A linear-time algorithm for finding induced planar subgraphs

Shixun Huang, Zhifeng Bao and J. Shane Culpepper
RMIT University, Australia

Ping Zhang
Wuhan University, China

Bang Zhang
DATA61 — CSIRO, Australia

Abstract

In this paper we study the problem of efficiently and effectively extracting induced planar subgraphs. Edwards and Farr proposed an algorithm with $O(mn)$ time complexity to find an induced planar subgraph of at least $3n/(d+1)$ vertices in a graph of maximum degree d. They also proposed an alternative algorithm with $O(mn)$ time complexity to find an induced planar subgraph graph of at least $3n/(\bar{d}+1)$ vertices, where $\bar{d}$ is the average degree of the graph. These two methods appear to be best known when d and $\bar{d}$ are small. Unfortunately, they sacrifice accuracy for lower time complexity by using indirect indicators of planarity. A limitation of those approaches is that the algorithms do not implicitly test for planarity, and the additional costs of this test can be significant in large graphs. In contrast, we propose a linear-time algorithm that finds an induced planar subgraph of $n-\nu$ vertices in a graph of n vertices, where ν denotes the total number of vertices shared by the detected Kuratowski subdivisions. An added benefit of our approach is that we are able to detect when a graph is planar, and terminate the reduction. The resulting planar subgraphs also do not have any rigid constraints on the maximum degree of the induced subgraph. The experiment results show that our method achieves better performance than current methods on graphs with small skewness.

2012 ACM Subject Classification G.2.2 Graph Theory, G.2.3 Applications

Keywords and phrases induced planar subgraphs, experimental analysis

Digital Object Identifier 10.4230/LIPIcs.SEA..

1 Introduction

A graph is *planar* if it admits a *planar drawing* which means that the graph can be drawn on the plane such that its edges only intersect at their endpoints. The goal of the graph planarization problem is to find a planar subgraph by removing edges or vertices from an input graph. It can be applied in many areas, such as facility layout design [8], circuit design [18], graph drawing [15], and automated graphical display systems [28]. One popular reformulation of the graph planarization problem, called the Maximum Induced Planar Subgraph (MIPS) problem, aims to find the largest number of vertices which induce a planar subgraph. This problem is known to be NP-hard, and also surprisingly hard to approximate [19, 23, 26]. The MIPS problem can also be used to compute the coefficient of fragmentability of a class of graphs, which is the proportion of vertices necessary to produce subgraphs of a bounded size [11].

Related Work. In this paper, we study the MIPS problem and we assume that the reader is familiar with basic graph theory (see for example [13, 30]). No graphs being considered contain edge loops, n denotes the number of vertices, m denotes the number of edges, d denotes the maximum degree, and $\bar{d}$ denotes the average degree in a graph.

Halldórsson and Lau [14] proposed a linear-time algorithm (denoted as HL) for the MIPS problem with a performance ratio of $1/\lceil(d+1)/3\rceil$ in a graph G. They presented several practical algorithms for partitioning graphs into a fixed number of vertex-disjoint subgraphs with degree constraints. In order to solve the problem, they capitalize on the Lovász [25] Theorem: Let $a_1, a_2, ..., a_k$ be non-negative integers such that $\sum_{i=1}^{k}(a_i+1)-1=d$. Then G can be partitioned into k induced subgraphs $G_1, G_2, ..., G_k$ such that the maximum degree of G_i is not greater than a_i. With this theorem, a graph can be partitioned into at most $\lceil(d+1)/3\rceil$ induced subgraphs of degree at most 2, and the largest subgraph is the planarized result. The approach of Halldórsson and Lau can induce such a partition in linear time. However, the maximum degree of the planarization result is restricted to be at most 2.

Edwards and Farr [10] proposed an algorithm (denoted as Vertex Addition) to find an induced planar subgraph of at least $3n/(d+1)$ vertices in $O(mn)$ time, which has a performance ratio of at least $3n/(d+1)$. Compared to the algorithm of Halldósson and Lau, the performance ratio is improved when $d \not\equiv 2 \pmod 3$. The induced planar subgraphs found by this algorithm is also not constrained to have maximum degree of 2. The algorithm works as follows. Suppose that P is an initially empty set and $R = V(G)\backslash P$, this algorithm works by adding vertices from R one by one into P while maintaining the planarity of $\langle P\rangle$. In some instances, a vertex from R is swapped with one from P. The restrictions on the swapping operations are stricter than that on maintaining planarity. By doing this, some properties in the graph are maintained, which allows the performance of the algorithm to be analyzed. For further information, please see [11, 10]. This leads to a fact that sometimes it still swaps some vertices even if planarity could be maintained when all vertices involved in the swapping operations are included in the planarization result.

Edwards and Farr [11] propose another algorithm (denoted as Vertex Removal) with time complexity $O(mn)$ for the MIPS problem in a graph of $\bar{d}$, which achieves a performance ratio of at least $3n/(\bar{d}+1)$ when $\bar{d} \geq 4$ or a graph is connected and $\bar{d} \geq 2$. This algorithm begins by removing any isolated vertex, any vertex of degree 1, and any vertex of degree 2. For a vertex of degree 2, if its neighbours are not adjacent, they are joined by an extra edge. Then a *reduced graph* is obtained by repeating the operations above until no further changes are possible. Then, it proceeds to remove the vertex of the highest degree in the reduced graph iteratively. In order to reduce complexity and improve efficiency, this algorithm avoids the planarity test in each iteration. Instead, a loose upper bound of the number of vertices to be removed is computed, which can result in the algorithm continuing to remove vertices until the upper bound is reached, regardless of whether the current result is already planar or not. Morgan and Farr [27] later proposed a modified algorithm (denoted as Vertex Subset Removal) which instead iteratively removes a vertex v with the largest number of neighbors with degree less than the degree of v in the reduced graph. There is no known investigation of the impact of the different vertex removal strategies on the planarization results [24].

There is a simple example made by Morgan and Farr [27], which roughly summarizes a limitation shared by all methods mentioned above — all previous methods fail to leave K_4 minors in the induced planar subgraph even though K_4 is already planar. For a K_5 graph, they can only find an induced planar subgraph of size at most degree 3.

