Dynamic Planar Embeddings of Dynamic Graphs

Jacob Holm and Eva Rotenberg

DIKU, Dept. of Computer Science, University of Copenhagen, Denmark
jh@poplar.dk, roden@di.ku.dk

Abstract

We present an algorithm to support the dynamic embedding in the plane of a dynamic graph. An edge can be inserted across a face between two vertices on the boundary (we call such a vertex pair linkable), and edges can be deleted. The planar embedding can also be changed locally by flipping components that are connected to the rest of the graph by at most two vertices. Given vertices u, v, linkable(u, v) decides whether u and v are linkable, and if so, returns a list of suggestions for the placement of (u, v) in the embedding. For non-linkable vertices u, v, we define a new query, one-flip-linkable(u, v) providing a suggestion for a flip that will make them linkable if one exists. We will support all updates and queries in $O(\log^2 n)$ time. Our time bounds match those of Italiano et al. for a static (flipless) embedding of a dynamic graph. Our new algorithm is simpler, exploiting that the complement of a spanning tree of a connected plane graph is a spanning tree of the dual graph. The primal and dual trees are interpreted as having the same Euler tour, and a main idea of the new algorithm is an elegant interaction between top trees over the two trees via their common Euler tour.

1998 ACM Subject Classification E.1 Data Structures, F.2.2 Nonnumerical Algorithms and Problems, G.2.2 Graph Theory

Keywords and phrases dynamic graphs, planar embeddings, data structures

Digital Object Identifier 10.4230/LIPIcs.STACS.2015.434

1 Introduction

We present a data structure for supporting and maintaining a dynamic planar embedding of a dynamic graph. In this article, a *dynamic graph* is a graph where edges can be removed or inserted, and vertices can be cut or joined, but where an edge (u, v) can only be added if it does not violate planarity. More precisely, the edges around each vertex are ordered cyclically by the embedding, similar to the edge-list representation of the graph. A *corner* (of a face) is the gap between consecutive edges incident to some vertex. Given two corners c_u and c_v of the same face f, incident to the vertices u and v respectively, the operation insert(c_u, c_v) inserts an edge between u and v in the dynamic graph, and embeds it across f via the specified corners. We provide an operation linkable(u, v) that returns such a pair of corners c_u and c_v if they exist. If there are more options, we can list them in constant time per option after the first. A vertex may be cut through two corners, and linkable vertices may be joined by corners incident to the same face, if they are connected, or incident to any face otherwise. That is, joining vertices corresponds to linking them across a face with some edge e, and then contracting e.

It may often be relevant to change the embedding, e.g. in order to be able to insert an edge. In a *dynamic embedding*, the user is allowed to change the embedding by what we call flips, that is, to turn part of the graph upside down in the embedding. Of course, the relevance of this depends on what we want to describe with a dynamic plane graph. If the application is to describe roads on the ground, flipping orientation would not make much

32nd Symposium on Theoretical Aspects of Computer Science (STACS 2015).
Editors: Ernst W. Mayr and Nicolas Ollinger; pp. 434–446

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany


sense. But if we have the application of graph drawing or chip design in mind, flips are indeed relevant. In the case of chip design, a layer of a chip is a planar embedded circuit, which can be thought of as a planar embedded graph. An operation similar to flip is also supported by most drawing software.

Given two vertices u, v, we may ask whether they can be linked after modifying the embedding with only one flip. We introduce a new operation, the one-flip-linkable(u, v) query, which answers that question, and returns the vertices and corners describing the flip if it exists.

Our data structure is an extension to a well-known duality-based dynamic representation of a planar embedded graph known as a tree-cotree decomposition [4]. We maintain top-trees [1] both for the primary and dual spanning trees. We use the fact that they share a common (extended) Euler tour - in a new way - to coordinate the updates and enable queries that either tree could not answer by itself. All updates and queries in the combined structure are supported in $O(\log^2 n)$, plus, in case of linkable(u, v), the length of the returned list.

1.1 Dynamic Decision Support Systems

An interesting and related problem is that of dynamic planarity testing of graphs. That is, we have a planar graph, we insert some edge, is the graph still planar, that is, does there still exist an embedding of it in the plane?

The problem of dynamic planarity testing appears technically harder than our problem, and in its basic form it is only relevant when the user is completely indifferent to the actual embedding of the graph. What we provide here falls more in the category of a decision support system for the common situation where the desired solution is not completely captured by the simple mathematical objective, in this case planarity. We are supporting the user in finding a good embedding, e.g., telling what are the options for inserting an edge (the linkable query), but leave the actual choice to the user. We also support the users in changing their mind about the embedding, e.g. by flipping components, so as to make edge insertions possible. Using the one-flip-linkable query we can even suggest a flip that would make a desired edge insertion possible if one exists.

1.2 Previous work

Dynamic graphs have been studied for several decades. Usually, a fully dynamic graph is a graph that may be updated by the deletion or insertion of an edge, while decremental or incremental refers to graphs where edges may only be deleted or inserted, respectively. A dynamic graph can also be one where vertices may be deleted along with all their incident edges, or some combination of edge- and vertex updates [14].

