Efficient Reassembling of Three-Regular Planar Graphs

Assaf Kfoury*  
Boston University  
Boston, Massachusetts  
kfoury@bu.edu

Benjamin Sisson*  
Boston University  
Boston, Massachusetts  
bmsisson@bu.edu

July 11, 2018

Contents

Abstract ii

1 Introduction 1

2 Graph Reassembling 2

3 Plane and Planar Graphs 5

4 Algorithm KS 10

5 Conditions for the Optimality of Algorithm KS 27

6 Conclusion 40

References 41

A Appendix: Pseudocode of Algorithm KS 43

*Partially supported by NSF awards CCF-0820138 and CNS-1135722.
Abstract

A reassembling of a simple graph $G = (V, E)$ is an abstraction of a problem arising in earlier studies of network analysis [2, 7, 8, 14]. There are several equivalent definitions of graph reassembling; in this report we use a definition which makes it closest to the notion of graph carving. A reassembling is a rooted binary tree whose nodes are subsets of $V$ and whose leaf nodes are singleton sets, with each of the latter containing a distinct vertex of $G$. The parent of two nodes in the reassembling is the union of the two children’s vertex sets. The root node of the reassembling is the full set $V$. The edge-boundary degree of a node in the reassembling is the number of edges in $G$ that connect vertices in the node’s set to vertices not in the node’s set. A reassembling’s $\alpha$-measure is the largest edge-boundary degree of any node in the reassembling. A reassembling of $G$ is $\alpha$-optimal if its $\alpha$-measure is the minimum among all $\alpha$-measures of $G$’s reassemblings.

The problem of finding an $\alpha$-optimal reassembling of a simple graph in general was already shown to be NP-hard [10, 11, among others].

In this report we present an algorithm which, given a 3-regular plane graph $G = (V, E)$ as input, returns a reassembling of $G$ with an $\alpha$-measure independent of $n = |V|$ and upper-bounded by $2k$, where $k$ is the edge-outerplanarity of $G$. (Edge-outerplanarity is distinct but closely related to the usual notion of outerplanarity; as with outerplanarity, for a fixed edge-outerplanarity $k$, the number $n$ of vertices can be arbitrarily large.) Our algorithm runs in linear time $O(n)$. Moreover, we construct a class of 3-regular plane graphs for which this $\alpha$-measure is optimal, by proving that $2k$ is the lower bound on the $\alpha$-measure of any reassembling of a graph in that class.
1 Introduction

We skip repeating the informal definition of graph reassembly given in the abstract above; it is further elaborated and made formally precise in Section 2, where we also spell out the connection with graph carving.

Background and Motivation. Besides questions of optimization and the variations which it naturally suggests, graph reassembly is an abstraction of an operation carried out by programs in a domain-specific language for the design of flow networks [2, 7, 8, 14]. In network reassembly, the network is taken apart and reassembled in an order determined by the designer.

Underlying a flow network is a directed graph, where vertices and edges are assigned various attributes that regulate flow through the network.1 Programs for flow-network design are meant to connect network components in such a way that typings at their interfaces, i.e., formally specified properties at their common boundaries, are satisfied. Network typings guarantee there are no conflicting data types when different components are connected, and insure that desirable properties of safe and secure operation are not violated by these connections, i.e., they are invariant properties of the whole network construction.

A typing $\tau$ for a network component $X$ (or vertex cluster $X$ in this report’s terminology) formally expresses a constraining relationship between the variables denoting the outer ports of $X$ (or the edge-boundary $\partial(X)$ in this report). The smaller the set of outer ports of $X$ is, the easier it is to formulate the typing $\tau$ and to test whether it is compatible with the typing $\tau'$ of another network component $Y$. Although every outer port of $X$ is directed, as input port or output port, the complexity of the formulation of $\tau$ depends only on the number of outer ports (or $|\partial(X)|$ in this report), not on their directions.

If $\delta$ is a uniform upper bound on the number of outer ports of all network components, the time complexity of reassembling the network without violating any component typing $\tau$ can be made linear in the size $n$ of the completed network and exponential in the bound $\delta$ – not counting the pre-processing time $f(n)$ to determine an appropriate reassembling order. Hence, the smaller are $\delta$ and $f(n)$, the more efficient is the construction of the entire network. From this follows the importance of minimizing the pre-processing time $f(n)$ for finding a reassembling strategy that also minimizes the bound $\delta$ (or $\alpha$-measure in this report). If the reassembling order minimizes the quantity $\delta$ among all possible reassemblings, we say that the reassembling is $\alpha$-optimal.

Main Results. Let $G$ be the underlying graph of a flow network as described above, where we ignore direction on edges. While the problem of finding an $\alpha$-optimal reassembly of $G$ in general is NP-hard [10, 11, among others], we show in this report that the problem is solvable in linear time for 3-regular planar graphs within an upper bound on the $\alpha$-measure which is independent of graph size. Specifically, we prove the existence of a linear-time algorithm which, given an arbitrary 3-regular plane graph $G = (V, E)$, returns a reassembling of $G$ with an $\alpha$-measure $\leq 2k$ and independent of $n = |V|$, where $k$ is the edge-outerplanarity of $G$. The significance of the parameter $k$ comes from the fact that $n$ does not depend on it, and indeed, for a fixed value of edge-outerplanarity, the size $n$ of $G$ can be arbitrarily large.

Although our algorithm does not return $\alpha$-optimal reassemblings for all 3-regular plane graphs in general, we prove that it does return $\alpha$-optimal realizations for a significant class of 3-regular plane graphs. This class of graphs satisfies a certain “high density condition” (spelled out clearly in Section 5); given any 3-regular plane graph $G = (V, E)$ not satisfying this “density condition”, our algorithm may return a sub-optimal reassembling, though its $\alpha$-measure is guaranteed not to exceed twice the graph’s edge-outerplanarity $k$.

Organization of the Report. Sections 2 and 3 are background material. They set the stage for the rest of the report, making precise many of the terms we use throughour. Several of the concepts are closely related to familiar ones, e.g., edge outerplanarity in connection with standard outerplanarity (herein called vertex

1Such networks are typically more complex than the capacitated directed graphs that algorithms for max-flow (and other related quantities) and its generalizations (e.g., multicommodity max-flow) operate on.
Our algorithm, which we call KS for lack of a better name, is presented in Section 4. It proceeds by a long exhaustive (and exhausting!) case analysis of possible configurations of subgraphs in 3-regular plane graphs. Section 4 also includes a proof of correctness and a complexity analysis of KS; the proof is elementary in that it does not invoke any deep theorem from elsewhere in graph theory. The pseudocode of KS is included in Appendix A, and a full Python implementation is available for download from the website Graph Reassembling.\(^2\)

Section 5 defines a class of 3-regular plane graphs for which our algorithm KS returns α-optimal reassemblings. Section 5 also defines a “density condition” on the topology of 3-regular plane graphs which, if satisfied, guarantees that the returned reassemblings are α-optimal.

The concluding Section 6 explains how results on graph carving can be transferred to results on graph reassembling, and vice-versa. Outside the aforementioned problems of network design and analysis, Section 6 also includes an application of our results to a flow problem, namely, the existence of a fixed-parameter linear-time algorithm for maximum flow in planar graphs in general (not restricted to 3-regular).

2 Graph Reassembling

We refer to the vertices and edges of a graph \(G\) by writing \(V(G)\) and \(E(G)\). If \(G\) is simple, an edge is uniquely identified by the two-element set of its endpoints \(\{v, w\}\), which we also write as \(vw\).

There are several equivalent definitions of graph reassembling [10]. We here use a definition which makes it closest to the notion of graph carving [13] and requires the preliminary notion of a binary tree, also defined in a way that makes the connection with carving easier.\(^3\)

We take a (rooted, unordered) binary tree \(B\) over a set \(V = \{v_1, \ldots, v_n\}\) where \(n \geq 1\) to be a collection of \((2n – 1)\) non-empty subsets of \(V\) – these are the nodes of \(B\) – satisfying three conditions:

1. For every \(v \in V\), the singleton set \(\{v\}\) is in \(B\). These are the \(n\) leaf nodes of \(B\).
2. The full set \(V\) is in \(B\). This is the root node of \(B\).
3. Every node \(X \in B - \{V\}\) other than the root has a unique sibling \(Y \in B - \{V\}\) such that: \(X \cap Y = \emptyset\)

We also call \(B\) a binary reassembling of \(V\), or also a binary reassembling of \(G\) when \(V = V(G)\).\(^4\) To denote the reassembling of graph \(G\) according to \(B\), we write \((G, B)\). Depending on the context, we may refer to the nodes of \(B\) as vertex clusters of \(G\).\(^5\)

If \(X\) and \(Y\) are disjoint subsets of \(V(G)\), we write \(\partial_G(X, Y)\) to denote the set of all edges connecting \(X\) and \(Y\), each such edge with one endpoint in \(X\) and one endpoint in \(Y\):

\[
\partial_G(X, Y) \triangleq \{ vw \in E(G) \mid v \in X \text{ and } w \in Y \}.
\]

If \(G\) is clear from the context, we just write \(\partial(X, Y)\). We also write \(\partial(X)\) to denote \(\partial(X, E(G) - X)\).

There are different ways of optimizing the reassembling of \(G\), depending on the measure we choose on it. For \(X \subseteq V(G)\), the edge-boundary size of \(X\) in \(G\) is \(\overline{\partial}_G(X) \triangleq |\partial_G(X)|\). If \(G\) is clear from the context, we write \(\overline{\partial}(X)\) instead of \(\overline{\partial}_G(X)\). If \(X\) is a singleton set \(\{v\}\), then \(\overline{\partial}(\{v\})\) is simply \(\deg(v)\), the degree of \(v\). What we

\(^2\)http://cs-peoople.bu.edu/bmsisson/

\(^3\)Our definition of binary tree is unusual but more convenient for our analysis. It is the same as full binary merging in [3].

\(^4\)A binary reassembling in our sense mimicks what is called “agglomerative, or bottom-up, hierarchical clustering” in data mining.

\(^5\)To keep them apart, we reserve the words “node” and “branch” for the tree \(B\), and the words “vertex” and “edge” for the graph \(G\).
call the $\alpha$-measure of the reassembling $(G, B)$ is defined by:

$$\alpha(G, B) \triangleq \max \{ \overline{\delta}(X) \mid X \in B \}.$$ 

We say the reassembling $(G, B)$ is $\alpha$-optimal iff:

$$\alpha(G, B) = \min \{ \alpha(G, B') \mid B' \text{ is a binary reassembling of } G \}.$$ 

For other ways of optimizing graph reassembling relative to other measures, consult the earlier [10], none used in this report.

### 2.1 Connections with Graph Carving

Graph reassembling is essentially a different name for graph carving [13], although the former is perhaps better understood as a bottom-up process (of rebuilding a graph back to its original form) whereas the latter is a top-bottom process (of repeatedly bipartitioning a graph’s vertex set). To be more precise, a carving is defined relative to a routing tree (sometimes called a call-routing tree) $T$ for $G$, which is an unrooted tree where every internal node has degree $3$. The number of leaf nodes in $T$ is $n = |V(G)|$, the number of internal nodes in $T$ is $(n - 2)$, and the number of branches in $T$ is $(2n - 3)$. The leaf nodes of $T$ correspond to the vertices of $G$ and, for every branch $e$ of $T$, deleting $e$ yields two trees whose leaf nodes define a bipartition of the vertices of $G$; we say that the edge cut in $G$ corresponding to this bipartition is induced by $e$.

To refer to the carving of $G$ relative to the routing tree $T$, we here write $(G, T)$. The measure width$(G, T)$ is the maximum size of an edge cut in $G$ that is induced by a branch $e$ of $T$. The carving-width of $G$, denoted $cw(G)$, is the minimum width of all possible carvings of $G$:

$$cw(G) \triangleq \min \{ \text{width}(G, T) \mid T \text{ is a routing tree for } G \}.$$ 

We say the carving $(G, T)$ is optimal iff width$(G, T) = cw(G)$.

Given a carving $(G, T)$, we obtain a reassembling $(G, B)$ by turning $T$ into a rooted binary tree; namely, by introducing a fresh node to be the root labelled with the entire set $V = V(G)$, deleting one of the branches $X \overline{Y}$ of $T$, and introducing two new branches: from the root (now labelled with $V$) to each of $X$ and $Y$. We associate with every internal node $X$ the union of the two vertex sets associated with $X$’s two children. There are $(2n - 3)$ different ways of obtaining such $(G, B)$ from $(G, T)$, one for each branch in $T$. We say $(G, B)$ is one of the binary reassemblings induced by the carving $(G, T)$. Note that $(G, B)$ may or may not be $\alpha$-optimal, even if the carving $(G, T)$ from which it is induced is optimal; however, at least one of the $(2n - 3)$ reassemblings $(G, B)$ induced by an optimal carving $(G, T)$ is $\alpha$-optimal.

Conversely, given a reassembling $(G, B)$ of $G$, where $B$ has $n$ leaf nodes, $(n - 1)$ internal nodes, and $(2n - 2)$ branches, we (uniquely) obtain a carving $(G, T)$ by deleting the two branches from the root node of $B$ to its two children $X$ and $Y$, deleting the root node of $B$, and introducing a new branch $X \overline{Y}$. We now ignore the association of every internal node of $B$ with a subset of $V(G)$, but the one-one correspondence between $B$’s leaf nodes and $V(G)$ is preserved. We say the reassembling $(G, B)$ (uniquely) induces the carving $(G, T)$. It is easy to see that if $(G, B)$ is $\alpha$-optimal, then the induced carving $(G, T)$ is optimal.

The following is a consequence of the preceding discussion.

**Proposition 1.** For an arbitrary graph $G$, the following hold:

1. If $(G, T)$ is an optimal carving, then $(G, T)$ induces (not uniquely) an $\alpha$-optimal reassembling $(G, B)$.
2. If $(G, B)$ is an $\alpha$-optimal reassembling, then $(G, B)$ induces (uniquely) an optimal carving $(G, T)$.

The construction of $(G, T)$ from $(G, B)$ is carried out in constant time, the construction of $(G, B)$ from $(G, T)$ is carried out in linear time.
2.2 Further Preliminary Notions

In the opening paragraphs of Section 2 we restricted reassemblings to simple graphs (no multi-edges, no self-loops). The definitions can be generalized in the obvious way to multigraphs, where multi-edges and self-loops are allowed, which will be encountered in Section 4 and later. However, we agree that the degree of a vertex $v$ in a multigraph $G$ omits self-loops, i.e.,

$$\deg(v) \triangleq \left| \{ w \mid \text{there is } e \in E(G) \text{ whose endpoints are } \{v, w\} \text{ with } v \neq w \} \right|.$$

Let $B_1$ and $B_2$ be binary reassemblings of the sets $V_1$ and $V_2$, respectively. By our definition, $V_1$ and $V_2$ are the roots of $B_1$ and $B_2$. If $V_1 \cap V_2 = \emptyset$, we can construct a new binary reassembling $B$ whose root is $V_1 \cup V_2$, whose two children are $B_1$ and $B_2$.

The earlier definition of graph reassembling $(G, B)$ is in fact a total reassembling because $B$ is defined over the full set of vertices $V(G)$. If $V'$ is a subset of $V(G)$ and $B'$ is a binary tree over $V'$, then $(G, B')$ is a partial reassembling of $G$. A total reassembling is a special case of a partial reassembling. The notions of ‘total reassembling’ and ‘partial reassembling’ apply equally well to multigraphs. Unless stated otherwise, ‘reassembling’ means ‘total reassembling’, and ‘graph’ means ‘simple graph’.

The next proposition is used in several places later in this report. Let $B$ be a binary reassembling of a simple graph $G$, with $V = V(G)$. Consider a node $X \in B - \{V\}$ other than the root $V$, and its sibling $Y \in B - \{V\}$, so that $X \cap Y = \emptyset$. We write $\text{merge}(X, Y)$ to denote the node $(X \cup Y) \in B$ which is the common parent of $X$ and $Y$. We say $\text{merge}(X, Y)$ has degree $m$ iff $\bar{\theta}(X, Y) = m \geq 0$.

**Proposition 2.** If $B$ is a binary reassembling of a simple connected graph $G$, then there is a binary reassembling $B'$ of $G$ such that:

- every $\text{merge}(X, Y) = (X \cup Y)$ of two sibling nodes in $X, Y \in B'$ has degree $\geq 1$, and
- $\alpha(G, B') \leq \alpha(G, B)$.

In words, we can assume that, whenever two vertex clusters $X$ and $Y$ are merged in a reassembling, there is at least one edge connecting $X$ and $Y$; put differently, merging two vertex clusters $X$ and $Y$ that do not have at least one edge connecting them is a wasted step in the reassembling.

**Proof.** Suppose the given binary reassembling $B$ contains $p \geq 1$ degree-0 merges of two siblings. It suffices to show how we can construct a new binary reassembling $B'$ from $B$ such that $B'$ contains $(p - 1)$ degree-0 merges of two siblings.