Preliminaries.. We need to review key definitions before introducing our contributions and the rest of content. According to Kuratowski [21], a graph is planar if and only if it does not contain a *Kuratowski subdivision* of K_5 (a complete graph of size 5) or of $K_{3,3}$ (a complete bipartite graph of size 6). A *subdivision* of a graph G is a graph resulting from the subdivision of edges in G. The subdivision of an edge e with endpoints (u, v) yields a graph containing one new vertex w, with an edge set replacing e by two new edges, (u, w) and (w, v). The *skewness* of a graph is the minimum number of edges whose removal results in a planar graph [6].

A graph is a *nearly planar graph* if it is a k-graph (i.e., it has at most k edge crossings) or a *k-skewness graph* when k is small. Several previous studies have studied graphs with similar properties in the context of straight-line drawing [17], visualization [12, 9, 7], and edge intersection [5]. Applications may also require non-planar graphs to be drawn on a plane even if edge crossings cannot be avoided [1, 8]. So it is naturally desirable to draw graphs as close to planar as possible. We therefore focus on these cases in our experiments.

Contributions and Outline. We present an algorithm including planarity test to solve the MIPS problem that does not remove any additional vertices once the graph becomes planar. An additional benefit of our approach is that the maximum degree of the planarization result is not constrained, which overcomes some of the limitations of previous work in this area. The algorithm runs in $O(n + m + E(S))$ time, with S being the set of detected Kuratowski subdivisions and $E(S)$ being the sum of number of edges of such subdivisions. The time complexity is linear w.r.t. $E(S)$, and graph size. The induced planar subgraph produced by our algorithm is of size $n - \nu$, with ν being the total number of vertices shared by the Kuratowski subdivisions detected. We conduct several experiments to show that our method outperforms all other methods for graphs with small skewness.

In Section 2, we first introduce a planarity test algorithm which is the basis of our approach. Next, we describe our planarization algorithm and proofs of correctness. In Section 3, we conduct intensive experiments on real-world graphs. Finally, we conclude this paper in Section 4.

2 Our Approach and Its Basis

Planarity Testing. Our work is based on one of the most efficient planarity test algorithms, originally presented by Boyer and Myrvold [3]. For more detailed information, please refer to the original work [3]. The algorithm (denoted as DETECT) works by checking if a graph produces a *planar drawing*. DETECT begins by creating a depth first search tree (DFS tree) of the graph. Each vertex is assigned to a depth first index (DFI), and edges are divided into tree edges forming the DFS tree and backedges (the remaining edges). In this paper, let v be the vertex currently being processed and $\bar{G}$ be the plane for embedding the graph. Initially, the DFS tree is embedded in $\bar{G}$. DETECT processes vertices in descending DFI order. In each iteration of v, DETECT attempts to embed each backedge (u, v), where u has a larger DFI than v, while maintaining planarity.

Initially, each tree edge (z, v) is represented as an independent biconnected component (z, v^z) formed by the vertex with a largest DFI and a *virtual vertex*. The vertex z is a DFS child of v, and we denote this virtual vertex as v^z to distinguish it from other copies of v. We use v' to denote a virtual vertex whose child is not specified. We say that the vertex v^z is the *root* of this biconnected component. Biconnected components are merged to form a larger subgraph when backedges are embedded.

Each iteration involves two processes: a WALKUP and a WALKDOWN. A WALKUP identifies relevant biconnected components for a backedge embedding, and classifies vertices as follows. A vertex w is *pertinent* if there is a backedge (w, v) to be embedded, or it has a child biconnected component in $\bar{G}$ which contains a pertinent vertex. A backedge (w, v) is *pertinent* if w is pertinent and w is marked with an `EdgeFlag`. A biconnected component is *pertinent*

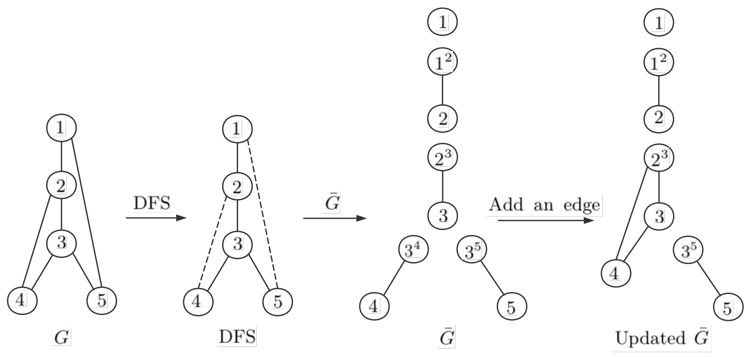

Figure 1 An example illustrating the DETECT algorithm.

if it contains a pertinent vertex. A vertex w is *external* if there is a backedge (w, u) to be embedded later, where u has a smaller DFI than v, or it has a child biconnected component in $\bar{G}$ which contains an external vertex. Each vertex is equipped with a `PertinentRoots` list that stores the roots of its pertinent child components. For every backedge (w, u), WALKUP traverses from w to u along the paths on the external face of the biconnected components.

A WALKDOWN then embeds pertinent backedges and merges relevant biconnected components traversed by WALKUP. The process is initiated with two traversals for each biconnected child component rooted by a virtual point v': one in the clockwise direction along the external faces of the biconnected child component, and a second one in the opposite direction. When WALKDOWN reaches a pertinent vertex u with an `EdgeFlag`, the relevant components are merged, and the backedge (u, v) is embedded. The process continues until reaching an external but non-pertinent vertex (denoted as *stopping vertex*), or v' is found again. This is the *halting condition* for the algorithm, and is the only possible indicator of non-embeddability of backedges. If a pertinent backedge cannot be embedded, the graph is not planar, as DETECT has identified a Kuratowski subdivision.

A Linear Time Solution for The MIPS Problem. DETECT terminates when a pertinent backedge exists that is not embedded due to a stopping vertex s – meaning the graph is non-planar. The reason is that if the algorithm were to proceed to embed an edge after passing s, then s cannot remain on the outer face of the graph. The embedding of a backedge (s, u) in a later iteration would result in intersecting edges, which cannot admit a planar drawing.

