In ALGORITHMICA, 1994. Copyright Springer-Verlag

Finding the Closed Partition of a Planar Graph*

Vijaya Ramachandran
Honghua Yang

Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712

December 3, 1992

Abstract

We consider the problem of finding a *closed partition* in a directed graph. This problem has applications in concurrent probabilistic program verification. The best sequential algorithm known for this problem runs in $O(mn)$ time where m is the number of directed edges and n is the number of vertices in the given digraph. In this paper, we present a linear time sequential algorithm to solve the closed partition problem for planar digraphs that are *compact*. We then build on this algorithm to obtain an $O(n^{1.5})$ time sequential algorithm to solve the closed partition problem for a general planar digraph.

1 Introduction

In this paper, we consider the *closed partition problem* [13]. Let $G = (V, E)$ be a digraph in which the vertices in a set $V' \subseteq V$ are colored blue. A *closed component* in G is a maximal induced subgraph S of G such that either S is a single vertex or S is strongly connected and no blue vertex in S has an outgoing edge in G to a vertex not in S. A closed component is *nontrivial* if it contains more than one vertex. Note that in general, a closed component is a subgraph of a strongly connected component. The *closed partition problem* is to find the collection of closed components in G. This problem is interesting as a graph-theoretical problem and it has applications in concurrent probabilistic program verification ([2], [8], [12]).

The following results are known about the complexity of the closed partition problem [13]. The problem can be solved by the following straightforward polynomial-time algorithm: Find the strongly connected components of the input graph, and repeatedly refine each strongly connected

*This work was supported in part by NSF grant CCR 89-10707.

component by deleting a blue vertex with an edge outgoing from the component (if such a vertex exists) and recomputing the strongly connected components in the resulting graph. If the input graph has n nodes and m edges then this algorithm runs in $O(mn)$ time. A reduction from the monotone circuit value problem shows that the problem is P-complete and hence unlikely to have an efficient highly parallel algorithm.

A *compact digraph* is an embedding of a planar digraph in which no strongly connected component encloses any other strongly connected component. In this paper, we present a linear time sequential algorithm to solve the closed partition problem for compact digraphs. We then extend this algorithm to obtain an $O(n^{1.5})$ time sequential algorithm to solve the closed partition problem for a general planar digraph where n is the number of vertices in the input digraph.

The main idea in our algorithms is to take advantage of the dual digraph of a plane digraph. As in the sequential algorithm described above, our algorithms works with a strongly connected subgraph of the input graph and repeatedly deletes a blue vertex that has an edge outgoing from the subgraph until no such blue vertex can be found. In general, it is difficult to decide, when deleting a blue vertex from a digraph, whether a strongly connected subgraph is separated. But the decision becomes easier when dealing with the dual of a plane digraph. We make use of the fact that a plane digraph is strongly connected if and only if its dual digraph is acyclic (see, e.g., [6]) and an extension of this result that the edges that are not in any strongly connected component of a plane digraph are exactly those edges whose dual edges are in the strongly connected components of the dual digraph.

We do not know if planar graphs are likely to appear in the applications cited in ([2], [8], [12]). However, planar digraphs form a natural subclass of directed graphs, and our algorithm can be viewed as a first step towards obtaining more efficient algorithms for the closed partition problem. Further, our technique of moving between the primal and dual of a plane embedding in order to obtain an efficient algorithm is one that may have applications in other problems on planar directed graphs.

The rest of this paper is organized as follows: Section 2 defines the terminology used in this paper. Section 3 gives the main lemmas that establish the relationship between the operations in a primal digraph and the operations in its dual digraph. Section 4 gives a high level description of the algorithm for compact digraphs. Section 5 presents a detailed description of this algorithm and its complexity analysis. Using the algorithm for compact digraphs, we give an algorithm to solve the closed partition problem for a general planar digraph in section 6. We discuss some open problems in section 7. An example is included in the appendix to illustrate how the first algorithm works on a compact digraph. Figure 5 and Figure 11 in the appendix show a compact digraph G and its closed partition, respectively.

2 Preliminaries

We start by presenting several definitions that will be used in the later sections of this paper.

A *(weakly) connected component* in a digraph G is a connected component in the undirected graph obtained from G by making the edges in G undirected. A digraph is *strongly connected* if every two vertices in the digraph are reachable from each other. A *strongly connected component* (scc for short) of a digraph is a maximal subgraph that is strongly connected. An scc which contains only one vertex is a *trivial* scc. An scc which contains more than one vertex is a *nontrivial* scc.

The start-point u of a directed edge (u, v) is called the *tail* of the edge and the end-point v is called the *head* of the edge. The directed edge (u, v) is said to be *outgoing from* u and *incoming to* v. *An edge is connected to a vertex* if it is either incoming to or outgoing from the vertex. A directed edge is *outgoing from a subgraph* if its tail is a vertex of the subgraph and its head is not. An edge that is outgoing from an scc is called *a link edge*.

A graph is said to be *embedded* in a surface S when it is drawn on S so that no two edges intersect. A graph is *planar* if it can be embedded in the plane. A *plane* graph is a graph that has already been embedded in the plane. Given a plane graph G, a *face* of G is a maximal portion of the plane for which any two points may be joined by a curve such that each point of the curve neither corresponds to a vertex of G nor lies on any curve corresponding to an edge of G. Every plane graph contains an unbounded face called the *external face*. For a plane graph G, the *boundary* of a face f consists of all those points x corresponding to vertices and edges of G having the property that x can be joined to a point of f by a curve, all of whose points different from x belong to f [1].

The *outer boundary of a plane graph* is the boundary of the external face. The *outer boundary of a face* in a plane digraph is the outer boundary of the subgraph induced by the boundary of the face. The *outer boundary of an scc* in a plane digraph is the outer boundary of the subgraph induced by the scc. The outer boundary of an scc divides a plane graph into two parts. The part of the plane that contains the external face (including the vertices and the edges of the original graph that are embedded in that face) is called *the external region of the scc*. The part of the plane that does not contain the external face is called the *internal region of the scc*. The outer boundary of a face also divides a plane graph into two parts. The *external region of a face* is the part of the plane that contains the external face. The *internal region of a face* is the part of the plane within the boundary of the face. A face *encloses* an scc if the scc is in the internal region of the face. A face f_1 *encloses* a face f_2 if f_2 is in the internal region of f_1. A face f *immediately encloses* an scc if the face encloses the scc but no other face enclosed by f encloses the scc. For two embedded sccs S_1 and S_2, we say S_1 *encloses* S_2 if S_2 is in the internal region of S_1. S_1 *immediately encloses* S_2 if S_1 encloses S_2 but no other scc enclosed by S_1 encloses S_2. An embedded scc is *compact* if it does not enclose any other scc (including any trivial scc). A plane digraph G is *compact* if all sccs in G are compact.

[1] By this definition, an isolated vertex belongs to the boundary of the face in which it lies.

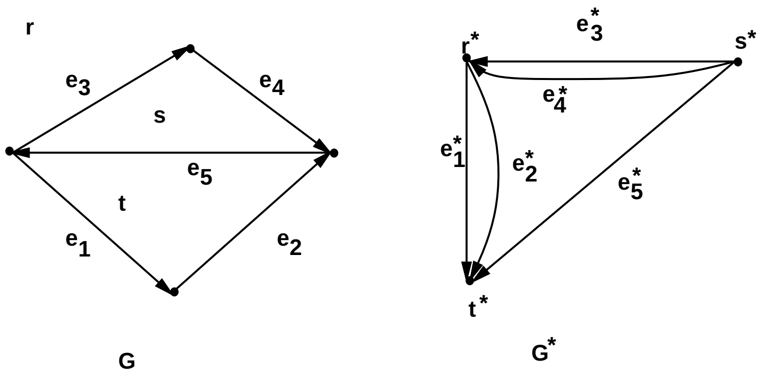

Figure 1: An example of a maximal divergent tree in a dual graph. r^* and t^* are tree vertices of the maximal divergent tree T rooted at r^*. s^* is a discrete vertex. e_1^* is a tree edge. e_2^* is an internal nontree edge. e_3^*, e_4^* and e_5^* are external nontree edges.