Hopcroft and Tarjan [9] were the first to solve planarity testing of static graphs in linear time. Incremental planarity testing was solved by La Poutre [12], who improved on work by Di Battista, Tamassia, and Westbrook [2,3,15], to obtain a total running time of $O(\alpha(q, n))$ where q is the number of operations, and where α is the inverse-Ackermann function. Galil, Italiano, and Sarnak [8] made a data structure for fully dynamic planarity testing with $O(n^{2/3})$ worst case time per update, which was improved to $O(n^{1/2})$ by Eppstein et al. [5]. For maintaining embeddings of planar graphs, Italiano, La Poutré, and Rauch [10] present a data structure for maintaining a planar embedding of a dynamic graph, with time $O(\log^2 n)$ for update and for linkable-query, where insert only works when compatible with the embedding. The dynamic tree-cotree decomposition was first introduced by Eppstein et al. [6] who used it to maintain the MST of a planar embedded dynamic graph subject to a sequence of

change-weight$(e, \Delta x)$ operations in $O(\log n)$ time per update. Eppstein [4] presents a data structure for maintaining the MST of a dynamic graph, which handles updates in $O(\log n)$ if the graph remains plane. More precisely the user specifies a combinatorial embedding in terms a cyclic ordering of the edges around each vertex. Planarity of the user specified embedding is checked using Euler's formula. Like our algorithm, Eppstein's supports flips. The fundamental difference is that Eppstein does not offer any support for keeping the embedding planar, e.g., to answer linkable(u, v), the user would in principle have to try all n^2 possible corner pairs c_u and c_v incident to u and v, and ask if insert(c_u, c_v) violates planarity.

As far as we know, the one-flip-linkable query has not been studied before. Technically it is the most challenging operation supported in this paper.

The highest lower bound for the problem of planarity testing is Pătraşcu's $\Omega(\log n)$ lower bound for fully dynamic planarity testing [13]. From the reduction it is clear that this lower bound holds as well for maintaining an embedding of a planar graph as we do in this article.

2 Maintaining a dynamic embedding

In this section we present a high-level overview of a data structure to maintain a dynamic embedding of a planar graph. In the following, unless otherwise stated, we will assume $G = (V, E)$ is a planar graph with a given combinatorial embedding and that $G^* = (F, E^*)$ is its dual.

Our primary goal is to be able to answer linkable(u, v), where u and v are vertices: Determine if an edge between u and v can be added to G without violating planarity and without changing the embedding. If it can be inserted, return the list of pairs of *corner*s (see Definition 3 below). Each corner-pair, (c_u, c_v), describes a unique place where such an edge may be inserted. If no such pair exists, return the empty list.

The data structure must allow efficient updates such as insert, remove, cut, join, and flip. We defer the exact definitions of these operations to Section 2.4.

As in most other dynamic graph algorithms we will be using a spanning tree as the main data structure, and note:

▶ **Observation 1.** *If $E_T \subseteq E$ induces a spanning tree T in G, then $(E \setminus E_T)^*$ induces a spanning tree $\overline{T}^*$ in G^* called the* co-tree *of T.*

▶ **Observation 2.** *If u and v are vertices, T is spanning tree, and e is any edge on the T-path between u and v, then any face containing both u and v lies on the cycle induced by e^* in the co-tree $\overline{T}^*$.*

Thus the main idea is to search a path in the co-tree. This is complicated by the fact (see Figure 1) that the set of faces that are adjacent to u and/or to v need not be contiguous in $\overline{T}^*$, so it is possible for the cycle to change arbitrarily between faces that are adjacent to any combination of neither, one, or both of u, v.

Our linkable query will consist of two phases. A marking phase, in which we "activate" (mark) all corners incident to each of the two vertices we want to link (see Section 2.2), and a searching phase, in which we search for faces incident to "active" (marked) corners at both vertices (see Section 2.3). But first, we define corners.

2.1 Corners and the extended Euler tour

The concept of a corner in an embedded graph turns out to be very important for our data structure. Intuitively it is simply a place in the edge list of a vertex where you might insert a new edge, but as we shall see it has many other uses.

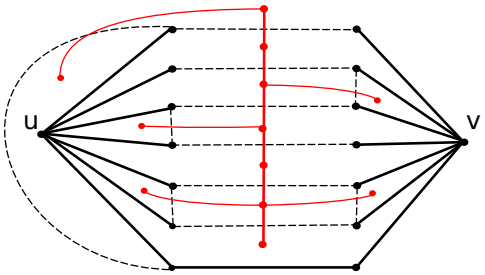

Figure 1 The co-tree path may switch arbitrarily between faces that are adjacent to any combination of neither, one, or both of u, v.

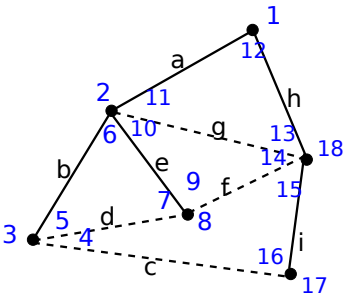

Figure 2 This graph has extended Euler Tour `1a2b3c4d5b6e7d8f9e...18h`, or, to write only the edges: **ab**cd**be**df**eg**g**ah**gf**ic**i**h**. Edges not in the spanning tree are drawn with dotted lines.

▶ **Definition 3.** If G is a non-trivial connected, combinatorially embedded graph, a *corner* in G is a 4-tuple (f, v, e_1, e_2) where f is a face, v is a vertex, and e_1, e_2 are edges (not necessarily distinct) that are consecutive in the edge lists of both v and f.
If $G = (\{v\}, \emptyset)$ and $G^* = (\{f\}, \emptyset)$, there is only one corner, namely the tuple (f, v).