Consider a degree-0 merge of two siblings $X_1, X_2 \in B$ in the binary tree $B$ such that every node/merge containing $\text{merge}(X_1, X_2) = (X_1 \cup X_2)$ has degree $\geq 1$, i.e., $\text{merge}(X_1, X_2)$ is the closest to the root $V(G)$ among all degree-0 merges in $B$. Let $X_3, X_4, \ldots, X_\ell$ be the nodes in $B$ such that:

- $X_3$ is the sibling of $(X_1 \cup X_2)$,
- $X_4$ is the sibling of $(X_1 \cup X_2 \cup X_3)$,
- $\ldots$
- $X_\ell$ is the sibling of $(X_1 \cup X_2 \cup X_3 \cup \cdots \cup X_{\ell-1})$,

and $X_1 \cup X_2 \cup \cdots \cup X_\ell = V$. By assumption, the merge $(X_1 \cup X_2 \cup \cdots \cup X_k)$ has degree $\geq 1$ for every $k \geq 3$.

Define the quantity $m_{i,j}$ as follows:

$$m_{i,j} \triangleq \bar{\theta}(X_i, X_j)$$
for all $1 \leq i < j \leq \ell$. By assumption $m_{1,2} = 0$, and $m_{1,3} \geq 1$ or $m_{2,3} \geq 1$ (or both). With no loss of

generality, let $m_{1,3} \geq 1$ and let $k \geq 3$ be the smallest index such that $m_{2,k} \geq 1$. Such an index $k$ must exist
since $G$ is connected. The desired $B'$ is obtained by re-arranging the nodes/merges above $X_1$ as follows:

$X_2$ is merged with $X_k$ (bypassing $X_3, \ldots , X_{k-1}$),

$X_1$ is merged with $X_3$ (bypassing $X_2$),

$(X_1 \cup X_3)$ is merged with $X_4$,

\[ \ldots \]

$(X_1 \cup X_3 \cup \cdots \cup X_{k-2})$ is merged with $X_{k-1}$,

$(X_1 \cup X_3 \cup \cdots \cup X_{k-1})$ is merged with $(X_2 \cup X_k)$,

$(X_1 \cup X_2 \cup X_3 \cup \cdots \cup X_k)$ is merged with $X_{k+1}$,

\[
\ldots .
\]

It is now easy to check that the number of 0-degree merges in $B'$ is one less than the number of 0-degree merges
in $B$ and that $\alpha(G, B') \leq \alpha(G, B)$. \qed

3 Plane and Planar Graphs

We recall standard notions and properties of planar graphs, some adapted to our own needs. These are used
explicitly in later sections or else explain the background in which to place our analysis.

A graph $G$ is a plane graph if it is drawn on the plane without any edge crossings. A graph $G$ is a planar graph
if it is isomorphic to a plane graph; i.e., it is embeddable in the plane in such a way that its edges intersect only
at their endpoints.

Given a plane graph $G$, a face is a maximal region $F$ of the plane such that $x, y \in F$ implies that $x$ and $y$ can
be joined by a curve which does not meet any edge of $G$. The unique unbounded face of $G$ is the exterior, or
outer face, of $G$. The edges of $G$ that are incident with a face $F$ form the boundary of $F$.

We distinguish between two kinds of edges, bounding and non-bounding. An edge $e$ of a plane graph $G$ is
bounding if $e$ is on the boundary of two adjacent faces of $G$, otherwise $e$ is non-bounding. Bounding and non-bounding edges are illustrated in Figure 1 and again in Figure 3.

Proposition 3. If $G$ is a biconnected plane graph, then every edge in $E(G)$ is a bounding edge and there are
no non-bounding edges in $G$.

Proof Sketch. If $G$ is biconnected, then every vertex is on a cycle. If every vertex is on a cycle, it is easy to see
that every edge is a bounding edge. Obvious details omitted. \qed

A natural measure on a planar graph is its outerplanarity index. Informally, if a planar graph $G$ is given with
one of its plane embeddings $G'$, then the outerplanarity of $G'$ (not that of $G$) is the number of times that all
the vertices on the outer face (together with all their incident edges) have to be removed in order to obtain the
empty graph. The outerplanarity index of $G$ is the minimum of the outerplanarities of all the plane embeddings
$G'$ of $G$. Deciding whether an arbitrary graph is planar can be carried out in linear time $O(n)$ and, if it is planar,
a plane embedding of it can also be carried out in linear time [12]. Given a planar graph $G$, the outerplanarity
index $k$ of $G$ and a $k$-outerplanar embedding of $G$ in the plane can be computed in time $O(n^2)$, and a 4-
approximation of its outerplanarity index can be computed in linear time [6].
For easier bookkeeping, we use a modified definition of outerplanarity, called edge-outerplanarity and already used by others [1]. There is a close relationship between this modified notion and the standard notion (Theorem 4 in Section 5.1 in [1]). In the case of three-regular plane graphs, the relationship is much easier to state: vertex-outerplanarity (the standard notion of outerplanarity) and edge-outerplanarity are “almost the same” (Proposition 8 below).

**Definition 4 (Edge Outerplanarity).** Let $G$ be a plane graph. If $E(G) = \emptyset$ and $G$ is a graph of isolated vertices, the edge outerplanarity of $G = 0$. If $E(G) \neq \emptyset$, we pose $G_0 = G$ and define $K_0 = L_0 \cup M_0$ as the set of edges lying on OuterFace$(G_0)$, where the edges in $L_0$ are bounding and the edges in $M_0$ are non-bounding.

For every $i > 1$, we define $K_i$ as the plane graph obtained after deleting all the edges in $K_{i-1} \cup \cdots \cup K_1 \cup C_i$ from the initial $G$ and $K_i = L_i \cup M_i$ the set of edges lying on OuterFace$(G_i)$, where the edges in $L_i$ are bounding and the edges in $M_i$ are non-bounding.

The edge outerplanarity of $G$, denoted $E$-outerplanarity$(G)$, is the least integer $k$ such that $G_k$ is a graph without edges, i.e., the edge outerplanarity of $G_k$ is 0. This process of peeling off the edges lying on the outer face $k$ times produces a $k$-block partition of $E(G)$, namely, $\{K_0, \ldots, K_{k-1}\}$.

An example of the decomposition of a plane graph $G$ according to Definition 4 is shown in Figure 3.

Let $G$ be a finite simple graph and $K \subseteq E(G)$. Let $G'$ be the subgraph of $G$ defined by:

$$E(G') = K \text{ and } V(G') = \{v \mid v \in \{v_1, v_2\} \subseteq V(G) \text{ and } \overline{v_1, v_2} \in K\}.$$  

We say $G'$ is the subgraph of $G$ induced by $K$ and write $G[K]$ to denote it.

As usual, a simple cycle in $G$ is a closed walk with no repeated vertices. A simple cycle $C$ in $G$ is a chordless cycle if no two distinct vertices on $C$ are connected by an edge that does not itself belong to $C$.

A cactus (plural: cacti) is a connected graph in which any two simple cycles have at most one vertex in common. Hence, in a cactus, every simple cycle is chordless, and every cactus is a planar graph. An (unrooted) tree, a connected acyclic graph, is a special case of a cactus.

**Proposition 5.** Let $G$ be a plane graph with $E$-outerplanarity$(G) = k$. Let $\{K_0, \ldots, K_{k-1}\}$ be the $k$-block partition of $E(G)$, with $K_i = L_i \cup M_i$ for every $0 \leq i \leq k - 1$, as specified in Definition 4. For every $i \in \{0, \ldots, k - 1\}$:

1. $G[K_i]$ is a finite collection of cacti.
2. $G[L_i]$ is a finite collection of simple cycles.
3. $G[M_i]$ is a finite collection of trees.

**Proof.** We provide some of the details for part 1, the proofs for parts 2 and 3 are just as straightforward. By the definitions, we have for every $i \in \{0, 1, \ldots, k - 1\}$:

$$E$$-outerplanarity$(G_i) = k - i$.

A straightforward induction on $i = 0, \ldots, k - 1$, shows that:

$$E$$-outerplanarity$(G_i) = E$$-outerplanarity$(G_{i+1}) + E$$-outerplanarity$(G[K_i])$.

---

6There is an unessential difference between our definition here and the definition in [1]. In Section 2.2 of that reference, “a $k$-edge-outerplanar graph is a planar graph having an embedding with at most $k$ layers of edges.” In our presentation, we limit the definition to plane graphs and say “a $k$-edge-outerplanar plane graph has exactly $k$ layers of edges.” Our version simplifies a few things later.
using the fact that $E(G_{i+1})$ is the set of all edges not lying on the outer face of $G_i$, and $K_i$ is the set of all edges lying on the outer face of $G_i$. Hence:

$$E_{\text{outerplanarity}}(G[K_i]) = E_{\text{outerplanarity}}(G_i) - E_{\text{outerplanarity}}(G_{i+1}).$$

Working backwards, $E_{\text{outerplanarity}}(G_k) = 0$ and:

$$E_{\text{outerplanarity}}(G[K_{k-1}]) = E_{\text{outerplanarity}}(G_{k-1}) - E_{\text{outerplanarity}}(G_{k-1} + 1),$$

which implies $G[K_{k-1}]$ is a finite collection of cacti. Similarly, for every $i = 0, \ldots, k - 1$:

$$E_{\text{outerplanarity}}(G[K_{k-i}]) = E_{\text{outerplanarity}}(G_{k-i}) - E_{\text{outerplanarity}}(G_{k-i+1}) = 1,$$

which implies $G[K_{k-i}]$ is a finite collection of cacti.

3.1 Three-Regular Plane Graphs

We specialize notions introduced earlier in Section 3 to the case of 3-regular graphs.

**Proposition 6.** Let $G$ be a 3-regular plane graph with $E_{\text{outerplanarity}}(G) = k \geq 1$. Let $\{K_0, \ldots, K_{k-1}\}$ be the $k$-block partition of $E(G)$, with $K_i = L_i \sqcup M_i$ for every $0 \leq i \leq k - 1$, as specified in Definition 4 and Proposition 5. We then have:

1. The edges in $L_0 \cup \cdots \cup L_{k-1}$ form vertex-disjoint cycles such that, for every such cycle $C$, the edges of $C$ are all in the same $L_i$ for some $0 \leq i \leq k - 1$.
2. The edges in $M_0 \cup \cdots \cup M_{k-1}$ form trees, whose non-leaf vertices have degree 3, such that for every such tree $T$, the edges of $T$ are all in the same $M_i$ for some $0 \leq i \leq k - 1$.
3. If in addition $G$ is biconnected, then $M_0 = \emptyset$.

We can thus view $G$ as a finite collection of vertex-disjoint cycles connected by trees. We thus call the latter inter-cycle trees (ICT’s). For later reference, we call each $K_i$ a layer of $G$, which is partitioned into cycle edges (those in $L_i$) and ICT edges (those in $M_i$). See Figure 3 for an example (which is not biconnected).

**Proof.** Straightforward by inspection. All details omitted.

The preceding proposition is not true for arbitrary plane graphs. Consider, for example, the non-regular plane graph $G$ in Figure 1: The cycles formed by the edges in $L_0 \cup L_1 \cup L_2$ are not vertex-disjoint.

In later sections we use the following definition. We identify a simple cycle and an ICT by the edges it contains.

**Definition 7 (Levels in Three-Regular Plane Graphs).** Let $G$ be a 3-regular plane graph as in Definition 4 and Propositions 5 and 6.

1. Let $C$ be a cycle in the induced subgraph $G[L_0 \cup \cdots \cup L_{k-1}]$ which therefore satisfies $C \subseteq L_i$ for some $0 \leq i \leq k - 1$ by part 1 of Proposition 6. We define:

$$\text{level}(C) \triangleq i$$

2. Let $T$ be an ICT in the induced subgraph $G[M_0 \cup \cdots \cup M_{k-1}]$ which therefore satisfies $T \subseteq M_i$ for some $0 \leq i \leq k - 1$ by part 2 of Proposition 6. We define:

$$\text{level}(T) \triangleq i$$
Assume $G$ is biconnected (part 3 in Proposition 6). It is easy to see that in the subgraph $G[L_0 \cup \cdots \cup L_{k-1}]$, there is one or more cycles of level $i$ for every $i \in \{0, \ldots, k-2\}$, and zero or more cycles of level $k-1$. In the subgraph $G[M_0 \cup \cdots \cup M_{k-1}]$, there is no ICT of level 0, two or more ICT’s of level $i$ for every $i \in \{1, \ldots, k-2\}$, and one or more ICT’s of level $k-1$.

We conclude by stating the relationship between the standard notion of outerplanarity and the notion of edge-outerplanarity used in this report. If $G$ is a plane graph, let $V$-outerplanarity$(G)$ denote the smallest $k$ such that $G$ is $k$-outerplanar (this is $k$-outerplanarity in the standard sense).

**Proposition 8.** If $G$ is a 3-regular plane graph, then $V$-outerplanarity$(G)$ and $E$-outerplanarity$(G)$ are “almost equal”, specifically: $V$-outerplanarity$(G) \leq E$-outerplanarity$(G) \leq 1 + V$-outerplanarity$(G)$.

This proposition is not true for arbitrary plane graphs, even if they are regular. Consider, for example, the four-regular plane graph $G$ in Figure 2, where $V$-outerplanarity$(G) = 2$ while $E$-outerplanarity$(G) = 4$.

**Proof Sketch.** For a 3-regular plane graph, the difference between $V$-outerplanarity$(G)$ and $E$-outerplanarity$(G)$ occurs in the last stage in the process of repeatedly removing (in the case of the standard definition) all vertices on the outer face and all their incident edges. The corresponding last stage in the modified definition may or may not delete all the edges; if it does not, then one extra stage is needed to delete all remaining edges.

---

**Figure 1:** A non-regular plane graph (on the left), with its bounding edges in boldface (in the middle), and its non-bounding edges in boldface (on the right). Cf. Proposition 3.

**Figure 2:** A four-regular plane graph $G$, with $V$-outerplanarity$(G) = 2$ and $E$-outerplanarity$(G) = 4$. Contrast with Proposition 8.
Figure 3: A 3-regular plane graph $G$ (at the top) with $E$-outerplanarity($G$) = 4 and its 4 layers of edges. Double-line edges in $K_i$ belong to $L_i$, the level-$i$ cycles, single-line edges in $K_i$ belong to $M_i$, the level-$i$ inter-cycle trees (ICT's), for $i \in \{0, 1, 2, 3\}$. This particular $G$ is decomposed into 6 vertex-disjoint cycles and 11 ICT's. The three edges of the single level-0 ICT are non-bounding, all the other edges are bounding; cf. Proposition 3.
4 Algorithm KS

We start by stating the main result of this section.

**Theorem 9.** There is an algorithm, herein named KS, which takes as input a biconnected 3-regular simple plane graph $G$ with $E$-outerplanarity($G$) = $k$ $\geq$ 2, and satisfies the following properties:

1. KS terminates in linear time $O(n)$, using $O(n)$ space, where $n = |V(G)|$, and
2. KS returns a binary reassembling $B = KS(G)$ of $G$ such that $\alpha(G, B) \leq 2k$.

*Note, in particular, the value of $\alpha(G, B)$ in the output $B$ is independent of $n$.*

The rest of this section is devoted to the proofs of Theorem 9, several lemmas, and several supporting examples. In Corollary 20, we lift the biconnectedness restriction. We begin with an informal description of algorithm KS. We use the terminology and notation introduced in Section 3.

4.1 Informal Description

At the topmost level of algorithm KS, there are two phases:

**Pre-Processing Phase.** This phase partitions the set $E(G)$ of edges into two non-empty disjoint sets, each partitioned into $k$ disjoint subsets (some of the latter possibly empty):

- the set $L_0 \cup \ldots \cup L_{k-1}$ of cycle edges,
- the set $M_1 \cup \ldots \cup M_{k-1}$ of ICT edges,

and carries out further classification of edges and vertices. We omit $M_0 = \emptyset$ (part 3 in Proposition 6).

**Processing Phase.** This is the actual algorithm at work, consisting of a sequence of repeated contractions, which we divide into two kinds, *collapses* and *merges*:

- a collapse contracts all the edges of an ICT and turns it into what we call a super vertex,
- a merge contracts all the cycle edges that connect a super vertex to what we call its clockwise neighbor, thus producing a larger super vertex.

A super vertex resulting from a collapse or a merge is not restricted to degree $= 3$. With the introduction of super vertices, two vertices may be connected by more than one edge and self-loops may be introduced, *i.e.*, with super vertices the graph becomes a multigraph. (More on super vertices below.)

Although the algorithm’s pseudocode and its Python implementation are sequential, it is better understood as carrying out its operations in parallel. Specifically, the algorithm starts by collapsing all eligible ICT’s, *i.e.*, ICT’s satisfying certain conditions $C$ (Section 4.3), then it carries out all merge operations satisfying certain conditions $D$ (Section 4.4). The purpose of carrying out the merges according to $D$ is to make a second group of ICT’s eligible for collapse according to $C$. After a second round of merges according to $D$, a third group of ICT’s becomes eligible for collapse according to $C$. This process continues by following every round of collapses according to $C$ by a round of merges according to $D$, with the latter making a new group of ICT’s eligible for collapse according to $C$ – until the whole graph $G$ becomes a single super vertex.
4.2 Further Classification and Terminology

The set $V(L_0 \cup \ldots \cup L_{k-1})$ is the set of cycle vertices, which is the same as the set of leaf vertices of the ICT’s. (See Figure 3 for an example.) We classify cycle vertices into two kinds, inward and outward. Let $v$ be a vertex on a cycle $X$ of level-$i$ for some $i \in \{0, \ldots, k-1\}$, and let $\{v,w_1,v,w_2,v,w_3\}$ be the three edges incident to $v$ with $\{v,w_1,v,w_2\} \subseteq X$ and $\{v,w_3\} \subseteq T$ for some ICT $T$ incident to $X$:

A. Inward Cycle Vertices: If $T$ is of level-$(i+1)$, then $v$ is an inward vertex.

B. Outward Cycle Vertices: If $T$ is of level-$i$, then $v$ is an outward vertex.

C. Super Vertices: Each collapse and each merge produces what we call a super vertex, which can be viewed as a ‘bag’ containing two or more ordinary vertices as well as the edges connecting them.

