $w_G(vw)$ in the remaining part of the proof.

As a result in graph $R_{i,j,c}$ each edge e is assigned a path in G of length at most $a+b$. Let H be a graph obtained from $R_{i,j,c}$ by replacing its edges by corresponding paths. Clearly, $H \subseteq G$. To prove that $R_{i,j,c}$ is an $(a+b)$-subdrawing we need to show that H is a subdivision of $R_{i,j,c}$, i.e. there is no pair of edges e_1, e_2 such that a certain inner vertex of path $p_G(e_1)$ belongs to $p_G(e_2)$. It will be convenient for us to show a stronger fact: there is no pair of edges e_1, e_2 such that a certain inner vertex of walk $w_G(e_1)$ belongs to walk $w_G(e_2)$. Assume on the contrary that such a pair exists. Let $e_1 = v_1w_1$ and let $e_2 = v_2w_2$. Let x_1 (respectively x_2) be the vertex that supports e_1 (respectively e_2). W.l.o.g. we assume that $x_1v_1, x_2v_2 \in A_{i,c}$ and $x_1w_1, x_2w_2 \in B_{j,c}$ There are three cases to be considered:

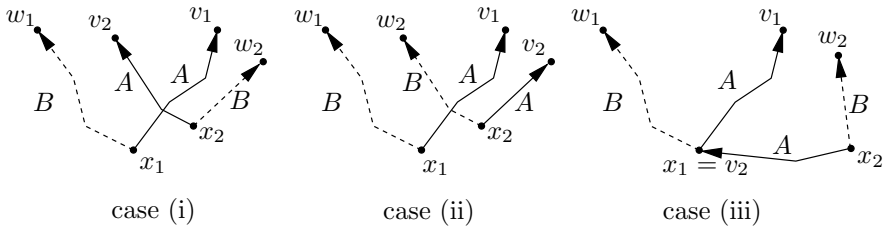

Fig. 7. Cases (i)-(iii)

(i) An inner vertex of path $p_G(x_1v_1)$ is a vertex of path $p_G(x_2v_2)$ (see Fig. 7 case (i)). This is impossible because $A_{i,c}$ is a subdrawing of G. Similarly it is impossible that an inner vertex of path $p_G(x_1w_1)$ is a vertex of path $p_G(x_2w_2)$.

(ii) An inner vertex of path $p_G(x_1v_1)$ is a vertex of path $p_G(x_2w_2)$ (see Fig. 7 case (ii)). We see that $x_2v_2 \in A_{i,c}$ points with B to $x_1v_1 \in A_{i,c}$ which is a contradiction. We can also exclude the symmetrical case when an inner vertex of path $p_G(x_1w_1)$ is a vertex of path $p_G(x_2v_2)$.

(iii) Vertex x_1 is a vertex of walk $w_G(e_2)$ (see Fig. 7 case (iii)). Since we excluded cases (i) and (ii), x_1 cannot be an inner vertex of either $p_G(x_2v_2)$ or $p_G(x_2w_2)$. As $A_{i,c}$ and $B_{j,c}$ are 1-oriented subdrawings of G, we know that $x_1 \neq x_2$. Hence, x_1 has to be either v_2 or w_2. Then x_1 and x_2 are adjacent in $A_{i,c}$ or in $B_{j,c}$, respectively. It cannot happen since x_1 and x_2 have got the same color c.

We have shown that $R_{i,j,c}$ is an (a+b)-subdrawing of G. This completes the proof that the set

$$\mathcal{R} = \{R_{i,j,c} : 1 \leq i \leq 2b+1;\ 1 \leq j \leq 2a+1;\ 1 \leq c \leq 5\}$$

is a $(G, a+b)$-representation of $A \odot B$. The decomposition $\mathcal{R}$ can be computed using Algorithm 4.1. It is easy to see that it works in linear time with respect to the size of G (we use the algorithm described in the proof of Lemma 2.1 for coloring graphs Γ_A, Γ_B and $A_i \cup B_j$).

Algorithm 4.1 Finding $(G, a+b)$-representation $\mathcal{R}$ for $A \odot B$ of thickness 5(2a+1)(2b+1)

Input: 1-oriented graphs A and B; A is a-subdrawing of G; B is b-subdrawing of G; every edge e of A or B stores a corresponding path $p_G(e)$.
Output: $(G, a+b)$-representation $\mathcal{R}$ for $A \odot B$ of thickness $5(2a+1)(2b+1)$

1: $\mathcal{R} \leftarrow \emptyset$
2: Compute graphs Γ_A, Γ_B described by the pointing relation on edges of A and B, resp.
3: $(2b+1)$-color graph Γ_A
4: $(2a+1)$-color graph Γ_B
5: **for** $i \leftarrow 1$ **to** $2b+1$ **do**
6: **for** $j \leftarrow 1$ **to** $2a+1$ **do**
7: $A_i \leftarrow$ graph formed by edges of A of color i
8: $B_j \leftarrow$ graph formed by edges of B of color j
9: 5-color $A_i \cup B_j$
10: **for** $c \leftarrow 1$ **to** 5 **do**
11: $A_{i,c} \leftarrow A_i$ restricted to the edges leaving vertices colored c
12: $B_{j,c} \leftarrow B_j$ restricted to the edges leaving vertices colored c
13: $R_{i,j,c} \leftarrow A_{i,c} \odot B_{j,c}$; for each edge of $R_{i,j,c}$ compute corresponding path in G
14: $\mathcal{R} \leftarrow \mathcal{R} \cup \{R_{i,j,c}\}$

□

COROLLARY 4.3. *Let G be a n-vertex graph. Let A be a digraph with given d_A-oriented (G, a)-representation of thickness t_A and let B be a digraph with given d_B-oriented (G, b)-representation of thickness t_B. Then the graph $A \odot B$ has a $(G, a+b)$-representation of thickness $5 \cdot d_A \cdot d_B \cdot t_A \cdot t_B \cdot (2a+1) \cdot (2b+1)$. If a and b are bounded, this representation can be computed in $\mathcal{O}(n \cdot d_A \cdot d_B \cdot t_A \cdot t_B)$ time.*

PROOF. Let $A = A_1 \cup A_2 \cup \ldots \cup A_{t_A}$ and $B = B_1 \cup B_2 \cup \ldots \cup B_{t_B}$ be (G, a)-representation and (G, b)-representation of graphs A and B respectively. Notice that the following holds:

$$A \odot B = \bigcup_{i=1}^{t_A} \bigcup_{j=1}^{t_B} A_i \odot B_j.$$

Each of graphs A_i can be further decomposed into a union of 1-oriented subdrawings of G: $A_{i,1}, \ldots A_{i,d_A}$. Similarly for every $j = 1, \ldots t_B$ we decompose B_j into a union of 1-oriented subdrawings of G: $B_{j,1}, \ldots B_{j,d_B}$. Observe that

$$A_i \odot B_j = \bigcup_{p=1}^{d_A} \bigcup_{q=1}^{d_B} A_{i,p} \odot B_{j,q}.$$

Finally we obtained

$$A \odot B = \bigcup_{i=1}^{t_A} \bigcup_{j=1}^{t_B} \bigcup_{p=1}^{d_A} \bigcup_{q=1}^{d_B} A_{i,p} \odot B_{j,q}.$$

Now we apply Theorem 4.2 to each of $d_A \cdot d_B \cdot t_A \cdot t_B$ components of the union. □

5. THE SHORTCUT GRAPH DATA STRUCTURE

From now on we assume that k is a fixed constant and G is an undirected planar graph with n vertices. We describe a data structure $\overrightarrow{S_k(G)}$ called *a shortcut graph of* G with the following properties:

(i) construction of $\overrightarrow{S_k(G)}$ takes $\mathcal{O}(n)$ time and space,
(ii) for arbitrary vertices u and v one can check in $\mathcal{O}(1)$ time whether u and v are at distance at most k in graph G.

5.1 Construction

$\overrightarrow{S_k(G)}$ is a directed graph with the vertex set $V(G)$. Edges have integer weights from the set $\{1 \ldots k\}$. We denote by $\overrightarrow{G_i}$ the subgraph of $\overrightarrow{S_k(G)}$ containing all the edges with weight i. The underlying undirected graph is denoted by G_i. In the sequel we will also refer to undirected version of graph $\overrightarrow{S_k(G)}$, denoted by $S_k(G)$.

We build $\overrightarrow{S_k(G)}$ by constructing successively graphs $\overrightarrow{G_1}, \overrightarrow{G_2}, \ldots \overrightarrow{G_k}$. For each graph $\overrightarrow{G_i}$ we compute and maintain a 3-oriented (G, i)-representation. This is achieved by repeating alternately the following two steps:

Step 1 – Computation of G_i. The initial graph G_1 is simply equal to G. For $i > 1$ we compute the graph G_i applying operation $\odot$ to the previously constructed graphs:

$$G_i = \bigcup_{\substack{1 \le p \le q < i \\ p+q=i}} \overrightarrow{G_p} \odot \overrightarrow{G_q} \tag{1}$$

Each of the operations $\overrightarrow{G_p} \odot \overrightarrow{G_q}$ is performed separately by applying Corollary 4.3. As a result we obtain a (G, i)-representation of the graph G_i. Notice that the

thickness of the representation is a function of i but independent of n, therefore it is $\mathcal{O}(1)$.