Note that faces and vertices appear symmetrically in the definition. Thus, there is a one-to-one correspondence between the corners of G and the corners of G^*. This is important because it lets us work with corners in G and G^* interchangeably. Even more interesting is:

▶ **Observation 4.** *Given a spanning tree T of G, there is a natural extension of the concept of an Euler tour of T into an* extended *Euler tour* $\mathrm{EET}(T)$ *as a cyclic arrangement that contains each edge in G exactly twice and each corner in G exactly once (see Figure 2). Furthermore, the corresponding tour* $\mathrm{EET}(\overline{T}^*)$ *in G^* defines exactly the opposite cyclical arrangement of the corresponding edges and corners in G^*.*

Thus, segments of the extended Euler tour translate directly between T and $\overline{T}^*$. By *segment*, we mean any contiguous sub-list of the cycle.

The high level algorithm (to be explained in detail later) is now to build a structure consisting of an arbitrary spanning tree T and its co-tree $\overline{T}^*$ such that we can

1. Find an edge e on the T-path between u and v. This is easy.
2. Mark all corners incident to u and v in T. This is complicated by the existence of vertices of high degree, so a lazy marking scheme is needed. However, it is easier than marking them in $\overline{T}^*$ directly, since each vertex has a unique place in T and no place in $\overline{T}^*$.
3. Transfer those marks from T to $\overline{T}^*$ using Observation 4. We can do this as long as the lazy marking scheme works in terms of segments of the extended Euler tour.
4. Search the cycle induced by e^* in $\overline{T}^*$ for faces that are incident to a marked corner on both sides of the path.

2.2 Marking scheme

We need to be able to mark all corners incident to the two query vertices u and v, and we need to do it in a way that operates on segments of the extended Euler tour. To this end

Figure 3 The vertex w is a boundary vertex of the green clusters, but not of their blue parent clusters.

we will maintain a top tree over T (see [1]).

Given a tree T, a top tree for T is a binary tree. At the lowest level, its leaves are the edges of T. Its internal nodes, called *clusters*, are sub-trees of T with at most two boundary vertices. At the highest level its root is a single cluster containing all of T. A non-boundary vertex of the subset $S \subset V$ may not be adjacent to a vertex of $V \setminus S$. A cluster with two boundary vertices is called a *path cluster*, and other clusters are called *leaf clusters*. Any internal node is formed by the merged union of its (at most two) children. All operations on the top tree are implemented by first breaking down part of the old tree from the top with $O(\log n)$ calls to a *split* operation, end then building the new tree with $O(\log n)$ calls to a *merge* operation.

▶ **Observation 5.** *We can maintain a top tree over T such that each cluster consists of edges from at most two segments of* $\mathrm{EET}(T)$*, using $O(\log n)$ calls to* merge *and* split *per update. Path clusters will have edges from two segments, and leaf clusters, one segment.*

The operation *expose* on the top tree takes one or two vertices and make them boundary of the level root cluster. Modifying this only slightly, one may even *expose* corners, giving complete control over which $\mathrm{EET}(T)$ segment is available for information or modification. We may even expose any constant number of vertices or corners, but then at the highest level of the top tree, in stead of only T, we may have some constant number of clusters.

▶ **Observation 6.** *Whenever a merge of two clusters in the top tree causes a vertex w to stop being a boundary vertex (see Figure 3), all corners incident to w are contained in one or two* $\mathrm{EET}(T)$ *segments. These segments will be sub-segments of the (one or two) segments corresponding to the parent cluster C (blue in Figure 3), and will not contain any corners incident to the (one or two) boundary vertices of C.*

Now suppose we associate a (lazy) *deactivation count* with each corner that is 0 before we start building the top tree. Define the merge operation on the top tree such that whenever a merge discards a boundary vertex we *deactivate* all corners on the at most two segments of $\mathrm{EET}(T)$ mentioned in Observation 6 by increasing that count (and define the split operation on the top tree to reactivate them as necessary). When the top tree is complete, the corners that are still *active* (have deactivation count 0) are exactly those incident to the boundary vertices of the root of the top tree. These boundary vertices are controlled by the expose operation on the top tree and changing the boundary vertices require only $O(\log n)$ merges and splits, so we have now argued the following

▶ **Lemma 7.** *We can mark/unmark all corners incident to vertices u and v by increasing and decreasing the deactivation counts on $O(\log n)$ segments of the extended Euler tour.*

What we really want is to be able to search for the marked corners in $\overline{T}^*$, so instead of storing the counts (even lazily) in the top tree over T, we will store them in a top tree over $\overline{T}^*$. Again, each cluster in this top tree covers one or two segments of the extended Euler tour. For each segment S we keep track of the minimum deactivation count $c_{\min}(S)$, and a $\delta(S)$ that needs to be applied to all corners in the segment. To update the deactivation counts of an arbitrary segment S, all we need to do is modify the $O(\log n)$ clusters that are affected, which can be done in $O(\log n)$ time, leading to

▶ **Lemma 8.** *We can maintain a top tree over $\overline{T}^*$ that has $c_{\min}$ and δ values for each EET segment in each cluster in $O(\log^2 n)$ time per change to the marked u and v vertices.*

▶ **Observation 9.** *This is enough for, given a face f and a vertex u, checking whether f is incident to u.*

2.3 Linkable query

Unfortunately, the $c_{\min}$ and δ values discussed in Section 2.2 are not quite enough to let us find the corners we are looking for. We can use it to ask what marked corners a given face is incident to, but we do not have enough to find *pairs* of marked corners on opposite sides of the same face on the co-tree path.

