Min st-cut oracle for planar graphs with near-linear preprocessing time

Glencora Borradaile
School of Electrical Engineering and Computer Science
Oregon State University
Email: glencora@eecs.orst.edu
WWW: www.glencora.org

Piotr Sankowski
Institute of Informatics
University of Warsaw and
DIS "Sapienza" University of Rome
Email: sank@mimuw.edu.pl

Christian Wulff-Nilsen
Department of Computer Science
University of Copenhagen
Email: koolooz@diku.dk
WWW: www.diku.dk/~koolooz

***Abstract*—For an undirected n-vertex planar graph G with non-negative edge-weights, we consider the following type of query: given two vertices s and t in G, what is the weight of a min st-cut in G? We show how to answer such queries in constant time with $O(n \log^5 n)$ preprocessing time and $O(n \log n)$ space. We use a Gomory-Hu tree to represent all the pairwise min st-cuts implicitly. Previously, no subquadratic time algorithm was known for this problem. Our oracle can be extended to report the min st-cuts in time proportional to their size. Since all-pairs min st-cut and the minimum cycle basis are dual problems in planar graphs, we also obtain an implicit representation of a minimum cycle basis in $O(n \log^5 n)$ time and $O(n \log n)$ space and an explicit representation with additional $O(C)$ time and space where C is the size of the basis. To obtain our results, we require that shortest paths be unique; this assumption can be removed deterministically with an additional $O(\log^2 n)$ running-time factor.**

***Keywords*-Graph theory; Algorithms; Networks;**

I. Introduction

A minimum cycle basis is a minimum-cost representation of all the cycles of a graph whereas the all-pairs min cut problem asks to find all the minimum cuts in a graph. In planar graphs the problems are intimately related (in fact, equivalent [1]) via planar duality. In this paper, we give the first sub-quadratic algorithm for these problems, running in $O(n \log^5 n)$ time. In the following, we consider undirected graphs with non-negative edge weights. Several details are omitted from this version of the paper; the full version is available on arXiv [2].

All-pairs min cut: The *all-pairs min cut problem* is to find the min st-cut for every pair s, t of vertices in a graph G. Gomory and Hu [3] showed that these cuts can be represented by an edge-weighted tree such that:

- the nodes of the tree correspond one-to-one with the vertices of G,
- for any distinct vertices s and t, the minimum edge-weight on the unique, simple s-to-t path in the tree has weight equal to the min st-cut weight in G, and
- removing the corresponding minimum-weight edge from the tree creates a partition of the nodes into two sets that is a partition of the vertices in G corresponding to a min st-cut.

We call such a tree a *Gomory-Hu* tree or GH tree; GH trees are also called cut-equivalent and cut trees in the literature. Gomory and Hu also showed how to find such a tree with $n-1$ calls to a min st-cut algorithm. To date, this is the best known method for general graphs and results in an $O(n^2 \log n)$-time algorithm for planar graphs using the best-known algorithm for min st-cuts in planar graphs [4], [5]. There is an algorithm for unweighted, general graphs that beats the $n-1$ times min st-cut time bound [6]; the corresponding time for planar graphs, however, is $O(n^2 \text{poly} \log n)$ time.

Minimum cycle basis: A cycle basis of a graph is a maximal set of independent cycles. Viewing a cycle as an incidence vector in $\{0,1\}^E$, a set of cycles is independent if their vectors are independent over $GF(2)$. The weight of a set of cycles is the sum of the weights of the cycles. The *minimum-cycle basis (MCB) problem* is to find a cycle basis of minimum weight. This problem dates to the electrical circuit theory of Kirchhoff [7] in 1847 and has been used in the analysis of algorithms by Knuth [8]. For a complete survey, see [9]. The best known algorithm in general graphs takes $O(m^\omega)$ time where ω is the exponent for matrix multiplication [10].

The best MCB algorithms for planar graphs use basic facts of planar embeddings: a simple cycle C in a planar embedded graph creates two bounded subsets of the plane corresponding to the bounded ($int(C)$) and unbounded ($ext(C)$) subsets. Hartvigsen and Mardon [1] prove that if G is planar, then there is a minimum cycle basis whose cycles are simple and nested[1] in the drawing in the embedding. As such, one can represent a minimum cycle basis of a planar embedded graph as an edge-weighted tree, called the *MCB tree*, such that:

- the nodes of the tree correspond one-to-one with the faces[2] of the planar embedded graph, and

[1]A set of simple cycles of G is called *nested* if, for any two distinct cycles C and C' in that set, either $int(C) \subset int(C')$, $int(C') \subset int(C)$, or $int(C) \subset ext(C')$.

[2]Faces are defined as usual by the planar embedding.

Partially supported by NSF grant CCF-0963921.

- each edge in the tree corresponds to a cycle in the basis, namely the cycle that separates the faces[3] in the components resulting from removing said edge from the tree.

Hartvigsen and Mardon also gave an $O(n^2 \log n)$-time algorithm for the problem that was later improved to $O(n^2)$ by Amaldi et. al. [10].

A. Planar duality

In planar graphs, the MCB and GH problems are related via planar duality. Corresponding to every connected planar embedded graph G (the *primal*) there is another connected planar embedded graph (the *dual*) denoted G^*. The faces of G are the vertices of G^* and vice versa. Two vertices in G^* are connected if the corresponding faces in G share a border. Note that the edges of G correspond one-to-one with those of G^*. Cycles and cuts are equivalent through duality:

> In a connected planar graph, a set of edges forms a cycle in the primal iff it forms a cut in the dual. [11]

Equivalence between MCB and GH trees: Just as cuts and cycles are intimately related via planar duality, so are the all-pairs min cut and minimum cycle basis problems:

Theorem 1 (Corollary 2.2 [1]): For a planar embedded graph G, a tree T represents a minimum cycle basis of G if and only if T is a GH tree for G^* (via mapping node-face relationships to node-vertex relationships).

Herein, we focus on the frame of reference of the minimum cycle basis. Our algorithm works by finding a minimum (weight) cycle that separates two as-yet unseparated faces f and g. By duality, this cycle is a min fg-cut in G^*. We recurse on unseparated faces, gradually building the tree that represents the minimum cycle basis which, by Theorem 1, is the GH tree for the dual graph. This alone will not achieve a sub-quadratic running time. In order to beat quadratic time, we guide the recursion with planar separators and use precomputed distances to efficiently find the minimum separating cycles.

B. Planar separators

A *decomposition* of a graph G is a set of subgraphs $P_1, \ldots, P_k$ such that the union of vertex sets of these subgraphs is the vertex set of G and such that every edge of G is contained in a unique subgraph. We call $P_1, \ldots, P_k$ the *pieces* of the decomposition. The *boundary vertices* of a piece P_i is the set of vertices u in that piece such that there exists an edge (u, v) in G with $v \notin P_i$. We use a recursive subdivision. When piece P is decomposed into child subpieces, the boundary nodes of a child are the boundary vertices inherited from P as well as the boundary vertices introduced by the decomposition of P. By recursively applying Miller's Cycle Separator Theorem [12] we get:

Definition 1 (Balanced recursive subdivision [13]): A decomposition of G such that at each level a piece with n nodes and r boundary nodes is divided into two subpieces each containing no more than $\frac{2}{3}n + c\sqrt{n}$ nodes and no more than $\frac{2}{3}r + c\sqrt{n}$ boundary nodes.