Step 2 – Orienting the Edges of G_i. We are given the graph G_i and its (G,i)-representation of thickness $\mathcal{O}(1)$. Observe that every subdrawing of a planar graph is planar. It follows that edges of each member of the representation can be oriented by applying the following result:

THEOREM 5.1 CHROBAK, EPPSTEIN [1991]. *For every planar graph one can compute its 3-orientation in linear time.*

After orienting all the edges of G_i we obtain the resulting graph $\overrightarrow{G_i}$. Notice that we have computed also a relevant 3-oriented (G,i)-representation for $\overrightarrow{G_i}$. The thickness of this representation is $\mathcal{O}(1)$ therefore the outdegrees in $\overrightarrow{G_i}$ are also bounded by $\mathcal{O}(1)$.

COROLLARY 5.2. *For every planar graph G the structure $\overrightarrow{S_k(G)}$ satisfies the following conditions:*

(i) Edge (u,v) in $\overrightarrow{S_k(G)}$ with weight t indicates that there is a path in G from u to v of length at most t.

(ii) The graph $\overrightarrow{S_k(G)}$ is d-oriented, $d = \mathcal{O}(1)$.

(iii) The construction of $\overrightarrow{S_k(G)}$ takes time and space $\mathcal{O}(n)$.

5.2 Processing the Queries

Weight of a path p in $\overrightarrow{S_k(G)}$ is the sum of the weights of all the edges that form p and will be denoted by $w(p)$ (analogously we define weight of a walk). Now we will prove a crucial property of the shortcut graph:

THEOREM 5.3. *Let G be a planar graph and let $\overrightarrow{S_k(G)}$ be a shortcut graph for G. Let vertices u and v be joined by a path of length $l \leq k$ in G. Then there are two directed paths in $\overrightarrow{S_k(G)}$, the first from u to some vertex x and the second from v to x such that their weights sum up to l.*

PROOF. The proof is by the induction on l. Theorem trivially holds for $l = 0$. Now we move to the induction step. Let P be the u,v-path in G of length $l > 0$ and let us denote by P_1 and P_2 the paths from u to x and from v to x that we want to find. Let u_1 be the neighbor of u on path P. By the induction hypothesis there are two dipaths in the shortcut graph, the first from u_1 to some vertex x and the second from v to x such that their weights sum up to $l-1$. We will denote these paths by Q_1 and Q_2 respectively.

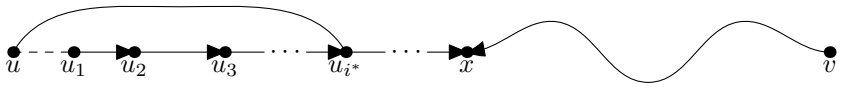

Fig. 8. Proof of Theorem 5.3.

Let $u_1, u_2, \ldots u_r = x$ be successive vertices of path Q_1 (see Fig. 8). Let $\mathrm{dist}_{Q_1}(a, b)$ denote the distance between vertices a and b in path Q_1, treated as a weighted graph. We define a set Z as follows:

$$Z = \{i : 1 \leq i \leq r \text{ and } uu_i \in E(G_{1+\mathrm{dist}_{Q_1}(u_1,u_i)})\}$$

As $uu_1 \in E(G_1)$, $1 \in Z$ and $Z \neq \emptyset$. Let $i^* = \max Z$.
If $i^* = r$, i.e. $u_{i^*} = x$ graph $\overrightarrow{G_{1+\mathrm{dist}_{Q_1}(u_1,x)}}$ contains either (u, x) or (x, u). If it contains (u, x) we can put $P_1 = \{(u, x)\}$ and $P_2 = Q_2$. Then $w(P_1) = 1 + w(Q_1)$ and $w(P_2) = w(Q_2)$. In the other case we put an empty path as P_1 and $P_2 = Q_2 \cup \{(x, u)\}$. Then $w(P_1) = 0$ and $w(P_2) = w(Q_2) + 1 + w(Q_1)$. Hence in both cases $w(P_1) + w(P_2) = l$.

Assume $i^* < r$. Let $a = \mathrm{dist}_{Q_1}(u_{i^*}, u_{i^*+1})$ and $b = 1 + \mathrm{dist}_{Q_1}(u_1, u_{i^*})$. We know that either $(u_{i^*}, u) \in E(\overrightarrow{G_b})$ or $(u, u_{i^*}) \in E(\overrightarrow{G_b})$. If $(u_{i^*}, u) \in E(\overrightarrow{G_b})$ then $u_{i^*+1}u \in E(G_{a+b})$ and $i^*+1 \in Z$, the contradiction. Finally when $(u, u_{i^*}) \in E(\overrightarrow{G_b})$ then $P_2 = Q_2$ and P_1 consists of edge (u, u_{i^*}) and the successive edges of Q_1, i.e. edges $(u_{i^*}, u_{i^*+1}), \ldots, (u_{r-1}, u_r)$. Again one can see that $w(P_1) + w(P_2) = l$. □

The Shortest Path Query Algorithm. Theorem 5.3 yields a simple $\mathcal{O}(1)$ time algorithm which verifies whether $\mathrm{dist}(u, v) \leq k$ and if so it computes a relevant shortest path between u and v. As it was proved the outdegree in the shortcut graph $\overrightarrow{S_k(G)}$ is bounded by $\mathcal{O}(1)$. Let us denote the maximum outdegree in $\overrightarrow{S_k(G)}$ by $\Delta(k)$. Let $\overrightarrow{S(u)}$ be the subgraph of the shortcut graph induced by all vertices that are reachable in $\overrightarrow{S_k(G)}$ from u via paths of weight at most k. Since each such path consists of at most k edges graph $\overrightarrow{S(u)}$ has at most $(\Delta(k))^{k+1} = \mathcal{O}(1)$ edges. Similarly we define graph $\overrightarrow{S(v)}$ which also has bounded size.

Let graph $T(u, v)$ be the union of undirected versions of $\overrightarrow{S(u)}$ and $\overrightarrow{S(v)}$, i.e. $T(u, v) = S(u) \cup S(v)$. Theorem 5.3 implies that if the distance between u and v in G is $l \leq k$ then there exists a path with weight l in $T(u, v)$. To find this path we simply use the standard Dijkstra's algorithm in graph $T(u, v)$. It works in constant time since graph $T(u, v)$ has bounded size. Let P denote the shortest path between u and v in $T(u, v)$ found using Dijkstra's algorithm. Clearly, P has weight at most l.

P is a path in graph $S_k(G)$. The shortest path between u and v in G can be reconstructed in $\mathcal{O}(1)$ time from P as follows. Recall that during the computation of (G, i)-representation of a graph G_i (see Section 4) for each edge ab of G_i we compute a corresponding path in G of length at most i, denoted by $p_G(ab)$. Thus a reconstruction of a shortest path in G from P can be done easily by replacing edges with weights greater than 1 by corresponding paths. As a result we get a walk w of length at most l from u to v. We see that the length of the walk is equal to l and each vertex of walk w appears in w only once, for otherwise the distance between u and v would be smaller than l. Thus walk w corresponds to a simple path of length l joining u and v. Clearly this last step also takes constant time.

5.3 More Precise Estimations

In this section we show how the time complexity of preprocessing and query algorithms described in sections 5.1 and 5.2 depends on constant k. We will show an upper bound for thickness of (G,i)-representation of graph G_i, denoted as $t(G_i)$, for $i = 1, \dots k$. From Equation (1) we get

$$\begin{cases} t(G_1) = 1 \\ t(G_i) \leq \sum_{j=1}^{\lfloor \frac{i}{2} \rfloor} t(\overrightarrow{G_j} \odot \overrightarrow{G_{i-j}}) \end{cases}$$

Then we use Corollary 4.3:

$$\begin{aligned} t(G_i) &\leq \sum_{j=1}^{\lfloor \frac{i}{2} \rfloor} 5 \cdot 3^2 \cdot (2j+1)(2(i-j)+1) t(G_j) t(G_{i-j}) \\ &\leq 45 \cdot \sum_{j=1}^{\lfloor \frac{i}{2} \rfloor} \left(\frac{2j+1+2(i-j)+1)}{2} \right)^2 t(G_j) t(G_{i-j}) \\ &\leq 45 \cdot (i+1)^2 \cdot \sum_{j=1}^{\lfloor \frac{i}{2} \rfloor} t(G_j) t(G_{i-j}) \end{aligned}$$