To distinguish vertices in the initial set $V(G)$ from super vertices, we sometimes call the former ordinary vertices. In contrast to ordinary vertices, always of degree $\geq 3$, super vertices can have arbitrary degrees $\geq 3$. We use late lower-case Roman letters $u, v, \text{and } w$, to denote ordinary vertices; late lower-case Greek letters $\varphi, \chi, \text{and } \psi$, to denote super vertices; and middle lower-case Greek letters $\mu, \nu, \text{and } \xi$, to denote both ordinary and super vertices. If $\varphi$ is a super vertex, then $V(\varphi)$ is the set of ordinary vertices contained in $\varphi$.

If a super vertex $\varphi$ is not the final super vertex containing the entire input graph $G$, then $\varphi$ always straddles one or more cycles. If we say super vertex $\varphi$ straddles cycle $X$, we mean $V(\varphi) \cap V(X) \neq \varnothing$ and $V(\varphi) \nsubseteq V(X)$, i.e., one or more of the vertices of $X$ are inside $\varphi$ and one or more are outside $\varphi$. If $\mu$ is a super vertex or an ordinary cycle-vertex, its set of cycles is:

$$\text{cycles}(\mu) \triangleq \{ X \mid X \text{ is a cycle and } \mu \text{ straddles } X \}.$$ 

The innermost cycle of $\mu$ is defined by:

$$\text{innermost-cycle}(\mu) \triangleq X \text{ where level}(X) = \max \{ \text{level}(X') \mid X' \in \text{cycles}(\mu) \}.$$ 

An ordinary cycle-vertex always straddles a single cycle, which is also its innermost cycle. Later in this section, Lemma 17 implies that $\text{innermost-cycle}(\mu)$ is uniquely defined, and that $\text{cycles}(\mu)$ is a chain of nested cycles of consecutive levels, with lower-level cycles nesting higher-level cycles and with the unique innermost-cycle($\mu$) having the highest level in that chain.

Once produced, a super vertex is viewed as a single vertex for the purposes of the algorithm, but the collection of ordinary vertices included in the super vertex are recorded and part of the final output returned by the algorithm. Super vertices are not classified into ‘inward’ and ‘outward’, in contrast to ordinary cycle-vertices which are always inward or outward.

D. Two Invariants of the Algorithm: During algorithm execution, cycle edges and ICT edges never change their designation: they remain ‘cycle edges’ and ‘ICT edges’, respectively, until they get included into super vertices and removed from further consideration.

For the notion of clockwise neighbor used below, we view every cycle edge as directed clockwise and this direction remains unchanged throughout algorithm execution. ICT edges do not have a direction.

E. Another Invariant of the Algorithm: The edges of an ICT $T$, but not necessarily its leaf vertices, remain unchanged until the collapse of $T$ places all of $E(T)$ in the same super vertex; some of the leaf vertices of $T$ may already be included in super vertices in earlier steps of the algorithm. More precisely, there is an ICT $U$ in the initial $G$ such that either $T$ is $U$ or else, if there is an edge $e_1 = \overline{\varphi w}$ in $T$ where $\varphi$ is a super vertex, then $\varphi$ is a leaf vertex of $T$ and there is an edge $e_0 = \overline{v w}$ in the initial $U$ such that:

$w$ is a leaf vertex of $U$, $w$ is included in $\varphi$, and $\text{cycles}(w) \subseteq \text{cycles}(\varphi)$,
and we can view \( e_0 \) and \( e_1 \) as the same edge. This implies that, even though the edges of \( T \) remain unchanged until a collapse operation places them all in a super vertex \( \psi \), the cycles straddled by a leaf vertex of \( T \) may become a larger set of cycles during algorithm execution such that, after the collapse operation, some edges of those cycles become self-loops of \( \psi \) or incoming edges to \( \psi \) or outgoing edges from \( \psi \). Only cycle edges become self-loops (of super vertices), ICT edges do not become self-loops.

We use the terms ‘sibling’ and ‘clockwise neighbor’ to qualify two different relationships between cycle vertices, whether ordinary or super:

F. Siblings: We call siblings the leaf vertices of an ICT \( T \). If they all – except possibly for one outward ordinary vertex \( v \) – have the same innermost cycle \( X \) and can be listed consecutively as \( X \) is traversed clockwise, i.e., without encountering interleaved vertices of \( X \) belonging to ICT’s other than \( T \), then we call them the consecutive siblings of \( T \) and the exception vertex \( v \), if it exists, the root of \( T \).

G. Clockwise Neighbors: Let \( \mu \) and \( \nu \) be cycle vertices, super or ordinary, and \( X \) the innermost cycle of \( \mu \). We say \( \nu \) is the clockwise neighbor of \( \mu \) if there is an edge of \( X \) in the clockwise direction from \( \mu \) to \( \nu \).

Note: We require that \( X \) be the innermost cycle of \( \mu \) only, not \( \nu \), and we do not disallow that \( \mu = \nu \) in which case \( \mu \) is a clockwise neighbor of itself.

### 4.3 Conditions \( \mathcal{C} \) for the Collapse Operation: How to Contract ICT Edges

We start applying the collapse operation whenever condition \((\mathcal{C}.0)\) is satisfied, and we apply it as long as conditions \((\mathcal{C}.1), (\mathcal{C}.2)\) are simultaneously satisfied. Condition \((\mathcal{C}.3)\) specifies how each ICT is collapsed.

\((\mathcal{C}.0)\) No merge is possible according to conditions \( \mathcal{D} \) below.

\((\mathcal{C}.1)\) Consider all the level-\((i + 1)\) ICT’s occurring between a level-\(i\) cycle \( X \) and zero or more level-\((i + 1)\) cycles. These are the ICT’s immediately enclosed in \( X \) and outside level-\((i + 1)\) cycles, if any: These ICT’s are not eligible for collapse before all the ICT’s that are incident to \( X \) from the outside – except possibly for one such incident ICT – have been collapsed.

\((\mathcal{C}.2)\) Assuming condition \((\mathcal{C}.1)\) is met, an ICT \( T \) is eligible for collapse when all its leaf vertices, except possibly for one outward ordinary vertex \( v \), are consecutive siblings on a cycle \( X \). Each of the consecutive siblings on \( X \) is: either a super vertex or an inward ordinary vertex.

\((\mathcal{C}.3)\) The collapse of an ICT \( T \) turns \( T \) into a super vertex, by contracting all of \( T \)’s tree edges, and is carried out to minimize the \( \alpha \)-measure and in linear time.

**Remark.** Condition \((\mathcal{C}.1)\) is not necessary for the algorithm to work correctly; we include it only to force execution to proceed in an ‘outside-in’ fashion, i.e., by applying the collapse operation to the ICT’s outside a cycle as much as possible before applying it to the ICT’s inside the same cycle. For later reference, we distinguish the two cases of “an ICT \( T \) is eligible for collapse” in \((\mathcal{C}.2)\):

**Type-a**: The leaf vertices of \( T \) are all consecutive siblings.

**Type-b**: The leaf vertices of \( T \), except for one outward ordinary \( v \), are consecutive siblings.

A type-a ICT is a rootless tree, where non-leaf vertices (all ordinary) have each degree three and leaf vertices (some ordinary, some super) have each degree one. A type-b ICT is a rooted tree, where the sole outward ordinary vertex is the root, non-leaf vertices (all ordinary) have each degree three, and the root and the leaf vertices (some ordinary, some super) have each degree one. The classification of ICT’s into type-a and type-b applies only to ICT’s eligible for collapse, not to ICT’s not eligible for collapse at any time during execution; the classification is thus best understood dynamically as the algorithm progresses. Examples 12 and 13 illustrate the difference between type-a and type-b.
4.4 Conditions \( \mathcal{D} \) for the Merge Operation: How to Contract Cycle Edges

A merge is applied in one of two cases: (i) two super vertices or (ii) one super vertex and one ordinary vertex. It is not applied to two ordinary vertices. Let \( \varphi \) be the super vertex involved in a merge operation and \( \mu \) the vertex, ordinary or super, involved in the same operation. The result of applying a merge to \( \varphi \) and \( \mu \) is a new super vertex obtained by contracting all the cycle edges (one or more) connecting \( \varphi \) and \( \mu \).

We start applying the merge operation from the moment condition \( \mathcal{D} \).0 \) is satisfied, and we apply it repeatedly as long conditions \( \{ (\mathcal{D} \).1, (\mathcal{D} \).2 \} \) are simultaneously satisfied.

(\( \mathcal{D} \).0) No collapse is possible according to conditions \( \mathcal{E} \) above.

(\( \mathcal{D} \).1) \( \varphi \) is a super vertex, \( \mu \) is the clockwise neighbor of \( \varphi \), and \( \varphi \) is not a leaf vertex of an ICT.

(\( \mathcal{D} \).2) If \( \mu \) is additionally an outward ordinary vertex, then \( \varphi \) is also the clockwise neighbor of \( \mu \); i.e., \( \varphi \) and \( \mu \) are distinct (because \( \varphi \) is super and \( \mu \) is ordinary) clockwise neighbors of each other.

Note the fact that \( \varphi \) is not a leaf vertex in (\( \mathcal{D} \).1), which implies that \( \varphi \) and \( \mu \) are not siblings, i.e., leaf vertices of the same ICT. Several remarks are in order in relation to conditions \( \{ (\mathcal{D} \).1, (\mathcal{D} \).2 \} \), which also spell out the different special cases subsumed by these two conditions:

1. \( \mu \) is an inward ordinary vertex: Contracting the clockwise cycle edge from \( \varphi \) to \( \mu \) produces a super vertex which is a leaf vertex, thus not satisfying condition (\( \mathcal{D} \).1) and not eligible for an additional merge.
2. \( \mu \) is an outward ordinary vertex: Condition (\( \mathcal{D} \).2) requires that \( \varphi \) is the clockwise neighbor of \( \mu \), thus implying that \( \mu \) is the only ordinary vertex on its cycle.
3. \( \mu \) is a super vertex: \( \varphi \) is distinct from \( \mu \) and \( \varphi \) is not the clockwise neighbor of \( \mu \).
4. \( \mu \) is a super vertex: \( \varphi \) is distinct from \( \mu \) and \( \varphi \) is the clockwise neighbor of \( \mu \) (in this case \( \varphi \) and \( \mu \) are super vertices that are clockwise neighbors of each other).
5. \( \mu \) is a super vertex: \( \varphi \) is the same super vertex as \( \mu \), which is thus a clockwise neighbor of itself.

This last special case is the only case when the merge operation does not create a new super vertex that contains a larger subset of ordinary vertices; its purpose is only to contract all the self-loops of \( \varphi = \mu \).

Because of conditions (\( \mathcal{E} \).0) and (\( \mathcal{D} \).0), the algorithm proceeds by alternating rounds of collapses and merges. Each of the two kinds of rounds executes a maximum number of operations of its kind. A round of collapses creates self-loops in general, the succeeding round of merges eliminates all resulting self-loops (and contracts other cycle edges in general). Since all the vertices in the initial input graph \( G \) are ordinary, the algorithm starts with a round of collapses and stops with a round of merges. We later identify these alternating rounds by numbering them; round \( k \) will be a round of collapses when \( k \geq 1 \) is odd, and it will be a round of merges when \( k \geq 2 \) is even.

**Remark About Self-Loops:** The only case of an ICT collapse that does not generate a self-loop is when the ICT is a single tree-edge connecting two distinct cycles (of the same level \( i \), or of two consecutive levels \( i \) and \( i+1 \), for some \( i \geq 0 \)). A merge never generates self-loops; it only eliminates them, if it is according to case 5 of condition (\( \mathcal{D} \).1).

Let \( G \) be the initial input graph and \( G' \) a graph obtained from \( G \) by a sequence of collapse and merge rounds, the last of which being a round of collapses. Let \( e' \) be a self-loop in \( G' \), which is the ‘descendant’ of a cycle edge \( e \) in \( G \), in the sense that the two endpoints of \( e' \) are a transformation of the two endpoints of \( e \). (More precisely, if \( e' \) is the self loop \( \varphi \varphi \) then \( e = \overline{v \varphi} \overline{\varphi w} \) where both of the ordinary vertices \( v \) and \( w \) are included in the super vertex \( \varphi \).) If we apply a round of merges to \( G' \) to obtain \( G'' \), then \( e' \) along with all other self-loops disappear in \( G'' \) and the cycles in \( G'' \) are therefore a subset of the cycles in \( G' \).
4.5 Examples

Before we tackle the correctness of our algorithm KS in Section 4.6, we include six examples illustrating the progression of KS. For the graph $G$ in each example, the constructed reassembling $\mathcal{B}$ consists of the super vertices produced by KS, in addition to the singletons $\{v\}$ for all $v \in V(G)$. We will thus have:

$$\alpha(G, \mathcal{B}) = \max \{ \deg(\mu) \mid \mu \text{ is an ordinary vertex or a super vertex produced by KS} \}$$

and $\alpha(G, \mathcal{B}) \leq 2 \cdot E$-outerplanarity($G$) in each example, as predicted by Theorem 9. The first example is very simple, and each successive example exhibits a few more complications than the preceding one.

In all the examples, super vertices are shown enclosed in colored boundaries: red if produced by a round of collapses, green if produced by a round of merges.

**Example 10.** On the left in Figure 4 is a 3-regular plane graph (the “cube”). It consists of two nested simple cycles, connected by 4 inter-cycle trees (ICT’s); in this case, each ICT is a single edge. On the right in Figure 4, we show the progression of our algorithm KS. The four innermost super vertices are obtained by the first round of collapses (round 1 of KS) which contract only ICT’s; these are enclosed in red boundaries. The ordinary vertices $\{a, b\}$ are enclosed in one super vertex, the ordinary vertices $\{c, d\}$ in a second super vertex, the ordinary vertices $\{e, f\}$ in a third super vertex, and the ordinary vertices $\{g, h\}$ in a fourth super vertex.

The following round of merges (round 2 of KS) contracts only cycle edges and produces three nested super vertices in succession. These are shown on the right in Figure 4 enclosed in green boundaries. The merge of super vertex $\{a, b\}$ with super vertex $\{c, d\}$, and then that of super vertex $\{a, b, c, d\}$ with super vertex $\{e, f\}$, do not create self-loops; these are two merge operations according to case 3 of condition (D.1). One more merge, according to case 4 of condition (D.1), puts together super vertex $\{a, b, c, d, e, f\}$ and super vertex $\{g, h\}$ to produce the final super vertex, and again this last merge does not create any self-loop.

![Figure 4: A 3-regular plane graph (left) and the progression of algorithm KS on it (right).](image)

**Example 11.** In Figure 5 we show a 3-regular plane graph decomposed into cycles and inter-cycle trees (ICT’s). In Figure 6 we show the progression of our algorithm KS on this graph. The five innermost super vertices enclosed in red boundaries on the left of Figure 6 are obtained from a first round of collapses (round 1 of KS), followed by several merges (part of round 2 of KS) that contract the self-loops of the five resulting super vertices. Actually, in this case, only the top super vertex among these five has self-loops to be contracted, shown as dashed edges on the left of Figure 6.

The remaining merges of round 2 contract several cycle edges, producing the two super vertices enclosed in green boundaries on the left of Figure 6. The four lower super vertices enclosed in red are put together to
produce the lower super vertex in green on the left of Figure 6; this is result of three merge operations according to case 3 of condition (D.1), followed by one more merge operation according to case 2 of condition (D.1).

At this point, there remains only one ICT, with two ordinary vertices and two super vertices. One more round of collapses (round 3 of KS) contracts this ICT into a super vertex with three self loops (the three remaining cycle edges that have not been yet contracted, shown as dashed edges on the right of Figure 6), and the latter are contracted by a final round of merges (round 4 of KS) which produce the final super vertex shown on the right of Figure 6 enclosed in the outermost green boundary.

Figure 5: A 3-regular plane graph (left figure) decomposed into 2 cycles and 6 ICT’s (right figure), 4 ICT’s are single-edge (light dashed), 2 multi-edge (bold dashed), only one not initially eligible for collapse (second from top).

Figure 6: KS’s progression on the graph in Figure 5. There is one ICT (left figure) with 2 ordinary vertices (bold black) and 2 super vertices (green), eligible for collapse (type-a) and contracted in KS’s round 3 (right figure). KS’s round 4 contracts the self-loops (dashed edges in right figure) and produces the last super vertex (green).