A piece inherits an embedding from G's embedding. For simplicity of presentation, we assume that pieces are connected and have no holes (bounded faces containing only boundary vertices). While it is not possible to guarantee this with a balanced recursive subdivision, these assumptions can be dropped; see the full version.

We define the $O(\log n)$ *levels* of the recursive subdivision in the natural way: level 0 consists of one piece (G) and level i-pieces are obtained by decomposing each level $i-1$-piece. We represent the recursive subdivision as a binary tree, the *subdivision tree (of G)*, with level i-pieces corresponding to vertices at level i in tree. Parent/child and ancestor/descendant relationships between pieces correspond to their relationships in the subdivision tree.

C. Precomputed distances

For a piece P, the *internal dense distance graph of* P or $int\mathrm{DDG}(P)$ is the complete graph on the set of boundary vertices of P, where the weight of each edge (u, v) is equal to the shortest path distance between u and v in P. The union of internal dense distance graphs of all pieces in the recursive subdivision of G is the *internal dense distance graph (of G)*, or simply $int\mathrm{DDG}$. Fakcharoenphol and Rao showed how to compute $int\mathrm{DDG}$ in $O(n \log^3 n)$ time [13]; Klein improved this to $O(n \log^2 n)$ [14].

Fakcharoenphol and Rao used a Dijkstra-like algorithm (Section 3.2.2 [13]) that runs in time $O(|\partial P| \log^2 |P|)$ in $int\mathrm{DDG}(P)$, where ∂P is the set of boundary vertices of P. We use this algorithm in graphs composed of dense distance graphs and pieces of the original graph:

Corollary 1: Dijkstra can implemented in $O((\sum_i |\partial G_i| + |E|) \log^2 |V|)$ time on a set of dense distance graphs G_i and a set of edges E over the vertex set V.

The *external dense distance graph of* P or $ext\mathrm{DDG}(P)$ is the complete graph on the set of boundary vertices of P where the weight of an edge (u, v) is the shortest path distance between u and v in $G \setminus E(P)$. The *external dense distance graph of* G or $ext\mathrm{DDG}$ is the union of all external dense distance graphs of the pieces in the recursive subdivision of G.

Theorem 2: The external dense distance graph of G can be computed in $O(n \log^3 n)$ time.

Proof: Fakcharoenphol and Rao compute $int\mathrm{DDG}$ bottom-up by applying a variant of Dijkstra to obtain $int\mathrm{DDG}(P)$ for a piece P from the internal dense distance graphs of its children. Similarly, we

[3]We say that a pair of faces of G are *separated* by C in G and that C *separates* this pair if one face is contained in $int(C)$ and the other face is contained in $ext(C)$.

can compute $ext\text{DDG}(P)$ from $ext\text{DDG}(P\text{'s parent})$ and $int\text{DDG}(P'\text{s siblings})$. After computing intDDG, extDDG can be computed top down. ■

D. Overview of the algorithm

We gradually build up the tree representing the minimum cycle basis. Initially the tree is a star centered at a root r and each leaf corresponding to a face in the graph (including the infinite face). We update the tree to reflect the cycles that we add iteratively to the basis. When the first cycle C is found, we create a new node x_C for the tree, make x_C a child of the root and make all the faces that C encloses children of x_C. Each non-face node in the tree corresponding to a cycle C defines a *region* R: the subgraph of G contained in the closed subset of the plane defined by the interior of C and the exterior of the children (if any) of C. We say that R is *bounded* by C, that C is a *bounding cycle* of R, and that R *contains* the child regions and/or child faces defined by the tree. The tree of regions is called the *region tree*. Observe that a pair of faces not yet separated by a basis cycle belong to the same region.

The root r will remain a special region that represents the entire plane. We only add cycles to the basis that nest with the cycles found so far. Whenever the basis is updated, the region tree is updated accordingly. This is illustrated in Figure 1. We show how to efficiently update the region tree in Section V.

In the final tree, all faces have been separated: each face is the only face-child of a region. We call such a region tree a *complete region tree*. Mapping each face to its unique parent, creating a tree with one node for each face in the graph, will create the MCB tree.

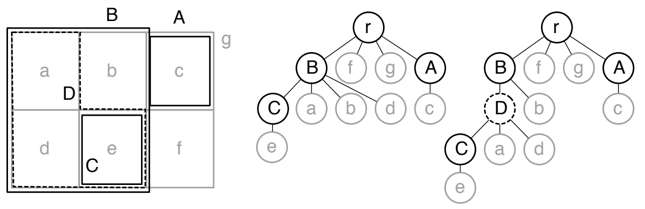

Figure 1. A graph with faces a through g; four nesting cycles A through D (left). A region tree for cycles A,B and C (center). A region tree for cycles A through D (right).

Our algorithm is guided by the recursive subdivision of G. Starting at the deepest level of the recursive subdivision, we separate all pairs of faces of G that have an edge in a common piece of the subdivision. Suppose we are at level i and that our algorithm has been applied to all levels deeper than i; consider a level-i piece P. Unseparated faces in P must belong to distinct children of P (of which there are at most two) and must belong to a common region R. We get:

Lemma 1: There are at most two unseparated faces of R in P.

Let the *region subpieces* of a piece P be the subgraphs defined by the non-empty intersections between P and regions defined by the region tree (Figure 2). We say that a region R is *associated* with a region subpiece P_R and that P_R is associated with R if $P_R = P \cap R$ is not empty. By Lemma 1, at most one pair of faces in a region subpiece needs to be separated. In Section III, we show how to separate such a pair of faces in $O((|\partial P_R|^2 + |P_R|)\log^3|P_R|)$ time assuming that pieces are connected. Without this assumption, we require $O(|P_R|\log^4|P_R|)$ time. This amounts to $O(|P|\log^4|P|)$ time over all region subpieces of P, $O(n\log^4 n)$ time at level i and $O(n\log^5 n)$ time overall.

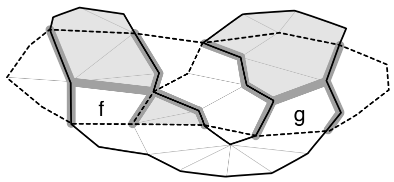

Figure 2. Dotted edges belong to boundaries of a piece P and children P_1 and P_2. f and g are unseparated faces. Solid black edges bound region R. Shaded areas are child regions of R. Thick grey edges belong to region subpiece P_R. The intersection of a minimum fg-separating cycle with P uses only edges of P_R.