The *surrounding faces* of a vertex in a plane digraph are the faces whose boundaries contain the vertex. An edge or a vertex is *adjacent* to a face if it is on the boundary of the face. An edge on the boundary of a face is *clockwise with respect to the face* (we will omit "with respect to the face" if there is no confusion) if the edge is on a directed cycle and is clockwise from the point of view of a person who is standing upright in the center of the face. We can define a *counterclockwise* edge on the boundary of a face similarly. Two edges on the boundary of a face are of *opposite directions* if one edge is clockwise and the other is counterclockwise.

The plane digraph G we consider here is called the *primal graph*. The edges in G are called *primal edges*. The *dual digraph* G^* of G is constructed as follows: (i) For each face t of G, define a vertex t^* in G^*. (ii) A directed edge $e^*{=}(t^*, s^*)$ is in G^* if the pair of corresponding faces t and s share a boundary edge e in G and when e is rotated counterclockwise, the head of e enters face s first. The edge e^* is called the *dual edge* of the primal edge e. Note that the dual digraph of a plane digraph is in general a multi-digraph which allows parallel directed edges.

A *divergent tree* is a rooted directed tree whose root has indegree 0 and all other vertices have indegree 1. Given a plane digraph G, a *maximal divergent tree* T of G is a divergent tree rooted at a specified vertex such that it includes all vertices in G reachable from the root. The vertices of T are called *tree vertices*. The vertices of G not in T are called *discrete vertices*. The edges of T are called *tree edges*. The edges of G that are either incoming to or outgoing from a discrete vertex are called *external nontree edges*. The nontree edges with both end points in T are called *internal nontree edges*. Figure 1 shows a compact digraph G and its dual digraph G^* with a maximal divergent tree.

We now define two operations on a multi-digraph: *merging* and *condensing*. *Merging* a vertex u to a vertex v in a multi-digraph is defined as follows: (i) Edges between u and v are deleted. (ii) Edges that are incoming to (or outgoing from) u are redirected as edges incoming to (or outgoing from) v. (iii) The vertex u is deleted from the digraph. *Condensing* an scc in a multi-digraph is defined as merging all the vertices of the scc to a specific vertex of the scc.

Let $G = (V, E)$ be a digraph in which the vertices in a set $V' \subseteq V$ are colored blue. A *closed component* in G is a maximal induced subgraph S of G such that either S is a single vertex or S is strongly connected and no blue vertex in S has an outgoing edge in G to a vertex not in S. A closed component is *nontrivial* if it contains more than one vertex. Note that in general, a closed component is a subgraph of a strongly connected component. It is easy to see that the set of closed components is unique. The *closed partition problem* is to find the set of closed components in G. A *bad blue vertex* is a blue vertex in the input digraph that is a singleton in the solution set. In the example given in Appendix A, a compact digraph G and its closed partition are shown in Figure 5 and Figure 11 respectively.

3 Main Lemmas

In this section, we introduce some lemmas to establish the relationship between the operations in a primal digraph and the operations in its dual digraph.

Lemma 1 *Deleting all the edges connected to a vertex on the outer boundary of an scc in a compact plane digraph G results in a digraph that is still compact.*

Proof: First we notice that deleting all the edges connected to a vertex on the boundary of an scc does not affect the compactness of other sccs.

Suppose after we have deleted all the edges connected to a vertex v on the outer boundary of a compact scc S, S breaks into several sccs among which S_1 encloses S_2 (see Figure 2). Let w be any vertex in S_2. Since v and S_2 were in the same scc S, there were two paths p and q in S such that p is a directed path from w to v and q is a directed path from v to w. Since S_1 encloses S_2 and v was on the outer boundary of S, p and q must intersect S_1. Let x be the first intersection of p and S_1 and y be the last intersection of q and S_1 (note that neither x nor y could be v). Then the part of the directed path from w to x and the part of the directed path from y to w connect S_1 and S_2 together. This contradicts the assumption that S_1 and S_2 are two distinct sccs. Hence the lemma holds. []

Lemma 2 *[6] A connected planar embedded digraph is strongly connected if and only if its dual digraph is acyclic. []*

Lemma 3 *[Extension of Lemma 2] Every connected component in a plane digraph is strongly connected if and only if the dual digraph of the plane digraph is acyclic.*

Proof: Let G be a plane digraph and let G^* be its dual digraph. We to prove the *if* part of the lemma by contradiction.

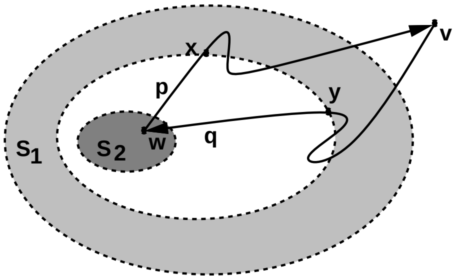

Figure 2: An example for the proof of Lemma 1.

Suppose G^* is acyclic but at least one connected component G' in G is not strongly connected. Let G'^* be the dual of G'. By Lemma 2, G'^* has a directed cycle. Since G'^* is a subgraph of G^* (G^* can be obtained by adding dual edges and vertices that correspond to the edges and faces of other connected components in G), G^* has a directed cycle. This contradicts the assumption that G^* is acyclic. Hence every connected component in G is strongly connected.

We now prove the *only if* part of Lemma 3 by induction on the number of connected components in G.

Base: By Lemma 2, if G has only one connected component and it is strongly connected, then G^* is acyclic.

Induction step: Let the statement be true for any plane digraph with $k-1$ connected components. Let G be a plane digraph with k connected components each of which is strongly connected. Let S be a compact scc in G, i.e., S does not enclose any other scc in G. Since there is no link edge in G, a region between two sccs of G is exactly a face. Let f be the face in G that immediately encloses S. We remove the vertices and edges of S from G and call the resulting graph G'. Let G'^* be the dual graph of G' and let f^* be the dual vertex of face f. Let S^* be the dual graph of S (by taking f as the external face of S). Then G^* is obtained from G'^* and S^* by identifying f^* and the dual vertex of the external face of S in S^*. By induction hypothesis, both G'^* and S'^* are acyclic. Furthermore, there is no edge in G^* outgoing from a vertex in G'^* to a vertex in S^* (except f^*) and there is no edge in G^* outgoing from a vertex in s^* to a vertex in G'^* (except f^*). Hence G^* is acyclic. []

Observation 1 *Let G be a plane digraph and let G^* be its dual graph. Let e be an edge in G and let e^* be its dual edge in G^*.*

If e is a boundary edge of exactly one face (we call e a single face boundary edge), then e^* is a selfloop. Removing e from G corresponds to removing the selfloop e^* from G^*.

If e is a boundary edge of two faces f and g in G (let f^* and g^* be the dual vertices of f and g respectively and let e^* be directed from f^* to g^*), then removing e from G corresponds to merging f^* to g^* in G^*. []

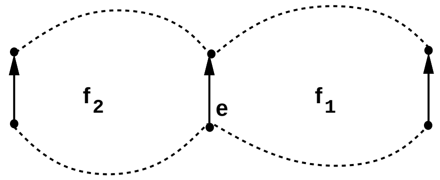

Figure 3: Diagram for the claim in the proof of Lemma 4.

Lemma 4 *An edge in a plane digraph G is a link edge if and only if its dual edge is in a nontrivial scc in the dual digraph G^*.*

Proof: First we claim that if a link edge e is adjacent to two faces f_1, f_2 in G, then there is a link edge that is of the opposite direction of e on the boundary of f_1 and there is a link edge that is of the opposite direction of e on the boundary of f_2 (see Figure 3). This is seen as follows.

Without loss of generality, assume the direction of e is clockwise with respect to f_1 (thus e is counterclockwise with respect to f_2). Then not all edges on the boundary of face f_1 are clockwise with respect to f_1 (otherwise e would be in an scc that contains the clockwise cycle). Furthermore, at least one of the counterclockwise edges on the boundary of face f_1 is a link edge. Otherwise each of the counterclockwise edges could be replaced by a clockwise simple path (since each edge is in some scc) and e would be on a clockwise cycle. Similarly, there is a clockwise link edge with respect to f_2 on the boundary of face f_2.