Let $c = 45 \cdot (k+1)^2$. Since $i \leq k$ in order to bound $t(G_i)$ we can solve the following recurrence:

$$\begin{cases} T_1 = 1 \\ T_i = c \cdot \sum_{j=1}^{i-1} T_j T_{i-j} \quad \text{for } i \geq 2 \end{cases} \tag{2}$$

Let C_n denote n-th Catalan number. Let us recall its definition:

$$\begin{cases} C_0 = 1 \\ C_i = \sum_{j=0}^{i-1} C_j C_{i-j-1} \quad \text{for } i \geq 1 \end{cases} \tag{3}$$

PROPOSITION 5.4. *$T_i = c^{i-1} C_{i-1}$.*

PROOF. We use the induction on i. Clearly, $T_1 = 1 = c^0 C_0$. Moreover, $T_i = c \cdot \sum_{j=1}^{i-1} T_j T_{i-j} = c \cdot \sum_{j=1}^{i-1} c^{j-1} C_{j-1} c^{i-j-1} C_{i-j-1} = c^{i-1} \sum_{j=0}^{i-2} C_j C_{i-j-2} = c^{i-1} C_{i-1}$. It settles the proof. □

To estimate the value of T_n we need the well-known estimation of Catalan numbers (see e.g. Graham et al. [1994]):

PROPOSITION 5.5. *$C_i = \frac{4^i}{\sqrt{\Pi i^3}}(1 + \mathcal{O}(1/i))$.*

Finally we obtained the following bound:

$$t(G_i) = \mathcal{O}((180(k+1)^2)^i) = 2^{\mathcal{O}(i \log k)}. \tag{4}$$

COROLLARY 5.6. *For any n-vertex planar graph G the shortcut graph structure $\overrightarrow{S_k(G)}$ can be created in $2^{\mathcal{O}(k \log k)} n$ time and space. The structure processes shortest path queries in $2^{\mathcal{O}(k \log k)}$ time.*

5.4 Compressing the Shortcut Graph to at Most $2|E(G)|$ Edges

In the previous section we showed that the data structure $S_k(G)$ occupies $2^{\mathcal{O}(k \log k)}n$ space. In this section we show that after computing the shortcut graph we can compress it in linear time to a very succinct data structure having similar properties as $S_k(G)$.

Now we describe a linear algorithm that compresses $\overrightarrow{S_k(G)}$. Let $\overleftrightarrow{G}$ be a digraph obtained from G by replacing every edge uv by a pair of directed edges (u,v) and (v,u). Initially, all the edges of $\overleftrightarrow{G}$ are marked as *unused*. Moreover, with every edge $(u,v) \in \overrightarrow{G_1}$ we store pointers to edges (u,v) and (v,u) in $\overleftrightarrow{G}$. Since $\overrightarrow{G_1}$ is 3-oriented we can access the edges of $\overleftrightarrow{G}$ in constant time. Next, for every edge $e = (x,y) \in \overrightarrow{S_k(G)}$ corresponding to a path $p_G(e) = xx_1 \dots y$ we mark the edge (x,x_1) as *used* in $\overleftrightarrow{G}$. The graph consisting of the edges of $\overleftrightarrow{G}$ marked as used will be denoted by $\overrightarrow{\sigma_k(G)}$ and called *succinct shortcut graph*. Thus, $\overrightarrow{\sigma_k(G)}$ is simply a biorientation of G and occupies $\mathcal{O}(|E(G)|)$ space for every value of k. The above construction guarantees that for every vertex v of G, $\mathrm{outdeg}_{\overrightarrow{\sigma_k(G)}}(v) \leq \mathrm{outdeg}_{\overrightarrow{S_k(G)}}(v)$.

COROLLARY 5.7. *For any planar graph G the succinct shortcut graph $\overrightarrow{\sigma_k(G)}$ is $\mathcal{O}(1)$-oriented.*

To show that the succinct shortcut graph can be used to process short-path queries we will need the following lemma.

LEMMA 5.8. *Let (u,v) be an edge of $\overrightarrow{G_i}$ corresponding to a path $p_G(e) = u \dots v_1 v$ of length i. Then the succinct shortcut graph $\overrightarrow{\sigma_i(G)}$ contains a directed path from u to v of length i and a directed path from v_1 to u of length $i-1$.*

PROOF. The proof is by the induction on i. In the case when $i = 1$ it suffices to observe that $\overrightarrow{\sigma_1(G)} = \overrightarrow{G_1}$. Assume that $i > 1$. Recall from the proof of Theorem 4.2 that there exists a vertex x and edges $(x,u) \in \overrightarrow{G_a}$, $(x,v) \in \overrightarrow{G_b}$ such that $a + b = i$ and $p_G(xu) \cup p_G(xv) = p_G(uv)$. Let u_1 be the neighbor of u in $p_G(uv)$. Clearly

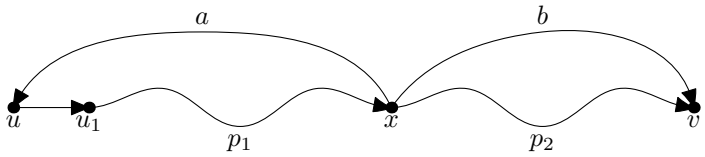

Fig. 9. Proof of Lemma 5.8.

$p_G(xu) = x \dots u_1 u$. By the induction hypothesis we get that there is a path p_1 of length $a-1$ from u_1 to x in graph $\overrightarrow{\sigma_a(G)}$ (See Fig. 9). Also there is a path p_2 of length b from x to v in $\overrightarrow{\sigma_b(G)}$. Since $\overrightarrow{\sigma_a(G)} \cup \overrightarrow{\sigma_b(G)} \subseteq \overrightarrow{\sigma_i(G)}$ and $(u,u_1) \in \overrightarrow{\sigma_i(G)}$ the desired u,v-path is formed by $\{(u,u_1)\} \cup p_1 \cup p_2$.

Similarly we can show the existence of the other path. We see that a path of length $b-1$ from v_1 to x is in $\overrightarrow{\sigma_b(G)}$, a path of length a from x to u is in $\overrightarrow{\sigma_a(G)}$ and the union of these paths forms the desired v_1,u-path. □

THEOREM 5.9. *For any n-vertex planar graph G and a natural number k it is possible to find in $2^{\mathcal{O}(k \log k)}n$ time $\overrightarrow{\sigma_k(G)}$, a biorientation of G such that*

(i) $\overrightarrow{\sigma_k(G)}$ has outdegrees bounded by $2^{\mathcal{O}(k \log k)}$,

(ii) for any pair of vertices u, v at distance at most k in G there is a shortest path p in G between u and v and a vertex $x \in p$ such that edges of p form two directed paths in $\overrightarrow{\sigma_k(G)}$: a path from u to x and a path from v to x.

Theorem 5.9 follows immediately from Theorem 5.3 and Lemma 5.8. It guarantees that the algorithm for finding shortest paths in $S_k(G)$ described in Section 5.2 can be also used in the succinct shortcut graph and still it works in $\mathcal{O}(1)$ time. It is even more simple: now there are no weights so it suffices to use BFS in graph $T(u, v)$.

6. DYNAMIC ENVIRONMENT

An important asset of our data structure is that it can be adapted to work in a fully dynamic environment. We show in this section that after deleting an arbitrary edge or vertex from G the shortcut graph $\overrightarrow{S_k(G)}$ can be updated in $\mathcal{O}(1)$ time. Additionally one can enable or disable edges and vertices in graph G still needing only constant time to update the shortcut graph. Finally we show how to refresh the shortcut graph after adding an edge (adding a vertex is trivial). The amortized time needed to perform such an operation is bounded by $\mathcal{O}(\log^k n)$ but the question whether this bound is tight is open. We emphasize that the query algorithm stays almost unchanged and it still works in constant time.

6.1 Deleting an Edge

Imagine that an edge $e = uv$ was deleted from graph G. Let us think which elements of the shortcut graph should be changed or deleted. Recall the successive phases of building the shortcut graph. In the very beginning one builds graph $\overrightarrow{G_1}$, an orientation of G. Hence in $\overrightarrow{G_1}$ there is an edge joining u and v, say (u, v). Clearly it has to be deleted. However, there may be another edge leaving u in $\overrightarrow{G_1}$, say (u, w). Then edges (u, v) and (u, w) cause edge vw in graph G_2 and u supports vw. This edge should also be deleted but only if there is no other vertex supporting vw. Deleting vw can further cause deleting some edges from G_3 an so on. Thus we get the whole set of edges that should be removed from graphs $\overrightarrow{G_1}, \overrightarrow{G_2}, \ldots, \overrightarrow{G_k}$. We need to answer the following two questions: *(i)* how numerous can be the set of deleted edges; *(ii)* how to find those edges quickly. It can be seen that the number of deleted edges is $\mathcal{O}(1)$ and they can be found in $\mathcal{O}(1)$ time.