Let s be a stopping vertex, and the *influenced region* of s be the collection of paths which can be visited by a WALKDOWN only after it has visited s. The vertex v being processed is an *obstruction vertex* if there exists at least one pertinent unembedded backedge. We observe that a stopping vertex s only influences the embedding of a backedge (w, v) when w is in the influenced region of s. We therefore propose the algorithm PLANARIZATIONBYREGIONSKIP which embeds all possible pertinent backedges in each iteration by skipping influenced regions of stopping vertices encountered during the WALKDOWN. When the embedding process is completed, all obstruction vertices are removed from the input graph, which produces an induced planar subgraph. This observation produces an algorithm which can test for planarity and produce a solution for the MIPS problem simultaneously.

Solution Overview. Algorithm 1 presents our solution for finding an induced planar subgraph. It begins by building a DFS tree, and initializing the embedding structure $\bar{G}$ (line 1 to 2). Then we use an `ObstructionsList` to store the indexes of obstruction vertices in an adjacency list. Each element is initialized to -1 (line 3). Next, the embedding loop is initiated (line 4 to 12). Obstruction vertices identified in each iteration are excluded from

Algorithm 1: PlanarizationByRegionSkip(G)

```
    Input    : A graph G
    Output : An induced planar subgraph $P$
 1  Construct a DFS tree of $G$ ;
 2  Initialize the embedding structure $\bar{G}$;
 3  Initialize ObstructionsList ;
 4  for each vertex $v$ in descending DFI order do
 5      foreach backedge $(w, v)$ of $G$ where $w < v$ do
 6          if $w$ is not an obstruction then
 7              WalkUp($\bar{G}$, $v$, $w$);
 8      foreach DFS child $c$ of $v$ in $G$ do
 9          WalkdownWithSkips($\bar{G}$, $v^c$);
10      foreach back edge $(w, v)$ of $G$ where $w < v$ do
11          if $(w, v)$ not in $\bar{G}$ then
12              ObstructionsList[$v.index$] ← $v.DFI$;
13  graph $P$ ← RemoveObstructions(ObstructionsList, $G$);
14  return $P$;
```

the the Walkup process (line 7). Details of Walkup are described in previous work [3].

In the WalkDownWithSkips, traversals for each biconnected child component rooted by the virtual point v^c are initiated. This process embeds all of the pertinent backedges which are not influenced by stopping vertices with skipping operations over the influenced regions (line 9). Then, v is added into the `ObstructionsList` if there exists an unembedded pertinent backedge. When the main loop finishes, all obstruction vertices are removed from the graph.

The WalkdownWithSkips algorithm. As mentioned earlier, in order to embed backedges that are not influenced by stopping vertices, we need to perform skipping operations. The process WaldownWithSkips terminates when it reaches v' or a stopping vertex on the component whose root is v'. We denote such a component as a *root component*. If the stopping vertex encountered is not on a root component, a skipping operation needs to be performed. When a traversal descends from vertex r to root vertex r' of a non-root component, it needs to choose a direction to proceed. Boyer and Myrvold [3] proposed `short circuit edges` which enable r' to be directly connected to neighbors such that they are either pertinent or a stopping vertex. Each `short circuit edge` is embedded in a previous iteration p between p' and the stopping vertex. This forms a new face such that interceding inactive vertices are removed from the external face. For more detailed information, please refer to Boyer and Myrvold [3]. For our purposes, when the WalkdownWithSkips encounters a stopping vertex, it checks if another neighbor of r' is not a stopping vertex. If so, it skips to this neighbor. Otherwise, it skips the components rooted by r' which is then deleted from the `PertinentRoots` of r, and returns to the parent component. The algorithm terminates on the stopping vertex on the root component since there does not exist a parent component for the process to ascend to.

Algorithm 2 describes the rationale of the WalkdownWithSkips. The algorithm begins a single traversal in a clockwise or counterclockwise direction (line 2). Let w be the next successor along the external face. If w has an `EdgeFlag`, the backedge (w, v^c) is embedded after the relevant components are merged (line 4 to 7). Then the traversal proceeds to the successor. When it encounters a pertinent vertex whose `PertinentRoots` list is not empty, it descends to the component rooted by the first element r in the list, and visits one of its

Algorithm 2: WalkdownWithSkips($\bar{G}, v^c$)

```
Input : The embedding structure $\bar{G}$ and a virtual vertex $v^c$.
1  foreach traversal from $v^c$ do
2      $w \leftarrow$The successor along the external face;
3      while $w$ is not $v^c$ do
4          if $w$ has an EdgeFlag then
5              Merge involved components;
6              Embed the backedge $(w, v^c)$ and clear $w$'s EdgeFlag;
7              $w \leftarrow$ The successor along the external faces;
8          if $w$.PertinentRoots is not empty then
9              $r \leftarrow w$.PertinentRoots[0];
10             Traverse down to the component rooted by $r$;
11             $w \leftarrow$ The successor along the external faces;
12         if $w$ is a stopping vertex then
13             if $w$ is on the root component then
14                 Embed the Short Circuit Edge $(w, v^c)$;
15                 break;
16             else
17                 $x \leftarrow$ Another neighbor of $r$, the root of the current component;
18                 if $x$ is a stopping vertex then
19                     Skip the components rooted by $r$;
20                 else $w \leftarrow x$;
21         else $w \leftarrow$ The successor along the external faces;
```

neighbors (line 8 to 11). If w is a stopping vertex in the root component, a `short circuit edge` is embedded, and the traversal stops, after which another traversal is initiated from v^c in the opposite direction (line 13 to 15). Otherwise, the traversal performs a skip based on whether another neighbour of r is a stopping vertex (line 17 to 20).

A Running Example of the Embedding Process. It is instructive to see an example of the embedding process on the pertinent subgraph in an iteration of c. The Walkup process is invoked for each vertex with an `EdgeFlags`. Two parallel traversals are started from each vertex and stops when either of them reaches the root of the current biconnected component. Afterwards, Walkup starts another two parallel traversals on the parent components. The process terminates when a traversal reaches c or a vertex that has been visited before is encountered.