We now prove this lemma by induction on the number of link edges in G.

Base: G has one link edge e. Let G_1 be $G - \{e\}$. Then G_1 has one more connected component than G has. By Lemma 3, its dual graph G_1^* is acyclic. Since e is the only link edge of G, e must be a single face boundary edge (otherwise, by the above claim, G would have at least two link edges). By Observation 1, the dual edge e^* of e is a selfloop and G^* is $G_1^* \cup \{e^*\}$. Therefore the selfloop e^* is the only scc in G^*.

Induction step: Suppose the statement holds for the case when G has $< k$ link edges. Consider the case when G has k link edges.

- If there is a link edge e in G that is a single face boundary edge, then the dual edge e^* is a selfloop. Let G_1 be $G - \{e\}$ and let G_1^* be the dual graph of G_1. By induction hypothesis, an edge in G_1 is a link edge if and only if its dual edge is in an scc in G_1^*. Since G^* is $G_1^* \cup \{e^*\}$ (by Observation 1), the statement is true for the case when G has k link edges.

- If all link edges in G are adjacent to two faces, then let e_1 be a link edge in G that is adjacent to two faces f_1 and f_2. Assume without loss of generality that e_1 is clockwise with respect to f_1 (see Figure 4). By the above claim, there is a counterclockwise (with respect to f_1) link edge e_0 on the boundary of face f_1 and a clockwise (with respect to f_2) link edge e_2 on the boundary of face f_2 (since e_1 is counterclockwise with respect to f_2). Let f_0 be the other face

that is adjacent to link edge e_0.
If e_2 is on the boundary of face f_0, then stop; else let f_3 be the other face that is adjacent to e_2 and let e_3 be a link edge on the boundary of f_3 that has the opposite direction of e_2 with respect to f_3. We keep finding link edges in this manner until we have a list of link edges $e_0, e_1, \ldots, e_i$ such that e_j and e_{j+1} $(0 \leq j < i)$ are link edges of opposite directions on the boundary of face f_{j+1} and e_i is also on the boundary of face f_h for some h, $0 \leq h < i$. Assume without loss of generality that $h = 0$. We identify face f_0 as the external face r of G. Let G_1 be $G - \{e_j | 0 \leq j \leq i\}$ and let G_1^* be its dual graph. Let x^* be the dual of x for each edge or vertex x. By induction hypothesis, an edge in G_1 is a link edge if and only if its dual edge is in an scc in G_1^*. By Observation 1, G^* can be obtained by expanding vertex r^* in G_1^* to a cycle $< r^*, f_1^*, \ldots, r^* >$; the edges on this cycle are $e_0^*, e_1^*, \ldots, e_i^*$. The edges $a_1^*, a_2^*, \ldots, a_k^*$ in G_1^* that are connected to r^* are connected to vertices $r^*, f_1^*, \ldots, f_i^*$ in G^* in the way that an edge a_l^* $(1 \leq l \leq k)$ is connected to v^* $(v^* \in \{r^*, f_1^*, \ldots, r^*\})$ if and only if the primal edge a_l is adjacent to the primal face v (see Figure 4).
We now show that the edges that are in sccs in G_1^* are still in sccs in G^* and the edges that are not in sccs in G_1^* are still not in sccs in G^*. This is seen as follows.

- If an edge b^* is in an scc in G_1^*, then b^* is in a cycle C in G_1^*.
  * If C does not contain r^*, then C remains unchanged in G^*.
  * If C contains r^*, then after r^* is expanded into cycle $< r^*, f_1^*, \ldots, r^* >$ in G^*, C is also expanded into a larger cycle in G^* containing part of the cycle $< r^*, f_1^*, \ldots, r^* >$.

  Therefore b^* is in a cycle in G^*.
- If an edge b^* is not in an scc in G_1^*, then suppose b^* is in a cycle C in G^*.
  * If C does not contain any vertex in $< r^*, f_1^*, \ldots, r^* >$, then C remains unchanged in G_1^*.
  * If C contains at least one vertex in $< r^*, f_1^*, \ldots, r^* >$, then after deleting edges in C that are in cycle $< r^*, f_1^*, \ldots, r^* >$ and identifying together, the vertices in C that are in cycle $< r^*, f_1^*, \ldots, r^* >$, the resulting C is still a cycle in G_1^*.

  Both cases contradict the fact that b^* is not in an scc in G_1^*. Therefore b^* is not in any scc in G^*.

Furthermore, the dual edges of e_j $(0 \leq j \leq i)$ are in an scc in G^* since they are in the cycle $< r^*, f_1^*, \ldots, r^* >$. Therefore, the statement holds when G has k link edges.

Hence Lemma 4 holds for any plane digraph G. []

Lemma 5 *If G is a compact digraph with external face r, then there is at most one nontrivial scc in its dual graph G^*. If G^* contains a nontrivial scc, the scc must include r^*, the dual vertex of r.*

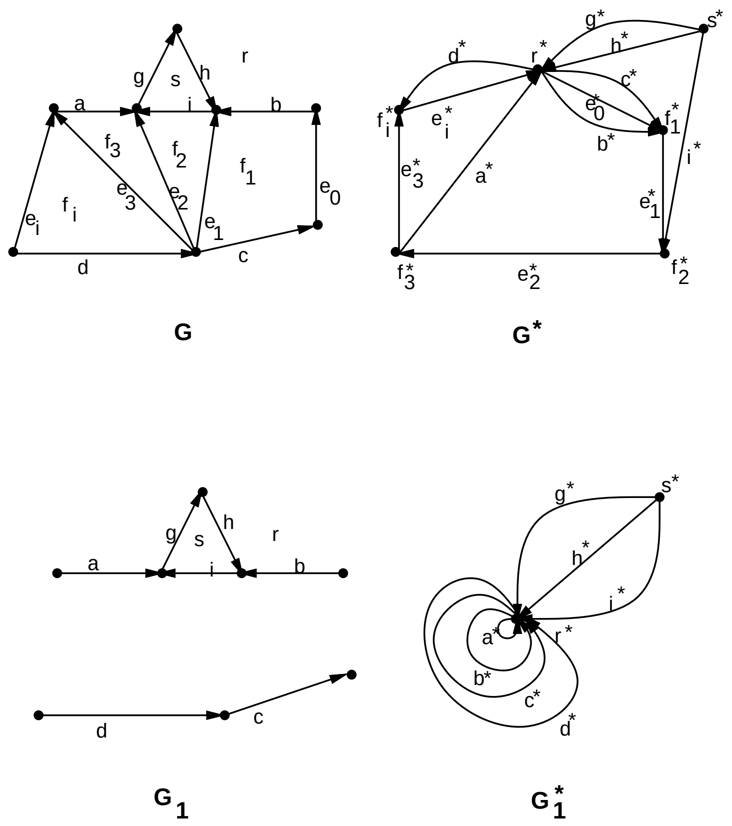

Figure 4: Figures for the proof of Lemma 4.

Proof: If there is no link edge in G, then by Lemma 4, there is no nontrivial scc in G^* and we are done. Otherwise, let $e = (u, v)$ be a link edge in G. By Lemma 4, its dual edge is in an scc S^* of G^*.

Let G' be the graph obtained from G after removing all link edges from G. Then by applying Lemma 4 and Observation 1 to each link edge in G, G'^*, the dual graph G', can be obtained from G^* by condensing all the sccs in G^* (see section 2 for the definition of condensing). Let S^* be condensed to a vertex s^* in G'^*. We now prove by contradiction that r^* is in S^*.

Suppose r^* is not in S^*. Then r^* must be condensed into another vertex, say r'^*, in G'^*, where $s^* \neq r'^*$. Let the primal face of s^* be f in G'.

We establish the following facts.

1. The primal face f is in the internal region of an scc H in G'.
   Proof of fact 1: The primal face f is different from the external face r in G'. Since G' contains no link edges, any face in G' except r is in the internal region of some scc.