As noted in Observation 2 all candidates to a common face for two given vertices u and v, must lie on some path in the dual tree. And a path which is easily found! Since the dual of a primal tree edge induces a cycle that separates u and v, we may use the path between the dual endpoints f, g of any edge on the primal tree path between u and v. Furthermore, once we expose the path (f, g) in the dual tree, if $f \neq g$, it will have two EET-segments: the minimum deactivation count of one EET-segment is 0 iff any non-endpoint faces are incident to v, the other iff any are incident to u. Checking the endpoint faces can be done (cf. Observation 9), but to find non-endpoint faces we need more structure.

To just output *one* common face, our solution is for each path cluster in the top tree over the co-tree to keep track of *a single* internal face $f_{\min}$ on the cluster path that is incident to minimally deactivated corners on either side of the cluster path if such a face exists.

▶ **Lemma 10.** *We can maintain a top tree over $\overline{T}^*$ that has $c_{\min}$ and δ values for each EET-segment in each cluster and $f_{\min}$ values for each path cluster in $O(\log^2 n)$ time per change to the marked u and v vertices.*

Proof. Each merge only has to check the at most two $f_{\min}$ values at the children and may discard or keep them based solely on the $c_{\min}$ and δ values available. ◀

▶ **Lemma 11.** *We can support each* linkable(u, v) *in $O(\log^2 n)$ time per operation.*

Proof. If u and v are not in the same connected component we pick any corners c_u and c_v adjacent to u and v and return them. Otherwise we use expose(u, v) on the top tree over T to activate all corners adjacent to u and v and to find an edge e on the T-path from u to v (e.g. the first edge on the path). Let f, g be the endpoints of e^*, and call expose(f, g) on the top tree over $\overline{T}^*$. Let h be the $f_{\min}$ value of the resulting root. We can now test each of f, g, h using the $c_{\min}$ values to find the desired corners if they exist. ◀

▶ **Lemma 12.** *If there are more valid answers to* linkable(u, v) *we can find k of them in $O(\log^2 n + k)$ time.*

Proof. For each leaf cluster and for each side of each path cluster we can maintain the list of minimally deactivated corners adjacent to each boundary vertex. Then, instead of maintaining a single face $f_{\min}$ for each path cluster, we can maintain a linked list of all relevant faces in the same time. And for each side of each face in the list we can point to a list of minimally deactivated corners that are adjacent to that side. For leaf-clusters, we point to a linked list of minimally deactivated corners incident to the boundary vertex. Upon the merge of clusters, face-lists and corner-lists may be linked together, and the point of concatenation is stored in the resulting merged cluster in case of a future split. Note that each face occurs in exactly one face-list.

As before, to perform linkable(u, v), expose u, v in the primal tree. Let e_0 be an edge on the tree-path between u and v, and expose the endpoints of e_0^* in the dual top tree. Now, the maintained face-list in the root of the dual top tree contains all faces incident to u, v, except maybe the endpoints of e_0^*, which can be handled separately, as before. The total time is therefore $O(\log^2 n)$ for the necessary expose operations, and then $O(1)$ for each reply. ◀

▶ **Observation 13.** *If we separately maintain a version of this data structure for the dual graph, then for faces f, g,* linkable(f, g) *in that structure lets us find vertices that are incident to both f and g.*

2.4 Updates

In addition to the query, our data structure supports the following set of update operations:

- insert(c_u, c_v) where c_u and c_v are corners that are either in different connected components, or incident to the same face. Adds a new edge to the graph, inserting it between the edges of c_u at one end and between the edges of c_v at the other. Returns the new edge.
- remove(e). Removes the edge e from the graph. Returns the two corners that could be used to insert the edge again.
- join(c_u, c_v) where c_u and c_v are corners that are either in separate components of the graph or in the same face. Combines the vertices u and v into a single new vertex w and returns the two new corners c_w and c'_w that may be used to split it again using cut(c_w, c'_w).
- cut(c_w, c'_w) where c_w and c'_w are corners sharing a vertex w. Splits the vertex into two new vertices u and v and returns corners c_u and c_w that might be used to join them again using join(c_u, c_v).
- flip(C) where C is any connected component of the graph. Reverses the order of the edges at each vertex/face cycle of the component.

When calling remove(e) on a non-bridge tree-edge e, we need to search for a replacement edge. Luckily, e^* induces a cycle in the dual tree, and any other edge on that cycle is a candidate for a replacement edge. If we like, we can augment the dual top tree so we can find the minimal-weight replacement edge, simply let each path cluster remember the cheapest edge on the tree-path, and expose the endpoints of e^*. If we want to keep T as a minimum spanning tree, we also need to check at each insert and join that we remove the maximum-weight edge on the induced cycle from the spanning tree.

In general, when we need to update both the top trees over T and $\overline{T}^*$ we must be careful that we first do the splits needed in the top tree over T to make each unchanged sub-tree into a (partial) top tree by itself, then update the top tree over $\overline{T}^*$ and finally do the remaining splits and merges to rebuild the top tree over T. This is necessary because the merge and split we use for T depend on T and $\overline{T}^*$ having related extended Euler tours.

Any change to the graph, especially to the spanning tree, implies a change to the extended Euler Tour. Furthermore, any deletion or insertion of an edge implies a merge or split in the dual tree. E.g. if an edge is inserted across a face, that face is split in two. As a more complex example, if the non-bridge tree-edge $e = (u, v)$ is deleted, the replacement edge is removed from the dual tree, and the endpoints of e^* are merged.