E. Results

The bulk of the paper will be devoted to presenting the details of the algorithm outlined in the previous section. In Section III, we describe an efficient way to compute minimum separating cycles for pairs of unseparated faces *given* region subpieces. In Section IV, we show how to find the region subpieces. In Section V, we show how to update the region tree with a new nesting separating cycle. This proves our main result:

Theorem 3: A complete region tree, MCB tree and GH tree of a planar undirected n-vertex graph with non-negative edge weights can be found in $O(n\log^5 n)$ time and $O(n\log n)$ space.

In order to find the weight of a min st-cut using the GH tree, we need to find the minimum-weight edge on the s-to-t path in the tree. With an additional $O(n\log n)$ preprocessing time, one can answer such queries in $O(1)$ time using a tree-product data structure [15], giving:

Theorem 4: With $O(n\log^5 n)$ time and $O(n\log n)$ space for preprocessing, the weight of a min st-cut between for any two given vertices s and t of an n-vertex planar, undirected graph with non-negative edge weights can be reported in constant time.

In the full version of this paper, we show how our algorithm can be adapted to report the edges of min st-cuts and basis cycles in time proportional to their size.

II. Preliminaries

A. Simplifying structural assumptions

To simplify the presentation of the algorithm and the analysis, we make a few structural assumptions on the input graph. These assumptions are not truly restrictive.

Simple faces: achieved by triangulating with infinite-weight edges.

Degree three: achieved by triangulating the dual graph with ϵ weight edges, where ϵ is much smaller than the smallest weight of an edge. We use $\epsilon > 0$ to coordinate with the next assumption. Triangulating the dual will increase the size of faces in the primal but they will remain simple.

Unique shortest paths: achieved by adding a small, random perturbation to the weight of each edge, making the probability of having non-unique shortest paths arbitrarily small. In the full paper version, we show how to *deterministically* ensure shortest path uniqueness at the cost of an extra $O(\log^2 n)$ running-time factor.

B. Isometric cycles

A cycle is *isometric* if for any pair of vertices on the cycle, one of the distinct paths between those vertices in the cycle is a shortest path in the graph. The following two lemmas imply that the minimum cycle basis we construct is isometric and nested.

Lemma 2: If shortest paths are unique, the intersection between an isometric cycle and a shortest path in G is a (possibly empty) shortest path, the intersection between two distinct isometric cycles is a (possibly empty) shortest path, and (in planar graphs) isometric cycles nest.
The proof is left as an exercise.

Lemma 3 (Proposition 4.4 [1]): Any minimum cycle basis of a graph is isometric.

C. Representing Regions

We represent the region tree using the top tree data structure [16] which support the following operations in logarithmic time:

- $lca(x, y)$: find the lowest common ancestor of nodes x and y
- $jump(x, y, d)$: find the node that is d edges away from x on the path between nodes x and y
- find the weight of a path between nodes x and y

For each region, we maintain a *compact representation*: vertices with degree 2 are removed by merging the incident edges creating *super edges*, which are associated with the first and the last edge on the corresponding path.

In Section V, we show how to efficiently add cycles to the basis using top-tree operations and how to maintain compact representations of regions.

III. Separating a pair of faces

In this section we show how to find the minimum fg-separating cycle for the unique pair of unseparated faces f, g (Lemma 1) in a region subpiece P_R by emulating an algorithm due to Reif [17].

We use an operation of cutting open a planar embedded graph G along a path X: duplicate every edge and vertex of X and create a new, simple face whose boundary is composed of edges of f, g, X, and the duplicate of X. The resulting graph is denoted G_X.

Paths P and Q cross if there is a quadruple of faces incident to P and Q that cover the set product $\{\text{left of } P, \text{right of } P\} \times \{\text{left of } Q, \text{right of } Q\}$.

A. Reif's minimum separating cycle algorithm

Let X be the shortest path between any vertex on the boundary of f and any vertex on the boundary of g. Reif's algorithm is based on the observation that there is a minimum gf-separating cycle that crosses X only once:

Theorem 5 (Proposition 3 [17]): Let X be the shortest g-to-f path. For each vertex $x \in X$, let C_x be the minimum weight cycle that crosses X exactly once and does so at x. Then a minimum fg-separating cycle is a cycle C_x of minimum weight. Further, C_x is the shortest path between duplicates of x in G_X.
Reif computes, for every vertex $x \in X$, the minimum separating cycle C_x. Finding C_x amounts to finding a shortest x-to-x' path in G_X, where x' is the duplicate of x. The running time of the algorithm is bounded by divide and conquer: start with x, the midpoint of X in terms of the number of vertices, and recurse on the subgraphs obtained by cutting along C_x. The algorithm takes $O(n \log n)$ time using the linear-time algorithm for shortest paths in planar graphs [18].

Corollary 1 will allow us to emulate Reif's algorithm in time $O((|\partial P_R|^2 + |P_R|) \log^3 |P_R|)$. In order to attain this running time, we must deal with the following peculiarities:

- X is not contained entirely in P_R. For a vertex $x \in X$ that is outside P_R, we will compute C_x by composing distances in extDDG and intDDG between restricted pairs of boundary vertices of P_R. We call such cycles *external cycles*. We show how to find these cycles in Section III-D.
- For vertices $x \in X$ that are inside P_R, C_x may not be contained by P_R. We find C_x by first modifying extDDG to disallow paths from crossing X. We call such cycles *internal cycles*. We show how to find these cycles in Section III-C.
- extDDG defines distances in G, not G_X. We compute modified dense distance graphs to account for this in Section III-B.

We can compute X using Dijkstra in $O((|\partial P_R|^2 + |P_R|) \log^2 |P_R|)$ time. In the next two sections we show how to find all the internal cycles and external cycles in time $O((|\partial P_R|^2 + |P_R|) \log^3 |P_R|)$ time. The minimum weight cycle over all internal and external cycles is the minimum fg-separating cycle in G.

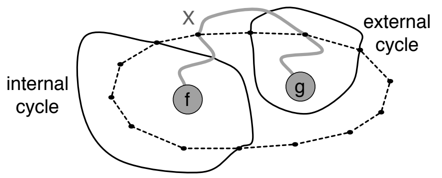

Figure 3. An external and internal cycle separating faces f and g in a region subpiece, whose boundary vertices are given by the bold vertices.

B. Modifying the external dense distance graph

We use $ext\mathrm{DDG}$ to compute $ext\mathrm{DDG}_X$, the dense distance graph that corresponds to distances between boundary vertices of P_R when the graph is cut open along X. However, we do not compute $ext\mathrm{DDG}_X$ explicitly as it can take too much time. We simply show how to determine its values when needed.

Let B be the set of boundary vertices of P_R. Cutting G open along X duplicates vertices of B that are in X, creating B'. $ext\mathrm{DDG}_X$ can be represented as a table of distances between every pair of (x, y) vertices of B':

$$ext\mathrm{DDG}_X(x,y) = \begin{cases} \infty, & x \text{ and } y \text{ separated in } G_x \setminus P_R \\ \infty, & x \text{ is a copy of } y \\ ext\mathrm{DDG}(x,y), & \text{otherwise} \end{cases}$$

Only the first condition is non-trivial to determine. The portions of X that appear outside P_R form a parenthesis of (a subset of) the boundary vertices (Figure 4). By walking along X we can label the start and endpoints of these parentheses. By walking along the boundary of the subpiece we can label a group of boundary vertices that are not separated by X by pushing the vertices onto a stack with a label corresponding to the start of a parenthesis and popping them off when the end of the parenthesis is reached, labelling the boundary vertices with the corresponding parenthesis. Two boundary vertices are not separated if they have the same parenthesis label.