6.1.1 *Extension of the Shortcut Graph.* To be more precise we need to introduce some definitions. Let $e_1 = (x, v)$, $e_2 = (x, w)$ be edges in graphs G, H respectively. Let e be an edge of $\overrightarrow{G_{i+j}}$ equal to (v, w) or (w, v). We say that edges e_1 and e_2 *support* e. We say also that e_1 and e_2 are *supporters* of e and e is a *dependant* of e_1 and e_2. We say that the pair $\{e_1, e_2\}$ is a *supporting couple* of e.

For convenience, we will use the notions defined above also for undirected versions of edges. Thus, when edge $(x, v) \in \overrightarrow{G_i}$ supports edge $(v, w) \in \overrightarrow{G_{i+j}}$ then also

$(x, v) \in \overrightarrow{G_i}$ supports $vw \in G_{i+j}$, $xv \in G_i$ supports $(v, w) \in \overrightarrow{G_{i+j}}$ and $xv \in G_i$ supports $vw \in G_{i+j}$.

Notice that each edge may have more than one supporting couples. For example consider the situation in Figure 10. Edge uv of graph G_2 has two supporting

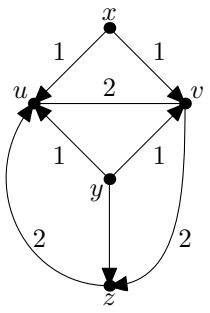

Fig. 10. Supporters and dependants. Labels denote weights of edges.

couples: $\{(x, u), (x, v)\}$ and $\{(y, u), (y, v)\}$. On the other hand edge (u, y) of graph $\overrightarrow{G_1}$ has two dependants: uv and uz.

The following theorem holds:

THEOREM 6.1. *Let G be a planar graph and $\overrightarrow{S_k(G)}$ a shortcut graph for G. Each edge $e = (v, w) \in \overrightarrow{S_k(G)}$ has $\mathcal{O}(1)$ dependants.*

PROOF. Recall that $\Delta(k)$ denotes a constant that bounds outdegrees in $\overrightarrow{S_k(G)}$. The number of the edges leaving v distinct from e is at most $\Delta(k) - 1$. It implies that the number of dependants of e is bounded by $\Delta(k) - 1 = \mathcal{O}(1)$. □

We assume that during the construction of $\overrightarrow{S_k(G)}$ for each edge $e \in \overrightarrow{S_k(G)}$ the following pieces of information are stored:

(i) the list $S(e)$ of all supporting couples of e,

(ii) the list $D(e)$ of all pairs (d, p) such that for some edge f, pair $\{e, f\}$ is a supporting couple of d and p is a pointer to $\{e, f\}$ in the list $S(d)$.

6.1.2 *An Algorithm for Deleting an Edge.* Assume that e is an edge of $\overrightarrow{G_t}$ and it is going to be deleted. For each pair $(d, p) \in D(e)$ we have to perform the following operations: remove the pair $\{e, f\}$ referenced by pointer p from list $S(d)$, remove the pair (d, p) from the list $D(f)$. If list $S(d)$ becomes empty delete edge d recursively. As each edge has $\mathcal{O}(1)$ dependants (see Theorem 6.1) and the depth of recursion is bounded by the constant k the whole operation takes $\mathcal{O}(1)$ time. In order to delete an edge from the initial graph G it is enough to delete the relevant directed edge from $\overrightarrow{G_1}$.

6.1.3 *Processing the Queries in the Dynamic Environment.* Recall that during execution of queries we need to find a path in G corresponding to a given edge in $S_k(G)$. In the static environment it was sufficient to store a fixed path $p_G(e)$ for each edge e. As we have seen it is possible that some edge of $p_G(e)$ is deleted but edge e remains in the shortcut graph. Thus in the dynamic environment we find a path corresponding to a given edge e using the structure of supporters stored in lists $S(\cdot)$. We start from a path $p = e$ and we successively replace edges of p by their supporting couples until all the edges in p have weight 1. Clearly, as a result we get some walk of length equal to the weight of e. Hence, in our query algorithm for two vertices u, v at distance d we transform a path of weight d in the shortcut graph to a walk of length d in the input graph. This walk must be a simple path for otherwise the walk would contain a shorter simple path between u and v and the distance between u and v would be smaller than d.

6.2 Deleting a Vertex

Observe that in order to delete a vertex v efficiently we cannot simply delete all the incident edges because there can be $\Omega(n)$ ones. Instead, we remove from graph $\overrightarrow{S_k(G)}$ all the edges leaving v using the recursive procedure described in Section 6.1.2. It takes $\mathcal{O}(1)$ time since the outdegree of v in $\overrightarrow{S_k(G)}$ is bounded by a constant. Then we mark v as deleted. It ends the vertex deletion algorithm. Certainly, the query algorithm ignores edges entering a deleted vertex (or, equivalently, every such edge is deleted immediately after the query algorithm finds it).

6.3 Inserting an Edge

In this section we show how to update the shortcut graph $S_k(G)$ after inserting an edge in such a way that the outdegrees remain bounded and it is still a shortcut graph of G. We show that the structure supports insertions in amortized $\mathcal{O}(\log^k n)$ time. To simplify the presentation, we assume that graph G initially contains no edges.

Let us start from the case when $k = 1$. Then the shortcut graph is simply an orientation $\overrightarrow{G_1}$ of G with bounded outdegrees. During insert operation we sometimes need to reorient some of the edges of $\overrightarrow{G_1}$ in order to keep outdegrees bounded.

Lazy Reorienting Scheme We use the result of G. Brodal and R. Fagerberg [1999]. They show how to keep bounded outdegree in an orientation of a graph with bounded arboricity. An *arboricity* of a graph G, denoted by $a(G)$, can be defined as the smallest number of forests needed to cover all edges of G. (It is widely known that arboricity of a planar graph does not exceed 3.) Let a be the arboricity of graph $\overrightarrow{G}$ and let $D = 4a$. Brodal and Fagerberg consider the following routine for keeping outdegrees in $\overrightarrow{G}$ bounded by D after inserting an edge (u, v). If $\operatorname{outdeg} u = D + 1$, repeatedly a node w with outdegree larger than D is picked, and the orientation of all the edges leaving w is changed. We will refer to this method as *lazy reorienting scheme*, as reorientation of edges is done as late as possible. They show in [Brodal and Fagerberg 1999] that for graphs of bounded arboricity this simple approach guarantees that amortized number of edge reorientations is $\mathcal{O}(1)$ for one insertion and $\mathcal{O}(\log n)$ for one deletion in a sequence of operations.

However, it is straightforward to modify their analysis to get $\mathcal{O}(\log n)$ amortized insertion time and $O(1)$ worst-case deletion time. Let us note that it is not known whether the above logarithmic bounds are tight; the best known lower bound for amortized insertion time is $\Omega(1)$. If the value of arboricity is not known *a priori*, the algorithm can estimate it (by doubling the value of D when too much reorientations is done).

Insertion Algorithm Now we move to general case and describe a recursive algorithm INSERT(uv,i) for inserting an edge uv to graph G_i, for any $i = 1, \ldots k$. We start from orienting uv arbitrarily in $\overrightarrow{G_i}$, say as (u, v). Then we use lazy reorienting scheme for keeping outdegrees in $\overrightarrow{G_i}$ bounded by $4 \cdot a(G_i)$. Whenever any edge changes its orientation we act as if it was deleted and we perform the algorithm described in Section 6.1.2. Moreover, when any edge (x, y) appears in $\overrightarrow{G_i}$, both after simply inserting it and after reorienting (y, x), we recursively add an edge yz to G_{i+j} for each edge (x, z) present in $\overrightarrow{G_j}$ such that $i + j \leq k$.

PROPOSITION 6.2. *Let $S_k(G)$ be a shortcut graph for graph G. Then the graph obtained as a result of* INSERT*(uv,1) algorithm is a shortcut graph of $G \cup \{uv\}$.*

PROOF. It follows immediately from the fact that the equation (1) holds after any insertion or deletion. □

LEMMA 6.3. *For any $i \leq k$, after any sequence of edge insertions and deletions the graph $\overrightarrow{G_i}$ is $\mathcal{O}(1)$-oriented and has (G, i)-representation of thickness $\mathcal{O}(1)$.*

PROOF. The proof is by the induction on i. For $i = 1$ it is actually the result of G. Brodal and R. Fagerberg [1999]. Now we move to the induction step. Assume that the lemma holds for all $j < i$, i.e. G_j is d_j-oriented and has (G, j)-representation of thickness t_j, where $d_j = \mathcal{O}(1)$, $t_j = \mathcal{O}(1)$. Each member of this (G, j)-representation is d_j-oriented. Observe that the equation (1) is preserved after any insertion and deletion. Hence by Corollary 4.3 for any a, b such that $a + b = i$ after any sequence of insert/delete operations the graph $G_a \odot G_b$ has $(G, a + b)$-representation of thickness $5 \cdot d_a \cdot d_b \cdot t_a \cdot t_b(2a + 1)(2b + 1) = \mathcal{O}(1)$. Thus G_i has thickness $t = \mathcal{O}(1)$. Since the arboricity of a planar graph G is at most 3, G_i has arboricity at most $3t = \mathcal{O}(1)$. Thus our algorithm keeps outdegrees of G_i bounded by $4 \cdot 3t = \mathcal{O}(1)$. □

Now it is clear that our insert routine properly updates the shortcut graph and after each insertion queries are still processed in $\mathcal{O}(1)$ time. Now we move to the amortized time complexity of insertions. The following lemma is a straightforward corollary from results contained in Brodal and Fagerberg [1999].