2. The two end vertices of e, u and v, are on the boundary of f in G'.
   Proof of fact 2: Since e was a link edge in G and does not exist in G', the primal faces in G that were adjacent to e become part of face f in G'. Therefore u and v are on the boundary of f in G'.

3. The two vertices u and v are in two different sccs in G'.
   Proof of fact 3: This follows from the fact that $e = (u, v)$ is a link edge in G.

From fact 1 and fact 2, u and v are either in the internal region of of the scc H or on the outer boundary of H in G'. From fact 3, either u or v is in an scc other than H. We assume without loss of generality that u is in another scc H'. Then u must be in the internal region of H since otherwise, u would be on the outer boundary of H and hence in H. It follows that H' is enclosed in H by the planarity of G'. Therefore the compactness of G' is violated (even if H' is a trivial scc that contains a single vertex). Hence by Lemma 1 the compactness of G is violated.

We conclude that the dual edge of any link edge in G is in an scc in G^* that contains r^*. Hence G^* has at most one nontrivial scc and if it has one, the nontrivial scc includes r^*. []

Corollary 1 *Let G be a compact digraph with external face r and let G^* be its dual digraph with r^* being the dual vertex of r. Let T be the maximal divergent tree rooted at r^* in G^*. Then G^* has a nontrivial scc if and only if there is an internal nontree edge of T incoming to r^*. []*

Corollary 2 *Let G be a compact digraph with external face r and let G^* be its dual digraph with r^* being the dual vertex of r. Suppose G^* has a nontrivial scc. Then condensing the nontrivial scc to r^* in G^* corresponds to deleting all link edges in G.*

Proof: By Lemma 4, an edge in a plane digraph G is a link edge if and only if its dual edge in G^* is in a nontrivial scc. By Lemma 5, the dual edges of all link edges are in the scc that contains r^*.

Deleting all link edges in G makes all faces adjacent to the link edges in G to become one external face, which is equivalent to merging all vertices in the scc to r^*. []

Lemma 6 *Let T be a maximal divergent tree in the dual digraph G^* of a plane digraph G. Then a discrete vertex in G^* with respect to T cannot be on any directed cycle of G^* that contains a tree vertex.*

Proof: Since T is maximal, there is no external nontree edge outgoing from a vertex in T to a discrete vertex, a discrete vertex cannot be reachable from any tree vertex. Therefore the lemma holds. []

Lemma 7 *Let G be a plane digraph with external face r and let G^* be its dual digraph with r^* being the dual vertex of r. Let T be the maximal divergent tree in G^* rooted at r^*. Then a vertex in G^* is a discrete vertex with respect to T if and only if its primal face in G is inside a clockwise cycle.*

Proof: By the definition of the dual digraph G^* (see section 2), the dual edges of the edges on a clockwise cycle are directed from the faces internal to the cycle to the faces external to the cycle. Therefore the dual vertices of the faces inside the clockwise cycle are not reachable from r^* in the dual graph G^*. Therefore the dual vertices of the faces that are inside a clockwise cycle are discrete vertices.

Conversely, let v^* be a discrete vertex in G^* and let f be its primal face in G. Let C be the boundary of f. If all the edges on C are clockwise with respect to f, then f is inside a clockwise boundary. Otherwise, let e be a boundary edge of f that is counterclockwise with respect to f and let f_1 be the other face that is adjacent to e. Replace e in C by the boundary of f_1 except e. Then there is a directed edge from the dual vertex of f_1 to the dual vertex of f. We keep replacing each counterclockwise edge in C in this manner and stop when either C is a clockwise cycle or C contains an edge on the boundary of the external face r. In the latter case, C must be a clockwise cycle since otherwise, there would be a directed path from r^* to v^* which contradicts the fact that v^* is a discrete vertex. []

4 High Level Description of the Algorithm for Compact Digraphs

We now outline the basic steps in our algorithm to solve the closed partition problem in a compact digraph G. The algorithmic notation here is from [10].

1. find all link edges in G;
2. **do** there exists a link edge $\longrightarrow$
    3. delete all link edges;

4. delete the bad blue vertices that are the tails of the link edges deleted in step 3;
5. repeatedly delete the bad blue vertices that have an outgoing edge to a deleted bad blue vertex;
6. find all link edges in the reduced graph G

od

These steps can be simulated on the dual digraph G^* of G. The corresponding steps are as follows:

$1'$. find the nontrivial scc in G^*;
$2'$. **do** there exists a nontrivial scc in G^* $\longrightarrow$
  $3'$. condense the nontrivial scc in G^*;
  $4'$. find the bad blue vertices in G that are the tails of the primal edges of the edges deleted in step $3'$;
  $5'$. repeatedly find the bad blue vertices in G that have outgoing edges to a bad blue vertex found so far and merge together in G^* the dual vertices of the surrounding faces of each such bad blue vertex;
  $6'$. find the nontrivial scc in the reduced dual graph G^*

od

To implement step $1'$, we construct a maximal divergent tree T rooted at r^* (which is the dual vertex of the external face r of G) in G^* and check if there is an incoming edge to r^*. We show in the next section that steps $3', 4'$ and $5'$ can be implemented in time that is linear in the number of vertices and edges of the input graph over all iterations of the do loop. In order to perform step $6'$ efficiently, our algorithm maintains the property that T is always a maximal divergent tree during the execution of the algorithm. Since the tree T we construct in step $1'$ is a maximal divergent tree, T has this property initially. T preserves this property throughout the algorithm we present in the next section for compact digraphs since whenever a nontree edge from a vertex in T to a discrete vertex is produced, the discrete vertex is forced to become a vertex in T so that the nontree edge becomes a tree edge.

5 Detailed Description of the Algorithm for Compact Digraphs

First we give an informal description of the algorithm for a compact digraph G. We construct the dual digraph G^* where the dual vertex of the external face r of G is labeled r^*. Then we form a maximal divergent tree T rooted at r^* and label the vertices of G^* that are not in T as discrete vertices. By Corollary 1, G^* has a nontrivial scc if and only if there is an internal nontree edge incoming to r^*. We apply a procedure called condense-scc to condense the scc of G^* to r^* (which corresponds to deleting all link edges in G by Corollary 2) and to record the dual vertices of the surrounding faces of the bad blue vertices found when deleting link edges. Two merge procedures are used repeatedly to perform the operations that corresponds to deleting the edges connected to bad blue vertices in G. But actually the two merge procedures do more than just deleting the edges connected to the bad blue vertices. They also delete the single face boundary edges of r already existing or created by deleting the edges connected to the bad blue vertices. The condense and merge procedures are applied repeatedly until we find the collection of closed components in G.

At the end of the algorithm, we form the reduced primal digraph from the reduced dual graph by deleting all the primal edges in G whose dual edges are deleted in G^*[2] and then apply the well known algorithm for finding sccs in G [11] once to get the solution.

There are four procedures in the algorithm.

1. detect-bad-blue-vertex(e^*). Whenever a dual edge e^* in G^* is deleted, this procedure is called to check if the tail of the primal edge of e^* is a blue vertex. If it is, which means the blue vertex is bad, the procedure adds the dual vertices of the surrounding faces of this bad blue vertex to a list called merge-list (which is implemented as a global variable). The vertices in the merge-list are to be merged in the procedures merge-tree and merge-dis.

2. condense-scc(mark-list). The vertices in the nontrivial scc of G^* are provided in the mark-list. Condense-scc procedure condenses the scc in G^* as follows. All edges of G^* in the scc are deleted. Each time a dual edge is deleted, detect-bad-blue-vertex procedure is called to find more bad blue vertices and to add the dual vertices of the surrounding faces of the bad blue vertices to merge-list. All dual edges outgoing from (or incoming to) a vertex in the scc to (or from) a vertex outside the scc are redirected as edges outgoing from (or incoming to) r^*.

3. merge-tree(v^*, r^*). This procedure is used to merge a tree vertex v^* to r^*. All edges between v^* and r^* are deleted. Each time a dual edge is deleted, detect-bad-blue-vertex procedure is called to find more bad blue vertices and to add the dual vertices of the surrounding faces of the bad blue vertices to merge-list. All dual edges outgoing from (or incoming to) v^* are also redirected as in 2.