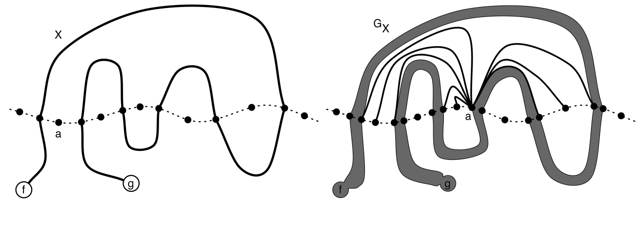

Figure 4. Modifying the external dense distance graph. (Left) X is given by the solid line and the boundary of the subpiece is given by the dotted line. The parts of X outside the subpiece form a parenthesis. (Right) In G_X, the only finite distances from a in $ext\mathrm{DDG}_X$ correspond to the thick lines. The shaded area represents the new face created by cutting along X.

C. Finding internal cycles

Let $D = V(X \cap P_R)$. Let D be ordered according to the order of the vertices along X. For each vertex $x \in D$, we compute the shortest x-to-x' paths in G_X where x' is the copy of x in G_X. We do this using Dijsktra's algorithm on the cut-open graph induced by the vertices in P_R (i.e. $G_X[P_R]$) and the modified dense distance graph $ext\mathrm{DDG}_X$. Each cycle can then be found in $O((|P_R| + |\partial P_R|^2) \log |P_R|)$ time. Let x_m be the mid-point vertex of D according to the order inherited from X. C_{x_m} splits P_R and $ext\mathrm{DDG}_X$ into two parts (not necessarily balanced). Recursively finding the cycles through the midpoint of D in each graph part[4] results in $\log |D|$ levels for a total of $O((|P_R| + |\partial P_R|^2) \log |P_R| \log |D|) = O((|P_R| + |\partial P_R|^2) \log^2 |P_R|)$ time to find all the internal cycles.

D. Finding external cycles

We use the following structural property (proof in full version) to efficiently find the shortest external cycle.

Lemma 4: The shortest external cycle is composed of a single edge ab in the *unmodified* $ext\mathrm{DDG}$ and a shortest path π_{ab} between boundary vertices of P_R in G that does not cross X.

Let $int\mathrm{DDG}_X$ be obtained from $int\mathrm{DDG}$ by cutting it open along X. We can compute π_{ab}, $\forall a, b$ using the modified Dijkstra algorithm of Fakcharoenphol and Rao (Section 3.2.2 and 3.2.2 [13]) in $O(|\partial P_R|^2 \log^2 |P_R|)$ time if $ext\mathrm{DDG}_X$ and $int\mathrm{DDG}_X$ are given: for each a, find a shortest path tree in $ext\mathrm{DDG}_X \cup int\mathrm{DDG}_X$ rooted at a in $O(|\partial P_R| \log^2 |P_R|)$ time. $int\mathrm{DDG}_X$ can be found in $O((|\partial P_R|^2 + |P_R|) \log^3 |P_R|)$ time by cutting open X, setting δP as border nodes and using the recursive internal dense distance graph algorithm of Fakcharoenphol and Rao.

In order to compute all the external cycles, one enumerates over all pairs a, b of vertices that are split *exactly once* by the parenthesis given by X, summing the weights of π_{ab} and of edge (a, b) in $ext\mathrm{DDG}$. Since there are $O(|\partial P_R|)$ boundary vertices, there are $O(|\partial P_R|^2)$ pairs to consider. The minimum-weight external cycle then corresponds to the pair with minimum weight. By the above, this cycle can be found in $O((|\partial P_R|^2 + |P_R|) \log^3 |P_R|)$ time.

IV. Finding region subpieces

In Section I-D, we defined the region subpieces of a piece as the intersection between a region and a piece (Figure 2). In this section, we show how to identify the region subpieces and the edges that are in them. We start by identifying the set of regions $\mathcal{R}_P$ whose corresponding region subpieces of piece P each contain a pair of unseparated faces (Section IV-A). For each region $R \in \mathcal{R}_P$ we initialize the

[4]In order to properly bound the running time, one must avoid repeating long paths in the subproblems: in a subproblem resulting from divide and conquer, we remove degree-two vertices by merging the incident edges.

corresponding region subpiece P_R as an empty graph. For each edge e of P we determine to what region subpieces e belongs using lowest common-ancestor and ancestor-descendent queries in the region tree (Section IV-B). In Section IV-C we show how to do all this in $O(r \log^2 n)$ time where r is the size of P.

A. Identifying region subpieces

Since each edge is on the boundary of two faces, we start by marking all the faces of G that share edges with P in $O(r)$ time. Since a pair of unseparated faces in P are siblings in the region tree, we can easily determine the set $\mathcal{R}_P$ of regions that contain unseparated faces in P.

There are $O(\sqrt{r})$ boundary vertices in P, so by a "chocolate-breaking" argument there cannot be more than $O(\sqrt{r})$ regions in $\mathcal{R}_P$ as well. We get the following bound (illustrated in Figure 5, formal proof in full version):

Lemma 5: $|\mathcal{R}_P| = O(\sqrt{r})$.

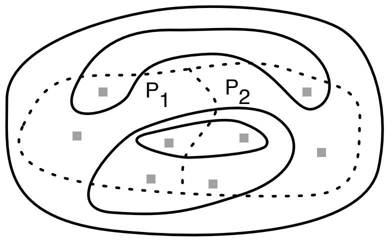

Figure 5. Shown: boundaries of sibling pieces P_1 and P_2 (dotted); $\mathcal{R}_P$ (solid cycles), each containing a pair of unseparated faces (grey). Since these faces must be separated by the child pieces (Lemma 1), each bounding cycle (except for one outer cycle) in $\mathcal{R}_P$ must cross the dotted boundaries, bounding $|\mathcal{R}_P|$.

B. Identifying edges of region subpieces

Having identified the set $\mathcal{R}_P$ of regions defining region subpieces, we now show how to identify the edges of these subpieces. To do this, we consider two types of edges of regions: *internal edges* and *boundary edges*. Let C be the bounding cycle of a region R. An edge e is an internal edge of R if the faces on either side of e are enclosed by C. An edge e is a boundary edge of R if e is an edge of C. Every edge is an internal edge of exactly one region: Lemma 6 shows how we can identify this region. Such an edge can belong to at most one region subpiece. We can also determine if an edge is a boundary edge of some region (Lemma 7). However, an edge can be a boundary edge of several regions and hence of several region subpieces. For performance reasons, we need the total number of edges over all region subpieces to be small. We deal with this problem in Section IV-C.