LEMMA 6.4 BRODAL AND FAGERBERG [1999]. *Let σ be an arbitrary sequence of p edge insertions and q edge deletions performed on an n-vertex graph, initially containing no edges. If at every stage the arboricity of the graph is bounded the lazy reorienting scheme is used after each insertion, there are $\mathcal{O}(p \log n)$ edge reorientations performed.*

LEMMA 6.5. *For arbitrary sequence of t insert operations in graph G (possibly alternated by deletions) for any $i \leq k$ there are $\mathcal{O}(t \log^i n)$ edge reorientations and $\mathcal{O}(t \log^{i-1} n)$ edge insertions/deletions in graph G_i.*

PROOF. The proof is by the induction on i. For $i = 1$ it follows from Lemma 6.4. For $i > 1$ the induction hypothesis implies that for every $j < i$ INSERT algorithm performed $\mathcal{O}(t \log^j n)$ edge reorientations and insertions in graph $\overrightarrow{G_j}$. Since the outdegrees in $\overrightarrow{S_k(G)}$ are bounded, each of these reorientations and insertions causes $\mathcal{O}(1)$ insertions and deletions in graph G_i. Hence the total number of insert/delete operations in G_i is $\mathcal{O}(t \log^{i-1} n)$. By Lemma 6.3 G_i has bounded arboricity so by Lemma 6.4 these operations require $\mathcal{O}(t \cdot \log^i n)$ edge reorientations in G_i. □

COROLLARY 6.6. *The amortized time complexity of the* INSERT *operation is* $\mathcal{O}(\log^k n)$.

What Means n When We Can Add/Delete Vertices? Observe that in the environment that allows deleting vertices of graph G insertions in a sequence τ of operations can be performed in amortized $\mathcal{O}(\log^k \hat{n})$ time where $\hat{n}$ refers to maximal number of vertices present in G during the execution of τ. Then the outdegrees of vertices cannot be stored. Each time the lazy reorienting scheme needs to check the outdegree of some vertex x one has to compute it by counting all the edges (x, v) such that vertex v is not disabled. Whenever an edge entering a disabled vertex is found it is immediately deleted.

Building the Shortcut Graph in Dynamic Environment Analyzing the time complexity of insertion algorithm we assumed that graph G initially contained no edges. However, in many applications we start from some nonempty graph G and then we perform some changes on it. Building a graph with m edges using INSERT algorithm takes $\mathcal{O}(m \log^k n)$ time. However, one can ask whether it is possible to build the shortcut graph for a given planar graph in linear time and then perform some updates with amortized $\mathcal{O}(\log^k n)$ time per insertion. We answer that it is possible and we show below how to do it.

We build the shortcut graph $\overrightarrow{S_k(G)}$ using the linear-time algorithm from section 5.1. Then appears some sequence σ of insert/delete operations performed in graph G. Let $\overrightarrow{A_k(G)}$ be the subgraph of $\overrightarrow{S_k(G)}$ induced by the edges added in the sequence σ. Each operation in σ either deletes an edge from $\overrightarrow{S_k(G)} - \overrightarrow{A_k(G)}$ or inserts/deletes an edge to/from $\overrightarrow{A_k(G)}$. The algorithms for inserting and deleting edges work like before, but the edges from $\overrightarrow{S_k(G)} - \overrightarrow{A_k(G)}$ do not change their orientations and we use the lazy reorientation scheme only in graph $\overrightarrow{A_k(G)}$. Clearly, outdegrees in the shortcut graph are still bounded and insertions work in $\mathcal{O}(\log^k n)$ time, since $\overrightarrow{A_k(G)}$ initially contained no edges.

7. VARIATIONS

7.1 Weighted Graphs

When we consider arbitrary nonnegative weights assigned to edges of the input graph any oracle that processes shortest path queries of our type can be used to obtain shortest paths and distances between *any* pair of vertices because it is possible to scale the weights of the input graphs to make them smaller than k. Even if we do not allow for weights smaller than one is seems to be hard to extend our approach to the continuous case. However we note that our method trivially

applies to weighted planar graphs when the weights are natural numbers. In the preprocessing phase we start from removing the edges of weights greater than k – as they cannot be a part of any path of length at most k. Then each edge of weight w is replaced by a path of length w. Clearly, since we treat k as a constant, the resulting graph has linear size and all our results still hold.

7.2 Directed Graphs

It is a very natural step to extend our results to directed input graphs. It can be achieved by minor modifications in the construction of the shortcut graph. Let G be the input directed graph and let G^u be the underlying undirected graph. In the directed case, the graph $\overrightarrow{G_1}$ is an $\mathcal{O}(1)$-orientation of G^u. Moreover, now for every edge $e = xy \in S_k(G)$ we store *two* directed paths in G corresponding to e: path $p_G(x, y)$ is a corresponding directed path from x to y and path $p_G(y, x)$ is a corresponding directed path from y to x. At most one of these paths may not exist and then we store an empty path in a corresponding variable. In the directed case, the graphs G_i for $i > 1$ are defined as follows: $uv \in E(G_i)$ if and only if there exists a vertex x and integers a, b such that $(x, u) \in E(G_a)$ and $(x, v) \in E(G_b)$ for $a + b = i$ and at least one of the following conditions is satisfied: (i) $p_G(u, x)$ is non-empty and $p_G(x, v)$ is non-empty; (ii) $p_G(v, x)$ is non-empty and $p_G(x, u)$ is non-empty. Certainly when (i) is satisfied, we assign $p_G(u, v)$ to a path contained in the walk $p_G(u, x) \cup p_G(x, v)$. We proceed similarly when (ii) holds. Now for every $i > 1$

$$G_i \subseteq \bigcup_{\substack{1 \le p \le q < i \\ p+q=i}} G_p \odot G_q$$

Thus, again by Corollary 4.3 for every i such that $1 < i \le k$ graph G_i is a (G, i)-representation of G of thickness $\mathcal{O}(1)$ and one can find in linear time an $\mathcal{O}(1)$-orientation of G_i, denoted by $\overrightarrow{G_i}$. Now edge (u, v) in graph $\overrightarrow{G_i}$ indicates that there is a directed path between u and v in G of length at most i.

Let p be a directed path in $\overrightarrow{S_k(G)}$. We call p a *forward path* if for every edge $(x, y) \in p$ the path $p_G(x, y)$ is non-empty. Similarly, p is a *backward path* if for every edge $(x, y) \in p$ the path $p_G(y, x)$ is non-empty. Note that p can be forward and backward path at the same time. It is straightforward to verify that to show the following result it suffices to slightly modify the proof of Theorem 5.3.

THEOREM 7.1 (A COUNTERPART OF TH. 5.3 FOR DIGRAPHS). *Let G be a directed planar graph and let $\overrightarrow{S_k(G)}$ be the shortcut graph. For any directed path of length $l \le k$ in G from a vertex u to a vertex v there there is a vertex x such that $\overrightarrow{S_k(G)}$ contains a forward u, x-path and a backward v, x-path and the sum of the weights of the paths is at most l.*

The above theorem guarantees the correctness of the shortest path query algorithm in the directed case. As the shortcut graph is $\mathcal{O}(1)$-oriented the query time remains constant. In fact the directed case is so similar to the undirected one that all the other of our results easily extend to digraphs.

7.3 Searching For Paths Avoiding Link- or Node-Failures

In real-life networks such as road networks or huge computer networks it is common that some local failures occur. We can focus on situation when certain links or nodes may be out of service for some time and we want the short path queries to act like the damaged elements were absent in the input graph. We could simply remove the failure edges and vertices from graph G and add them again as soon as the failure is fixed. However, adding edges takes amortized polylogarithmic time. Instead, we can allow to perform new operations on edges and vertices of the input graph: *disable* and *enable*, both with only constant time complexity.

Initially, all the vertices and edges are marked as *enabled*. For each edge $e \in \overrightarrow{S_k(G)}$ we store a new list denoted $dS(e)$. The list $dS(e)$ stores all pairs $\{e_1, e_2\}$ which are supporting couples of e and such that at least one of edges e_1, e_2 is disabled.

Operation DISABLE(e) executed for an edge e acts similarly as 'delete': It starts from marking e as disabled. Then for each pair $(d, p) \in D(e)$ we move the pair $\{e, f\}$ referenced by the pointer p from the list $S(d)$ to the list $dS(d)$; if the list $S(d)$ becomes empty we disable edge d recursively. When we want to disable a vertex it suffices to mark it as disabled and disable all the edges leaving it.