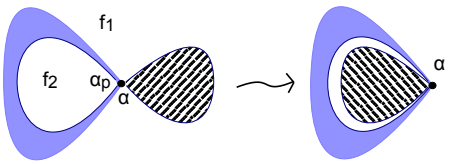

Figure 4 An articulation flip at the vertex α.

Figure 5 A separation flip at a separation pair (blue). The flip makes vertex u linkable with vertex v.

Finally, for flip to work we have to use a version of top trees that is not tied to a specific clockwise orientation of the vertices. The version in [1] that is based on a reduction to Frederickson's topology trees [7] works fine for this purpose.

▶ **Definition 14** (Articulation flip)**.** Having vertex split and vertex join functions, we may perform an *articulation-flip* — a flip in an articulation point: Given a vertex α incident to the face f_1 in two corners, c and c', we may cut through c, c', obtaining two graphs G_1, G_2, having split α in vertices $\alpha_1 \in G_1, \alpha_2 \in G_2$, and having introduced new corners c_1, c_2 where we cut. Now, given a corner α_p incident to α_1 and incident to some face f_2, we may join α_1 with α_2 by the corners α_2, α_p, with or without having flipped the orientation of G_2.

▶ **Definition 15** (Separation flip)**.** Similarly, given a separation pair α, β, incident to the faces f, g with corners $c_1, \ldots c_4$, we may split through those corners, obtaining two graphs. We may then flip the orientation of one of them, and rejoin. We call this a *separation-flip*.

3 One-flip linkable query

Given vertices u, v, we have already presented a data structure to find a common face for u, v. Given they do not share a common face, we will determine if an articulation flip exists such that an edge between them can be inserted, and given no such articulation-flip exists, we will determine if a separation-flip that makes the edge insertion (u, v) possible exists.

Let f_1 and f_2 be faces in G, and let S be a subgraph of G. We say that S *separates* f_1 and f_2 if f_1 and f_2 are not connected in $G^* \setminus (E[S])^*$. Here, $E[X]$ denotes the set of edges of the subgraph X, $E[f]$ the edges incident to the face f, and $V[f]$ the incident vertices.

▶ **Observation 16.** *Given a cycle C that is induced in $T \cup \{e\}$ by some edge e and given any two faces f_1, f_2 not separated by C, any face f such that $C \cup E[f]$ separates f_1 and f_2 lies on the path $f_1 \cdots f_2$ in $\overline{T}^*$.*

Let f_1 and f_2 be faces of G, and let $S = V[f_1] \cap V[f_2]$ be the set of vertices they have in common. Let C denote the set of corners between vertices in S and faces in $\{f_1, f_2\}$. The sub-graphs obtained by cutting G through all the corners of C are called *flip-components* of G w.r.t. f_1 and f_2. Flip-components which are only incident to one vertex of S can be flipped with an articulation-flip, and flip-components incident to two vertices can be flipped with a separation-flip. (See Figure 6.)

▶ **Observation 17.** *Note that the perimeter of a flip component always consists of the union of a path along the face of f_u with a path along the face of f_v. One of these paths is trivial (equal to a point) exactly when u, v are linkable via an articulation-flip.*

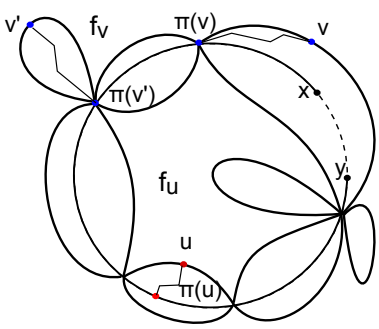

Figure 6 The faces f_u and f_v have five common vertices, and there are eight flip-components with respect to them.

Given vertices u, v in G, that are connected and not incident to a common face, we wish to find faces f_u and f_v such that u and v are in different flip-components w.r.t. f_u and f_v.

3.1 Finding one face

Let u and v be given vertices, and assume there exists faces f_u and f_v such that $u \in V[f_u] \setminus V[f_v]$, $v \in V[f_v] \setminus V[f_u]$, and u and v are in different flip-components w.r.t. f_u and f_v.

Let u_L, u_R be the left and right faces adjacent to the first edge on the path from u to v. Similarly let v_L, v_R be the left and right faces adjacent to the first edge on the path from v to u.

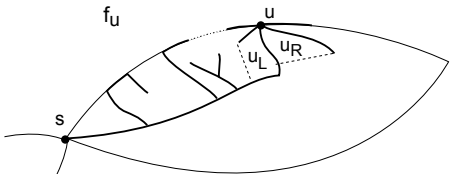

Figure 7 The co-tree path from u_L to u_R goes through f_u. The proof uses that the tree-path from u to v goes through some $s \in S$ on the boundary of u's flip-component.