Lemma 6: Let e be an edge of G and let f_1 and f_2 be the faces incident to e. Then e is an internal edge of a region R iff R is the lowest common ancestor of f_1 and f_2 in the region tree.

Proof: There must exist some region R satisfying the lemma. Let C be its bounding cycle. Then both f_1 and f_2 are contained in $int(C)$ and it follows that R must be a common ancestor of f_1 and f_2. If R' is another common ancestor then either R is an ancestor of R' or R' is an ancestor of R. If R is an ancestor of R' then R' is contained in a face of R so e cannot belong to R. But this contradicts the choice of R. Hence, $R = lca(f_1, f_2)$. ■

Iterating over each edge e of P, we can find the region R for which e is an internal edge. If $R \in \mathcal{R}_P$, we add e to the corresponding region subpiece P_R. Applying Lemma 6 with the top tree data structure, the total time to add internal edges to their corresponding region subpieces is $O(r \log n)$.

Lemma 7: Let e be an edge of G and let f_1 and f_2 be the faces incident to e. Let R' be the lowest common ancestor of f_1 and f_2 in the region tree. Then e is a boundary edge of a region R iff R is a descendant of R' and one of f_1, f_2 is a descendant of R.

Proof: Assume first that $e \in C$, where C is the cycle bounding R. Then exactly one of the faces f_1 and f_2 is in $int(C)$ and the other is in $ext(C)$. Assume w.l.o.g. that $f_1 \in int(C)$ and $f_2 \in ext(C)$. Then f_1 is a descendant of R and since f_2 is not, R must be a descendant of R'.

Now assume that R is a descendant of R' and that, say, f_1 is a descendant of R. Then f_2 is not a descendant of R since otherwise, R' could not be an ancestor of R. This implies that $e \in C$. ■

Let R be a region in $\mathcal{R}_P$ and let C be the bounding cycle of R. Let P_1 and P_2 be the children of P and let B be the union of boundary vertices of P_1 and P_2. We have seen (Lemma 5) that the intersection between C and P are subpaths in P between pairs of vertices of B. We shall refer to these as *cycle paths* (of C). Note that the edges we still need to add in order to form the region subpieces all belong to cycle paths. To identify these edges, we start by finding starting points of cycle paths with the following algorithm.

Cycle path starting points identification algorithm: Pick a boundary vertex $u \in B$. For every edge e incident to u (there are at most three such edges), check to see if e is a boundary edge of R. If there is no such edge, then there is no cycle path through u. Otherwise, mark e as a starting point of a cycle path for R. Repeat this process for every vertex in B.

This process takes $O(\sqrt{r} \log n)$ time for each region R using a constant number of tree queries for every vertex in B (Lemma 7). Repeating for all regions in $\mathcal{R}_P$ takes $O(r \log n)$ time (Lemma 5).

After identifying staring points of cycle-paths we can find all edges belonging to them by *linear search* (ie. simply walking along the cycle-path). If the cycles are edge-disjoint over all regions $R \in \mathcal{R}_P$, then the cycle paths will also be edge-disjoint and the time for linear search is $O((|\mathcal{R}_P|\sqrt{r} + |P|)\log n) = O(r \log n)$. This is also a bound on the time to identify the remaining edges of region subpieces. However, the cycles are not necessarily edge disjoint. We overcome this complication in the next section.

C. *Efficiently identifying boundaries of region subpieces*

Since cycles will share edges, the total length of cycle paths over all cycles can be as large as $O(r^{3/2})$. However, we can maintain efficiency by using a compact representation of each cycle path. The compact representation consists of edges of P and *cycle edges* that represent paths in P shared by multiple cycle paths.

View each edge of G as two oppositely directed darts and view the cycle bounding a region as a clockwise cycle of darts. By Lemma 2:

Corollary 2: If two isometric cycles C and C' of G share a dart, then either $int(C) \subseteq int(C')$ or $int(C') \subseteq int(C)$.

Let $\mathcal{F}$ be the forest representing the ancestor/descendant relationship between the bounding cycles of regions in $\mathcal{R}_P$. By Lemma 5, there are $O(\sqrt{r})$ bounding cycles and since we can make descendent queries in the region tree in $O(\log n)$ time per query, we can find $\mathcal{F}$ in $O(r \log n)$ time. Let d be the maximum depth of a node in $\mathcal{F}$. For $i = 0, \ldots, d$, let $\mathcal{C}_i$ be the set of cycles corresponding to nodes at depth i in $\mathcal{F}$. By Corollary 2:

Corollary 3: For any $i \in \{0, \ldots, d\}$, cycles in $\mathcal{C}_i$ are pairwise dart-disjoint.

Bottom-up algorithm: We find cycle paths for cycles in $\mathcal{C}_d$, then $\mathcal{C}_{d-1}$, and so on. The cycles in $\mathcal{C}_d$ are dart disjoint, so any edge appears in at most two cycles of $\mathcal{C}_d$. We find the corresponding cycle paths using the above algorithm. While Corollory 3 ensures that the cycles in $\mathcal{C}_d$ are mutually dart-disjoint, they can share darts with cycles in C_{d-1}. In order to efficiently walk along subpaths of cycle paths Q that we have already discovered, we use a balanced binary search tree (BBST) to represent Q. We augment the BBST to store in each node the length of the subpath it represents. Now, given two nodes in Q, the weight of the corresponding subpath of Q can be determined in logarithmic time.

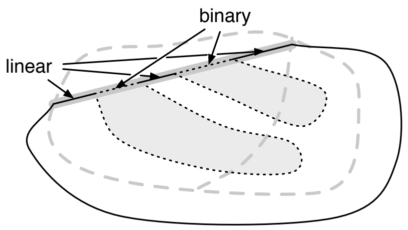

Figure 6. Finding a cycle path (highlighted straight line) for a cycle $C \in \mathcal{C}_{d-1}$ between boundary nodes of P_1 and P_2 (grey dashed lines) is found by alternating linear (solid) and binary (dotted) searches. Binary searches corrrespond to cycle paths of region subpieces (shaded) bounded by cycles in $\mathcal{C}_d$.

To find the cycle paths of a cycle $C \in \mathcal{C}_{d-1}$ that bounds a region R, we emulate the above cycle path identification algorithm: start walking along a cycle path Q of C, starting from a vertex of B, and stop if reaching an edge $e = uv$ that has already been visited *(linear search)*. In this case, e must be an edge of a cycle path Q' of a cycle $C' \in \mathcal{C}_d$. By Lemma 2, the intersection of Q and Q' is a single subpath and so we can use the BBST to find the last vertex w common to Q and Q' *(binary search)*. We add to P_R an edge uw of weight equal to the weight of the u-to-w subpath of Q to compactly represent this subpath. If $w \in B$, we stop our walk along Q. Otherwise we continue walking (and adding edges to the corresponding region subpiece) in a linear fashion, alternating between linear and binary searches until a boundary vertex is reached. See Figure 6.