Example 12. The graph in Figure 7 (on the left) is decomposed into 3 cycles and 5 ICT’s (on the right). The middle three-edge ICT is not initially eligible for collapse, and becomes eligible for collapse after rounds 1, 2, 3 and 4 (middle figure in Figure 8); and when it becomes eligible for collapse, it is a type-a ICT, because its three leaf vertices (one ordinary and two super) are consecutive siblings on the same cycle.
Figure 7: A 3-regular plane graph (left figure) decomposed into 3 cycles and 5 ICT’s (right figure), 4 ICT’s are single-edge, one is three-edge. Two single-edge ICT’s are initially eligible for collapse, the leftmost and the rightmost.

Figure 8: KS’s progression on the graph in Figure 7, after rounds 1 and 2 (top left), rounds 3 and 4 (top right), and rounds 5 and 6 (bottom). KS’s rounds 4 and rounds 6 contract two and three self-loops (dashed edges).

Example 13. This is a variation on the graph in Example 12: All ICT’s are single-edge, except for one which is three-edge, just like in Example 12. Whereas the single three-edge ICT in Example 12 is type-a when it becomes eligible for collapse (after round 4), the single three-edge ICT in this example is type-b when it becomes eligible for collapse (after round 4). This illustrates how an ICT is classified as type-a or type-b dynamically, i.e., it is so classified at the time when it first becomes eligible for collapse.

Figure 9: A 3-regular plane graph (left) decomposed into 4 cycles and 7 ICT’s (right). One ICT is three-edge, 6 ICT’s are single-edge, two of the latter initially eligible for collapse, the leftmost and the rightmost, both type-b.
Figure 10: KS’s progression on the graph in Figure 9, after rounds 1 and 2 (top figure), rounds 3 and 4 (second figure from top), rounds 5 and 6 (third figure from top), and rounds 7 and 8 (bottom figure). KS’s rounds 4, rounds 6, and rounds 8, contract two, one, and four self-loops (dashed edges), respectively.

Example 14. Figure 11 shows a 3-regular plane graph with 54 vertices, complicated enough to illustrate different aspects of our algorithm KS. Figures 12 and 13 show the progression of algorithm KS on this graph.

On the left in Figure 12, the 14 innermost super vertices result from the first round of collapses (round 1 of
KS) which contract only ICT’s; these are enclosed in red boundaries. As a result of round 1, two self-loops are created, shown as dashed edges on the left in Figure 12. The following round of merges (round 2 of KS) contracts the self-loops and produces 7 new super vertices, enclosed in green boundaries on the left in Figure 12. Six of these 7, enclosed in square green boundaries, are each obtained in two steps: first, by merging a super vertex with its clockwise neighbor, also a super vertex, according to case 3 of (\(D.1\)); second, by merging the resulting super vertex with its clockwise neighbor, an outward ordinary vertex, according to case 2 of conditions \{(\(D.1\), (\(D.2\)}). The remaining super vertex in green of those 7, on the left in Figure 12, is obtained by merging a super vertex with its clockwise neighbor, an inward ordinary vertex, according to case 1 of (\(D.1\)).

Out of the 14 innermost super vertices in red on the left in Figure 12, one is not involved in any contractions of round 2 (i.e., first round of merges); conditions \{(\(D.1\), (\(D.2\)} and their 5 special cases do not apply to it.

The next round of collapses (round 3 of KS) produces the super vertices enclosed in red boundaries on the right in Figure 12. There are 8 of these super vertices, 7 new and 1 from round 1 that was not involved in any contractions in the intermediate round 2. Also on the right in Figure 12, three new super vertices in green boundaries are shown, resulting from the following round of merges (round 4 of KS).

The result of the next round of collapses and the following round of merges, round 5 and round 6 of KS respectively, is shown on the left in Figure 13. At this point, all the ordinary vertices in the initial graph are included in one of two disjoint super vertices. The latter two super vertices are connected by two cycle edges and one inter-cycle edge; these are the three remaining uncontracted edges.

The contraction of that last inter-cycle edge is the result of one more round of collapses, round 7 of KS, which creates two self-loops from the two remaining uncontracted cycle edges (shown as dashed edges on the right in Figure 13). The latter are contracted by one more round of merges, round 8 of KS.

Figure 11: A 3-regular plane graph (left) decomposed into 11 cycles and 24 ICT’s (right), 21 ICT’s are single-edge (light dashed edges) and 3 are multi-edge (bold dashed edges). Fourteen of the 24 initial ICT’s are eligible for collapse; of these 14, one ICT is type-\(a\) (nested multi-edge on the right), and 13 ICT’s are type-\(b\).
Figure 12: Progression of algorithm KS on the graph in Figure 11. All ICT’s eligible for collapse on the left, 7 of them, are type-\( b \). All ICT’s eligible for collapse on the right, 2 of them, are type-\( b \); one of these two involves the topmost super vertex, the other involves the bottommost super vertex.

Figure 13: Progression of algorithm KS on the graph in Figures 11 and 12. There is only one (single-edge) ICT on the left which is type-\( a \) and eligible for collapse. There are no ICT’s on the right.
Example 15. In Figure 14 is a 3-regular plane graph with 42 vertices, again complicated to exhibit additional aspects of our algorithm KS. The top of Figure 14 shows the initial input graph $G$, and the bottom shows $G$’s decomposition into 5 cycles and 18 ICT’s. In this case, it is easy to see that $E$-outerplanarity$(G) = 4$.

The progression of algorithm KS is shown in Figure 15. The top of the figure shows the result of round 1 of collapses (7 of them) followed by round 2 of merges (also 7 of them). Super vertices resulting from round 1 are enclosed in red boundaries, and super vertices resulting from round 2 are enclosed in green boundaries. There are only 6 super vertices shown in green, not 7, because one of them (at the north-west corner of the graph) is obtained by two merges: the first according to case 3 of condition ($D$.1) and the second according to case 1 of condition ($D$.1).

The middle of Figure 15 shows the result of round 3 of collapses followed by round 4 of merges. As in the two previous rounds, super vertices resulting from collapses are enclosed in red boundaries (9 such boundaries) and super vertices resulting from merges are enclosed in green boundaries (7 such boundaries). Two of the 7 super vertices in green are each the result of two consecutive merges, one according to case 3 of condition ($D$.1) followed by one according to case 1 of condition ($D$.1); these two super vertices in green are at the north-west corner of the graph in the middle of Figure 15 and at the south-center in the middle of the same figure.

One of the merges from round 4 does not create a new super vertex; it only contracts self-loops, shown as dashed edges in the middle graph in Figure 15. A merge operation that contracts self-loops is one according to case 5 of condition ($D$.1).

At this point, there are only two ICT’s that have not yet been contracted. Each of these two consists of a single ordinary vertex (shown as a boldface vertex in the middle graph in Figure 15) and three super vertices that share a same innermost cycle.

The bottom of Figure 15 shows the result of round 5 of collapses (two of them) followed by round 6 of merges (four of them). The two collapses produce two super vertices, enclosed in red boundaries at the bottom of Figure 15, each with 5 self-loops, shown as dashed edges. These 10 self-loops are contracted by two merges according to case 5 of condition ($D$.1). There are now 3 super vertices, connected by a total of 5 cycle edges that are yet to be contracted. A merge according to case 3 of condition ($D$.1), followed by a merge according to case 4 of condition ($D$.1), contract these 5 cycle edges and produce the final and only super vertex. □
Figure 15: KS's progression on the graph in Figure 14. The top figure shows the graph after rounds 1 and 2, the middle figure after rounds 3 and 4, and the bottom figure after rounds 5 and 6. In the top figure, there are 9 ICT's eligible for collapse; 8 of these are single-edge and type-b, and one (at the west) is multi-edge and type-a. In the middle figure, there are 2 ICT's eligible for collapse, both multi-edge and type-a. In the bottom figure, there are no ICT's eligible for collapse.
4.6 Proof of Correctness and Complexity

We prove the correctness of our algorithm KS by referring to the conditions $\mathcal{C}$ for collapses (Section 4.3) and the conditions $\mathcal{D}$ for merges (Section 4.4). We purposely avoid any explicit reference to the pseudocode in Appendix A for two reasons: first, any such reference would obscure the intuition underlying the proofs of Lemmas 16, 17, 18, and 19, as well as an informal understanding of their correctness; second, the pseudocode in Appendix A proposes one particular way (there are others) of implementing KS and programming the conditions in $\mathcal{C}$ and $\mathcal{D}$.

Let $G$ be a biconnected 3-regular simple plane graph as in Theorem 9. The execution of our algorithm can be represented by a sequence of graphs (multigraphs after the first $G_0$):

$$(\spadesuit) \quad G_0 = G \quad G_1 \quad G_2 \quad G_3 \quad \ldots \quad G_i \quad G_{i+1} \quad \ldots \quad G_p$$

where for every odd $i \geq 1$ (resp. even $i \geq 2$), graph $G_i$ is obtained from the preceding $G_{i-1}$ by a maximum round of collapses (resp. merges). Thus, round 1 is a round of collapses, round 2 is a round of merges, round 3 is a round of collapses, etc. This sequence is bound to terminate after a finite number $p$ of rounds, because the collapse and merge operations contract edges and there are finitely many edges. We say the sequence $(\spadesuit)$ terminates successfully if the last graph $G_p$ is a single super vertex with no edges. We use Lemma 16 to prove Part 1 of Theorem 9.

**Lemma 16.** The sequence $(\spadesuit)$ terminates successfully.

**Proof.** We prove the lemma by contradiction. Assume the sequence $(\spadesuit)$ does not terminate successfully, i.e.:

1. $\widetilde{G}$ is not a single super vertex, and
2. no collapse operation and no merge operation can be applied to $\widetilde{G}$.

For an easy case first, assume that there are no ICT edges in $\widetilde{G}$, because they have all been contracted in earlier rounds, and that all the edges left in $\widetilde{G}$ are cycle edges. An ICT $T$ before a collapse operation, whether of type-$a$ or type-$b$ (see Remark in Section 4.3), has at least one ordinary vertex; after $T$’s collapse, all of it vertices, ordinary or super, become merged into a single super vertex. Hence, all vertices in $\widetilde{G}$ are now super vertices.

Consider an innermost cycle $X$ in $\widetilde{G}$. Then $X = \text{innermost-cycle}(\varphi)$ for some super vertex $\varphi$. If there is no super vertex $\psi \neq \varphi$ such that $X = \text{innermost-cycle}(\psi)$, then $X$ is a self-loop of $\varphi$, otherwise we can choose $\psi$ as the clockwise neighbor of $\varphi$. In either case, a round of merges can be applied to $\widetilde{G}$, contradicting our initial assumption.

Assume next that there is an ICT $T$ which has not been collapsed in $\widetilde{G}$. Let all of $T$’s leaf vertices, except possibly for one outward ordinary $w$ in case $T$ is of type-$b$, be siblings on a cycle $X$, which are necessarily super vertices or inward ordinary vertices. Among all ICT $T' \neq T$ in $\widetilde{G}$ that have not been collapsed, select $T$ so that $\text{level}(X) \leq \text{level}(X')$ where $X'$ is the cycle on which the leaf vertices of $T'$ are located. Hence, there are no ordinary outward vertices on $X$, because any such ordinary outward vertex would be the root of a type-$b$ ICT $T'$ incident to $X$ from the outside and such that: (i) $T'$ has not been collapsed and (ii) $\text{level}(X') < \text{level}(X)$, which would contradict the conditions for our selection of $T$.

Moreover, all the super vertices on $X$ must be leaf vertices of ICT’s enclosed in $X$, i.e., ICT’s that are incident to $X$ from the inside of $X$. Indeed, if any of these super vertices, say $\varphi$, were not a leaf vertex of any such enclosed ICT, it would be possible to apply the merge operation to contract the cycle edge connecting $\varphi$ with its clockwise neighbor on $X$ and $\widetilde{G}$ would not be the last graph in the sequence $(\spadesuit)$. Hence, it follows that:

($$) all the vertices on $X$ are either super vertices or ordinary inward vertices, and all of them are leaf vertices of ICT’s that are enclosed in $X$. 

22
If all the leaf vertices of the selected \( T \) are consecutive siblings on \( X \), it contradicts the assumption that no \textit{collapse} operation can be applied to \( \hat{G} \). So, the leaf vertices of \( T \) cannot be consecutive on \( X \). But from this, together with fact (S), it is easy to argue there must be another ICT \( T_0 \) whose leaf vertices are consecutive siblings on \( X \), implying again that the \textit{collapse} operation can be applied to \( \hat{G} \). We conclude that the sequence (♠) always terminate successfully. \( \square \)

We use Lemma 17 in the proof of Lemma 18, and we use the latter to prove Part 2 of Theorem 9. In the lemmas to follow, \( G \) is possibly a plane multigraph resulting from earlier applications of the \textit{collapse} and \textit{merge} operations.

**Lemma 17.** Let \( T \) be an ICT in \( G \) satisfying condition (6.2) all of whose leaf vertices, except possibly for one outward ordinary \( w \), are consecutive siblings on a cycle \( X \). Let \( M = \{ \mu_1, \ldots, \mu_p \} \) be the consecutive siblings on \( X \), each of which is a super vertex or an inward ordinary vertex. Assume none of the super vertices in \( M \) have self-loops (as a result of preceding merge operations). It then holds that:

1. For every super vertex \( \varphi \) in \( G \), if its clockwise neighbor is a super vertex \( \psi \in M \), then \( \varphi \in M \) (\( \varphi \) is thus a leaf vertex of \( T \)); in which case innermost-cycle(\( \varphi \)) = innermost-cycle(\( \psi \)) = \( X \) and there are no ordinary vertices on \( X \) between \( \varphi \) and \( \psi \).
2. For every leaf vertex \( \mu_i \in M \), the set cycles(\( \mu_i \)) is a chain of nested cycles of consecutive levels, with level(\( X \)) being the highest level and \( X \) being its innermost cycle.
3. For all leaf vertices \( \mu_i, \mu_j \in M \), it holds that cycles(\( \mu_i \)) \subseteq cycles(\( \mu_j \)) or cycles(\( \mu_i \)) \supseteq cycles(\( \mu_j \)).

**Proof.** For Part 1, let innermost-cycle(\( \varphi \)) = \( Y \). If \( \psi \) is the clockwise neighbor of \( \varphi \), then \( Y \in \text{cycles}(\psi) \). If \( Y \neq X = \text{innermost-cycle}(\psi) \), then \( \varphi \) would be merged with \( \psi \) in a preceding round of \textit{merge} operations. Hence, if \( \varphi \) and \( \psi \) are distinct, it must be that \( Y = X \). And since \( \psi \) is \( \varphi \)'s clockwise neighbor, there are no intervening ordinary vertices on \( X \) between \( \varphi \) and \( \psi \).

We can prove Part 2 by induction on the number \( q \geq 1 \) of rounds of applying the \textit{collapse} operation on the initial simple three-regular plane graph. For \( q = 1 \), the desired conclusion is immediate, because all the leaf vertices of ICT’s that are eligible for collapse in the initial graph are ordinary vertices, and each of these ordinary vertices straddles exactly one cycle. For the induction hypothesis, we assume the conclusion holds for an arbitrary \( q \geq 1 \) rounds of \textit{collapses} interleaved with \( q \) rounds of \textit{merges}, and we next prove the conclusion after the \((q+1)\)-st round of \textit{collapses} is applied. The \( q \)-th round of \textit{merges}, immediately preceding the \((q+1)\)-st round of \textit{collapses}, contracts all self-loops of the same vertex.

Suppose the \((q+1)\)-st round of \textit{collapses} results in the contraction of an ICT \( T \) with leaf vertices \( M = \{ \mu_1, \ldots, \mu_p \} \) or \( M \cup \{ w \} = \{ \mu_1, \ldots, \mu_p \} \cup \{ w \} \), depending on whether \( T \) is type-a or type-b (see Remark in Section 4.3), with \( X \) being the innermost cycle of all the members of \( M \) and \( X' \) being the sole cycle of \( w \). The contraction of such an ICT \( T \) into a super vertex \( \chi \) generally creates self-loops of \( \chi \). The self-loops thus created, if any, are all edges of cycles in the chain of nested cycles formed by the members of \( \bigcup \{ \text{cycles}(\mu_i) \ | \ \mu_i \in M \} \). Suppose this chain of nested cycles is:

\[
X_1, \ldots, X_r, X_{r+1}, \ldots, X_{r+s}, X_{r+s+1}, \ldots, X_{r+s+t}
\]

with \( X = X_{r+s+t} \), which is listed in order of increasing consecutive levels and divided into three groups:

\[
\bullet \ \{ X_1, \ldots, X_r \} \text{ do not contribute self-loops of } \chi \text{ and are included in } \text{cycles}(\chi),
\]

\[1\]If innermost-cycle(\( \varphi \)) = \( X \), it does not necessarily follow that \( \varphi \in M \), unless \( \varphi \)'s clockwise neighbor is in \( M \) – as asserted here.

\[2\]In general, \( 0 \leq r \leq 1 \), while \( s \) and \( t \) can be arbitrarily large integers \( \geq 0 \). Moreover, the same cycle can contribute more than one self-loops of \( \chi \). This is provided by a finer analysis which we can ignore here.
\[ \{X_{r+1}, \ldots, X_{r+s}\} \text{ contribute self-loops of } \chi \text{ and are included in } \text{cycles}(\chi), \]
\[ \{X_{r+s+1}, \ldots, X_{r+s+t}\} \text{ contribute self-loops of } \chi \text{ and are not included in } \text{cycles}(\chi), \text{ because all their edges are turned into self-loops.} \]

After applying the \((q + 1)\)-st round of \textit{merges}, the self-loops of \(\chi\), all necessarily of levels \(\geq \text{level}(X')\), are eliminated by contraction. Let \(\chi'\) be \(\chi\) after the elimination of all self-loops. If \(T\) is a type-\(a\), then \(\text{cycles}(\chi') = \{X_1, \ldots, X_{r+s}\}\) and innermost-cycle(\(\chi') = X_{r+s}. \) If \(T\) is a type-\(b\), it is not difficult to check that level(\(X') = 1 + \text{level}(X_{r+s})\), with \(\text{cycles}(\chi') = \{X_1, \ldots, X_{r+s}\} \cup \{X'\}\) so that also innermost-cycle(\(\chi') = X'. \) This implies the conclusion of Part 2.