▶ **Lemma 18.** *Face f_u is on the $\overline{T}^*$-path $u_L \cdots u_R$ and face f_v is on the $\overline{T}^*$-path $v_L \cdots v_R$.*

Proof. For symmetry reasons, we need only be concerned with the case f_u. The perimeter of a flip-component consists of edges incident to f_u and edges incident to f_v (see Observation 17). Furthermore, in order for u, v to be linkable via a flip, u needs to lie on the perimeter of its flip-component. We also know that the tree-path from v to u must go through a point s in S which lies on the boundary of u's flip-component. Thus, there must exist a path p in G from $\pi \in S$ to u, consisting only of edges incident to f_u. Note that $u \notin S$ since u, v were not already linkable. If the first edge e_u on the tree path from u to v is not already incident to f_u, then the union of p and the tree must contain an induced cycle containing e_u, separating u_L from u_R, induced by an edge e_i incident to f_u. (See Figure 7.) But then, the co-tree path from u_L to u_R goes through e_i^*, which means it goes through f_u. ◀

▶ **Lemma 19.** *If there exists an induced cycle C separating f_u from f_v such that $u \notin V[C]$ and $v \in V[C]$, then $f_u = \text{meet}(u_L, u_R, f)$ where $f \in \{v_L, v_R\}$ is the face that is on the same side of C as u. Here, $\text{meet}(a, b, c)$ denotes a's projection to the path $b \cdots c$.*

Proof. By Lemma 18 f_u is on the path $u_L \cdots u_R$. And since $C \cup E[f_u]$ separates f from u_L and u_R it is on the paths $u_L \cdots f$ and $u_R \cdots f$ by Observation 16. ◀

▶ **Lemma 20.** *If there exists an induced cycle C separating f_u from f_v such that $u \notin V[C]$ and $v \notin V[C]$, then either $f_u = \text{meet}(u_L, u_R, v_L) = \text{meet}(u_L, u_R, v_R)$ or $f_v = \text{meet}(v_L, v_R, u_L) = \text{meet}(v_L, v_R, u_R)$.*

Proof. Let e be the edge in $C \setminus T$, and let e_u, e_v be the faces adjacent to e that are on the same side of C as f_u and f_v respectively. Then e is on all 4 paths in $\overline{T}^*$ with u_L or u_R at one end and v_L or v_R at the other. At least one of u, v is in a different flip-component from e, so we can assume without loss of generality that u is. By Lemma 18 f_u is on the path $u_L \cdots u_R$. And since $C \cup f_u$ separates u_L and u_R from e_u, f_u is on both the paths $u_L \cdots e_u$ and $u_R \cdots e_u$ by Observation 16. Thus $f_u = \text{meet}(u_L, u_R, e_u) = \text{meet}(u_L, u_R, v_L) = \text{meet}(u_L, u_R, v_R)$. ◀

▶ **Lemma 21.** *If an induced cycle C separates f_u from f_v such that $u \in V[C]$ and $v \in V[C]$, then either $f_u = \text{meet}(u_L, v_L, v_R) = \text{meet}(u_L, u_R, v_R)$ or $f_u = \text{meet}(u_R, v_L, v_R) = \text{meet}(u_L, u_R, v_L)$ or $f_v = \text{meet}(v_L, u_L, u_R) = \text{meet}(v_L, v_R, u_R)$ or $f_v = \text{meet}(v_R, u_L, u_R) = \text{meet}(v_L, v_R, u_L)$.*

Proof. Let e be the edge in $C \setminus T$, and let e_u, e_v be the faces adjacent to e that are on the same side of C as f_u and f_v respectively. Then e is on all 4 paths in $\overline{T}^*$ with u_L or v_R at one

end and v_L or u_R at the other. Assume that u_L and v_R are on the side of C containing f_u and u_R and v_L are on the side of C containing f_v. At least one of u, v is in a different flip-component from e, so assume that v is. By lemma 18 f_u is on the path $u_L \cdots e_u \subset u_L \cdots u_R$. And since $C \cup f_u$ separates u_L and e_u from v_R it is on both the paths $u_L \cdots v_R$ and $e_u \cdots v_R$ by Observation 16. Thus $f_u = \text{meet}(u_L, e_u, v_R) = \text{meet}(u_L, v_L, v_R) = \text{meet}(u_L, u_R, v_R)$. The remaining cases are symmetric. ◀

▶ **Theorem 22.** *If f_u, f_v exist, either $f_u \in \{\text{meet}(u_L, u_R, v_L), \text{meet}(u_L, u_R, v_R)\}$ or $f_v \in \{\text{meet}(u_L, v_L, v_R), \text{meet}(u_R, v_L, v_R)\}$.*

Proof. If they exist there is at least one induced cycle C separating them. This cycle must have the properties of at least one of Lemmas 19, 20, or 21. ◀

By computing the at most two different meet values and checking which ones (if any) contain u or v we therefore get at most two candidates and are guaranteed that at least one of them is in $\{f_u, f_v\}$ if they exist.

▶ **Lemma 23.** *Given a top tree over a tree T, with vertices $a, b, c \in T$, we can find $\text{meet}(a, b, c)$ in logarithmic time.*

Proof. Split all clusters containing a, b, or c as a non-boundary vertex. There are only $O(\log n)$ of those. After these split-operations, we have a tree with $O(\log n)$ vertices. Use this tree to find $\text{meet}(a, b, c)$ in linear time. ◀

3.2 Finding the other face

▶ **Lemma 24.** *Let u, v, and f_u be given. Then the first edge e_L on $f_u \cdots v_L$ or the first edge e_R on $f_u \cdots v_R$ induces a cycle $C(e_R)$ or $C(e_L)$ in T that separates f_u from f_v.*

Proof. By lemma 18, f_v is on $v_L \cdots v_R$ in $\overline{T}^*$, so the first edge on $f_u \cdots f_v$ is also the first edge on either $f_u \cdots v_L$ or $f_u \cdots v_R$. ◀

Thus given the correct f_u we can find at most two candidates for an edge e that induces a cycle $C(e)$ in T that separates f_u from f_v, and be guaranteed that one of them is correct.