In order to repeat the above idea to find cycle paths for cycles in $\mathcal{C}_{d-2}$, we need to build BBSTs for cycle paths of cycles in $\mathcal{C}_{d-1}$. Let Q be one such cycle path. Q can be decomposed into subpaths $Q_1 Q'_1 \cdots Q_k Q'_k$, where $Q_1, \ldots, Q_k$ are paths obtained with linear searches and $Q'_1, \ldots, Q'_k$ are paths obtained with binary searches (possibly Q_1 and/or Q'_k are empty). To obtain a binary search tree $\mathcal{T}$ for Q, we start with $\mathcal{T}$ the BBST for Q_1. We extract a BBST for Q'_1 from the BBST we used to find Q'_1 and merge it into $\mathcal{T}$. We continue merging with BBSTs representing the remaining subpaths.

Running time: We now show that the bottom-up algorithm runs in $O(r \log^2 n)$ time over all region subpieces by bounding time required for linear and binary searches and BBST construction.

A subpath identified by a linear search consists only of darts that have not yet been discovered. Since each step of a linear search takes $O(\log n)$ time, the total time for linear searches is $O(r \log n)$.

The number of cycle paths corresponding to a cycle C is bounded by the number of boundary vertices, $O(\sqrt{r})$. We consider three types of cycles paths. Those where

1) all edges are shared by a single child of C in $\mathcal{F}$,
2) no edges are shared by a child, and
3) some but not all edges are shared by a single child.

Cycle paths of the first type are identified in a single binary search for a total of $O(r)$ binary searches (Lemma 5) over all cycles $C \in \mathcal{F}$. Cycle paths of the second type do not require binary search. For a cycle path Q in the third group, Q can only share one subpath with each child (in $\mathcal{F}$) cycle (Lemma 2); hence, there can be at most two binary searches per child.

In total there are $O(r)$ binary searches. Each BBST has $O(r)$ nodes. In traversing the binary search tree, an edge is checked for membership in a given cycle path in $O(\log n)$ time (Lemma 7). Each binary search therefore takes $O(\log r \log n) = O(\log^2 n)$ time. The total time spent performing binary searches is $O(r \log^2 n)$.

It remains to bound the time needed to construct all BBSTs. BBSTs T_1 and T_2 are merged in time $O(\min\{|T_1|, |T_2|\} \log(|T_1| + |T_2|)) =$

$O(\min\{|T_1|, |T_2|\} \log n)$ by inserting elements from the smaller tree into the larger.

When forming a BBST for a cycle path of a cycle C, it may be necessary to delete parts of cycle paths of children of C. By Lemma 2, these parts intersect $int(C) \setminus C$ and will not be needed for the remainder of the algorithm. The total number of deletions is $O(r)$ and they take $O(r \log r)$ time to execute. Ignoring deletions, paths represented by BBSTs are pairwise dart disjoint (Corollary 3). Theorem 6 follows from the next lemma with $k = \log n$ and $W = r$.

Lemma 8: Consider a set of objects, where each object o is assigned a positive integer weight $w(o)$. Let $merge(o, o')$ be an operation that replaces two distinct objects o and o' by a new object whose weight is at most $w(o) + w(o')$. Assume that the time to execute $merge(o, o')$ is bounded by $O(\min\{w(o), w(o')\}k)$ for some value k. Then repeating the $merge$-operation on pairs of objects in any order until at most one object remains takes $O(kW \log W)$ time where W is the total weight of the original objects.
The proof follows by backward simulation of the algorithm and by a "join by rank" charging argument.

Theorem 6: The region subpieces of a piece of size r can be identified in $O(r \log^2 n)$ time.

V. Adding a separating cycle to the region tree

In Section III, we showed how to find a compact representation of a minimum cycle cycle C separating a pair of faces in a region R. Now we show how to add this cycle to the basis by updating the region tree $\mathcal{T}$ accordingly. As in the previous section, let P_R be the region subpiece $P \cap R$ of piece P.

When C is added to the partial basis, R is split into two regions, R_1 and R_2. Equivalently, in $\mathcal{T}$, R will be replaced by two nodes R_1 and R_2. The children $\mathcal{F}$ of R will be partitioned into children $\mathcal{F}_1$ of R_1 and $\mathcal{F}_2$ of R_2. Define R_1 to be the region defined by the children of R that are contained to the left of C for the orientation of C inherited by the shortest path computed in Section III. Likewise define R_2 to be the region for the children to the right of C. We describe an algorithm that finds $\mathcal{F}_1$ and detects whether $\mathcal{F}_1$ is contained by $int(C)$ or $ext(C)$. Finding $\mathcal{F}_2$ is symmetric. The algorithms take $O(|\mathcal{F}_i| \log^3 n + (|P_R| + |\partial P_R|^2) \log n)$ time; we can identify the smaller side of the partition in $O(\min\{|\mathcal{F}_1|, |\mathcal{F}_2|\} \log^3 n + (|P_R| + |\partial P_R|^2) \log n)$ time. By Lemma 8 the total time for all region tree updates is $O(n \log^4 n)$.

Updating the region tree: Given the smaller side of the partition (wlog, $\mathcal{F}_1$), we use cut-and-link operations to update $\mathcal{T}$ (Figure 1) in $O(|F_1| \log n)$ additional time. If $\mathcal{F}_1$ is contained by $int(C)$ then update $\mathcal{T}$ by: cutting the edges between R and each element in $\mathcal{F}_1$, linking each element in $\mathcal{F}_1$ to R_1, making R the parent of R_1, identifying R with R_2. If $\mathcal{F}_1$ is contained by $ext(C)$ then update $\mathcal{T}$ by: cutting the edges between R and each element in $\mathcal{F}_1$, linking each element in $\mathcal{F}_1$ to a new node u, making u the parent of R; identifying R with R_1 and u with R_2.

A. Partitioning the faces

Regions are represented compactly: vertices of degree 2 are removed by merging the incident edges creating *super edges*, which are associated with the first and last edge on the corresponding path. In addition to partitioning the faces when adding a cycle to the basis, we must find the compact representation for the resultant new regions, R_1 and R_2.

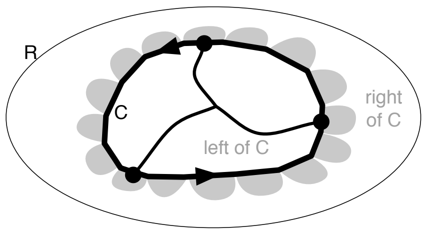

Figure 7. C (bold cycle) is given with a counterclockwise orientation. The children of R (boundary given by thin cycle) incident and to the right of C are grey. The edges to the left of C (and not on C) will never reach a boundary edge of R: therefore the left of C forms $int(C)$. Vertices of L are given by dark circles.

The algorithm for finding $\mathcal{F}_1$ starts with an empty set and consists of three steps:

Left-root vertices Identify the set L of vertices v on C having an edge emanating to the left of C; also identify, for each $v \in L$, the two edges on C incident to v (in G, not the compact representation).