For Part 3, if \(\mu_i\) or \(\mu_j\) is an ordinary vertex, then the conclusion is immediate, because \(X\) is the sole cycle of all the ordinary vertices in \(M\) and \(X\) is also the innermost cycle of all the vertices in \(M\). Suppose next that neither \(\mu_i\) nor \(\mu_j\) are ordinary vertices and, by way of getting a contradiction, that cycles(\(\mu_i\) \(\subseteq\) cycles(\(\mu_j\)) and cycles(\(\mu_i\) \(\supseteq\) cycles(\(\mu_j\)). Hence, there are cycles \(Y_i \in \text{cycles}(\mu_i)\) and \(Y_j \in \text{cycles}(\mu_j)\) such that \(Y_i \notin \text{cycles}(\mu_j)\) and \(Y_j \notin \text{cycles}(\mu_i).\) But cycles \(Y_i\) and \(Y_j\) both enclose cycle \(X\), and each of cycles(\(\mu_i\)) and \(\text{cycles}(\mu_j)\) is a set of nested cycles of consecutive levels. Since cycles \(Y_i\) and \(Y_j\) cannot cross each other, it must be that either \(Y_i\) is nested in \(Y_j\) or \(Y_j\) is nested in \(Y_i\), i.e., \(Y_i \in \text{cycles}(\mu_j)\) or \(Y_j \in \text{cycles}(\mu_i)\) – but this is a contradiction. \(\square\)

**Lemma 18.** Let \(T\) be an ICT in \(G\) satisfying condition (6.2) all of whose leaf vertices, except possibly for one outward ordinary \(w\), are consecutive siblings on a cycle \(X\). Let \(M = \{\mu_1, \ldots, \mu_p\}\) be the consecutive siblings on \(X\), with \(p \geq 1\), each of which is a super vertex or an inward ordinary vertex. Let \(N = \{v_1, \ldots, v_q\}\) be the non-leaf vertices of \(T\), which are all ordinary of degree 3, with \(q \geq 0\). Assume none of the super vertices in \(M\) have self-loops (as a result of preceding merge operations).

It then holds that there is an algorithm \textit{COLLAPSE} which on input \(T\) returns in linear time \(O(p + q)\) a reassembling of \(T\), i.e., \(\text{COLLAPSE}(T) = \mathcal{B}\) where \(\mathcal{B}\) is a binary reassembling of the vertices in \(M \cup N \cup \{w\}\) (if \(T\) is type-\(a\)) or in \(M \cup N \cup \{w\}\) (if \(T\) is type-\(b\)) such that:\(^9\)

1. \(\alpha(T, \mathcal{B}) \leq 3.\)
2. \(\alpha(G, \mathcal{B}) \leq 1 + \max \{ \deg_G(\mu_i) \mid \mu_i \in M \}.\)
3. If \(\varphi\) is the super vertex resulting from contracting all the edges of \(T\), and \(\psi\) is the super vertex resulting from contracting all the self-loops of \(\varphi\), then

\[
\deg_G(\psi) \leq \begin{cases} 
\max \{ \deg_G(\mu_i) \mid \mu_i \in M \} & \text{if } T \text{ is type-}a, \\
2 + \max \{ \deg_G(\mu_i) \mid \mu_i \in M \} & \text{if } T \text{ is type-}b.
\end{cases}
\]

**Proof.** We first define a traversal of \(T\), which includes all the tree edges of \(T\) and excludes all the cycle edges connecting its leaf vertices \(\{\mu_1, \ldots, \mu_p\}\) or \(\{\mu_1, \ldots, \mu_p\} \cup \{w\}\), depending on whether \(T\) is type-\(a\) or type-\(b\), respectively. The traversal starts at any of the leaf vertices, say \(\mu_1\), and moves along the edge, call it \(e_1\), that connects \(\mu_1\) to a non-leaf vertex, say \(v_1\). From \(v_1\) (and from every subsequent non-leaf vertex), the traversal continues recursively by visiting the left subtree (first), then the right subtree (second), and then finally edge \(e_1\) in the reverse direction from \(v_1\) to \(\mu_1\) (third). It is easy to check that this traversal visits the starting leaf vertex \(\mu_1\) twice, every non-leaf vertex three times, and every other leaf vertex in \(\{\mu_2, \ldots, \mu_p\}\) or \(\{\mu_2, \ldots, \mu_p\} \cup \{w\}\) once, and can be carried out in time \(O(p + q)\). As the traversal proceeds recursively, it is useful to think that every non-leaf vertex \(v_j\) with \(1 \leq j \leq q\), which is first reached by traversing an edge, say \(e_j\), is the root of a binary tree whose right and left subtrees are the subtrees that can be aligned with edge \(e_j\) by a counterclockwise and a clockwise rotation around \(v_j\), respectively.

\(^9\)Algorithm \textit{COLLAPSE} is the function \textit{collapse_tree} in the pseudocode in Appendix A and in the full Python implementation downloadable from the website \textbf{Graph Reassembling}.  

24
Algorithm COLLAPSE carries out the traversal of $T$ just defined and simultaneously builds the vertex clusters of a binary reassembling $B$ as in the statement of the lemma. It builds a singleton cluster $X_i \equiv \{\mu_i\}$ the first time it visits leaf vertex $\mu_i$, which is also the only time for $i \in \{2, \ldots, p\}$. If $T$ is a type-$b$ and $w$ exists, it also builds a singleton cluster $X_{p+1} \equiv \{w\}$ the first and only time it visits leaf vertex $w$. For every non-leaf vertex $v_j$, algorithm COLLAPSE builds a cluster $X_j$ where $1 \leq j \leq q$ right after it visits $v_j$ the third and last time; the desired $X_j$ is defined as:

$$X_{p+j} \equiv \{v_j\} \cup Y \cup Z \quad \text{or} \quad X_{p+1+j} \equiv \{v_j\} \cup Y \cup Z,$$

depending on whether $T$ is type-$a$ or type-$b$, where $Y$ and $Z$ are the clusters of all the vertices (both leaf and non-leaf) of the right and left subtrees of $v_j$, respectively.

Part 1 of the lemma now readily follows from the preceding analysis. Part 2 is an easy consequence of Part 1. In Part 1, the degree of every leaf vertex is one because $T$ is considered in isolation from the rest of $G$; in Part 2, the degrees of the leaf vertices (also cycle vertices in $G$) are $\deg_G(\mu_1), \ldots, \deg_G(\mu_p)$ and $\deg_G(w) = 3$.

Part 3 is a straightforward consequence of Lemma 17. If $T$ is type-$b$, let $\text{cycles}(w) = X'$ which is necessarily $\neq X$, in which case $\text{level}(X') = 1 + \text{level}(X)$. Whether $T$ is type-$a$ or type-$b$, it is easy to see that:

$$\text{cycles}(\psi) = \text{cycles}(\varphi) - \{\text{all the self-loops of } \varphi\}$$

where $\text{cycles}(\varphi) = \bigcup \{\text{cycles}(\mu_i) | \mu_i \in M\}$ or $\text{cycles}(\varphi) = \{X'\} \cup \bigcup \{\text{cycles}(\mu_i) | \mu_i \in M\}$, depending on whether $T$ is type-$a$ or type-$b$, respectively.

For the next lemma, review the pre-processing phase and processing phase of algorithm KS in Section 4.1.

**Lemma 19.** The pre-processing phase and processing phase of KS on graph $G$ are each carried out in $O(n)$ steps, each using $O(n)$ space, where $n = |V(G)|$.

**Proof.** The initial input graph $G$ is represented by an adjacency list, where each vertex $v$ is identified by a pair of numbers ($v$’s coordinates in the Cartesian plane) together with a list containing the three vertices to which $v$ is connected. This requires $O(n)$ space. The pre-processing phase decomposes the input graph $G$ into ICT’s and cycles at each level of edge-outerplanarity, where every ICT can be identified by a postorder traversal (left, right, root); this whole pre-processing does not need to exceed $O(n)$ time and $O(n)$ space for its work.

In the processing phase, every ICT $T$ is collapsed in time linear in the size $|T|$; this is the result of COLLAPSE($T$) in Lemma 18 which, according to its proof, also uses a postorder traversal (left, right, root). All vertices (including all cycle vertices), whether ordinary or super, are each part of exactly one ICT, and every ICT is collapsed exactly once. Because any reassembling can be viewed as a tree containing $(2n - 1)$ nodes (review definitions of reassembling trees in Section 2), corresponding to $n$ initial ordinary vertices plus $(n - 1)$ super vertices produced in the course of KS’s operation, the entire processing phase also takes $O(n)$ time and $O(n)$ space. 

**Proof of Theorem 9.** Part 1 of Theorem 9 is an immediate consequence of Lemma 19. Lemma 16, Lemma 17, and Lemma 18, together imply Part 2 of Theorem 9. More precisely, by Lemma 16, algorithm KS terminates when there is only one super vertex $\varphi$ left to consider such that $V(\varphi) = V(G)$. By Lemma 17, if $\psi$ is one of the super vertices produced during KS’s execution at the end of a round of collapses (and prior to the following round of merges) with $\text{cycles}(\psi) = \{X_1, \ldots, X_\ell\}$, then the latter form a chain of nested cycles of consecutive levels, which implies that $\ell \leq E$-outerplanarity($G$) = $k$. By Lemma 18, each additional cycle in $\text{cycles}(\psi)$ contributes at most 2 to $\deg(\psi)$. Hence, for every super vertex $\chi$ produced at the end of a round of collapses during KS’s execution, it holds that $\deg(\chi) \leq 2k$. Hence, KS returns a reassembling $B$ such that $\alpha(G, B) \leq 2k$. 

25
4.7 Lifting the Restriction of Biconnectedness

We do not give the pseudocode, nor do we implement, the algorithm whose existence is asserted by the conclusion of the next corollary. We leave these to the interested reader.

Corollary 20. Identical to the statement of Theorem 9, except that $G$ is not required to be a biconnected graph.

An example of a 3-regular plane graph $G$ which is not biconnected is shown in Figure 3. It is reproduced on the left in Figure 16. On the right of the latter figure, there are 4 biconnected components, shown as $\{G_1, G_2, G_3, G_4\}$ such that $E$-outerplanarity$(G) = 4$ while $E$-outerplanarity$(G_i) \leq 3$ for $1 \leq i \leq 4$. This is a general fact, implicit in the proof of the corollary: the edge outerplanarity of every biconnected component is bounded by the edge outerplanarity of the full graph.

Proof Sketch. Identifying the biconnected components of a simple graph is a classical result, and the operation can be carried out in linear time (see the original [5] or any of the standard textbooks discussing graph algorithms, and also [4]). Let $KS'$ be the algorithm to be defined in order to satisfy the conclusion of this corollary.

Throughout this proof, biconnected means maximal biconnected and containing at least three vertices, ordinary or super. With no loss of generality, we can assume that the input graph $G$ is connected. Initially, all vertices are ordinary, but as algorithm $KS'$ is progressing, super vertices are created. A biconnected component has an outermost cycle consisting of all the edges that form the boundary of the component’s outerface. Initially, all vertices have degree $= 3$, but as algorithm $KS'$ starts executing, super vertices of degree $= 1$ (and other super vertices of arbitrary degrees) are created. Let the initial $G$ have $p \geq 2$ biconnected components, denoted:

$$G_1, G_2, \ldots, G_p.$$  

Each $G_i$ is such that $2 \leq E$-outerplanarity$(G_i) \leq E$-outerplanarity$(G)$.

$KS'$ calls algorithm $KS$ $p$ times. The reassembling of each $G_i$ is carried out separately, by applying $KS$ to it, which is thus turned into a single super vertex. The order in which $\{G_1, G_2, \ldots, G_p\}$ are reassembled is not arbitrary: The next $G_i$ selected for reassembling is innermost and with edge-boundary degree $\overline{\partial}(V(G_i)) = 1$:

- $G_i$ is innermost if none of its faces contains another $G_j$ with $i \neq j$ and/or (super) vertices of degree $= 1$; put differently, $G_i$ is innermost if $V(G_i)$ is the set of all the vertices, ordinary or super, which are on, or enclosed in, the outermost cycle of $G_i$.
- Among the biconnected components $\{G_1, G_2, \ldots, G_p\}$ there is always one $G_i$ such that $\overline{\partial}(V(G_i)) = 1$.

Because the initial $G$ is 3-regular, a biconnected component $G_i$ is connected to the rest of $G$ by a bridge $e = \mu\nu$ both of whose endpoints $\{\mu, \nu\}$ are articulation vertices. Suppose vertex $\nu \in V(G_i)$, so that $\mu \notin V(G_i)$. (We denote the endpoints of $e$ by the letters “$\mu$” and “$\nu$” because they may be super vertices as algorithm $KS'$ progresses in its execution.) Suppose also that $G_i$ is innermost. Applying algorithm $KS$ to $G_i$ produces a super vertex $\varphi$ of degree $= 1$ containing exactly all the vertices in $V(G_i)$. The initial edge $e = \mu\nu$ is transformed into the edge $e' = \mu\varphi$, and contracting $e'$ produces a super vertex $\varphi'$ of degree $= 2$ containing all the vertices in $\{\mu\} \cup V(G_i)$. There are now two edges $\xi_1 \varphi'$ and $\xi_2 \varphi'$ for some distinct vertices $\xi_1$ and $\xi_2$, which may be ordinary or super.

Before proceeding to select the next innermost biconnected component $G_j$, algorithm $KS'$ contracts one of the two edges, $\xi_1 \varphi'$ or $\xi_2 \varphi'$, to produce a new super vertex $\varphi''$ of degree $= 3$.  

5 Conditions for the Optimality of Algorithm KS

We show that for a family of 3-regular plane graphs with a sufficiently high “inter-cycle density” (density of inter-cycle trees), algorithm KS returns $\alpha$-optimal reassemblings (Theorem 35). For the same family of graphs with a low “inter-cycle density”, KS does not return $\alpha$-optimal reassemblings (Proposition 36). What the informal expressions “high density” and “low density” mean is made precise right after Theorem 35.

Let $f$ be a monotonically increasing or constant function on the natural numbers such that $f(x) \geq 3$ for all $x$. We define an infinite family $H_f$ of 3-regular plane graphs parametrized with $f$. Each member of $H_f$ is assigned a second parameter $k$, a natural number $\geq 2$:

$$H_f \triangleq \{ H_{f,k} \mid k \geq 2 \}$$

such that $E$-outerplanarity($H_{f,k}$) = $k$. Until the proof of Lemma 32, we do not need to be specific about the function $f$, only $k$ needs to be operated on; until then, however, it is useful to keep in mind that we will choose $f$ so that $f(k) \gg k$. The graph $H_{f,k}$ is shown in Figure 17 when $k = 4$ and $f(x) = 2x - 1$, so that $f(k) = 7$.

$H_{f,k}$ consists of $k$ concentric cycles $\{ C_0, \ldots, C_{k-1} \}$ such that cycle $C_i$ and $C_{i+1}$ are connected by $f(k)$ one-edge ICT’s, henceforth called ICE’s in this section (ICE = inter-cycle edge). 10 By our earlier conventions,

$$\text{level}(C_0) = 0, \quad \text{level}(C_1) = 1, \quad \ldots \quad \text{level}(C_{k-1}) = k - 1$$

For convenience, we use a double-indexing to denote the ICE’s. The ICE $e_{i,j}$ is an edge whose first index $i$ denotes its level and its second index $j$ ranges over the set $\{1, \ldots, f(k)\}$. Moreover, all the level-$i$ ICE’s occur in the following order: $e_{i,1}, e_{i,2}, \ldots, e_{i,f(k)}$ in a clockwise direction, for every $1 \leq i \leq k - 1$.