Figure 2 shows the process of the Walkup of an example set of biconnected components (diamonds), external and pertinent vertices (dashed squares), stopping vertices (solid squares) and pertinent vertices with `EdgeFlags`. Only the traversals that reach root vertices first are shown. Walkup begins at f. When it reaches d^e, d^e is then added to the `PertinentRoots` of d, and it starts traversals at d until reaching c. Then the Walkup traversals of i are initiated. The vertex g^h is added to the `PertinentRoots` of g after being visited. When it reaches g, the traversal terminates since g has been visited before. The main purpose of Walkup is to determine which components are involved in the embedding. Hence the traversals initiated from i do not have to continue. This process is repeated until all vertices with `EdgeFlags` have all done a Walkup.

The main purpose of the WalkdownWithSkips is to embed as many pertinent backedges as possible by skipping the influenced regions of the stopping vertices, and identify if the

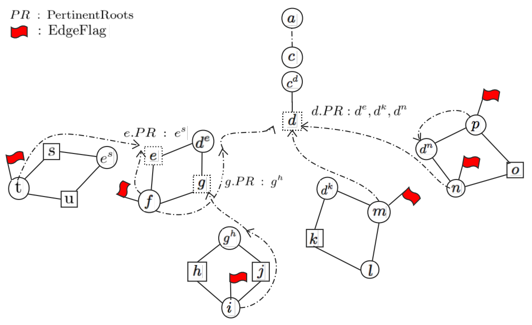

Figure 2 An example of the Walkup.

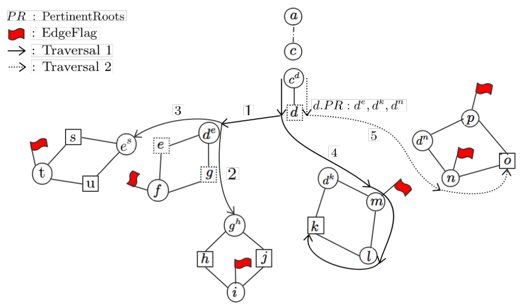

Figure 3 The WalkdownWithSkips in the iteration of c.

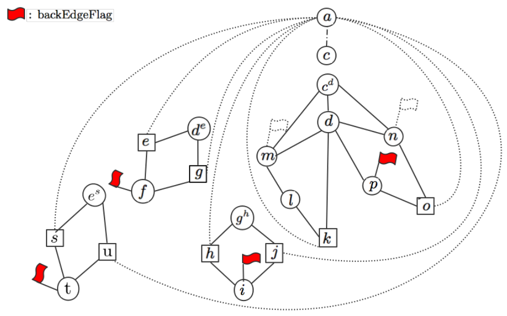

Figure 4 The status after the embedding.

vertex being processed is an obstruction. Figure 3 describes WalkdownWithSkips on the same example set of biconnected components as Walkup. A traversal starts in one direction of the child component of c. Then it descends to the component rooted by d^e which is the first element in the `PertinentRoots` of d. Which direction to go from d^e depends on the types of the neighbors: a neighbor can be pertinent but not external, pertinent and external, or a stopping vertex (external but non-pertinent). The direction towards the neighbor of the first type is preferred, and the direction towards a stopping vertex will be chosen if no neighbors of the first two types exist. Since both e and g are of the same type, a neighbor g is randomly selected.

Afterwards, the algorithm descends to the component rooted by g^h which is deleted from the `PertinentRoots` of g, and then returns to g because it cannot reach i without passing a stopping vertex. Next a skip to another neighbor e of d^e is initiated, and the process is repeated. Since external and pertinent vertices are stopping vertices once their `PertinentRoots` lists become empty, the traversal cannot reach f. So the components rooted by d^e are skipped and deleted from `PertinentRoots` of d, and processing returns to d. Then the algorithm descends to the component rooted by the next element d^k in the `PertinentRoots` of d. The backedge (m, c^d) is embedded and the relevant components rooted by c^d and d^k respectively are merged. The traversal terminates at k after embedding the `short circuit edge` (k, c^d) since k is on the root component after merging operations. Another traversal begins from c^d, and the process is repeated until a termination condition is reached.

Figure 4 shows the embedding for this example. The dashed edges refer to backedges which will be embedded by the algorithm. The vertices which still have `EdgeFlags` correspond to unembedded backedges in the last iteration. As we can see, the component rooted by c^d is larger after merging, and backedges (m, c^d) and (n, c^d) have been embedded. The embeddings of any other backedges in this figure would result in an intersection with the dashed edges. Thus, they cannot be embedded. Since there exist unembedded backedges, the vertex c is added to the `ObstructionsList`.

Removing Obstruction Vertices. After the main loop of the embedding process, the obstruction vertices are collected, which need to be removed from the graph to induce the planar subgraph (line 13 in Algorithm 1). Note that we use the terms element and vertex interchangeably henceforth. The input graph is represented as an adjacency list, which is a collection of vertex lists. The first element in each vertex list is adjacent to the rest of the elements. Each element denotes an index of a vertex. We denote a vertex list E as a list of e_1 if the vertex e_1 is the first element in E.

As discussed in Section 2, each element of the `ObstructionsList` refers to the index of a vertex in the adjacency list, and is initialized to -1. Since we assume that all index values are non-negative, after the main loop, we can identify which vertex is an obstruction based on whether the content of the corresponding element is non-negative. For each vertex list of e_1 where `ObstructionsList[`e_1`]` ≥ 0, we just remove them directly from the adjacency list. For each vertex list of e_1 where `ObstructionsList[`e_1`]` ≤ 0, we process each of the rest elements e_i in the vertex list of e_1 by checking `ObstructionsList[`e_i`]`. If `ObstructionsList[`e_i`]` ≤ 0, we leave this element and process the next one. Otherwise, this element is deleted from this vertex list and we continue processing. After all vertex lists have been processed, the adjacency list is an induced graph where each obstruction vertex o (`ObstructionsList[o]` ≥ 0) has been removed. The overall cost includes the $O(n)$ vertex lists, and the total number of elements in vertex lists are $O(m)$. Since each element is processed in $O(1)$, the total time complexity is $O(n + m)$.

Proof of Correctness. In this section, we prove that the induced subgraph found by our algorithm is planar and has a linear time complexity.