4. merge-dis(v^*, r^*). This procedure is used to merge a discrete vertex v^* to r^* and is designed to keep the property of the maximal divergent tree while merging. All edges between v^* and r^* are deleted. Each time a dual edge is deleted, detect-bad-blue-vertex procedure is called to find more bad blue vertices and to add the dual vertices of the surrounding faces of the bad blue vertices to merge-list. We redirect edges incoming to v^* as edges incoming to r^* in this procedure. Problems could occur when there is an edge outgoing from v^* to another discrete vertex w^* since after merging v^* to r^*, r^* would have an edge incoming to the discrete vertex w^* which destroys the property of the maximum divergent tree. What we do in this

[2] Here we do not construct G from G^* by finding the dual of G^* since the dual of G^* is in general not the same as G.

procedure is to place w^* into tree T, i.e., let w^* become a child of r^* and put w^* in a list called subtree-list (this list is used to temporarily hold the discrete vertices that are going to be placed in tree T). We then repeatedly place the discrete vertices that are reachable from a vertex in subtree-list into T until no more discrete vertex can be reached from T.

As noted above, the algorithmic notation in this paper is from [10]. We use three control structures: **if** ... **fi**, **do** ... **od**, and **for** ... **rof**.

The form of an **if** statement is:

$$\textbf{if}\ condition_1 \longrightarrow statement\ list_1 \mid condition_2 \longrightarrow statement\ list_2 \mid \ldots$$
$$\mid condition_n \longrightarrow statement\ list_n\ \textbf{fi.}$$

The effect of this statement is to cause the conditions to be evaluated and the statement list for the first **true** condition to be executed; if none of the conditions is **true** none of the statement lists is executed.

The form of a **do** statement is:

$$\textbf{do}\ condition_1 \longrightarrow statement\ list_1 \mid condition_2 \longrightarrow statement\ list_2 \mid \ldots$$
$$\mid condition_n \longrightarrow statement\ list_n\ \textbf{od.}$$

The effect of this statement is similar to that of an **if** except that after the execution of a statement list the conditions are reevaluated, the appropriate statement list is executed, and this is repeated until all conditions evaluated to **false**.

The form of a **for** statement is:

$$\textbf{for}\ iterator \longrightarrow statement\ list\ \textbf{rof.}$$

This statement causes the statement list to be evaluated once for each value of the iterator. We allow the following abbreviations: "**for** $x \in s : condition_1 \longrightarrow statement\ list_1 \mid \ldots \mid condition_n \longrightarrow statement\ list_n$ **rof**" is equivalent to "**for** $x \in s \longrightarrow$ **if** $condition_1 \longrightarrow statement\ list_1 \mid \ldots \mid condition_n \longrightarrow statement\ list_n$ **fi rof.**"

In the following algorithms, edge (x, y) denotes a directed edge pointing from x to y. A '*' will be attached to the edges and vertices in the dual graph. Comments are enclosed between a pair of double curly brackets ('{{' and '}}').

Algorithm 1: Finding the Closed Partition of a Compact Digraph

Input: A compact plane digraph G with a subset of vertices marked blue.

Output: The closed partition of G with respect to the blue vertices.

1. construct the dual digraph G^* with r^* being the dual vertex of the external face r of G;
2. merge-list, mark-list := empty, empty;
3. construct a maximal divergent tree T rooted at r^* in G^*, and label all vertices that are not in T as discrete vertices;
4. **do** there exists an internal nontree edge incoming to r^* $\longrightarrow$
    {{ finding the nontrivial scc in G^* }}
    5. **for** every internal nontree edge (v^*, r^*) $\longrightarrow$
        6. mark the ancestors of v^* (including v^*) that are not already marked and add them to mark-list;
        7. **for** every internal nontree edge (x^*, y^*) for which y^* is marked $\longrightarrow$
            8. mark all ancestors of x^* (including x^*) and add them to mark-list until a marked ancestor is met
            rof
        rof;
    {{ condense the scc to r^* and detect bad blue vertices }}
    9. condense-scc(mark-list);
    {{ delete bad blue vertices by merging and detecting more bad blue vertices }}
    10. **do** merge-list is not empty $\longrightarrow$
        11. **for** every vertex v^* in merge-list :
            12. v^*.type = discrete vertex $\longrightarrow$ merge-dis(v^*, r^*)
            13. | v^*.type = tree vertex $\longrightarrow$ merge-tree(v^*, r^*)
            rof
        od
    od;
14. delete primal edges in the primal digraph G whose dual edges were deleted in G^*;
15. find all the sccs in the reduced primal digraph and output the set of nontrivial sccs

end.

We now specify the four procedures used in Algorithm 1.

procedure detect-bad-blue-vertex(e^*);
  if the tail v of the primal edge of e^* is a blue vertex $\longrightarrow$
    add all dual vertices (except r^*) of the surrounding faces of v to merge-list
  fi
end;

procedure merge-dis(v^*, r^*);
1. **for** multiple edges $e^* = (v^*, r^*) \longrightarrow$
  2. delete e^*;
  3. detect-bad-blue-vertex(e^*)
  rof;
4. subtree-list := empty;
5. **for** every edge $e^* = (w^*, v^*) \longrightarrow e^* := (w^*, r^*)$ **rof**;
6. **for** every edge $e^* = (v^*, w^*) \longrightarrow$
    7. $e^* := (r^*, w^*)$;
    8. **if** w^*.type = tree vertex $\longrightarrow$ e^*.type:= internal nontree edge
      9. | w^*.type = discrete vertex $\longrightarrow$ e^*.type:= tree edge;
        w^*.type:= tree vertex;
        add w^* to subtree-list
    fi
  rof;
B {{ add more discrete vertices to tree T to keep it a maximal divergent tree }}
10. **do** subtree-list is not empty $\longrightarrow$
    11. pick a vertex v'^* from subtree-list;
    12. **for** every edge $e'^* = (v'^*, w'^*)$:
      13. w'^*.type = tree vertex $\longrightarrow$ e^*.type = internal nontree edge
      14. | w'^*.type = discrete vertex $\longrightarrow$ e'^*.type= tree edge;
        w'^*.type:=tree vertex;
        if w'^* is not already on subtree-list $\longrightarrow$
          add w'^* to subtree-list
        fi
    rof
  od;
15. delete v^*
end;

procedure merge-tree(v^*, r^*);
1. **for** $e^* = (v^*, r^*)$ or $e^* = (r^*, v^*) \longrightarrow$
   2. delete e^*;
   3. detect-bad-blue-vertex(e^*)
   rof;
4. **for** every edge $e^* = (w^*, v^*) \longrightarrow$
      5. $e^* := (w^*, r^*)$;
      {{ step 6 is to handle the case when w^* is the parent of v^*}}
      6. **if** w^*.type = tree vertex $\longrightarrow$ e^*.type:= internal nontree edge **fi**
   rof;
7. **for** every edge $e^* = (v^*, w^*) \longrightarrow$
      8. $e^* := (r^*, w^*)$
   rof;
9. delete v^*
end;

procedure condense-scc(mark-list);
1. **for** every vertex v^* (except r^*) in the mark-list $\longrightarrow$
      2. **for** every edge $e^* = (w^*, v^*)$:
            3. w^* is marked $\longrightarrow$ delete e^*; detect-bad-blue-vertex(e^*)
            4. | w^* is not marked $\longrightarrow$ $e^* := (w^*, r^*)$
         rof;
      5. **for** every edge $e^* = (v^*, w^*)$:
            6. w^* is marked $\longrightarrow$ delete e^*; detect-bad-blue-vertex(e^*)
            7. | w^* is not marked $\longrightarrow$ $e^* := (r^*, w^*)$
         rof;
      8. delete v^*
   rof
end;

We now establish the correctness of Algorithm 1.