For every $1 \leq i \leq k - 1$, we identify the two endpoints of ICE $e_{i,j}$ by the vertices $x_{i,j}$ and $y_{i,j}$, such that $x_{i,j} \in V(C_{i-1})$ and $y_{i,j} \in V(C_i)$. The vertices and edges of cycle $C_0$ are therefore:

$$V(C_0) = \{ x_{1,1}, x_{1,2}, \ldots, x_{1,f(k)} \}$$

$$E(C_0) = \{ x_{1,1} \ x_{1,2}, \ x_{1,2} \ x_{1,3}, \ldots, x_{1,f(k)-1} \ x_{1,f(k)}, \ x_{1,f(k)} \ x_{1,1} \}$$

10Note carefully that “$C_i$” here is unrelated to the standard notation “$C_i$” which refers to the cycle graph with $i$ vertices.
The vertices and edges of cycle $C_i$, for $1 \leq i \leq k - 2$, are:

$$V(C_i) = \{ y_{i,1}, x_{i+1,1}, y_{i,2}, x_{i+1,2}, \ldots, y_{i,f(k)}, x_{i+1,f(k)} \}$$

$$E(C_i) = \{ y_{i,1} \ x_{i+1,1}, y_{i,2} \ x_{i+1,2}, \ldots, y_{i,f(k)} \ x_{i+1,f(k)}, x_{i+1,f(k)} \ y_{i,1} \}$$

The vertices and edges of cycle $C_{k-1}$ are:

$$V(C_{k-1}) = \{ y_{k-1,1}, y_{k-1,2}, \ldots, y_{k-1,f(k)} \}$$

$$E(C_{k-1}) = \{ y_{k-1,1} \ y_{k-1,2}, y_{k-1,2} \ y_{k-1,3}, \ldots, y_{k-1,f(k)-1} \ y_{k-1,f(k)}, y_{k-1,f(k)} \ y_{k-1,1} \}$$

Figure 17 shows the graph $H_{f,k}$ and the naming conventions of edges and vertices, when $k = 4$ and $f(k) = 7$. The sequence of definitions and lemmas, from Definition 21 to Lemma 31, is to show that, if we want to maximize the size $|X|$ of a vertex cluster $X \subseteq V(H_{f,k})$ whose edge-boundary degree $\partial(X)$ is a fixed constant $c$ strictly less than $2k$, then we can restrict attention to clusters that we call strongly regular (Definition 28).

**Definition 21 (Clusters, Holes in Clusters, Full Clusters).** A non-empty subset $X \subseteq V(H_{f,k})$ is connected iff between any two vertices of $X$ there is a path. A cluster in graph $H_{f,k}$ is a non-empty connected subset $X \subseteq V(H_{f,k})$.

Let $X, Y \subseteq V(H_{f,k})$ be clusters in graph $H_{f,k}$. We say $Y$ is a hole in $X$, or that $X$ contains the hole $Y$, iff all of the following conditions are satisfied:

1. $X \cap Y = \emptyset$, i.e., $X$ and $Y$ are disjoint.
2. $\partial(X, Y) \neq \emptyset$, i.e., there are edges with one endpoint in $X$ and one endpoint in $Y$.
3. $\partial(V(H_{f,k}) - (X \cup Y), Y) = \emptyset$, i.e., there is no edge with one endpoint in $V(H_{f,k}) - (X \cup Y)$ and one endpoint in $Y$.
We say a cluster $X \subseteq V(H_{f,k})$ is full iff $X$ contains no holes. An example of a cluster $X$ with a hole $Y$ is shown in Figure 18, and a full cluster is $X \cup Y$.

\[\text{Figure 18: Example for Definition 21: Graph } H_{f,k}\text{ is the same as in Figure 17. The black vertices form a cluster } X \text{ in } H_{f,k}, \text{ the red vertices form a hole } Y \text{ in } X. \text{ The union } Z = X \cup Y \text{ is a full cluster.}\]

\[\text{Lemma 22. Let } X \subseteq V(H_{f,k}) \text{ be a cluster in } H_{f,k}. \text{ If } X \text{ contains holes, then there is a cluster } Z \subseteq V(H_{f,k}) \text{ without holes such that:}\]

1. $\partial(Z) \leq \partial(X)$ and $Z \supseteq X$.
2. For every full cluster $Z' \supseteq X$, it holds that $|Z| \leq |Z'|$.

In words, we can minimally augment $X$ and eliminate all the holes in it without increasing $\partial(X)$.

\[\text{Proof. Full clusters exist, with } V(H_{f,k}) \text{ being the largest full cluster. The desired } Z \text{ in the lemma statement is the smallest full cluster such that } Z \supseteq X. \text{ It is straightforward to see that } \partial(X) \geq \partial(Z), \text{ since each hole in } X \text{ delete some vertices from } X \text{ and increases } \partial(X). \]

The situation described in the statement of Lemma 22 is illustrated in Figure 18: For the full cluster $Z = X \cup Y$, we have $8 = \partial(Z) \leq \partial(X) = 12$.

\[\text{Definition 23 (Frontiers of Full Clusters). Other than its outermost face and its innermost face, every other face of } H_{f,k}\text{ is bounded by 5 or 6 edges. Call the outermost and innermost faces the large faces of } H_{f,k}\text{ (there are only two of them), and all the other faces the small faces of } H_{f,k}\text{ (there are } (k - 1) \cdot f(k) \text{ of them).}\]

Let $X \subseteq V(H_{f,k})$ be a full cluster in graph $H_{f,k}$. Every face of $H_{f,k}$ is either inside or outside $X$. We say a face of $H_{f,k}$ is inside (resp. outside) $X$ iff all of the vertices (resp. one or more of the vertices) on the bounding edges of the face are in $X$ (resp. not in $X$). The frontier of $X$ is a set of edges defined as follows:

\[\text{frontier}(X) \triangleq \{ e \in E(H_{f,k}) \mid e = xy \text{ bounds a face outside } X \text{ and } \{x, y\} \subseteq X \}.\]

Note that if $e \in \text{frontier}(X)$, it does not necessarily follow that $e$ bounds a face inside $X$, even though the two endpoints of $e$ are in $X$; this is illustrated in Figure 19. And if $X$ is a face of $H_{f,k}$, large or small, then $\text{frontier}(X)$ coincides with the boundary of $X$.

\[\text{11 A stronger conclusion in fact holds: The two inequalities } \geq \text{ and } \leq \text{ can be changed to strict inequalities } > \text{ and } <. \text{ We do not need the stronger conclusion.}\]
Figure 19: Example for Definitions 23 and 25: Graph $H_{f,k}$ is the same as in Figure 17. The black vertices form a full cluster $X$ in $H_{f,k}$. The faces in light gray are inside $X$; all the other faces are outside $X$. The edges in boldface form $\text{frontier}(X)$; not all edges in $\text{frontier}(X)$ bound a face inside $X$ (the dashed boldface edges).

Lemma 24. If $X \subseteq V(H_{f,k})$ be a full cluster in $H_{f,k}$, then the set of vertices $V(\text{frontier}(X))$ forms a cluster, i.e., it is a non-empty connected subset of $V(H_{f,k})$.

It is worth noting that, unless the vertices of the outermost cycle $C_0$ and/or the vertices of the innermost cycle $C_{k-1}$ are all in $X$, the vertices in $V(\text{frontier}(X))$ form a connected cactus.

Proof. Straightforward from the definition. Details omitted.

Definition 25 (Cut Edges, Dangling Edges, and Regular Clusters). Let $X \subseteq V(H_{f,k})$ be a full cluster in graph $H_{f,k}$, and consider $\text{frontier}(X)$. If $e \in \text{frontier}(X)$ does not bound a face inside $X$, then $e$ is one of two kinds:

- $e$ is a cut edge of $\text{frontier}(X)$,
- $e$ is a dangling edge of $\text{frontier}(X)$.

If the deletion of $e \in \text{frontier}(X)$ disconnects $\text{frontier}(X)$ into two components, then $e$ is a cut edge; otherwise, $e$ is a dangling edge. See Figure 19 for an illustration: it shows one dangling edge and three cut edges.

If $X$ is a full cluster and $\text{frontier}(X)$ contains no cut edges and no dangling edges, then we say $X$ is a regular cluster; this is illustrated in Figure 20. Observe that a regular cluster is constructed from ‘piling on top of each other’ 5-edge faces and 6-edge faces. The 5-edge faces are those adjacent to the outer face (bounded by $C_0$) and the innermost face (bounded by $C_{k-1}$).
**Lemma 26.** Let $X \subseteq V(H_{f,k})$ be a full cluster with $|X| \geq 3$. If $\text{frontier}(X)$ contains cut edges and/or dangling edges, then there is a regular cluster $Z \subseteq V(H_{f,k})$ such that:

1. $\partial(Z) \leq \partial(X)$ and $|Z| \geq |X|$.
2. For every regular cluster $Y$ such that $|Y| \geq |X|$, it holds that $|Z| \leq |Y|$.

In words, we can minimally augment the size $|X|$ and eliminate all cut edges and dangling edges from $\text{frontier}(X)$ without increasing $\partial(X)$.

**Proof.** The proof is an exhaustive case analysis. We proceed repeatedly to eliminate cut edges and dangling edges, one by one. There is no loss of generality in assuming that $X$ satisfies one of two conditions (or both):

1. $X \cap V(C_0) \neq \emptyset$ and $X \cap V(C_1) \neq \emptyset$,
2. $X \cap V(C_{k-2}) \neq \emptyset$ and $X \cap V(C_{k-1}) \neq \emptyset$.

In words, $X$ overlaps with the two outermost cycles (condition 1) and/or the two innermost cycles of $H_{f,k}$ (condition 2).

Consider a particular $e_0 \in \text{frontier}(X)$ which is a cut edge or a dangling edge. Keep in mind that the two large faces of $H_{f,k}$ do not have a common boundary, that every small face of $H_{f,k}$ is bounded by 5 or 6 edges, and that every edge bounds exactly two faces.

Because $e_0 \in \text{frontier}(X)$, there is a small face $F$ of $H_{f,k}$ outside $X$ which is bounded by $e_0$. Because $\text{frontier}(X)$ is connected, we can choose the face $F$ to be bounded by at least two edges of $\text{frontier}(X)$, say $e_1$ in addition to $e_0$. Let the set of edges bounding the small face $F$ be $\{e_0, e_1, e_2, e_3, e_4\}$ or $\{e_0, e_1, e_2, e_3, e_4, e_5\}$.

In all cases satisfying one of the two following conditions:

3. $F$ is bounded by 5 edges, with at least two of them $\{e_0, e_1\} \subseteq \text{frontier}(X)$,
4. $F$ is bounded by 6 edges, with at least three of them $\{e_0, e_1, e_2\} \subseteq \text{frontier}(X)$,

it is straightforward to add all the vertices of $V(F)$ to $X$, thus increasing $|X|$ and making $F$ a face inside $X$, without increasing $\partial(X)$. All details of this straightforward case analysis are omitted.

The remaining cases satisfy the following condition:

5. $F$ is bounded by 6 edges, with exactly two of them $\{e_0, e_1\} \subseteq \text{frontier}(X)$.

---

Note that we augment the size $|X|$ not $X$ itself, i.e., it is not necessarily that $Z \supseteq X$, but only that $|Z| \geq |X|$.
Because $F$ is bounded by 6 edges, $F$ is not adjacent to the outer face (bounded by cycle $C_0$) nor to the innermost face (bounded by cycle $C_{k-1}$).

We eliminate cut edges of $\text{frontier}(X)$ first, by starting from cut edges of lowest level (those that are closest to cycle $C_0$), and we then proceed inward until we reach cut edges of highest level (closest to cycle $C_{k-1}$). In this order, it is easy to see that we only need to eliminate cut edges that satisfy condition 3 of condition 4 above.

We are left with the case when there are only dangling edges and condition 5 is satisfied. Let therefore $e_0$ be a dangling edge of $\text{frontier}(X)$, which implies that $F$ is a 6-edge face outside $X$ which shares $e_1$ as a bounding edge with another face inside $X$.

Let $e_0 = \overrightarrow{v_0 v_1}$ and $e_1 = \overrightarrow{v_1 v_2}$. Let $Y = X - \{v_0\}$, so that $\overrightarrow{\partial(Y)} = \overrightarrow{\partial(X)} - 1$ and $|Y| = |X| - 1$. Then $Y$ is a full cluster, such that $\text{frontier}(Y)$ contains no cut edges and one dangling edge less than $\text{frontier}(X)$. The desired full cluster $Z$ in the conclusion of the lemma is obtained by appropriately adding two or more vertices to $Y$ and by increasing $\overrightarrow{\partial(Y)}$ by at most 1.

Cluster $Y$ satisfies condition 1 or condition 2. Assume $Y$ satisfies condition 1. This implies there is a face $F'$ with 5 bounding edges, say $\{e_0', e_1', e_2', e_3', e_4'\}$, such that edge $e_0'$ is an ICE in $\text{frontier}(Y)$, edge $e_1'$ is an edge of $C_0$ and also in $\text{frontier}(Y)$, edge $e_2'$ is an ICE which may or may not be in $\text{frontier}(Y)$, and $\{e_3', e_4'\}$ are consecutive edges of cycle $C_1$ which may or may not be in $\text{frontier}(Y)$. It is now easy to see that we can add two (or more) vertices to $Y$ such that: (i) $\overrightarrow{\partial(Y)}$ is increased by at most 1 and (ii) no cut edge and dangling edge are added to $\text{frontier}(Y)$. The resulting cluster is the desired $Z$.

\begin{definition}[\(\prec\)-Sequences] A maximal \(\prec\)-sequence $S$ of ICE’s in the graph $H_{f,k}$ is a sequence of the form:
\[ S \triangleq e_{1,j} \prec \cdots \prec e_{k-1,j} \]
where $j \in \{1, \ldots, f(k)\}$. A maximal \(\preceq\)-sequence $S$ of ICE’s in the graph $H_{f,k}$ is a sequence of the form:
\[ S \triangleq e_{1,j_1} \preceq \cdots \preceq e_{k-1,j_{k-1}} \]
where $j_1 \in \{1, \ldots, f(k)\}$ and:
\[ j_2 = 1 + ((j_1 - 2) \mod f(k)), \]
\[ j_3 = 1 + ((j_1 - 3) \mod f(k)), \]
\[ \ldots \]
\[ j_{k-1} = 1 + ((j_1 - k + 1) \mod f(k)). \]
A maximal \(\preceq\)-sequence $S$ of ICE’s in the graph $H_{f,k}$ is a sequence of the form:
\[ S \triangleq e_{1,j_1} \preceq \cdots \preceq e_{k-1,j_{k-1}} \]
where every pair $e_{p,j_p} \preceq e_{p+1,j_{p+1}}$ in the sequence, with $1 \leq p \leq k-1$, is:
\[ \text{either } e_{p,j_p} \preceq e_{p+1,j_{p+1}} \text{ or } e_{p,j_p} \preceq e_{p+1,j_{p+1}}, \]
\text{i.e., in a maximal }\prec\text{-sequence it does not matter whether }e_{p+1,j_{p+1}}\text{ is a clockwise successor or counter-clockwise successor of }e_{p,j_p},\text{ as we traverse the sequence from the outermost cycle to the innermost cycle.} \]
Figure 21: Four possible shapes of maximal $\prec$-sequences in graph $H_{f,k}$ when $k = 4$ and $f(k) = 2 \cdot k = 8$ (slightly different from the graph in Figure 17). The leftmost above is a maximal $\prec^r$-sequence, the rightmost is a maximal $\prec^\ell$-sequence, and the two in the middle are neither maximal $\prec^r$ nor maximal $\prec^\ell$.

A $\prec^\ell$-sequence is a subsequence of a maximal $\prec^\ell$-sequence, i.e., the former is obtained by omitting a prefix and/or a suffix from the latter.

Similarly, a $\prec^r$-sequence is a subsequence of a maximal $\prec^r$-sequence, and a $\prec$-sequence is a subsequence of a maximal $\prec$-sequence.

Definition 28 (Strongly Regular Clusters). Let $X \subseteq V(H_{f,k})$ be a regular cluster with $|X| \geq 3$. We say $X$ is a strongly regular cluster bounded by the outermost cycle $C_0$ iff $\text{frontier}(X)$ can be partitioned into 4 disjoint subsets:

- a subsequence of $n$ consecutive edges in $C_0$,
- a prefix of $p$ ICE’s in a maximal $\prec^r$-sequence together with their $(p - 1)$ intermediate cycle edges,
- a subsequence of $q$ consecutive edges in $C_p$,
- a prefix of $p$ ICE’s in a maximal $\prec^\ell$-sequence together with their $(p - 1)$ intermediate cycle edges,

for some $n, p, q \geq 1$. Symmetrically, we say $X$ is a strongly regular cluster bounded by the innermost cycle $C_{k-1}$ iff $\text{frontier}(X)$ can be partitioned into 4 disjoint subsets:

- a subsequence of $n$ consecutive edges in $C_{k-1}$,
- a suffix of $p$ ICE’s in a maximal $\prec^r$-sequence together with their $(p - 1)$ intermediate cycle edges,
- a subsequence of $q$ consecutive edges in $C_{k-1-p}$,
- a suffix of $p$ ICE’s in a maximal $\prec^\ell$-sequence together with their $(p - 1)$ intermediate cycle edges,

for some $n, p, q \geq 1$. We call $n, p$, and $q$ the parameters of the strongly regular cluster $X$, the values of which are not totally arbitrary, as stated in the next lemma; $p$ is the height of $X$, and $n$ and $q$ its bases. Figure 22 shows an example of a strongly regular cluster bounded by the outermost cycle $C_0$ and Figure 23 shows an example of a strongly regular cluster bounded by the innermost cycle $C_{k-1}$.
Lemma 29. If $X$ is a strongly regular cluster in $H_{f,k}$ with parameters $n$, $p$, and $q$, then:

1. $1 \leq n \leq f(k) - 1$ and $1 \leq p \leq k - 1$.
2. If $p < k - 1$, then $q = 2i$ for some $1 \leq i \leq f(k) - p$,
   \[ p + (q/2) = n + 1, \quad \partial(X) = 2(p + 1) + (q/2), \text{ and } |X| = p^2 + pq + p + (q/2). \]
3. If $p = k - 1$, then $q = i$ for some $1 \leq i \leq f(k) - p$,
   \[ p + q = n + 1, \quad \partial(X) = 2k, \text{ and } |X| = p^2 + 2pq + p. \]

Proof. Let $X$ be a strongly regular cluster bounded by the outermost cycle $C_0$. (The argument applies again symmetrically if $X$ is bounded by the innermost cycle $C_{k-1}$.)