▶ **Lemma 1.** *Given a graph G, the main embedding loop finds a planar subgraph of G.*

Proof. Boyer and Myrvold [3] have proved that, in the iteration of v, Kuratowski subdivisions will occur if and only if the WALKDOWN passes stopping vertices to embed backedges. Since the embedding process works by skipping the influenced regions of stopping vertices in each iteration, any Kuratowski subdivision cannot exist in the graph. Kuratowski [21] proved that a graph is not planar if and only if it contains a Kuratowski subdivision. The embedding loop preserves planarity since no Kuratowski subdivisions exist in the graph. ◀

▶ **Theorem 2.** *Given a graph G, removal of obstruction vertices leads to an induced planar subgraph of G.*

Proof. Although the graph is already planar after the embedding process, it is not an induced graph since we only remove certain edges. In order to have an induced planar subgraph, we need to remove one of two endpoints of each removed edge. Since all removed edges were connected to obstruction vertices, the removal of such vertices leads to an induced planar subgraph. ◀

▶ **Theorem 3.** *Given a graph G with n vertices and m edges, our algorithm is bounded by $O(n + m + E(S))$ and therefore linear.*

Proof. The construction of the DFS tree can be accomplished in linear time with a well-known algorithm [29]. The initialization of the embedding structure $\bar{G}$ and the `ObstructionsList` is also a linear time process. During the main backedge embedding loop, embedding edges runs in linear time since the cost of embedding each edge is $O(1)$. If the input graph is planar, the cost of WALKUP is bounded by the faces formed by embedded edges. The faces formed by embedded backedges and short circuit edges bound the cost of the WALKDOWNWITHSKIPS. Thus the cost of WALKUP and WALKDOWN is linear since the size of the faces is at most twice the number of edges in the graph. So, each edge can only traversed at most two times throughout the entire embedding loop. However, if the input graph is not planar, the cost of WALKUP and WALKDOWN cannot be bounded by the faces formed by backedges since some of the backedges are unembedded in order to preserve planarity. This means that some edges along the external faces of the graph are traversed multiple times before new external faces are formed, which then include these edges in the internal faces. Such an edge is traversed at most k times where k denotes the number of Kuratowski Subdivisions which contain this edge. If S is a collection of all Kuratowski Subdivisions detected in the entire embedding process, and $E(S)$ denotes the size of subdivisions in S, then our algorithm runs in $O(n + m + E(S))$ time, which is output sensitive and linear w.r.t. $E(S)$ and the graph size. ◀

3 Experimental Evaluation

In this section, we compare our PLANARIZATIONBYREGIONSKIP algorithm (RS) with baselines mentioned in Section 1: HL [14], Vertex Addition (VA) [10], Vertex Removal (VR) [11], and Vertex Subset Removal (VSR) [27]. All baselines were implemented by Morgan and Farr [27], and are publicly available. Morgan and Farr [27] also proposed additional algorithms for the MIPS problem. We have selected the subset of algorithms listed above for the following reasons: 1. VR is best kown for average degree d, and it achieved second best accuracy in the original work [27]. 2. VSR, as a modified algorithm of VR, has the same approximate ratio as VR, and achieved the best accuracy previously. 3. VA is besk known for maximum degree $\bar{d}$. 4. HL has linear-time complexity and was the most efficient. Note that we do not include the EPS algorithm as it is a post-processing enhancement [27]. This operation can be applied to the planarization result of any of the algorithms explored in this work to improve the approximation ratio further.

All codes are written in C, compiled using GCC 4.2.1, and are available online[1]. All experiments are performed on a machine with two Intel Core i5 (2.6 GHz) and 8 GB RAM. In this section, we use the term **Percentage** to describe how many vertices from the input graph are retained in the planarization result. We conduct experiments on real-world graphs collected from KONECT [20] and SNAP [22]. Table 1 summarizes the basic properties of the datasets.

Experiments on graphs with small skewness. In this section, we conduct experiments on two datasets: *RT5000* which contains 5000 vertices from *RT*, and *RD100000* which contains 100000 vertices from *RD*. We construct the graphs with increased skewness by randomly inserting edges between existing vertices. We insert edges up to 0.1% of the input

[1] https://github.com/rmitbggroup/GraphPlanarization

Dataset	n	m	Description
RT	1,379,917	1,921,660	The planar road network of Texas.
RD	1,088,092	1,541,898	The planar road network of Pennsylvania.
IN	26,475	53,581	Non-planar network of autonomous systems in the CAIDA project.
PG	10,680	24,316	A non-planar social network of the Pretty Good Privacy algorithm.
UG	4,941	6,594	The non-planar power grid network of the Western States in US.
MP	212	244	A non-planar network of protein-protein interactions from PDZBase.

Table 1 Basic properties of the test collections.

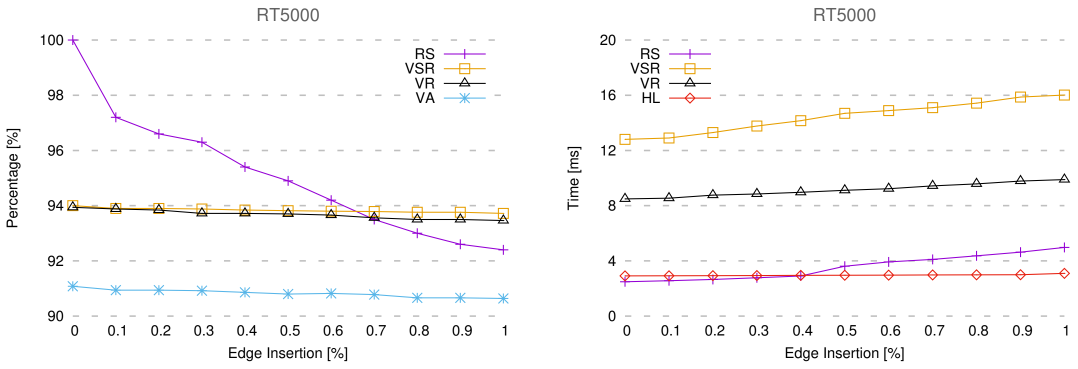

Figure 5 Experiments on RT5000 with edge insertion. The left figure shows the percentage achieved by different algorithms (HL achieves 35% on average, and is not shown on the graph). The right figure shows the running time of different algorithms (VA requires 24,888 ms on average, and is not shown).