Lemma 8 *Steps 5-8 in Algorithm 1 mark all vertices in the nontrivial scc in G^*.*

Proof: By Lemma 5, we know that there is at most one nontrivial scc in G^* and this scc includes r^*. If G^* does not have a nontrivial scc (in which case every connected component in G is strongly connected), then Lemma 8 holds vacuously. Suppose G^* has one nontrivial scc. Let v be any vertex in the scc. Then there is a directed path $P = [v = v_1, \ldots, v_{n-1}, v_n = r^*]$ in G^*. We prove by induction on the indices of the vertices in P that v is marked.

Base: (v_{n-1}, r^*) is an internal nontree edge. So v_{n-1} is marked in step 6 of Algorithm 1.
Induction step: Suppose v_{i+1} is marked for some i, $1 \leq i < n$. Consider edge $e = (v_i, v_{i+1})$. If e is a tree edge, then v_i is the parent of v_{i+1} and is marked in step 6 of Algorithm 1; If e is an internal nontree edge, then v_i is marked in step 8 of Algorithm 1. []

Observation 2 *In a plane digraph G with no link edge, deleting the edges connected to a vertex v on the boundary of the external face r will cause the surrounding faces of v to become a single face (we still call it r) and some edges to become single face boundary edges of r (which are obviously link edges). Let G^* be the dual graph of G and let r^* be the dual vertex of r. Then merging the dual vertices of the surrounding faces of v to r^* in G^* corresponds to deleting all edges connected to v and the single face boundary edges of r in G. []*

Theorem 1 *Algorithm 1 correctly solves the closed partition problem on a compact digraph.*

Proof: We denote the reduced dual graph G^* at the beginning of phase i of the do loop in step 4 of Algorithm 1 by G_i^*. We denote the primal graph obtained by deleting from G the primal edges whose dual edges are deleted in G^* up to phase $i-1$ of the do loop by G_i. Then we have $G_1 = G$ and $G_1^* = G^*$.

We claim that at the beginning of the ith phase of the do loop in step 4 of Algorithm 1, the following four conditions hold.

1. T is a maximal divergent tree of G_i^*.
2. The edges deleted in G_i^* in steps 5-9 of Algorithm 1 in phase i are dual edges of the link edges of G_i.
3. The blue vertices deleted during each phase are bad blue vertices.
4. G_i^* is the dual graph of G_i and G_i is a compact digraph.

We prove the above claim by induction on the number of phases executed of the do loop in step 4 of Algorithm 1.
Base: $i = 1$. Clearly 1, 2, 3 and 4 hold after the execution of step 3 of Algorithm 1.
Induction step: Suppose 1, 2, 3 and 4 hold at the beginning of phase i.

1. Condition 1 is preserved during the i th phase by steps 8 to 14 of the merge-dis procedure.
2. By Lemma 8, steps 5-8 of Algorithm 1 find the nontrivial scc in G^* and mark the vertices in the scc. The dual edges deleted by the condense-scc procedure in phase i correspond to link edges in G_i by Corollary 2.
3. We notice that the blue vertices found in the condense-scc procedure are indeed bad blue vertices since they are detected by procedure detect-bad-blue-vertex(e^*) when the dual edge e^* of a link edge e is deleted. A blue vertex is identified as a bad blue vertex only if it is the tail of a deleted link edge.

4. Let G^*_{i1} be the reduced dual graph after the nontrivial scc is condensed in G^*_i and let G_{i1} be $G_i - \{ e \mid e$ is a link edge in $G_i \}$. By Corollary 2, G^*_{i1} is the dual graph of G_{i1}. Since removing all link edges in G_i leaves all sccs in G_{i1} immediately enclosed by the external face, G_{i1} is still compact.

   Let G^*_{i2} be the reduced dual graph after applying steps 10-13 of Algorithm 1 to G^*_{i1} and let G_{i2} be $G_{i1} - \{ e \mid e$ is an edge whose dual edge is deleted in steps 10-13 of Algorithm 1 in phase i$\}$. Then by Observation 2 in this section, G^*_{i2} is the dual of G_{i2}. A bad blue vertex that is the tail of a link edge of G_i is on the outer boundary of the scc that contains the bad blue vertex. By Lemma 1, deleting the edges connected to the bad blue vertices that are the tails of link edges does not affect the compactness of G_{i1}. Nor does deleting the single face boundary edges of the external face r. A bad blue vertex v with an outgoing edge to a deleted bad blue vertex w is also on the outer boundary of an scc (the original scc that contains both v and w may have been broken into several sccs after deleting w). Again, by Lemma 1, G_{i2} is still compact.

   Since $G^*_{i2} = G^*_{i+1}$ and $G_{i2} = G_{i+1}$, G^*_{i+1} is the dual of G_{i+1} at the beginning of phase i+1 and G_{i+1} is compact.

Therefore conditions 1, 2, 3 and 4 hold at the beginning of phase $i+1$. Hence the claim holds for all phases.

The number of edges in G is finite and each phase reduces the number of edges in the primal graph of that phase. Therefore the do loop in step 4 of Algorithm 1 will stop eventually. On termination of the do loop, there is no bad blue vertex in the reduced G since there is no link edges in the reduced G. Therefore we can find the collection of closed components in the reduced primal graph G in step 15 of Algorithm 1 by finding all the sccs in the reduced G.

In conclusion, our algorithm correctly solves the closed partition problem. []

Theorem 2 *Algorithm 1 works in sequential time $O(n)$, where n is the number of vertices in the input compact digraph G.*

Proof: We first notice that the input graph is a plane digraph and by Euler's formula the number of edges is $O(n)$.

Steps 1-2 and 14-15 of Algorithm 1 can be performed in $O(n)$ time using known techniques. Step 3 can be implemented by applying depth first search on G^* starting from r^* to construct T. This step requires $O(n)$ time. The do loop in step 4 of Algorithm 1 consists of four substeps: find the nontrivial scc (find-scc), condense-scc, merge-tree and merge-dis.
The total execution time T is

$$T = T_{find-scc} + T_{condense-scc} + T_{merge-tree} + T_{merge-dis}$$

where $T_{find-scc}, T_{condense-scc}, T_{merge-tree}$ and $T_{merge-dis}$ are the total times needed in the above four substeps respectively during the entire execution of the do loop in step 4 of Algorithm 1. We claim that each of the times is $O(n)$.

- Claim 1: $T_{find-scc} = O(n)$
  Since all vertices in the mark-list (except r^*) will be deleted later in the condense-scc procedure and the total number of vertices in G^* is $O(n)$, Claim 1 follows.

- Claim 2: $T_{merge-tree} = O(n)$
  All edges processed in this procedure are made connected to r^* if not deleted. There are three cases:
  Case (1). An internal nontree edge incoming to r^* (back edge to r^*) will be deleted in the next condense-scc procedure.
  Case (2). An internal nontree edge (or a tree edge) outgoing from r^* will be processed in this procedure or the condense-scc procedure for at most one more time.
  Case (3). An external nontree edge incoming to r^* from a discrete vertex w^* will be processed in the merge-dis procedure for at most once and it will become an internal nontree edge incoming to r^* (which will be deleted in the condense-scc procedure at the next iteration of the do loop) if not deleted. This is because when w^* is processed in the merge-dis procedure, it is either deleted or pulled into the tree. No external nontree edge outgoing from r^* exists in T.

  Therefor an edge in G^* is processed in this procedure for at most three times and Claim 2 holds.

- Claim 3: $T_{merge-dis} = O(n)$
  We notice that all edges processed in this procedure are external edges since an end point of the edge to be merged is a discrete vertex. When an edge is processed, if it is not deleted, it either changes to a tree edge or an internal nontree edge in which case the edge will not appear in this procedure again or it becomes an external nontree edge incoming to r^* and the argument in Claim 2 case (3) can be applied here. Therefore the time spent on this procedure is $O(n)$.

- Claim 4: $T_{condense-scc} = O(n)$
  Any edge processed in this procedure is made connected to r^* if it is not deleted. We can use the same argument as in Claim 2, case (2) and (3) to prove $T_{condense-scc} = O(n)$.