Search Start a search (say, depth first) in R from each vertex of L avoiding edges on C or emanating to the right of C; for each super edge $\hat{e}$ of R visited, find the first (or last) edge e on the path represented by $\hat{e}$.

Add For the pair of faces f_1, f_2 incident to e, find the two children of R in $\mathcal{T}$ having f_1 and f_2 as descendants, respectively, and add these nodes to $\mathcal{F}_1$.

Above we assume that $L \neq \emptyset$ since otherwise, identifying $\mathcal{F}_1$ is trivial. This algorithm correctly builds $\mathcal{F}_1$: The algorithm visits all super edges $\hat{e}$ that are strictly inside R and on the left side of C. Let A_1 and A_2 be the children of R that are added corresponding to edge e. A_i is a region or a face of G; let C_i be the bounding cycle. Since f_i is a descendent of A_i, f_i is contained by $int(C_i)$. Since e is in C_1 and C_2, so must $\hat{e}$. A_1 and A_2 are therefore the child regions of R on either side of $\hat{e}$.

The algorithm can easily determine if $\mathcal{F}_1$ is contained by $int(C)$ or $ext(C)$: let f_1 and f_2 be the incident faces of a searched edge e. Lemma 7 tells us if e is in the bounding cycle of R. If it is, then $\mathcal{F}_1$ must be contained by $ext(C)$: otherwise, the search could never reach an edge on the cycle bounding R (since edges of C are not visited) and $\mathcal{F}_1$ must be contained by $int(C)$.

Analysis: The above-described algorithm can be implemented in $O(|\mathcal{F}_1|\log^3 n + (|P_R| + |\partial P_R|^2)\log n)$ time. Finding the left-root vertices is the trickiest part; while $|L| = O(|\mathcal{F}_1|)$, $|L|$ could be much smaller than the number of vertices in C, even in the compact representation, so searching through all of C to find L is too slow. We give details in Section V-B. Assuming that left-root vertices can be found quickly, we analyze the remaining steps.

Lemma 9: The number of super edges to be searched in C is $O(|\mathcal{F}_1|)$.

Proof: Since G has degree three and degree-2 vertices are removed, the compact representation of R is 3-regular. Therefore, the number of edges is $\frac{3}{2}$ times the number of vertices. The lemma follows from Euler's formula. ■

The search step can be done in $O(|\mathcal{F}_1|)$ time starting with vertices of L. Given a super edge $\hat{e}$ found by this search, we find the first or last edge e (of G) on the path the super edge represents; this takes $O(1)$ time since e is associated with $\hat{e}$. For this to work, we need to identify, for each $v \in L$, the set of super edges incident to v which are not on C and which do not emanate to the right of C. Since the compact representation of R is three-regular, no super edge incident to v emanates to the right of C. We identify the super edge incident to v which is not on C, by using Lemma 7 on the first edge on each of the three super edge paths.

The total time spent in the Add step is $O(|\mathcal{F}_1|\log n)$: Faces f_1 and f_2 incident to e can be found in constant time using the graph representation of G. Applying Lemma 7 to e (to check if $\mathcal{F}_1$ is contained by $int(C)$ or $ext(C)$) takes $O(\log n)$ time. Finding the children of R that are ancestors of f_1 and f_2 also takes $O(\log n)$ time using the operations $jump(R, f_1, 1)$ and $jump(R, f_2, 1)$ in the top tree for $\mathcal{T}$.

B. Finding left-root vertices

We show how to find the set L of left-root vertices along C in $O(|\mathcal{F}_1|\log^3 n + |C|\log n)$ time where $|C|$ is the number of super edges in the compact representation of C. As a result of Section III, C has $O(|P_R| + |\partial P_R|^2)$ super edges and of three different types corresponding to: edges in $ext\mathrm{DDG}$ and $int\mathrm{DDG}(P_R)$ and edges and cycle paths in P_R. We will show how to use binary search to prune certain super edges of C that do not contain vertices of L. We assume that each super edge is on the boundary of a child region (as opposed to a child face) of R that is to the left of C. We extend the algorithm to child faces in the full version of the paper.

The following lemma is the key to using binary search along C:

Lemma 10: Let P be the shortest u_1-to-u_2 path in G that is also a subpath of C. For $i = 1, 2$, let e_i be the edge on P incident to u_i and let r_i be the child-region of R that is left of C and is bounded by e_i. Then $r_1 = r_2$ if and only if no interior vertex of P belongs to L.

Proof: The "if" part is trivial.

By our assumption, r_1 and r_2 are regions, not faces. Their bounding cycles must therefore be isometric. If $r_1 = r_2$ then Lemma 2 implies that P is a subpath of the boundary of r_1 so no interior vertex of P can belong to L. This shows the "only if" part. ■

1) Shortest path covering: In order to use Lemma 10, we cover the left-root vertices of C with two shortest paths P and Q. Let r be a vertex that is the endpoint of a super edge of C. Since C is isometric, there is a unique edge e such that C is the union of e and two shortest paths P' and Q' between r and the endpoints of e. Note that e could be in the interior of a super edge of C. The paths P and Q that we use to cover L are prefixes of P' and Q'.

To find e, we first find $\hat{e}$, the super edge that contains e. Since P and Q are shortest paths and shortest paths are unique, the weight of each path is at most half the weight of the cycle. To find $\hat{e}$, simply walk along the super edges of C and stopping when more than half the weight is traversed: $\hat{e}$ is the last super edge on this walk.

Given $\hat{e}$, we continue this walk according to the type of super edge that $\hat{e}$ is. If $\hat{e}$ corresponds to a cycle path, then, by definition, all the interior vertices of $\hat{e}$ have degree two in R and so cannot contain a left-root vertex; there is no need to continue the walk. P and Q are simply the paths along C from r to $\hat{e}$'s endpoints. This takes $O(|C|)$ time.

If $\hat{e}$ is an edge of $int\mathrm{DDG}(P_R)$ or $ext\mathrm{DDG}$, we continue the walk. We describe the process for $int\mathrm{DDG}(P_R)$ as $ext\mathrm{DDG}$ is similar: we continue the walk started above through the subdivision tree of P_R that is used to find $int\mathrm{DDG}(P_R)$. $\hat{e}$ is given by a path of edges in the internal dense distance graph of P_R's children in the subdivision tree. We may assume that we have a top tree representation of the shortest path tree containing this path and so we can find the child super edge $\hat{e}_c$ that contains e using binary search taking $O(\log^2 n)$ time. Recursing through the subdivision tree finds a cycle path or edge that contains e for a total of $O(\log^3 n)$ time.

When we are done, P and Q are paths of super edges from $ext\mathrm{DDG}$ or $int\mathrm{DDG}(P_R)$. They each have $O(|C| + \log n)$ super edges and they are found in $O(|C| + \log^3 n)$ time.

2) Building L: Using Lemma 10, we will decompose P into maximal subpaths $P_1, \ldots, P_k$ such that no interior vertex of a subpath belongs to L. Each subpath P_i will be associated with the child region of R to the left of C that is bounded (partly) by P_i. We repeat this process for Q and find L in $O(k)$ time by testing the endpoints of the subpaths.