Operation ENABLE(e) executed for an edge e works as follows. For each element (d, p) in list $D(e)$ we perform the following operations: let $\{e, f\}$ be the pair referenced by pointer p; if f is enabled move the pair $\{e, f\}$ from $dS(d)$ to $S(d)$. If f is enabled, d is disabled and d leaves an enabled vertex execute ENABLE(d) recursively. Again, to enable a vertex v it suffices to mark it as enabled and enable every edge e leaving v such that $S(e)$ is not empty.

To show that the above operations work in constant time we can use the same arguments as in the case of deletions. Certainly, we assume that query algorithms ignore disabled edges and vertices – it is straightforward to see that the queries are answered properly. Note that there can be an arbitrary number of disabled edges and vertices.

7.4 An Application: Computing the Girth

Let k be a fixed constant and G a planar graph. We consider the following problem: verify whether the girth of G is bounded by k and if so compute the corresponding shortest cycle. We show here a simple linear-time algorithm for this problem. The algorithm uses the shortest graph $\overrightarrow{S_k(G)}$ adapted to allow enabling and disabling edges (see the previous section). After building the shortcut graph we perform the following three operations for each edge $e = (v, w) \in G$:

(i) disable e,

(ii) verify whether $\mathrm{dist}_G(v, w) \leq k - 1$ and if so find the cycle compound of e and the shortest path between v and w; store this cycle if it is shorter than the cycles found before,

(iii) enable e.

Each step takes $\mathcal{O}(1)$ time. It follows that the whole algorithm works in linear time.

7.5 Searching for Paths of Given Length

In this section we show how to modify our structure to allow a new type of queries called *given length path queries*. For two given vertices u and v and an integer $t \leq k$ the query returns a u-v path of length exactly t or reports that there is no such path in graph G. The new query is answered in constant time. As in these queries we allow $u = v$ it is also possible to find a cycle of specified fixed length containing given vertex in $\mathcal{O}(1)$ time. Obviously we can use it for finding given fixed length cycle in planar graph in linear time.

7.5.1 *Generating Walks.* Let W be a walk of weight t in $S_k(G)$ and let W contain an edge $e = uv$ of weight greater than 1. Edge e has at least one supporting couple, say $\{(x,u),(x,v)\}$. Then we can replace e by path uxv obtaining another walk of weight t in $S_k(G)$. We can repeat this step several times each time obtaining a walk of weight t. Let W' be the resulting walk. We say that W generates W'.

Formally, walk W *generates* walk W' if there is a sequence of walks $W_0, W_1, \ldots W_l$ such that $W_0 = W$, $W_l = W'$ and for every $i = 1, \ldots l$ the walk W_i can be obtained from W_{i-1} by replacing some edge $e = uv \in E(W_{i-1})$ by path uxv such that edges $(x,u),(x,v) \in \overrightarrow{S_k(G)}$ form a supporting couple of e.

Note that if the sequence of walks is long enough we obtain a walk in the initial graph G. Since an edge may have many supporting couples it follows that a walk may generate many walks in G. Observe that in particular every walk generates itself. For convenience we will extend the notion of generating to paths, since we can treat paths as walks. Hence a path may generate another path or a walk. We will also use the idea of generating for directed walks or paths, i.e. directed walk $\overrightarrow{W}$ generates walk W' iff its undirected version W generates W'. We will also say that edge e generates a walk W when the path consisting of edge e generates W.

Assume we are looking for a path of length t between u and v. If it exists Theorem 5.3 implies that there are two directed paths beginning in u and v and meeting in a common vertex x such that their weights sum up to t. We can find such pair of paths. From these paths we can generate two walks in G. Further, these two walks form a walk of length t joining u and v in G. However, we cannot be sure that this walk is a simple path. Even if we consider all walks generated by all pairs of paths Theorem 5.3 does not imply that any of these walks is a simple path. Hence we need a stronger result:

THEOREM 7.2. *For every u,v-path P in G of length at most k (possibly $u = v$) there exists a vertex $x \in P$ and two directed paths in $\overrightarrow{S_k(G)}$, $\overrightarrow{P_1}$ from u to x and $\overrightarrow{P_2}$ from v to x, such that walk $P_1 \cup P_2$ generates P.*

PROOF. The proof is analogous to the proof of Theorem 5.3. We use the induction on the length of P. When the length is 1 the theorem is obvious.

Now let us move to the induction step. For any vertices $a, b \in V(P)$ let $P[a,b]$ denote the path between a and b consisting of edges of P. As before, let u_1 be the neighbor of u on path P. By the induction hypothesis there are two dipaths in the shortcut graph, $\overrightarrow{Q_1}$ from u_1 to x and $\overrightarrow{Q_2}$ from v to x, such that $Q_1 \cup Q_2$ generates $P[u_1, v]$. Let $u_1, u_2, \ldots u_r = x$ be successive vertices of path Q_1. Let $\mathrm{dist}_{Q_1}(a,b)$ denote the distance between vertices a and b in path Q_1, treated as a weighted

graph. We define a set Z as follows:

$$Z = \{i : 1 \leq i \leq r \text{ and } E(G_{1+\mathrm{dist}_{Q_1}(u_1,u_i)}) \text{ contains edge } uu_i \text{ which generates path } P[u,u_i]\}.$$

Again, since $uu_1 \in E(G_1)$ and uu_1 generates itself, i.e. path $P[u,u_1]$, we get that $1 \in Z$ and $Z \neq \emptyset$. We can finish the proof exactly like the proof of Theorem 5.3. □

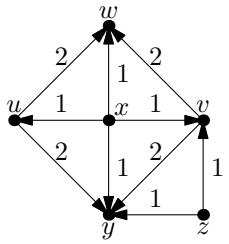

Fig. 11. Searching for a u,v-path of length 4.

7.5.2 *Difficulties to Cope With.* Consider the shortcut graph presented in Figure 11. Assume we are looking for a path of length $t = 4$ between vertices u and v. There is one such path, i.e. path $uxyzv$. Theorem 7.2 says us that it is generated by some pair of dipaths beginning in u and v and meeting in the same vertex. There is a path of weight 2 from u to w consisting of edge (u,w). There is also a path of weight 2 from v to w and it consists of edge (v,w). The weights of these paths sum up to 4 but they do not generate any simple path in G.

Luckily, this is not a serious problem – we can consider all pairs of directed paths beginning in u and v and meeting in the same vertex. Since the outdegrees in the shortcut graph are bounded the number of such pairs is also bounded. Let us come back to the situation in Figure 11. There is one more pair of paths: the first path consists of edge (u,y) and the second one of edge (v,y). The first path generates path uxy. Edge (v,y) has two supporting couples, i.e. $\{(x,y),(x,v)\}$ and $\{(z,y),(z,v)\}$. Hence, our pair of dipaths generates two walks in G: $uxyxv$ and $uxyzv$. Only the latter one is a simple path. We see that it is not enough to generate only one walk in G for each pair of paths. However since an edge may have $\Omega(n)$ supporters a path may generate $\Omega(n)$ walks. Hence we cannot generate all walks because would like to get constant query time. We will cope with this problem as follows:

—We observe that some supporting couples are "useless". For example see Figure 12. Edge $uv \in E(G_4)$ has one supporting couple, i.e. $(x,u),(x,v)$. However, the path consisting of edges $(x,u),(x,v)$ does not generate any simple path in G. We will not store in the shortcut graph useless supporting couples.

—During execution of the query algorithm we will generate only a constant-sized set of carefully selected walks in G. We will show that if none of these walks is a simple path then in G there is no simple path of requested length.

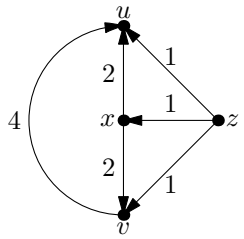

Fig. 12. Useless supporting couple.

7.5.3 *Algorithm for Given Length Path Query.* We will say that $S_k(G)$ is l-*clean* when for every edge e of weight $w \leq l$ and for every supporting couple of e, say $r = \{e_1, e_2\}$, the path consisting of edges e_1, e_2 generates some simple path in G. We will call this simple path *a path corresponding to* r. Note that the length of this path is equal to the weight of e. We assume that in an l-clean shortcut graph every supporting couple r of an edge of weight at most l stores a corresponding path (an arbitrary one if there are many of them) denoted as $p_G(r)$.

Now we describe the given length path query algorithm which finds a path in G of given length t between a given pair of vertices u and v. The algorithm uses information stored in the shortcut graph. We assume that the shortcut graph is k-clean. In Section 7.5.4 we discuss how to make the shortcut graph k-clean in linear time.