Summarizing the above discussion, we have $T = O(n)$. Hence Algorithm 1 runs in $O(n)$ time.
[]

6 An Algorithm to Find the Closed Partition of a General Planar Digraph

If an embedded planar digraph G is not compact, its dual digraph G^* may have more than one nontrivial scc. We could apply Algorithm 1 repeatedly, which would give us an $O(n^2)$ time sequential algorithm for the closed partition problem on a planar digraph. The reason why we require the graph to be compact in Algorithm 1 is that there is at most one nontrivial scc in G^* which can be easily detected by examining the back edges to r^* in T. But the other operations in Algorithm 1 are still valid in the case of a general plane digraph if we choose to work on one nontrivial scc in G^* at a time. We recall that Lemma 3, Lemma 4, Lemma 6, Lemma 7, Observation 1 and Observation 2 all hold for a general plane digraph. By Lemma 4, there is still a one to one correspondence between a link edge in G and a dual edge in a nontrivial scc in G^*. The order in which we remove link edges and bad blue vertices in G is not important. Therefore we can first reduce the number of nontrivial sccs in G^* to less than $\sqrt{n}$ and then work on the nontrivial sccs in G^* one at a time.

In the following, we use Algorithm 1 to find the closed partition of any planar digraph in $O(n^{1.5})$ time.

Algorithm 2: Finding the Closed Partition of a General Planar Digraph
Input: A planar digraph $G = (V, E)$ with a subset of V marked blue. $|V| = n$.
Output: The closed partition of G with respect to the blue vertices.
1. embed G in the plane (we still call the embedding G for simplicity);
2. construct the dual digraph G^* of the embedding G;
3. **do** G is not empty and the number of nontrivial sccs in $G^* \geq \sqrt{n} \longrightarrow$
   4. delete primal edges in G whose dual edges are in nontrivial sccs of G^*;
   5. delete bad blue vertices in G that are the tails of the edges deleted in step 4;
   6. reconstruct the dual digraph G^* from the reduced primal graph G;
7. **od**;
8. **do** the number of nontrivial sccs in $G^* > 0 \longrightarrow$
   9. choose a vertex v^* in a nontrivial scc in G^*;
   {{ let v be the primal face of v^*}}
   10. identify face v as the external face in G;
   11. apply steps 1-14 of Algorithm 1 on G;
12. **od**;
13. find all the sccs in the reduced primal graph G and output the set of nontrivial sccs
end.

The correctness of Algorithm 2 follows from the informal discussion above. The details of the proof are omitted.

Step 1 can be implemented in linear time by various algorithms presented in ([5], [7], [3], [1], [9]).

Steps 2 and 13 can be implemented in linear time.

Steps 4, 5 and 6 can be implemented in linear time. Since the number of nontrivial sccs in G^* is greater than or equal to $\sqrt{n}$ at the beginning of each iteration of the do loop in step 3 of Algorithm 2, at least $\sqrt{n}$ edges are deleted during each iteration. Therefore the do loop in step 3 can be executed for at most $\sqrt{n}$ times. Each iteration of the do loop takes $O(n)$ time. Therefore the total time needed in the do loop in step 3 is $O(n^{1.5})$.

There are less than $\sqrt{n}$ nontrivial sccs in G^* at the beginning of step 8. Steps 9, 10 and 11 can be implemented in linear time. Since each time Algorithm 1 is called, the number of nontrivial sccs in G^* is reduced by at least one (see Lemma 9 below), the do loop in step 8 can be executed for at most $\sqrt{n}$ times and the total time this do loop takes is $O(n^{1.5})$.

Hence the time complexity of Algorithm 2 is $O(n^{1.5})$.

Lemma 9 *After each iteration of the do loop in step 8 of Algorithm 2, the number of nontrivial sccs in G^* decreases by at least one.*

Proof: During each iteration of the do loop in step 8 of Algorithm 2, we choose a vertex v^* in a nontrivial scc S in G^* and identify the primal face v as the external face r of G and then call Algorithm 1.

Algorithm 1 then finds the nontrivial scc S in G^* and condenses it to r^* (in steps 5-9 of Algorithm 1). The condense-scc procedure does not create a new nontrivial scc in G^* since the edges deleted in G^* in this procedure correspond to the primal edges in G whose removal combines some faces with the external face r. Furthermore, the bad blue vertices found in the condense-scc procedure are on the boundary of the external face r of G. Therefore the only possible nontrivial scc formed in G^* by deleting the edges connected to the bad blue vertices in G (in the two merge procedures of Algorithm 1) should include the vertex r^*. This newly formed nontrivial scc in G^* will be repeatedly condensed in the do loop in step 4 of Algorithm 1. Therefore at the end of each iteration of the do loop in step 8 of Algorithm 2, no nontrivial scc in G^* includes r^* and no new nontrivial scc is formed in G^*. Hence the lemma holds. []

Theorem 3 *Algorithm 2 works in sequential time $O(n^{1.5})$. []*

7 Conclusion and Open Problems

In this paper, we have presented sequential algorithms for finding the closed partition of a compact digraph and of a general planar digraph. Both algorithms use the fact that the link edges in a plane digraph G are the primal edges of the edges in the nontrivial sccs of the dual graph G^*. For a compact digraph, we used the additional fact that there is at most one nontrivial scc in

G^* if G is compact, and we designed a linear time algorithm for finding the closed partition of a compact digraph based on this fact. Using this algorithm, we gave an algorithm for finding the closed partition in a general planar digraph which runs in $O(n^{1.5})$ sequential time.

Our algorithms have a better time complexity than the algorithm known for general digraphs. A couple of open problems left by this work are:

1. Finding a linear time algorithm for the closed partition problem on a general planar digraph.

2. Improving the time bound for finding the closed partition of a general digraph.

Acknowledgment We thank Mihalis Yannakakis for drawing our attention to the closed partition problem, and the referees for helpful comments on the presentation of this material.

References

[1] Booth, K. and Lueker, G.
"Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms"
J. Comp. Syst. Sci., vol 13, 1976, p335-379.

[2] Courcoubetis, C. and Yannakakis, M.
"Verifying temporal properties of finite-state probabilistic programs"
FOCS 1988, p338-345.

[3] Even, S. and Tarjan, R.
"Computing an st-numbering"
Theoretical Computer Science, vol. 2, 1976, p339-344.

[4] Goldschlager, L. M.
" The monotone and planar circuit value problems are log space complete for P"
SIGACT News, vol. 9, 1977, p25-29.

[5] Hopcroft, J. E. and Tarjan, R. E.
"Efficient planarity testing"
J. ACM, vol. 21, 1974, p549-568.

[6] Kao, M. and Shannon, G.
"Local reorientation, global order, and planar topology"
Proc. 18th Annual ACM Symp. on Theory of Computing, 1986, p160-168.

[7] Lempel, A., Even, S. and Cederbaum, I.
"An algorithm for planarity testing of graphs"
Theory of Graphs: International Symp., Gordon and Breach, New York, NY, 1967, p215-232.

[8] Pnueli, A. and Zuck, L.
"Probabilistic verification by tableaux"
Proc. 1st Symp. on Logic in Computer Science, 1986, p322-331.

[9] Ramachandran, V. and Reif, J. H.
"Planarity testing in parallel"
Technical Report, TR 90-15, CS Dept, UT Austin, 1990;
Preliminary version appears as *"An optimal parallel algorithm for graph planarity"*
Proc. 30th Ann. IEEE Symp. on Foundations of Comp. Sci., 1989, p282-287.

[10] Tarjan, R. E.
"Data structures and network algorithms"
SIAM, Philadelphia, PA, 1983.

[11] Tarjan, R. E.
"Depth-first search and linear graph algorithms"
SIAM J. Comp. 1:2, p146-160.

[12] Vardi, M.
"Automatic verification of concurrent probabilistic finite state programs"
Proc. 26th Symp. on Foundations of Computer Science, 1985, p327-338.

[13] Yannakakis, M.
" Private communication"
June 1990.