▶ **Observation 25.** *For each vertex, v, we may consider the projection $\pi(v)$ of v onto the cycle C. For each flip-component, X, we may consider the projection $\pi(X) = \{\pi(v) \mid v \in X\}$. If X is an articulation-flip component, the projection $\pi(X)$ is a single point in $S = V[f_u] \cap V[f_v]$. If X is a separation-flip component, its projection is a segment of the cycle, $\pi_1 \cdots \pi_2$, between the separation pair $(\pi_1, \pi_2) \subset C(e)$ where $\pi_1, \pi_2 \in S$.*

3.2.1 Finding an articulation-flip

Let (x, y) be any edge inducing a cycle C in $T \cup \{(x, y)\}$ that separates f_u from f_v, let $\pi(u) = \text{meet}_T(u, x, y)$ be the projection of u on C.

Now the articulation-flip cases are not necessarily symmetrical. First we present how to detect an articulation-flip, given u, v, and f_u, if f_v plays the role of f_2 (see Definition 14).

If the flip-component containing v is an articulation-flip component, then $\pi(v)$ is an articulation point incident to both f_u and f_v, but the opposite is not necessarily the case. Assume $\pi(v)$ is incident to both f_u and f_v and let c_u denote a corner between $\pi(v)$ and f_u.

Note that if $\pi(v)$ is an articulation point with corners c_1, c_2 both incident to f_v, then f_v is an articulation point in the dual graph with corners c_1, c_2 both incident to $\pi(v)$. Removing

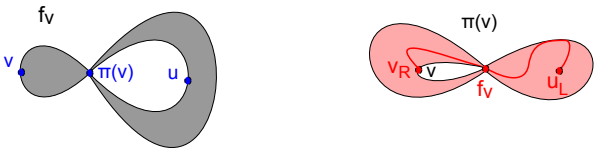

Figure 8 If f_v is an articulation point, so is $\pi(v)$. But then the co-tree path from u_L to v_L must go through f_v. Left: Primal graph. Right: Dual graph.

f_v from the dual graph would split its component into several components, and clearly, aside from f_v, only faces in *one* of these components may contain faces incident to v. Any path in the co-tree starting and ending in different components w.r.t the split will have the property that the first face incident to v on that path is f_v. (See Figure 8.)

Now, in the case $f_u = f_1$ and $f_v = f_2$, to find the corner of $\pi(v)$ incident to f_u, we can simply use $\texttt{query}(\pi(v), u)$ from before, which will return a corner of f_u incident to $\pi(v)$. To find the two corners of f_v: With the dual structure (see Observation 13) we may mark the face f_v, and expose the vertices u, v. Now, $\pi(v)$ has a unique place in the face-list of some cluster — if and only of that place is in the root cluster, and $c_{\min} = 0$ for both segments of that cluster, f_v plays the role of f_2. That is, iff $\pi(v)$ has a corner incident to f_v to one side, and a corner incident to f_v to the other side. In affirmative case, $\pi(v)$ appears with at least one corner to either vertex list; those corners can now be used as cutting-corners for the articulation-flip.

If instead f_v played the role of f_1, a similar procedure is done with $\pi(u)$.

▶ **Theorem 26.** *Given u, v are not already linkable, we can determine whether u, v are linkable via an articulation-flip in time $O(\log^2 n)$.*

3.2.2 Finding a separation-flip

Assume v, u are not linkable via an articulation-flip, determine if they are linkable via a separation-flip.

▶ **Lemma 27.** *Let (x, y) be any edge inducing a cycle C in $T \cup \{(x, y)\}$ that separates f_u from f_v, let $\pi(u) = \text{meet}_T(u, x, y)$ be the projection of u on C. Let e_1, e_2 be the edges incident to $\pi(u)$ on C. Then at least one of e_1, e_2 is in the same flip-component as u w.r.t f_u and f_v.*

Proof. This follows from Observation 25: If $\pi(u)$ is an endpoint of an arc $f_1 \cdots f_2$, then only one of the edges is in the same flip-component. If the projection is not an endpoint, then both of the edges are in the same flip-component. ◀

▶ **Lemma 28.** *Let C be any induced cycle separating f_u from f_v, let e_u be an edge on C in the same flip-component as u, let f_1 be the face adjacent to e_u that is separated from f_u by C, and let $f_2 \in \{v_L, v_R\}$ be a face on the same side of C as f_1. Then f_v is the first face on $f_1 \cdots f_2$ that contains v.*

Proof. $C \cup E[f_v]$ separates f_1 and f_2, so by Observation 16 f_v is on the $f_1 \cdots f_2$ path. It must be the first face on that path that contains v because for any face f after that, $C \cup V[f]$ does not separate f_1 and f_2, since it can only touch the part of C between $u' = \text{meet}(u, x, y)$ and $v' = \text{meet}(v, x, y)$ where (x, y) is the edge inducing C. ◀

3.3 Finding the separation pair and corners

Assume u, v are not linkable and not linkable via an articulation-flip.

▶ **Lemma 29.** *Given u, v, f_u, and f_v, let $(x,y)^*$ be any edge on $f_u \cdots f_v$ inducing a separating cycle C. If $\pi(u) = \pi(v) = \alpha$, then α is one of the separation points if it is adjacent to both f_u and f_v, and otherwise no separation pair for u, v exists. The other separation point, β, is then the first vertex $\neq \alpha$ adjacent to both f_u and f_v on either $\alpha \cdots x$ or $\alpha \cdots y$. If instead $\pi(u) \neq \pi(v)$, then α, β are amongst the first two vertices adjacent to both f_u and f_v either on $\pi(u) \cdots x$ and $v \cdots x$, or on $u \cdots y$ and $v \cdots y$.*

Proof. If the projection of u equals the projection of v, but u and v are in different flip-components, then the next point incident to both f_u and f_v along the cycle to either side will be the one we are looking for. However, (x,y) may be internal in the flip component containing u or that containing v, and thus one of the searches may return the empty list. But then the other will return the desired pair of vertices.