For part 1, note that $n$ is restricted to be at most $f(k) - 1$, it cannot be $f(k)$, because Definition 28 requires that the two bounding $\prec$-sequences be disjoint. Parts 2 and 3 follow from straightforward calculations (all details omitted).
Lemma 30. Let \( X \subseteq V(H_{f,k}) \) be a regular cluster with \( f(k) \gg k \). Let \( N = |V(H_{f,k})| \), the total number of vertices in \( H_{f,k} \). If \( |X| < N/2 \) and \( \overline{d}(X) \leq 2k - 1 \), then there is a strongly regular cluster \( Z \subseteq V(H_{f,k}) \) such that:\(^{13}\)

\[
\overline{d}(Z) \leq \overline{d}(X) \quad \text{and} \quad |X| \leq |Z| < N/2.
\]

Proof. There is no loss of generality in assuming that \( X \) satisfies condition 1 or condition 2 in the proof of Lemma 26. Since \( \overline{d}(X) \leq 2k - 1 \), it is only one of these two conditions that \( X \) can satisfy, not both. We assume that \( X \) satisfies condition 2 and thus view that \( X \) has one or more consecutive 5-edge faces at the bottom (all adjacent to the innermost face), on top of which there are 6-edge faces, piled upward no higher than cycle \( C_1 \) in order not to violate the restriction \( \overline{d}(X) \leq 2k - 1 \). Moreover, because \( |X| < N/2 \) and \( \overline{d}(X) \leq 2k - 1 \), not all the vertices of the innermost cycle \( C_{k-1} \) are in \( X \), and because \( X \) is full, there are paths connecting vertices in \( C_0 \) and \( C_{k-1} \) whose vertices are all not in \( X \).

Let \( h \geq 1 \) be the height of \( X \), which is necessarily \( \leq k - 2 \), otherwise we would have \( \overline{d}(X) \geq 2k \) contradicting the hypothesis. There are two kinds of edges connecting \( X \) to \( (V(H_{f,k}) - X) \) contributing to the total in \( \overline{d}(X) \): ICE’s and cycle edges. Consider all the ICE’s in the set \( \partial(X) \) at height \( h \), say, these are \( e_1, e_2, \ldots, e_r \) for some \( r \geq 1 \). These ICE’s are not necessarily consecutive, as there may be “dips” along \( \text{ frontier}(X) \).

The construction of the desired cluster \( Z \) in the lemma conclusion proceeds in two stages. First, we remove all the “dips” between the top-level ICE’s \( e_1, e_2, \ldots, e_r \) in \( \partial(X) \), to obtain an intermediate full cluster \( X' \) of the same height \( h \) and where all the top-level ICE’s \( e_1', e_2', \ldots, e_s' \) in \( \partial(X') \) are now consecutive and such that \( \overline{d}(X) \geq \overline{d}(X') \) and \( |X| \leq |X'| \). By a ‘top-level ICE’ in \( \partial(X) \) we mean an ICE which is closest to the outermost cycle \( C_0 \). Note that

\[
2k - 1 \geq \overline{d}(X) \geq \overline{d}(X') \geq 2(h + 1) + s
\]

In the last term, \( 2(h + 1) \) is the number of cycle edges in \( \partial(X') \) and \( s \) is the number of ICE’s in \( \partial(X') \) at height \( h \). In general, there may be other ICE’s in \( \partial(X') \) at heights lower than \( h \).

Second, we obtain the desired regular cluster \( Z \) by setting its parameters \( p \) and \( q \), as specified in Definition 28, and by Lemma 29, we do not need to also set its parameter \( n \). We set \( p = h + q = 2s \). By part 2 of Lemma 29, we have \( \overline{d}(Z) = 2(h + 1) + s \) and also \( |Z| \leq |X'| \). \( \square \)

Lemma 31. Let \( X \subseteq V(H_{f,k}) \) be a strongly regular cluster with parameters \( n, p, q \), as specified in Definition 28. If \( \overline{d}(X) < 2k \), then \( |X| \leq \lceil (16k^2 - 32k + 13)/12 \rceil \).

Proof. By part 3 of Lemma 29, if \( \overline{d}(X) < 2k \) and therefore \( \overline{d}(X) \neq 2k \), then \( p \neq k - 1 \). Because \( X \) is strongly regular, it follows that \( p < k - 1 \), by part 1 of Lemma 29. Hence, by part 2 of Lemma 29, we have:

\[
|X| = p^2 + pq + p + (q/2) \quad \text{and} \quad \overline{d}(X) = 2(p + 1) + (q/2).
\]

Let \( c \) be a constant such that \( \overline{d}(X) = c \) which we keep fixed throughout the proof. Relative to \( c \), we determine how to set the values of the parameters \( p \) and \( q \) in order to maximize \( |X| \). We first express \( p \) in terms of \( q \) and \( c \), namely, if \( 2(p + 1) + (q/2) = c \) then:

\[
p = \frac{2c - q - 4}{4}
\]

\(^{13}\)We do not need the restriction \( |X| \geq 3 \) here, which is part of the hypothesis in Lemma 26, because if there are no cut edges and no dangling edges, it follows that \( |X| \geq 3 \) — in fact, that \( |X| \geq 5 \).
which we substitute in \( |X| = p^2 + pq + p + (q/2) \) to obtain a function \( f_c(q) \) depending on \( q \) as a variable and \( c \) as a constant:

\[
f_c(q) \triangleq p \cdot (p + q + 1) + \frac{q}{2} = \left(\frac{2c - q - 4}{4}\right) \cdot \left(\frac{2c - q - 4}{4} + q + 1\right) + \frac{q}{2} = \frac{(2c - q - 4) \cdot (2c + 3q) + 8q}{16} = -(3/16)q^2 + (1/4)(c - 1)q + (1/4)(c^2 - 2c)
\]

where the last expression is obtained from the preceding one by straightforward calculations. In the argument to follow, even though possible values of \( c = 2(p + 1) + (q/2) \) are integers \( \geq 5 \), we deal with \( q \) and \( c \) as real numbers and we use the derivative of \( f_c(q) \) relative to \( q \) as such.

The function of \( f_c(q) \) defines a parabola which is maximized at its vertex, \( i.e. \), at the value of \( q \) for which the derivative:

\[
f_c'(q) \triangleq \frac{df}{dq} = -(6/16)q + (1/4)(c - 1)
\]

is zero. Hence, \( f_c(q) \) as a function over the reals is maximized at \( \hat{q} \) given by:

\[
\hat{q} \triangleq (2/3)(c - 1).
\]

After substituting \( \hat{q} \) in \( f_c(q) \) and carrying out straightforward calculations, we obtain:

\[
f_c(\hat{q}) = \frac{4c^2 - 8c + 1}{12}
\]

It is easily checked that \( f_c(\hat{q}) \) is monotonically increasing for all \( c \geq 1 \). By the lemma hypothesis, \( c \) cannot exceed \( 2k - 1 \geq 1 \). Substituting \( 2k - 1 \) for \( c \) in \( f_c(\hat{q}) \), we obtain after simple calculations:

\[
f_{2k-1}(\hat{q}) = \frac{16k^2 - 32k + 13}{12}.
\]

We conclude that \( |X| \) is an integer which cannot exceed the real number \( (16k^2 - 32k + 13)/12 \).

In the graph \( H_{f,k} \) there are \( k \) concentric cycles, such that two consecutive cycles are connected by \( f(k) \) ICE’s. There is therefore a total of \( (k - 1) \cdot f(k) \) ICE’s in \( H_{f,k} \), and since each ICE contributes two vertices, a total of \( 2(k - 1) \cdot f(k) \) vertices. Hence, \( |V(H_{f,k})| = 2(k - 1) \cdot f(k) \).

**Lemma 32.** Consider the family \( \mathcal{H}_f = \{H_{f,k}\} \) where \( k \geq 2 \) and \( f \) is any function such that the inequality:

\[
2(k - 1) \cdot f(k) > 4 \cdot \left(\frac{16k^2 - 32k + 13}{12}\right) = \frac{16k^2 - 32k + 13}{3}
\]

is satisfied. Given any \( H_{f,k} \in \mathcal{H}_f \) there is no binary reassembling \( \mathcal{B} \) of \( H_{f,k} \) such that \( \alpha(H_{f,k}, \mathcal{B}) < 2k \).

**Proof.** If \( \mathcal{B} \) is a binary reassembling of \( H_{f,k} \) such that \( \alpha(H_{f,k}, \mathcal{B}) < 2k \), then for every node/cluster \( X \in \mathcal{B} \) it holds that \( \overline{\mathcal{D}}(X) < 2k \). Let \( N = |V(H_{f,k})| \). To prove that no such \( \mathcal{B} \) can exist, it suffices to show that for every cluster \( X \in \mathcal{B} \), it must be that \( |X| < N/2 \), \( i.e. \), \( X \) contains fewer than half the vertices of \( H_{f,k} \), contradicting that \( \mathcal{B} \) is a binary reassembling of \( H_{f,k} \).
Given any two disjoint clusters \( X_1, X_2 \in \mathcal{B} \) such that \( \overline{d}(X_i) < 2k \) and \( |X_i| < N/2 \), with \( i = 1, 2 \), we show that if \( \overline{d}(X_1 \cup X_2) < 2k \), then again \( |X_1 \cup X_2| < N/2 \). By Lemma 31, every strongly regular cluster \( X \) with \( \overline{d}(X) < 2k \) cannot include more than:
\[
\frac{16k^2 - 32k + 13}{12}
\]
vertices. Hence, by Lemmas 22, 24, 26, 29, and 30, any cluster \( X \), strongly regular or not, with \( |X| < N/2 \) and \( \overline{d}(X) < 2k \) cannot include more than \( (16k^2 - 32k + 13)/12 \) vertices. Hence,
\[
|X_1 \cup X_2| < \frac{16k^2 - 32k + 13}{6} < \frac{N}{2}
\]
The desired conclusion follows.

\[\square\]

**Remark 33.** The inequality in the hypothesis of Lemma 32 can be simplified as follows. First, observe that:
\[
(k - 1) \cdot (16k - 13) = 16k^2 - 29k + 13 > 16k^2 - 32k + 13
\]
for all \( k \geq 2 \). Hence, if \( f \) satisfies the inequality:
\[
2(k - 1) \cdot f(k) > \frac{(k - 1) \cdot (16k - 13)}{3} > \frac{16k^2 - 32k + 13}{3}
\]
then the hypothesis of Lemma 32 is satisfied. Hence, it suffices to require that \( f \) is any number-theoretic function such that \( f(k) > (16k - 13)/6 \).

\[\square\]

**Lemma 34.** For every graph \( H_{f,k} \in \mathcal{H}_f \) where \( f \) is any monotonically increasing or constant function \( \geq 3 \), algorithm \( \text{KS} \) on input \( H_{f,k} \) returns a binary reassembling \( \mathcal{B} = \text{KS}(H_{f,k}) \) such that \( \alpha(H_{f,k}, \mathcal{B}) = 2k \).

**Proof.** The function \( f \) plays no role in this proof. Assuming that \( f(k) \geq 3 \) for all \( k \), it is easy to see that the binary reassembling \( \mathcal{B} = \text{KS}(H_{f,k}) \) returned by algorithm \( \text{KS} \) is such that \( \alpha(H_{f,k}, \mathcal{B}) = 2k \).

\[\square\]

**Theorem 35** (Optimality of Algorithm \( \text{KS} \)). For every graph in the family \( \mathcal{H}_f \) where \( f(k) > (16k - 13)/6 \), algorithm \( \text{KS} \) is an \( \alpha \)-optimal reassembling algorithm.

**Proof.** Immediate consequence of Lemma 32, Remark 33, and Lemma 34.

\[\square\]

For the preceding theorem to hold, the number of ICE’s must increase as the edge-outerplanarity \( k \) increases – this is the “density condition” mentioned in the opening paragraph of Section 5 – otherwise our algorithm \( \text{KS} \) is not an \( \alpha \)-optimal reassembling algorithm as we explain next.

We define a family of 3-regular plane graphs, called \( \mathcal{G}_\ell \) where \( \ell \geq 3 \), which exhibits a low “density of ICT’s” (or “inter-cycle density”) in the sense that their number is bounded by \( \ell \) at every level. For every graph \( G \in \mathcal{G}_\ell \):

- there is no restriction on \( n = |V(G)| \) or \( k = E\text{-outerplanarity}(G) \), and both can be unboundedly large,
- at every level of \( G \), the number of ICT’s is \( \leq \ell \).

The second bullet point formalizes what we mean by low “inter-cycle density”. For the definition of level in 3-regular plane graphs, review Definition 7.

It is possible to show (not in this report) that there is linear-time reassembling algorithm which, given an arbitrary \( G \in \mathcal{G}_\ell \), returns an \( \alpha \)-optimal reassembling \( (G, \mathcal{B}) \) such that \( \alpha(G, \mathcal{B}) \in \mathcal{O}(\ell) \). A consequence of this fact is that our algorithm \( \text{KS} \) is not \( \alpha \)-optimal for the family \( \mathcal{G}_\ell \). In the next proposition, we prove a particular
case: We consider the family $H_f$ as defined in the opening paragraphs of Section 5, where every ICT is an inter-cycle edge (ICE), and choose a particular function $f$, called $\bar{f}$, which makes $H_\bar{f}$ a proper subclass of $G_\ell$ for sufficiently large $\ell$, i.e., $H_\bar{f} \subseteq G_\ell$. The $f$ in question is a constant function, specifically here: $f(k) = c$ for some constant $c \geq 3$ which makes $H_f \subseteq G_c$. Thus, for every $H_{f,k} \in H_f$, the number of inter-cycle one-edge trees at every level of $H_{f,k}$ is $c$ and independent of $k$.

**Proposition 36.** Let $c$ be a constant $\geq 3$. There is linear-time $\alpha$-optimal reassembling algorithm $A$ for the family $H_f$ where $\bar{f}(k) = c$. Specifically, given an arbitrary $H_{f,k} \in H_f$, algorithm $A$ returns in time $O(|V(H_{f,k})|)$ an $\alpha$-optimal reassembling $B$ of $H_{f,k}$ such that $\alpha(H_{f,k}, B) = c + 2$.

By contrast, given an arbitrary $H_{f,k} \in H_f$ as input, algorithm KS returns a reassembling $B$ of $G$ such that $\alpha(H_{f,k}, B) = 2 \cdot E$-outerplanarity$(H_{f,k}) = 2k$. Hence, KS is not an $\alpha$-optimal reassembling algorithm for the family $H_f$.

**Proof Sketch.** We restrict the proof to the case $c = 3$, so that $\bar{f}(k) = 3$ for every $k = E$-outerplanarity$(H_{f,k})$; the proof is easily generalized for any $c \geq 3$. With this restriction, there is a total of $6k'$ vertices in $H_{f,k} \in H_f$ where we pose $k' = k - 1$. Let $V(H_{f,k}) = \{a_1, a_2, \ldots, a_{6k'}\}$, with “$a_1 a_2 a_3$” be the clockwise sequence of vertices of the innermost cycle $L_{k-1}$. (We use the naming conventions for the nested cycles introduced in Definition 4.) Continuing inside-out, “$a_4 \cdots a_9$” is the clockwise sequence of vertices of the next cycle $L_{k-2}$, etc., and “$a_{6k'-2} a_{6k'-1} a_{6k'}$” is the clockwise sequence of vertices of the outermost cycle $L_0$. See the top of Figure 24 for the case $k = 5$.

It is easy to check that an $\alpha$-optimal reassembling of $H_{f,k}$ (not the only one) proceeds as follows:

$$
\{\{\{\{\{\{a_1, a_2\}^4, a_3\}^3, a_4\}^4, a_5\}^5, a_6\}^4, a_7\}^5, a_8\}^4, a_9\}^3 \cdots a_{6k'-3}\}^3, a_{6k'-2}\}^4, a_{6k'-1}\}^3, a_{6k'}\}^0
$$

where the superscript of every cluster is its edge-boundary degree. For example, the contraction of the edge $a_1 a_2 a_3$ produces a super vertex/cluster of degree 4, i.e., $\bar{\sigma}(\{a_1, a_2\}) = 4$, and then the contraction of the edge $a_1 a_2 a_3$ produces a super vertex/cluster of degree 3, i.e., $\bar{\sigma}(\{a_1, a_2, a_3\}) = 3$, etc. The first three contractions of this reassembling are depicted in the bottom of Figure 24 for the case $k = 5$.

The resulting reassembling tree $B$, as just described, is such that $\alpha(H_{f,k}, B) = 5$, whereas algorithm KS on input $H_{f,k}$ returns a reassembling tree $KS(H_{f,k}) = B'$ such that $\alpha(H_{f,k}, B') = 2k$.