Let $\hat{e}$ be one of the $O(|C| + \log n)$ super edges of P. $\hat{e}$ is either an edge of $ext\mathrm{DDG}$ or $int\mathrm{DDG}(P_R)$. Suppose $\hat{e}$ is in $int\mathrm{DDG}(P_R)$. We can apply Lemma 10 to the first and last edges on the path that $\hat{e}$ represents, and stop if there are no vertices of L in the interior of the path. Otherwise, with the top tree representation of the shortest path tree containing the shortest path representing $\hat{e}$, we find the midpoint of this path and recurse. If $\hat{e}$ is in $ext\mathrm{DDG}$, the process is similar.

Adjacent subpaths may still need to be merged after the above process, but this can be done in time proportional to their number.

How long does it take to build L? Let $\hat{e}$ be a super edge representing subpath $P_{\hat{e}}$ of P and let m be the number of interior vertices of $P_{\hat{e}}$ belonging to L. Then there are m leaves in the recursion tree for the search applied to $\hat{e}$. We claim that the height of the recursion tree is $O(\log^2 n)$. Let S be some root-to-leaf path in the recursion tree. If $\hat{e}$ is in $int\mathrm{DDG}(P_R)$, S is split into $O(\log n)$ subpaths, one for each level of the subdivision tree; in each level, the corresponding subpath is halved $O(\log n)$ times before reaching a single edge. If $\hat{e}$ is in $ext\mathrm{DDG}$, the search may go root-wards in the subdivision tree but once we traverse down, we are in $int\mathrm{DDG}$ and will thus not go up again. The depth of the recursion tree is still $O(\log^2 n)$.

At each node in the recursion tree, we apply two top tree operations to check the condition in Lemma 10 and one top tree operation to find the midpoint of a path for a total of $O(\log n)$ time. The total time spent finding the m vertices of L in $P_{\hat{e}}$ is $O(m \log^3 n)$ time. If $m = 0$, we still need $O(\log n)$ time to check the condition in Lemma 10. Summing over all super edges of P and Q, the time to identify L is $O(|C| \log n + |L| \log^3 n) = O(|C| \log n + |\mathcal{F}_1| \log^3 n)$.

C. Obtaining new regions

While we have found the required partition of the children of R and updated the region tree accordingly, it remains to find compact representations of the new regions R_1 and R_2. Recall that we only explicitly find one side of the partition, w.l.o.g., $\mathcal{F}_1$.

To find R_1, start an initially empty graph. In the *search* step, we explicitly find all the super edges of R_1 that are not on the boundary of C. Remove these edges from R and add them to R_1. The remaining super edges are simply subpaths of C between consecutive vertices of L. Edges corresponding to these subpaths are added to R_1.

The super edges left in R are exactly those in R_2. However, there may be remaining degree-two vertices that should be removed by merging adjacent super edges. All such vertices, by construction, must be in L, and so can be removed in $O(|\mathcal{F}_1|)$ time.

That super edges are associated with the first and last edges on their respective paths is easy to maintain given the above construction. The entire time required to build the new compact representation is $O(|\mathcal{F}_1|)$.

References

[1] D. Hartvigsen and R. Mardon, "The all-pairs min cut problem and the minimum cycle basis problem on planar graphs," *SIAM Journal on Discrete Mathematics*, vol. 7, no. 3, pp. 403–418, 1994.

[2] G. Borradaile, P. Sankowski, and C. Wulff-Nilsen, "Min st-cut oracle for planar graphs with near-linear preprocessing time," arXiv, Tech. Rep. 1003.1320, 2009.

[3] R. Gomory and T. Hu, "Multi-terminal network flows," *Journal of SIAM*, vol. 9, no. 4, pp. 551–570, 1961.

[4] G. N. Frederickson, "Fast algorithms for shortest paths in planar graphs with applications," *SIAM Journal on Computing*, vol. 16, pp. 1004–1022, 1987.

[5] G. Borradaile and P. Klein, "An $O(n \log n)$ algorithm for maximum st-flow in a directed planar graph," *Journal of the ACM*, vol. 56, no. 2, pp. 1–30, 2009.

[6] A. Bhalgat, R. Hariharan, D. Panigrahi, and K. Telikepalli, "An $\tilde{O}(mn)$ Gomory-Hu tree construction algorithm for unweighted graphs," in *Proceedings of the 39th Annual ACM Symposium on Theory of Computing*, 2007, pp. 605–614.

[7] G. Kirchhoff, "Ueber die auflösung der gleichungen, auf welche man bei der untersuchung der linearen vertheilung galvanischer ströme geführt wird." *Poggendorf Ann. Physik*, vol. 72, pp. 497–508, 1847, english transl. in Trans. Inst. Radio Engrs. CT-5 (1958), pp. 4-7.

[8] D. E. Knuth, *The Art of Computer Programming*. Addison-Wesley, 1968, vol. 1.

[9] J. Horton, "A polynomial time algorithm to find the shortest cycle basis of a graph," *SIAM Journal on Computing*, vol. 16, pp. 358–366, 1987.

[10] E. Amaldi, C. Iuliano, T. Jurkiewicz, K. Mehlhorn, and R. Rizzi., "Breaking the $o(m^2 n)$ barrier for minimum cycle bases." in *Proceedings of the 17th European Symposium on Algorithms*, ser. Lecture Notes in Computer Science, A. Fiat and P. Sanders, Eds., no. 5757, 2009, pp. 301–312.

[11] H. Whitney, "Planar graphs," *Fundamenta mathematicae*, vol. 21, pp. 73–84, 1933.

[12] G. L. Miller, "Finding small simple cycle separators for 2-connected planar graphs," *Journal of Computer and System Sciences*, vol. 32, no. 3, pp. 265–279, 1986.

[13] J. Fakcharoenphol and S. Rao, "Planar graphs, negative weight edges, shortest paths, and near linear time," *J. Comput. Syst. Sci.*, vol. 72, no. 5, pp. 868–889, 2006.

[14] P. N. Klein, "Multiple-source shortest paths in planar graphs," in *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms*, 2005, pp. 146–155.

[15] N. Alon and B. Schieber, "Optimal preprocessing for answering on-line product queries," Tel Aviv, Tech. Rep. 71/87, 1987.

[16] S. Alstrup, J. Holm, K. de Lichtenberg, and M. Thorup, "Maintaining information in fully dynamic trees with top trees," *ACM Transactions on Algorithms*, vol. 1, no. 2, pp. 243–264, 2005.

[17] J. Reif, "Minimum s-t cut of a planar undirected network in $O(n \log^2 n)$ time," *SIAM Journal on Computing*, vol. 12, pp. 71–81, 1983.

[18] M. R. Henzinger, P. N. Klein, S. Rao, and S. Subramanian, "Faster shortest-path algorithms for planar graphs," *Journal of Computer and System Sciences*, vol. 55, no. 1, pp. 3–23, 1997.