A Appendix: An Example to Illustrate Algorithm 1

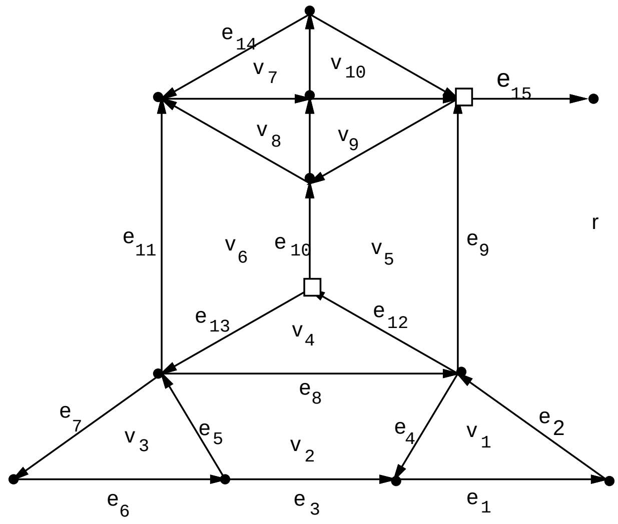

Figure 5: A compact digraph G, where the e_i's are edges and the v_i's are faces. The face r is the external face of G. A square node represents a blue vertex.

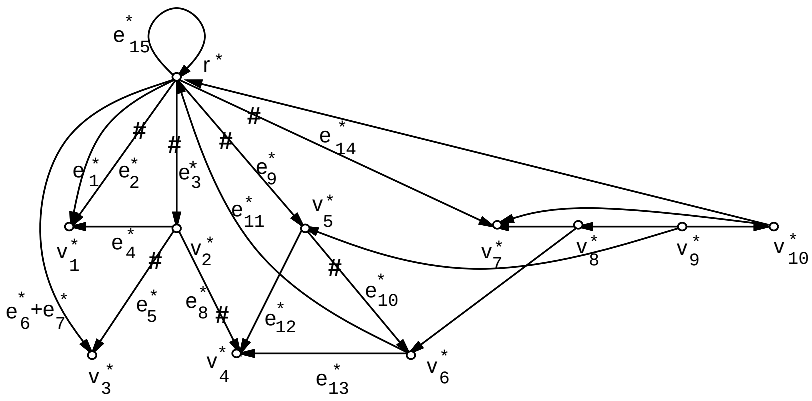

Figure 6: The dual digraph G^* of G with r^* being the dual vertex of r. The edges marked with # are tree edges of the maximal divergent tree T rooted at r^*.

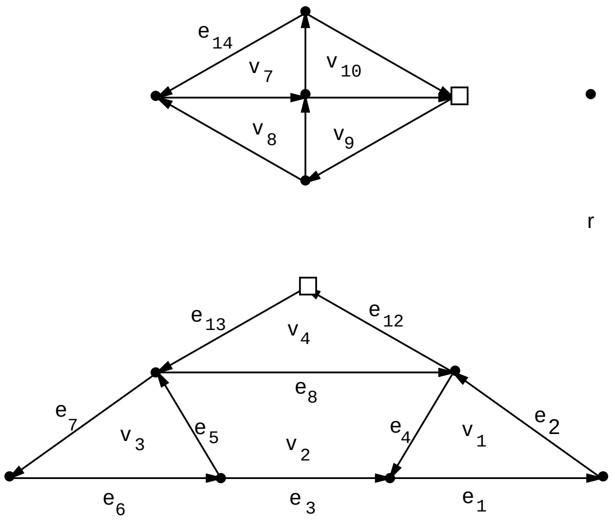

Figure 7: Delete link edges e_9, e_{10}, e_{11} and e_{15} in G.

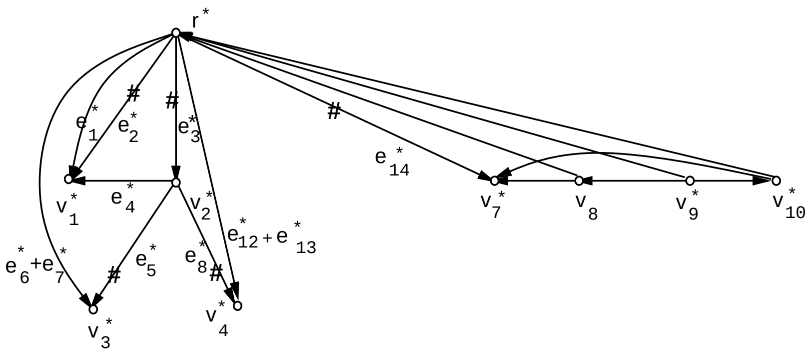

Figure 8: Condense the nontrivial scc (r^*, v_5^*, v_6^*) to r^* in G^*.

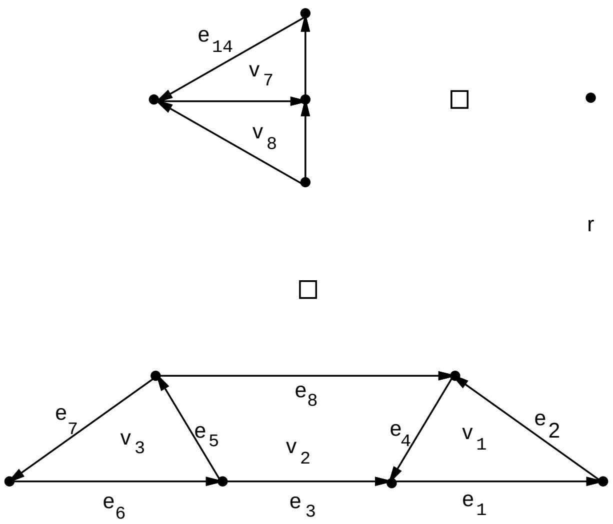

Figure 9: Delete the edges connected to the two bad blue vertices in G.

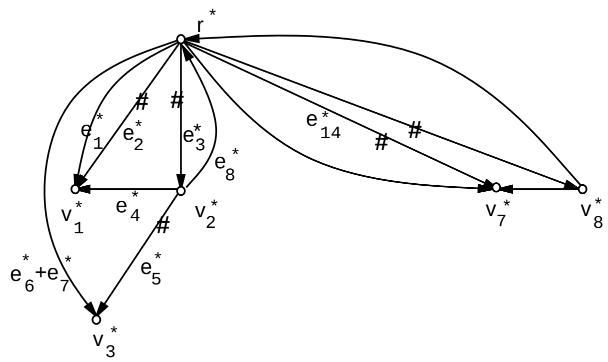

Figure 10: Merge v_4^*, v_9^* and v_{10}^* to r^*. The vertex v_8^* becomes a tree vertex of T.

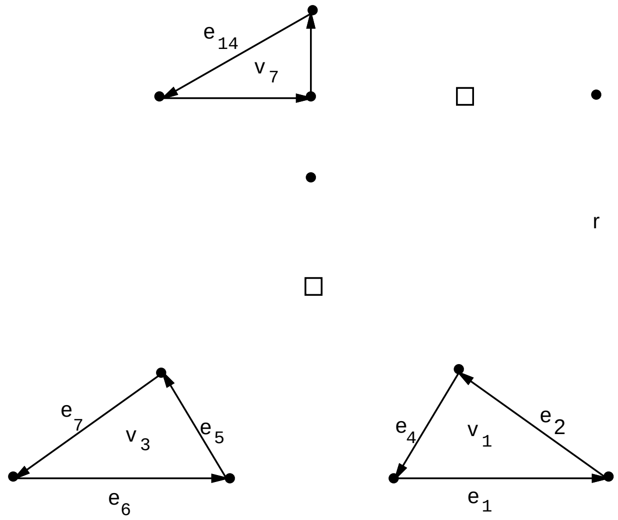

Figure 11: The primal graph G when the do loop in step 4 of Algorithm 1 terminates.

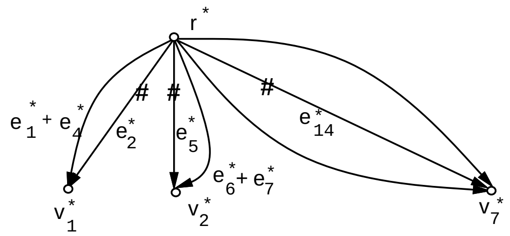

Figure 12: The dual graph G^* after the nontrivial scc (r^*, v_8^*, v_2^*) is condensed to r^*.