Whereas algorithm KS proceeds in an ‘outside-in’ fashion (starting from the outermost cycles), the reassembling in the preceding proof proceeds ‘inside-out’ (starting from the innermost cycle). However, the order of the latter can be reversed to work ‘outside-in’ optimally too.
Figure 24: For Proposition 36: Graph $H_{\bar{f},k}$ when $k = 4$ and $\bar{f}(k) = 3$.

The lower figure depicts the first three contractions of an $\alpha$-optimal reassembling:

- $a_1 a_2$
- $a_1 a_2 a_3$
- $a_1 a_2 a_3 a_4$

$\bar{f}(k)$ is the $\alpha$-optimal reassembling size.
6 Conclusion

We developed a linear-time algorithm $KS$ for the reassembling of 3-regular plane graphs. Given such a graph $G$ as input, with $E$-outerplanarity$(G) = k \geq 2$, $14$ algorithm $KS$ returns a binary reassembling $B = KS(G)$ of $G$ such that $\alpha(G, B) = 2k$.

In Section 4 we proved the correctness of algorithm $KS$ and its time complexity, preceded by several examples illustrating the progression of $KS$ on several graphs exhibiting different peculiarities (Section 4.5). In Section 5 we showed that our algorithm is optimal for the class of 3-regular plane graphs that have a sufficiently high “inter-cycle density” which increases as the edge-outerplanarity increases (this informal description is made precise in Section 5).

6.1 Related Work

In Section 2.1 we spelled out the connection between graph carving and graph reassembling. The first of these two notions has been studied for many years. A number of results about graph carving are readily translated into results about graph reassembling. We mention two of the most salient.

**Proposition 37.** There exists an algorithm which, given a plane graph $G$ (not necessarily 3-regular) as input with $n$ vertices, returns in time $O(n^3)$ a binary reassembling $B$ of $V(G)$ such that $(G, B)$ is $\alpha$-optimal.

**Proof.** This is an immediate consequence of Proposition 1 in this report and Theorem 2.1 in [3].

The next proposition is about the existence of a fixed-parameter linear-time algorithm where the parameter-bound not to be exceeded is the $\alpha$-measure. This is to be contrasted with our algorithm $KS$, which can be recast (not here) as a fixed-parameter linear-time algorithm where the parameter-bound not to be exceeded is the edge-outerplanarity. As usual, ‘linear time’ means ‘linear as a function of the number $n$ of vertices’.

**Proposition 38.** Let $p \geq 1$ be fixed. There exists a linear-time algorithm that checks whether an input graph $G$ (not necessarily plane) has a binary reassembling $B$ of its vertices $V(G)$ such that $\alpha(G, B) \leq p$ and, if this is the case, the algorithm returns a reassembling $B$ such that $(G, B)$ is $\alpha$-optimal.

**Proof.** This is a straightforward consequence of Proposition 1 in this report and the main result in [15] (Theorem 1 and Theorem 2 in Section 6).

We can also state, in reverse, implications from results on graph reassembling to results on graph carving, illustrated by the following proposition. Review the definitions related to graph carving in Section 2.1.

**Proposition 39.** There exists a linear-time algorithm which, given an arbitrary 3-regular plane graph $G$ as input with $E$-outerplanarity$(G) = k \geq 2$, returns a routing tree $T$ for $G$ such that width$(G, T) \leq 2k$.

**Note,** in particular, the value of width$(G, T)$ in the returned routing tree $T$ is independent of $n = |V(G)|$.

Moreover, for the family of graphs $H_f$ defined at the beginning of Section 5, the bound $2k$ in Proposition 39 is an optimal carving width, by Theorem 35.

**Proof.** Straightforward consequence of Proposition 1, Theorem 9, and Corollary 20, in this report.

$14$There is no 3-regular plane graph $G$ such that $E$-outerplanarity$(G) = 1$. 

40
6.2 Future Research

One obvious task for the future is to extend our method for 3-regular plane graphs to arbitrary plane graphs. However, short of such an extension (which seems to require a considerable effort to formulate with an equal loss in transparency), we know there is at least one important application, discussed next, for which there is no explicit need for extending our method beyond its current form. Other applications are yet to be identified.

Consider an application involving the computation of maximum flows in networks. We take a flow network to be a capacitated directed graph $G$ with no self-loops and two distinguished vertices, a source vertex $s$ and a sink vertex $t$, which we identify by the quadruple $(G, c, s, t)$ where $c : E(G) \rightarrow \mathbb{R}_+$ is the (non-negative) capacity function. A flow $f : E(G) \rightarrow \mathbb{R}_+$ is feasible if $f(e) \leq c(e)$ for every $e \in E(G)$ and $f$ satisfies the usual flow-conservation condition at every vertex $v \in V(G) - \{s, t\}$. We write $|f|$ to denote the value of the flow $f$, which is the excess flow exiting $s$ or, equivalently, the excess flow entering $t$.

Given a network $(G, c, s, t)$, we can transform it into an equivalent $(G^*, c^*, s^*, t^*)$ in time $O(n)$ where $n = |V(G)|$ and every vertex $v \in V(G^*)$ has degree $3$. Moreover, if $G$ is plane, then so is $G^*$ such that $E$-outerplanarity$(G) = E$-outerplanarity$(G^*)$. The crux of the transformation $(G, c, s, t) \mapsto (G^*, c^*, s^*, t^*)$ is an expansion of every $v \in V(G^*)$ of degree $\geq 4$; Figure 25 shows the expansion of a degree-5 vertex.

The details of the transformation $(G, c, s, t) \mapsto (G^*, c^*, s^*, t^*)$ will be in a separate report [9], which will also include a proof of the following result.

**Proposition 40.** There is a fixed-parameter linear-time algorithm to compute max flow in planar flow networks $(G, c, s, t)$, where the parameter-bound not to be exceeded is $k = E$-outerplanarity$(G)$.

We conclude with a conjecture which, we believe, can be examined using the methodology developed in this paper. To make sense of the conjecture statement, review how we classify and define the level of inter-cycle trees (ICT’s) in Definition 4, Propositions 5 and 6, and the Remark on page 12.

**Conjecture 41.** There is a function $f : \mathbb{N} \rightarrow \mathbb{N}$ such that, for every 3-regular plane graph $G$, if for every level $j \in \{0, \ldots, k - 1\}$ where $k = E$-outerplanarity$(G)$ the number of level-$j$ ICT’s is $\geq f(k)$, then KS is an $\alpha$-optimal reassembling algorithm for $G$, i.e., KS returns a reassembling $B = KS(G)$ such that $\alpha(G, B) = 2k$ is optimal.

If Conjecture 41 is true, it defines a larger class of 3-regular plane graphs than Theorem 35 for which KS is an $\alpha$-optimal reassembling algorithm. (Theorem 35 is about 3-regular plane graphs where every ICT is an ICE, an inter-cycle edge.)

A final open problem is: Characterize the class of all 3-regular plane graphs for which KS is an $\alpha$-optimal reassembling algorithm.
References

[1] Cedric Bentz. Disjoint paths in sparse graphs. *Discrete Applied Mathematics*, 157(17):3558–3568, 2009. 6

[2] Azer Bestavros and Assaf Kfoury. A Domain-Specific Language for Incremental and Modular Design of Large-Scale Verifiably-Safe Flow Networks. In *Proc. of IFIP Working Conference on Domain-Specific Languages (DSL 2011), EPTCS Volume 66*, pages 24–47, Sept 2011. ii, 1

[3] Qian-Ping Gu and Hisao Tamaki. Optimal Branch-Decomposition of Planar Graphs in $O(n^3)$ Time. *ACM Transactions on Algorithms*, 4(3), June 2008. 2, 40

[4] Dorit S. Hochbaum. Why Should Biconnected Components Be Identified First. *Discrete Appl. Math.*, 42(2-3):203–210, April 1993. 26

[5] John Hopcroft and Robert Tarjan. Algorithm 447: Efficient Algorithms for Graph Manipulation. *Communications of the ACM*, 16(6):372–378, June 1973. 26

[6] Frank Kammer. Determining the Smallest $k$ Such That $G$ Is $k$-Outerplanar. In Lars Arge, Michael Hoffmann, and Emo Welzl, editors, *Proc. of 15th Annual European Symposium on Algorithms, ESA 2007*, pages 359–370. LNCS 4698, Springer Verlag, September 2007. 5

[7] Assaf Kfoury. The Denotational, Operational, and Static Semantics of a Domain-Specific Language for the Design of Flow Networks. In *Proc. of SBLP 2011: Brazilian Symposium on Programming Languages*, Sept 2011. ii, 1

[8] Assaf Kfoury. The Syntax and Semantics of a Domain-Specific Language for Flow-Network Design. *Science of Computer Programming*, 93(Part A):19–38, November 2014. ii, 1

[9] Assaf Kfoury. A Fixed-Parameter Linear-Time Algorithm for Maximum Flow in Planar Flow Networks. Uploaded to arxiv.org, July 2018. 41

[10] Assaf Kfoury and Saber Mirzaei. Efficient Reassembling of Graphs, Part 1: The Linear Case. *J. of Combinatorial Optimization*, 33(3):1057–1089, April 2017. ii, 1, 2, 3

[11] Saber Mirzaei and Assaf Kfoury. Efficient Reassembling of Graphs, Part 2: The Balanced Case. *CoRR*, abs/1602.02863, 2016. preprint, http://arxiv.org/abs/1602.02863v1. ii, 1

[12] Maurizio Patrignani. Planarity Testing and Embedding. In Roberto Tamassia, editor, *Handbook of Graph Drawing and Visualization*, pages 1–42. CRC Press, Baton Rouge, FL, 2013. 5

[13] Paul D. Seymour and Robin Thomas. Call Routing and the Ratcatcher. *Combinatorica*, 14(2):217–241, 1994. 2, 3

[14] Nate Soule, Azer Bestavros, Assaf Kfoury, and Andrei Lapets. Safe Compositional Equation-based Modeling of Constrained Flow Networks. In *Proc. of 4th Int’l Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*, Zürich, September 2011. ii, 1

[15] Dimitrios M. Thilikos, Maria J. Serna, and Hans L. Bodlaender. Constructive Linear-Time Algorithms for Small Cutwidth and Carving-Width. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, D. T. Lee, and Shang-Hua Teng, editors, *Algorithms and Computation: 11th International Conference, ISAAC 2000 Taipei, Taiwan, December 18–20, 2000 Proceedings*, pages 192–203. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000. 40

42
A Appendix: Pseudocode of Algorithm KS

Below is the pseudocode of algorithm KS, of which the full Python code is an elaboration. The latter can be downloaded from the website Graph Reassembling. The pseudocode is based on the description in Sections 4.1, 4.2, 4.3, and 4.4. For easier cross-referencing, we use the following conventions in the pseudocode:

1. Lines in black boldface appear verbatim in the Python code; these are headings of function definitions.
2. Lines in black Roman characters, which do not start with a hash #, describe each an action that is implemented with several Python instructions; they are inserted as comments throughout the Python code in order to make the Python code more readable.
3. Lines in grey Roman characters, which start with a hash #, are inserted as comments in the Python code; they describe the state of the execution at the points where they are inserted.
4. Lines in red are related to the collapse operation, lines in green are related to the merge operation. We follow the same color convention in our examples in Section 4.5.

```python
def pre-process():
    let the current E-outerplanarity level be 0

    # by marking edges at each E-outerplanarity layer one at a time
    # the trees and cycles can be differentiated
    initialize every vertex such that all vertices are unmarked
    while the graph has edges left unmarked
        starting at the upper leftmost unmarked vertex,
        traverse the graph by taking the next clockwise edge at each vertex encountered,
        marking each time a vertex is visited,
        until the traversal returns to the original vertex
        the edge of this particular E-outerplanarity are now defined to be all the edges traversed

    for all edges traversed
        if the edge was traversed once
            the edge is part of a cycle
        otherwise
            # the edge has been marked exactly twice
            the edge is part of a cycle

    for all vertices that have two incident cycle edges
        # the vertex will have exactly one incident tree edge
        if the tree edge is on the same E-outerplanarity as the vertex
            then it is an outward cycle vertex
        otherwise
            then it is an inward cycle vertex

    to determine the vertices of a particular cycle or tree
    begin a depth-first search at both endpoints of a tree or cycle edge
    and traverse every edge adjacent of the same type
    two cycles or two trees will never share the same vertex,
    so the depth-first search will traverse the entire tree or cycle and then end
    # the entire graph will be traversed once as a whole
    # all leaf vertices will be visited twice
    # while other vertices will be visited once

    increment the E-outerplanarity by 1
```


def process():
    at each E-outerplanarity level in the graph
    define a queue that holds trees to be collapsed and merged

    prepare_cycle(outermost cycle enclosing the whole graph)

    while any queue is not empty
        pop the oldest tree in the queue at the deepest E-outerplanarity level
        if the tree has not collapsed
            collapse the tree
        otherwise
            merge the tree

    def collapse_type_a(tree):
        refer to the upper left most vertex of tree as v
        collapse_tree(tree,v)
        add tree to the queue for this layer so it merges as a type A tree

    def collapse_type_b(v):
        refer to the tree v is part of as tree

        refer to the root of tree as u
        # note that u and v will never be the same vertex

        collapse_tree(tree,v)
        add tree to the queue for this layer so it merges as a type B tree

    def merge_type_a(tree):
        refer to the cycle enclosing tree as the outer cycle
        refer to the super node that tree is part of as super

        if the outer cycle has no incident trees left uncollapsed
            merge_cycle(the outer cycle, super)
            return

        otherwise, if the outer cycle has a single incident tree left uncollapsed
            prep_incident_tree(the outer cycle, super)
            return

        find a vertex adjacent to tree on the outer cycle that
        is part of a tree on the same layer as tree
        and is not in the same super vertex as tree

        if that vertex does not exist
            # there is nothing to do until the outer cycle is ready to merge
            because the outer cycle is waiting on a tree that is not adjacent to this tree
            return

        refer to that vertex as the successor vertex
        refer to the tree that the successor vertex is part of as the successor tree

        if the successor tree is Type-B
if the successor tree has merged
    # there is nothing to do because there is nowhere for us to merge
    that has not already merged
    return
# the successor vertex will never be the root vertex of the successor tree
merge super and the super node of which the successor vertex is a part

otherwise, if the successor tree is Type-B or it has not collapsed yet
    merge super and the successor vertex
    if successor tree is ready to collapse
        # the successor tree may be either type A or type B
        add the successor tree to the queue for this layer
        so it merges with the proper type

otherwise,
    # the successor tree is a type A tree that has collapsed but not yet merged
    merge super and the super node of which the successor tree is a part

def merge_type_b(v):
    refer to the super node that v is part of as the super
    refer to the cycle v is a on as cycle
    refer to the clockwise successor of v on cycle as the successor vertex
    refer to the tree v is part of as tree

    marked v as collapsed as it was not marked as collapsed earlier

    if the successor vertex is an outer vertex
        if there is nothing stored with the successor vertex
            refer to the tree that the successor vertex is part of
            as the successor tree
            merge the super and the successor vertex
            if the successor tree is the only tree left on cycle uncollapsed
                and it is ready to collapse
                add the successor tree to the queue for this layer
                so it merges as a type B tree

        otherwise, if there are any trees on cycle left uncollapsed
            merge super and the super node of which the successor vertex is a part

    otherwise,
        # cycle has been entirely collapsed and is ready to merge
        merge_cycle(layer_states, layer, rs, c, super)

otherwise
    merge super and the successor vertex
    if cycle has 0 or 1 incident trees left uncollapsed
    and it has not been prepared
        prep_cycle(cycle)

def merge_cycle(cycle, super):
    refer to the cycle enclosing cycle as the outer cycle

    find the clockwise successor tree of this cycle on the layer of the outer cycle
if there is such a tree
    merge super and the vertex closest to this cycle

otherwise, if the outer cycle is the outermost cycle
    the graph has been fully collapsed and no more work is left

otherwise, if the outer cycle has any incident trees left uncollapsed
    # it will have exactly one incident tree left uncollapsed
    prepare_incident_tree(the outer cycle, super)

otherwise,
    # we must recurse upwards to continue
    merge_cycle(the outer cycle, super)

def prep_incident_tree(cycle, super):
    # there is exactly one tree uncollapsed incident to this cycle
    merge cycle and the root vertex of the incident tree on cycle
    if the incident tree is ready to collapse
        add the incident tree to the queue for this layer so it as a type B tree

def collapse_tree(tree,x):
    refer to the vertex adjacent to x as v
    note that x will always be leaf vertex of tree

    if v is a leaf vertex of tree
        assign an index to x
        assign an index to v
        return the super vertex of x and v

    while v != x:
        if v is a leaf vertex
            assign an index to v
            set v to the predecessor of v
            continue

        refer to the left child of v as l
        refer to the right child of v as l

        if l does not have an index
            v is the predecessor of l
            set v to l
            continue

        if r does not have an index
            v is the predecessor of l
            set v to l
            continue

    note that at this point, l and r may be super vertices,
    but v is definitely not
    assign an index to v
    collapse l and v into a super vertex
collapse the result with r and store it under v
set v to the predecessor of v

assign an index to x
return the super vertex of x and v