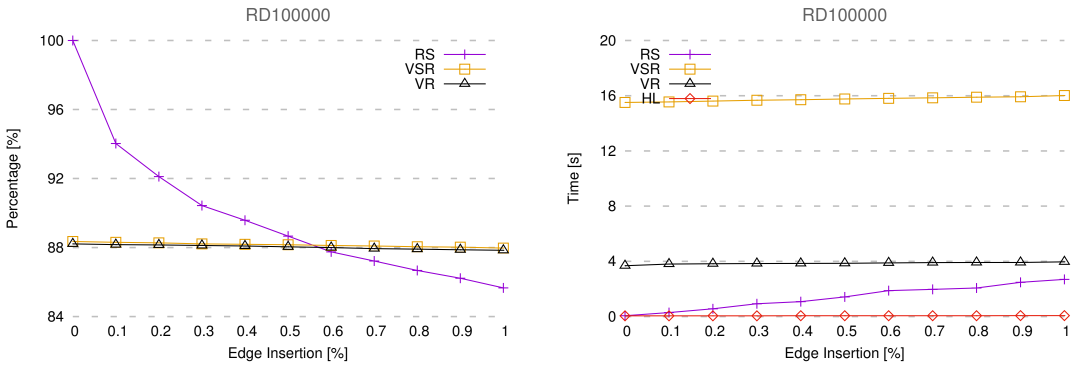

Figure 6 Experiments on RD100000 with edge insertion excluding VA. The left figure shows the percentage achieved by the algorithms (HL achieves 34% on average, and is not shown). The right figure shows the running time achieved by all of the algorithms.

graph size. Figure 5 shows the experimental results on *RT5000*. As we can see, even if the graph is already planar (no edge insertions), only RS achieves a percentage of 100%. All other methods remove vertices based on the requirements of their corresponding indirect indicators of planarity. With incremental edge insertions, the performance of RS can vary significantly since each inserted edge may introduce multiple Kuratowski subdivisions. This behavior also indicates that the performance of RS is related to the direct indicator of planarity. On the other hand, the performance of other methods do not change much since a small number of edge insertions do not change the size of graph in any meaningful way. VSR only achieves around 0.2% percentage more than VR on average. In term of efficiency, the running time

Vertex Increase (%)	Percentage (%)				Time (s)			
	RS	VSR	VR	HL	RS	VSR	VR	HL
10	99.99	88.29	88.13	34.28	0.07	17.61	4.51	0.06
20	99.99	89.13	88.97	34.31	0.11	67.03	16.64	0.12
30	99.99	89.28	89.14	34.33	0.20	148.58	34.63	0.18
40	99.99	89.95	89.75	34.35	0.28	272.61	62.98	0.22

Table 2 Experimental results on almost planar graphs using the RD dataset.

of RS grows linearly, which indicates that its performance is linearly associated with the Kuratowski subdivisions detected in the graph since the graph sizes are similar. Figure 6 shows the experiment results on *RD100000*, and produces similar observations. As the graph size increases, the superiority of the efficiency of HL becomes more pronounced.

We also perform experiments on almost planar graphs which belong to a special class of graphs with skewness equal to one [16]. Previous studies working on almost planar graphs have taken a similar approach [16, 2, 4]. We construct almost planar graphs based on the *RD* dataset. The number of vertices of those graphs range from 10% to 40% of *RD*. As we can see in Table 2, RS always achieves a percentage of 99.99%, which corresponds with the definition of almost planar graphs. The percentage achieved by other methods are all below 90%, and are sensitive to graph size, which reflects the over-reliance on the indirect indicators of planarity used by these methods. The average running time of HL and RS are 0.15 s and 0.17 s respectively. Performance of RS varies little since Kuratowski subdivisions are rarely introduced, and this is the main property which affects its performance. On average, RS is 150 times faster than VR and 640 times faster than VSR.

Experiments on non-planar graphs. In this section, we explore the performance on real-world non-planar graphs: *MP*, *UG*, *PG* and *IN*. Based on each graph, we construct graphs by removing a certain percentage of vertices from the original graphs in descending order of the maximum degree. When the skewness of the input graph is not small, VR and VSR tend to perform well since they iteratively remove a vertex with the maximum degree, and this has the same effect as removing multiple Kuratowski subdivisions at once. On the other hand, RS consistently achieves a local optima by removing the obstruction vertex shared by Kuratowski subdivisions detected in each iteration. Due to limited space, we use only *MP* and *IN* to demonstrate this effect. Additional results are in the Appendix.

Figure 7 shows the results on the dataset *MP*. RS always achieves the best percentage, and reaches 100% when the vertex removal rate is 3.5%. Other methods cannot achieve 100% even though the graph is already planar. The performance of VR and VSR are almost the same. VSR achieves at most 0.001% more than VR. Methods such as VA and VR exhibit higher efficiency than HL, and VA is more efficient than RS in many cases. The reason is that the graph size is so small that these methods converge very quickly. For example, the reduced graph mentioned in Section 1 is so small that VR only has to remove a few vertices from the graph. On this dataset, RS and VSR outperforms all other approaches if both accuracy and efficiency are considered.

Figure 8 shows experimental results on *IN*. Initially, RS achieves 1.2% less than VSR and 0.3% less than VA. As the percentage of vertex removals increases, the gap between RS and VSR is narrowed and RS outperforms all other methods when 2% of vertices are removed. A higher percentage indicates a higher vertex removal rate, which also indicates a smaller graph size. From the right figure, it is worth noting that there is a rapid change of efficiency of VR and VSR when the vertex removal rate increases to 0.2%. The increased cost in VR and VSR are caused by the iterative removal of maximum degree vertices. Since, we have already removed vertices of the maximum degree before the algorithms are initialized, their costs

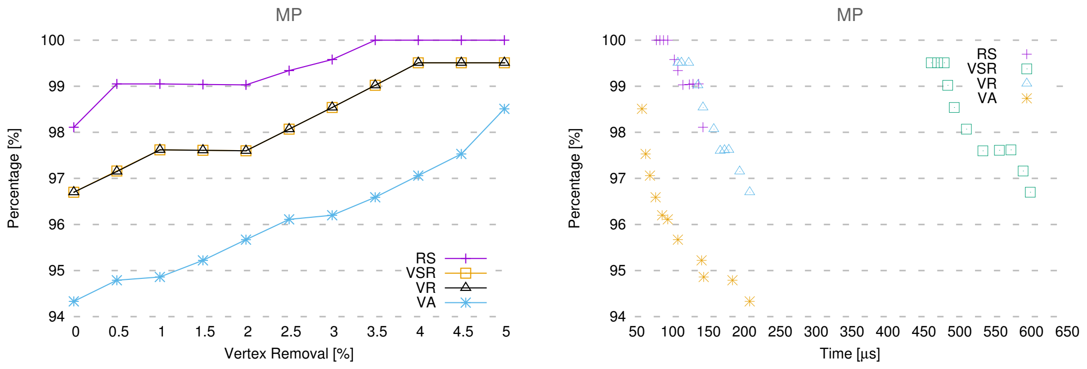

Figure 7 Experiments on *MP* with vertex removal. The left figure shows the percentages achieved by different algorithms (HL achieves only 25% on average and is not shown). The right figure shows the Efficiency / Effectiveness relationship (the running time of HL is 356 ms on average and not shown).