As we mentioned before, in our algorithm we consider all pairs of paths $\overrightarrow{P_1}, \overrightarrow{P_2}$ in $\overrightarrow{S_k(G)}$ such that $\overrightarrow{P_1}$ is a directed path from u to certain vertex x, $\overrightarrow{P_2}$ is a directed path from v to x, and the weights of paths $\overrightarrow{P_1}$ and $\overrightarrow{P_2}$ sum up to t. Since $\overrightarrow{S_k(G)}$ is $\mathcal{O}(1)$-oriented, the number of such pairs of paths is bounded by a constant.

Assume that the requested path between u and v exists and it is denoted by P. For each pair of dipaths $\overrightarrow{P_1}, \overrightarrow{P_2}$ described above we consider the walk in $S_k(G)$ corresponding to the union of undirected paths P_1 and P_2. Theorem 7.2 implies that one of these walks generates P. Clearly we can restrict ourselves only to walks which correspond to simple paths in $S_k(G)$. Let Q be such the path. In the following we will show how to find in constant time a simple path in G which is generated by a simple path Q in $S_k(G)$.

Let λ be some constant which will be defined later. A walk W will be called a λ-*walk* if each edge of W either has weight 1 or has at least λ supporting couples. A path is called a λ-*path* when it corresponds to a λ-walk.

The algorithm for finding a simple path in G generated by a path Q consists of two steps.

Step One: From a Path Q in $S_k(G)$ to a λ-path Q_λ. We execute routine λ-List(Q); see Alg. 7.1. It is clear that it returns a set of λ-walks. We will show that it is of bounded size. The recursion depth is at most $k-1$, because the initial path Q has weight at most k and before each recursive call the path passed as the parameter grows by one edge but its weight is not altered. Each invocation of routine λ-List performs fewer than λ recursive calls. Hence the size of the returned set of walks is bounded by $(\lambda-1)^{k-1}$, which is a constant. Consequently algorithm

λ-List works in constant time.

Algorithm 7.1 (λ-List)

Input: path Q in $S_k(G)$
Output: a set of λ-walks generated by Q.

λ-List(Q):
1: **if** Q is a λ-walk **then return** $\{Q\}$
2: **else**
3: $\mathcal{L} \leftarrow \emptyset$
4: $e \leftarrow$ an edge of Q of weight > 1 and such that $|S(e)| < \lambda$
5: **for all** $r \in S(e)$ **do**
6: $Q' \leftarrow$ path obtained from Q by replacing e by $p_G(r)$
7: $\mathcal{L} \leftarrow \mathcal{L} \cup \lambda$-List($Q'$)
8: **return** $\mathcal{L}$

Observe that if Q generates some walk, say w, then some λ-walk returned by λ-List(Q) generates w. Hence, if none of the returned walks is a simple path there is no simple path in G generated by Q. Otherwise, let Q_λ be any λ-path generated by Q. If Q_λ is a path in G, i.e. all edge weights in Q_λ are equal to 1 we are done. In the other case we perform step two.

Step Two: Finding a Simple Path P in G generated by Q_λ. In order to find a path in G that is generated by Q_λ we apply the following simple method (see Alg. 7.2: λ-Search). While Q_λ contains an edge of weight > 1 we try to replace it by a path corresponding to one of its supporting couples. If the resulting walk is not a path we try another supporting couple. Below we prove Proposition 7.4 which guarantees that when λ is large enough, after checking at most λ supporting couples we obtain a simple path. It follows that the algorithm *always* finds a path in G generated by Q_λ and it works in constant time.

Algorithm 7.2 (λ-Search)

Input: λ-path Q_λ in $S_k(G)$
Output: path P in G generated by Q_λ .

1: $P \leftarrow Q_\lambda$
2: **while exists** $xy \in E(P)$ **such that** xy has weight > 1 **do**
3: $X \leftarrow$ arbitrary λ supporting couples of xy
4: **for all** $r = \{(z, x), (z, y)\} \in X$ **do**
5: **if** $(V(p_G(r)) \setminus \{x, y\}) \cap V(P) = \emptyset$ **then**
6: Replace the edge xy in path P by the path $p_G(r)$.
7: **exit** from the **for** loop
8: **return** P

We say that edge $xy \in S_k(G)$ *crosses* vertex x when xy generates a path R such that x is an inner vertex of R.

LEMMA 7.3. *There exists a constant $\lambda_1(k)$ such that for every vertex x the number of edges in $S_k(G)$ crossing x is bounded by $\lambda_1(k)$.*

PROOF. It suffices to prove that for any weight $w = 1, \ldots, k$ the number of edges of weight w crossing x, which will be denoted by $\gamma(w)$, is bounded by a constant – we can add these constants to get $\lambda_1(k)$.

Obviously, there are no edges of weight 1 crossing x. Hence we can put $\gamma(1) = 0$. Now let us consider an edge $e = uv$ of weight $w > 1$ that crosses x. There are two types of such edges. Edges of type 1 have a supporting couple of the form $(x,u),(x,v)$. Recall that $\Delta(k)$ denotes the constant that bounds outdegrees in $\overrightarrow{S_k(G)}$. Clearly, the number of type 1 edges is bounded by $(\Delta(k))^2$. When e is not of type 1 then it has is a supporting couple $(z,u),(z,v)$ such that one of edges zu, zv crosses x. Note that both zu and zv have weights smaller than w. Assume that zu crosses x and let i be its weight. Then there is at most one edge (z,v) of weight $w-i$. Hence the number of type 2 edges is bounded by $\sum_{i=1}^{w-1}\gamma(i)$. It follows that for $w > 1$, $\gamma(w) \leq (\Delta(k))^2 + \sum_{i=1}^{w-1}\gamma(i)$. As $\Delta(k)$ is bounded, for each $w = 1, \ldots, k$, $\gamma(w)$ is also bounded and consequently $\lambda_1(k)$ is a constant. □

PROPOSITION 7.4. *Let $S_k(G)$ be an l-clean shortcut graph. Then there exists a constant $\lambda(k)$ such that for arbitrary λ-path Q_λ with each edge of weight at most l Algorithm 7.2 finds a simple path in G generated by Q_λ.*

PROOF. We put

$$\lambda(k) := k(\lambda_1(k) + k - 1) + 1.$$

Let xy be an edge chosen by Algorithm 7.2. The algorithm picks a supporting couple $r = \{(z,x),(z,y)\}$ and verifies whether after replacing xy in the path P by $p_G(r)$ one obtains a path. If so, r is called *successful*. Otherwise, r is called *failed* and subsequent supporting couples are checked. Observe that if r is failed then (a) one of edges zx, zy crosses some vertex of P or (b) $z \in V(P)$. (It may happen that both conditions are satisfied). Observe that since P has at most k vertices the number of supporters of xy that cross one of these vertices is bounded by $k\lambda_1(k)$. Hence there are at most $k\lambda_1(k)$ failed couples of type (a). Now note that for any vertex $q \in V(P)$ there are at most $k-1$ supporting couples $\{(q,x),(q,y)\}$, since weights of (q,x) and (q,y) sum up to weight of xy which does not exceed k and in graphs $\overrightarrow{G_i}$ each pair of vertices is joined by at most one edge. Since $|V(P)| \leq k$ it follows that there are at most $k(k-1)$ failed couples of type (b). Hence there can be at most $k(\lambda_1(k) + k - 1)$ failed supporting couples of xy. It follows that after checking at most $\lambda(k)$ supporting couples the algorithm finds a successful one. □

In steps one and two we showed how to find a simple path P in G that is generated by a path Q in the shortcut graph. Let $\mathrm{mw}(Q)$ be the maximal edge weight in path Q. Note that these steps take constant time, provided that the shortcut graph is $\mathrm{mw}(Q)$-clean. We will need this fact in the following subsection.

7.5.4 *Cleaning the Shortcut Graph.* Now let us show how to make a shortcut graph k-clean in linear time. Observe that $S_k(G)$ is always 2-clean. We make it 3-clean as follows.

Let e be an arbitrary edge of weight 3 and let $r = \{e_1, e_2\}$ be one of its supporting couples. Let Q be the path consisting of edges e_1, e_2. Observe that the maximal

edge weight in Q is at most 2. Hence using steps one and two from the previous section we can verify in constant time whether Q generates a simple path in G. If so, we store the path we found as $p_G(r)$. Otherwise we remove r from the shortcut graph. If e has no more supporting couples we remove e in constant time using the algorithm from Section 6.1.2. Clearly, if we apply this algorithm to each supporting couple of each edge in $E(G_3)$ the shortcut graph becomes 3-clean.

Analogously, having c-clean shortcut graph we can make it $(c+1)$-clean in linear time. After $k-2$ such phases we obtain k-clean shortcut graph.