If the projections are different, and do not themselves form the desired pair (α, β), then we may assume without loss of generality that $\pi(v)$ does not belong to the flip-component containing u. Let X_v, X_u denote the flip-components containing u and v, respectively. If (x,y) is in X_v, such that no edge on $\pi(v) \cdots x$ is incident to both f_u and f_v, then the first vertex on $\pi(v) \cdots y$ incident to f_u and f_v is α. Recall (Observation 25) that $\pi(X_u)$ is an arc $\pi_1 \cdots \pi_2 \subset C$, and suppose without loss of generality π_1 is on the path $u \cdots v$. If $\pi(u) = \pi_1$, β is the second vertex on the path u to y incident to both f_u and f_v, as $\pi(u)$ itself is the first. Otherwise, the first such vertex on the path is β. If, on the other hand, (x,y) did not belong to X_v, let x be the vertex of x, y with the property that the path $u \cdots x$ goes through $\pi(u)$. Then the first vertices on the paths to x which are incident to f_u and f_v both, will be the desired separation pair. ◀

▶ **Lemma 30.** *In the scenario above, we may find the first two vertices on the path incident to both faces in time $O(\log^2 n)$.*

Proof. We use the dual structure (see Observation 13) to search for vertices incident to f_u and f_v. Now since the path $\pi(u) \cdots x$ is a sub-path of the cycle C induced by (x,y) which separates f_u from f_v, all corners incident to f_v will be on one side, and all corners incident to f_u will be on the other side of the path, or at the endpoints. Thus, we expose f_u and f_v in the dual structure, which takes time $O(\log^2 n)$. Now expose $\pi(u), x$ in the primal tree. Since this path is part of the separating cycle, if $c_{\min} = 0$ for both segments, then the maintained vertex-list will contain exactly those vertices incident to both faces, and a corner list for each of them. We now deal separately with the endpoints exactly as with linkable, by exposing the endpoint faces one by one in the dual structure, and noting whether $c_{\min} = 0$ and in that case, the corner list, for each endpoint. ◀

We conclude with the following theorem.

▶ **Theorem 31.** *We can maintain an embedding of a dynamic graph under insert, remove, split, join, and flip, together with queries that*

1. *Answer whether an edge can be inserted between given endpoints with no other changes to embedding, and if so, where.*
2. *Answer whether there exists a flip that would change the answer for query 1 from "no" to "yes", and if so, what flip.*

The worst case time per operation is $O(\log^2 n)$.

Acknowledgments We would like to thank Christian Wulff-Nilsen and Mikkel Thorup for many helpful and interesting discussions and ideas.

References

1 Stephen Alstrup, Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Maintaining information in fully dynamic trees with top trees. *ACM Transactions on Algorithms*, 1(2):243–264, 2005.

2 Giuseppe Di Battista and Roberto Tamassia. Incremental planarity testing. In *FoCS, 1989.*, pages 436–441. IEEE, 1989.

3 Giuseppe Di Battista and Roberto Tamassia. On-line planarity testing. *SIAM Journal on Computing*, 25:956–997, 1996.

4 David Eppstein. Dynamic generators of topologically embedded graphs. *SODA '03*, pages 599–608, 2003.

5 David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Thomas H. Spencer. Separator based sparsification: I. planarity testing and minimum spanning trees. *J. CSS*, 52(1):3 – 27, 1996.

6 David Eppstein, Giuseppe F. Italiano, Roberto Tamassia, Robert E. Tarjan, Jeffery R. Westbrook, and Moti Yung. Maintenance of a minimum spanning forest in a dynamic planar graph. *J. Algorithms*, 13(1):33–54, March 1992. Special issue for 1st SODA.

7 Greg N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM J. Comput.*, 14(4):781–798, 1985.

8 Zvi Galil, Giuseppe F. Italiano, and Neil Sarnak. Fully dynamic planarity testing with applications. *J. ACM*, 46:28–91, 1999.

9 John Hopcroft and Robert E. Tarjan. Efficient planarity testing. *J. ACM*, 21(4):549–568, October 1974.

10 Giuseppe F. Italiano, Johannes A. La Poutré, and Monika H. Rauch. Fully dynamic planarity testing in planar embedded graphs (extended abstract). *ESA '93, Proceedings*, pages 212–223, 1993.

11 David R. Karger. Random sampling in cut, flow, and network design problems. *Mathematics of Operations Research*, pages 648–657, 1994.

12 Johannes A. La Poutré. Alpha-algorithms for incremental planarity testing (preliminary version). In *STOC '94*, pages 706–715. ACM, 1994.

13 Mihai Pătraşcu and Erik D. Demaine. Logarithmic lower bounds in the cell-probe model. *SIAM Journal on Computing*, 35(4):932–963, 2006. See also STOC'04, SODA'04.

14 Mihai Pătraşcu and Mikkel Thorup. Planning for fast connectivity updates. In *FOCS '07*, pages 263–271, 2007.

15 Jeffery Westbrook. Fast incremental planarity testing. In W. Kuich, editor, *ALP*, volume 623, pages 342–353. Springer Berlin Heidelberg, 1992.