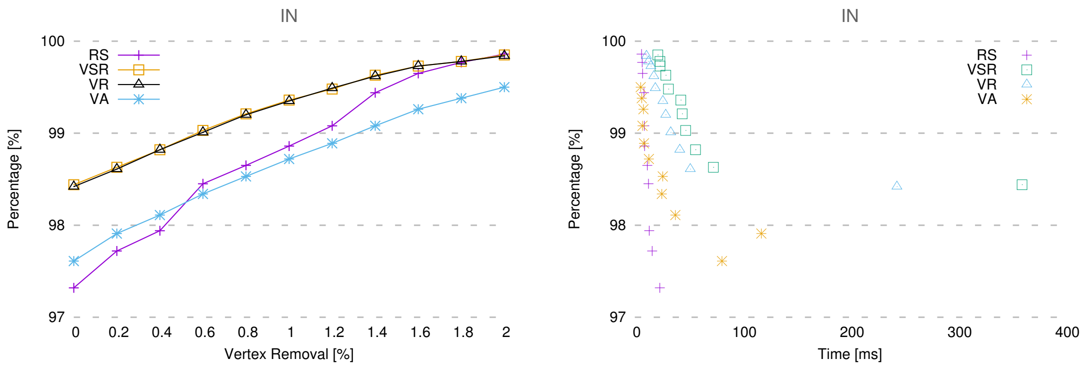

Figure 8 Experiments on *IN* with vertex removal. The left figure shows the percentages achieved by different algorithms (HL achieves 9% on average and is not shown). The right figure shows the Efficiency / Effectiveness relationship (the running time of HL is 10 ms on average and not shown).

are therefore greatly reduced. Another behavior needs worth noting is that VA runs slower even though the graph size is smaller when the vertex removal rate increases to 0.2%. The performance of VA cannot be predicted based on the graph size since it depends on finding paths between vertices, which can vary significantly based on the connectivity in the graph. Small changes in the overall structure of the graph can lead to large changes in efficiency.

In summary, RS outperforms all other methods on nearly planar graphs. When the graph is not ‘close to’ planar, RS provides a good option when a tradeoff between efficiency and accuracy needs to be made since RS is more efficient than all previous methods, and its accuracy is still competitive.

4 Conclusion

In this paper, we studied the Maximum Induced Planar Subgraph (MIPS) problem which aims to find the largest size of vertices which induce a planar subgraph. As in many related problems, there is a trade-off between the quality of the approximation and the efficiency of the algorithm. By observing that both planarity testing and planarization can be accomplished simultaneously, we were able to produce a linear time algorithm for the MIPS problem, and the new approach is competitive in both efficiency and effectiveness.

References

1 Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G Tollis. *Graph drawing: algorithms for the visualization of graphs.* Prentice Hall PTR, 1998.

2 Itai Benjamini and Oded Schramm. Harmonic functions on planar and almost planar graphs and manifolds, via circle packings. *Inventiones mathematicae*, 126(3):565–587, 1996.

3 John M Boyer and Wendy J Myrvold. On the cutting edge: Simplified $o(n)$ planarity by edge addition. *J. Graph Algorithms Appl.*, 8(2):241–273, 2004.

4 Sergio Cabello and Bojan Mohar. Crossing and weighted crossing number of near-planar graphs. In *International Symposium on Graph Drawing*, pages 38–49. Springer, 2008.

5 Gek L Chia and Chan L Lee. Regular raphs with small skewness and crossing numbers. *Bulletin of the Malaysian Mathematical Sciences Society*, 39(4):1687–1693, 2016.

6 Robert J Cimikowski. Graph planarization and skewness. *Congressus Numerantium*, pages 21–21, 1992.

7 Alice M Dean, William S Evans, Ellen Gethner, Joshua D Laison, Mohammad Ali Safari, and William T Trotter. Bar k-visibility graphs. *J. Graph Algorithms Appl.*, 11(1):45–59, 2007.

8 Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G Tollis. Algorithms for drawing graphs: an annotated bibliography. *Computational Geometry*, 4(5):235–282, 1994.

9 Walter Didimo and Giuseppe Liotta. The crossing-angle resolution in graph drawing. In *Thirty Essays on Geometric Graph Theory*, pages 167–184. Springer, 2013.

10 Keith Edwards and Graham Farr. An algorithm for finding large induced planar subgraphs. In *International Symposium on Graph Drawing*, pages 75–83. Springer, 2001.

11 Keith Edwards and Graham Farr. Planarization and fragmentability of some classes of graphs. *Discrete Mathematics*, 308(12):2396–2406, 2008.

12 William Evans, Michael Kaufmann, William Lenhart, Tamara Mchedlidze, and Stephen Wismath. Bar 1-visibility graphs and their relation to other nearly planar graphs. *J. Graph Algorithms Appl*, 18(5):721–739, 2014.

13 Alan Gibbons. *Algorithmic graph theory.* Cambridge university press, 1985.

14 Magnús M Halldórsson and Hoong Chuin Lau. Low-degree graph partitioning via local search with applications to constraint satisfaction, max cut, and coloring. In *Graph Algorithms And Applications I*, pages 45–57. World Scientific, 2002.

15 Mohsen MD Hassan and Gary L Hogg. A review of graph theory application to the facilities layout problem. *Omega*, 15(4):291–300, 1987.

16 Petr Hliněnỳ and Gelasio Salazar. On the crossing number of almost planar graphs. In *International Symposium on Graph Drawing*, pages 162–173. Springer, 2006.