8. GENERALIZATIONS

It is clear that all the algorithms presented in this paper can be performed for an arbitrary input graph. Nevertheless the performance of the shortcut graph structure was analyzed for the class of planar graphs. We can ask a natural question: which properties of planar graphs were actually needed in our proofs? In other words, what conditions should a class of graphs $\mathcal{C}$ satisfy to replace planar graphs in our approach? One can easily check that planarity was used only twice. Recall orienting edges of G_i in section 5.1. It was needed that every subdrawing of a planar graph is planar and it can be $\mathcal{O}(1)$-oriented in linear time. The second time was in Section 6.3 when we used the fact that planar graphs have bounded arboricity. However, any 1-orientation is a collection of paths and cycles and has arboricity at most 2. Hence a graph that can be $\mathcal{O}(1)$-oriented has bounded arboricity. Thus it suffices to require that:

(a) $\mathcal{C}$ is closed under taking subdrawings,
(b) there exists a constant d such that every graph from $\mathcal{C}$ can be d-oriented in linear time.

Let us consider the following three more basic conditions:

(i) $\mathcal{C}$ is *sparse*, i.e. there exists a constant c such that for every $G \in \mathcal{C}$, $|E(G)| \leq c|V(G)|$,
(ii) $\mathcal{C}$ is closed under taking subgraphs,
(iii) if a graph G from $\mathcal{C}$ is a subdivision of a graph H, then $H \in \mathcal{C}$.

We claim that class $\mathcal{C}$ satisfies conditions (a) and (b) if and only if it satisfies (i)–(iii). (b) implies (i) – we can put $c = d$. Trivially (a) implies (ii) since every subgraph is a 1-subdrawing. Also (a) implies (iii) because if G is a subdivision of H then H is a subdrawing of G. On the other hand (ii) and (iii) imply (a). Now assume (i) and (ii). It is easy to show that every graph in $\mathcal{C}$ contains a vertex of degree at most $2c$ and therefore a simple greedy algorithm $(2c)$-orients graphs from $\mathcal{C}$ in linear time (it was observed by Aichholzer *et al.* [1995]). It follows that (b) is satisfied with constant $d = 2c$. Thus (i) and (ii) imply (b) and the proof of the equivalence of conditions (a)-(b) and (i)-(iii) is finished.

COROLLARY 8.1. *For any class of graphs satisfying conditions (i)-(iii) the shortcut graph is created in linear time. The shortest path queries and the queries on given length paths are serviced in constant time. Updating the shortcut graph after deleting a vertex or an edge and after adding a vertex takes constant time while updates after edge insertions are processed in amortized* $\mathcal{O}(\log^k n)$ *time.*

Observe that if a class of graphs is closed under taking minors then it immediately implies that conditions (ii) and (iii) above are satisfied. This proves the following proposition.

PROPOSITION 8.2. *Any sparse class of graphs which is closed under taking minors satisfies conditions (i)-(iii).* □

Let $\mathrm{Forb}_{\preccurlyeq}(\mathcal{H})$ denote the class of all graphs without a minor in $\mathcal{H}$. In the rest of this section we show that when $\mathcal{H}$ is a finite set of fixed graphs then $\mathrm{Forb}_{\preccurlyeq}(\mathcal{H})$ satisfies our three conditions. The following lemma is due to Mader [1968] (see also textbook by R. Diestel [2000]). *Average degree* of graph G is defined as $\frac{1}{|V(G)|}\sum_{v\in V(G)} \deg_G(v) = 2|E(G)|/|V(G)|$.

LEMMA 8.3. *There is a function $h : \mathbb{N} \to \mathbb{N}$ such that every graph of average degree at least $h(r)$ contains K_r as a minor, for every $r \in \mathbb{N}$.*

PROPOSITION 8.4. *For any finite set of fixed graphs $\mathcal{H}$, the class $\mathrm{Forb}_{\preccurlyeq}(\mathcal{H})$ satisfies conditions (i)-(iii).*

PROOF. Since the minor relation is transitive $\mathrm{Forb}_{\preccurlyeq}(\mathcal{H})$ is closed under taking minors and hence by Proposition 8.2 it suffices to show that $\mathrm{Forb}_{\preccurlyeq}(\mathcal{H})$ is sparse. Let $G \in \mathrm{Forb}_{\preccurlyeq}(\mathcal{H})$ and $M \in \mathcal{H}$. As G does not contain M as a minor, then also $K_{|V(M)|}$ is not a minor of G. Then by Lemma 8.3 the average degree of G is smaller than $h(|V(M)|)$, which is a constant number. Since G was an arbitrary graph in $\mathrm{Forb}_{\preccurlyeq}(\mathcal{H})$, then this class is sparse. It establishes the proof. □

By the Graph Minor Theorem any minor-closed family of graphs can be defined as $\mathrm{Forb}_{\preccurlyeq}(\mathcal{H})$ for some finite set of graphs $\mathcal{H}$. Hence we get the following corollary.

COROLLARY 8.5. *Any minor-closed class of graphs satisfies conditions (i)-(iii).*

ACKNOWLEDGMENTS

We thank Krzysztof Diks for comments on early versions of this paper and many fruitful discussions. We are grateful to Mikkel Thorup for pointing out the reference [Thorup 2001] to shortest paths oracles in planar digraphs. We are also deeply indebted for anonymous referees for their excellent work which helped to improve the presentation of the paper.

REFERENCES

AHUJA, R. K., MAGNANTI, T. L., AND ORLIN, J. B. 1993. *Network Flows: Theory, Algorithms and Applications*. Prentice Hall, Englewood Cliffs.

AICHHOLZER, O., AURENHAMMER, F., AND ROTE, G. 1995. Optimal graph orientation with storage applications. SFB-Report F003-51, SFB 'Optimierung und Kontrolle', TU Graz, Austria.

ALON, N., YUSTER, R., AND ZWICK, U. 1997. Finding and counting given length cycles. *Algorithmica 17,* 3, 209–223.

ARIKATI, S. R., CHEN, D. Z., CHEW, L. P., DAS, G., SMID, M. H. M., AND ZAROLIAGIS, C. D. 1996. Planar spanners and approximate shortest path queries among obstacles in the plane. In *European Symposium on Algorithms*. 514–528.

BRODAL, G. S. AND FAGERBERG, R. 1999. Dynamic representations of sparse graphs. In *Proc. 6th International Workshop on Algorithms and Data Structures*. Lecture Notes in Computer Science, vol. 1663. 342–351.

Chen, D. A. and Xu, J. 2000. Shortest path queries in planar graphs. In *Proc. of the 32nd Annual ACM Symposium on Theory of Computing*. 469–478.

Chiba, N. and Nishizeki, T. 1985. Arboricity and subgraph listing algorithms. *SIAM Journal on Computing 14,* 1, 210–223.

Chrobak, M. and Eppstein, D. 1991. Planar orientations with low out-degree and compaction of adjacency matrices. *Theoretical Computer Science 86,* 2, 243–266.

Demetrescu, C. and Thorup, M. 2002. Oracles for distances avoiding a link-failure. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 838–843.

Diestel, R. 2000. *Graph Theory*. Springer-Verlag.

Djidjev, H. 1996. Efficient algorithms for shortest path queries in planar digraphs. In *Proc. 22nd Int. Worksh. Graph-Theoretic Concepts in Computer Science (WG 1996)*. 151–165.

Djidjev, H. 2000. Computing the girth of a planar graph. In *Proc. 27th International Colloquium on Automata, Languages and Programming*. 821–831.

Eppstein, D. 1999. Subgraph isomorphism in planar graphs and related problems. *J. Graph Algorithms & Applications 3,* 3, 1–27.

Federickson, G. N. 1987. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM Journal on Computing 16,* 6 (Dec.), 1004–1022.

Frederickson, G. N. 1997. Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. *SIAM Journal on Computing 26,* 2 (April), 484–538.

Graham, R., Knuth, D., and Patashnik, O. 1994. *Concrete Mathematics*. Addison-Wesley.

Klein, P. 2002. Preprocessing an undirected planar network to enable fast approximate distance queries. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 820–827.

Kowalik, Ł. 2003. Short cycles in planar graphs. In *Proc. 29th Int. Worksh. Graph-Theoretic Concepts in Computer Science (WG 2003)*, H. Bodlaender, Ed. Lecture Notes in Computer Science, vol. 2880. Springer-Verlag, 284–296.

Lipton, R. J. and Tarjan, R. E. 1979. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics 36,* 2 (Apr.), 177–189.

Mader, W. 1968. Homomorphiesätze für graphen. *Math. Ann. 178*, 154–168.

Papadimitriou, C. H. and Yannakakis, M. 1981. The clique problem for planar graphs. *Information Processing Letters 13,* 4–5, 131–133.

Ramalingam, G. 1996. *Bounded Incremental Computation*. LNCS, vol. 1089. Springer.

Richards, D. 1986. Finding short cycles in planar graphs using separators. *Journal of Algorithms 7*, 382–394.

Thorup, M. 2001. Compact oracles for reachability and approximate distances in planar digraphs. In *Proceedings of the 42nd IEEE Symposium on Foundations of Computer Science*. 242–251.

Received Month Year; revised Month Year; accepted Month Year;