17 Seok-Hee Hong, Peter Eades, Giuseppe Liotta, and Sheung-Hung Poon. Fáry's theorem for 1-planar graphs. In *Computing and Combinatorics - 18th Annual International Conference, COCOON 2012, Sydney, Australia, August 20-22, 2012. Proceedings*, pages 335–346, 2012.

18 Michael Jünger and Petra Mutzel. Maximum planar subgraphs and nice embeddings: Practical layout tools. *Algorithmica*, 16(1):33–59, 1996.

19 Mukkai S Krishnamoorthy and Narsingh Deo. Node-deletion np-complete problems. *SIAM Journal on Computing*, 8(4):619–625, 1979.

20 Jérôme Kunegis. KONECT: the koblenz network collection. In *22nd International World Wide Web Conference, WWW '13, Rio de Janeiro, Brazil, May 13-17, 2013, Companion Volume*, pages 1343–1350, 2013.

21 Casimir Kuratowski. Sur le probleme des courbes gauches en topologie. *Fundamenta mathematicae*, 15(1):271–283, 1930.

22 Jure Leskovec and Andrej Krevl. Snap datasets: Stanford large network dataset collection. 2015. URL: `http://snap.stanford.edu/data`.

23 John M Lewis and Mihalis Yannakakis. The node-deletion problem for hereditary properties is np-complete. *Journal of Computer and System Sciences*, 20(2):219–230, 1980.

24 Annegret Liebers. Planarizing graphs—a survey and annotated bibliography. In *Graph Algorithms And Applications 2*, pages 257–330. World Scientific, 2004.

25 László Lovász. On decomposition of graphs. *Studia Sci. Math. Hungar*, 1(273):238, 1966.

26 Carsten Lund and Mihalis Yannakakis. The approximation of maximum subgraph problems. In *International Colloquium on Automata, Languages, and Programming*, pages 40–51. Springer, 1993.

27 Kerri Morgan and Graham Farr. Approximation algorithms for the maximum induced planar and outerplanar subgraph problems. *J. Graph Algorithms Appl.*, 11(1):165–193, 2007.

28 Mauricio GC Resende and Celso C Ribeiro. A grasp for graph planarization. *Networks*, 29(3):173–189, 1997.

29 Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

30 Douglas Brent West et al. *Introduction to graph theory*, volume 2. Prentice hall Upper Saddle River, 2001.

A Experiments on non-planar graphs

Figure 9 and Figure 10 show experimental results for *UG* and *PG* respectively. As in the experiments on *MP* and *IN*, when the vertex removal rate increases, the gaps between the percentages achieved by VA and VSR are reduced, and VA outperforms the other methods once the vertex removal rate is high. Even though the graph size of *UG* is smaller than *IN*, VA runs around five times slower on *UG* than on *IN*. The efficiency of VA is remarkably unstable on *PG* as the vertex removal rate increases.

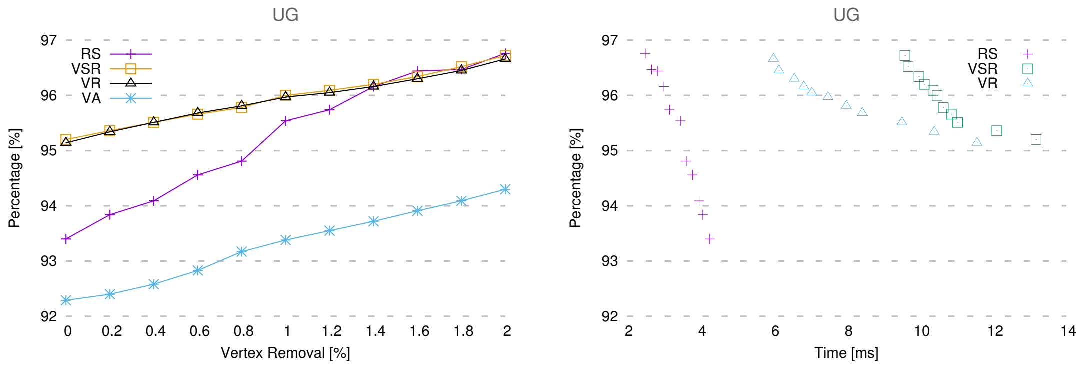

Figure 9 Vertex Removal experiments on the *UG* dataset. The left figure shows the percentage achieved by different algorithms (HL achieves 28% on average and is not shown). The right figure shows the Effectiveness / Efficiency trade-off (the running time of HL is 4 ms, and VA is 1,526 ms on average – neither are shown to maintain the graph scale).

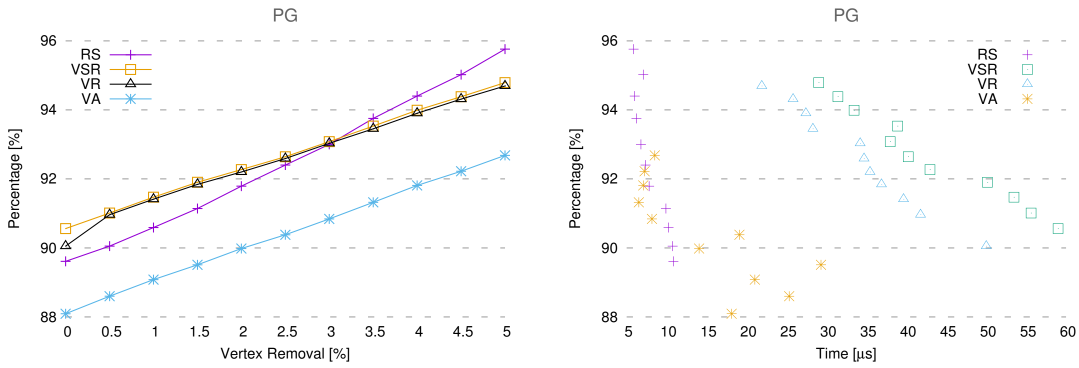

Figure 10 Vertex removal experiments on the *PG* dataset. The left figure shows the percentage achieved by the algorithms (HL achieves 12% on average and is not shown). The right figure shows the Effectiveness / Efficiency trade-off (the running time of HL is 10 ms on average and not shown).