Approximation Algorithms for NP-Complete Problems on Planar Graphs*

Brenda S. Baker

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This paper describes a general technique that can be used to obtain approximation algorithms for various NP-complete problems on planar graphs. The strategy depends on decomposing a planar graph into subgraphs of a form we call *k*-outerplanar. For fixed *k*, the problems of interest are solvable optimally in linear time on *k*-outerplanar graphs by dynamic programming. For general planar graphs, if the problem is a maximization problem, such as maximum independent set, this technique gives for each *k* a linear time algorithm that produces a solution whose size is at least $k/(k+1)$ optimal. If the problem is a minimization problem, such as minimum vertex cover, it gives for each *k* a linear time algorithm that produces a solution whose size is at most $(k+1)/k$ optimal. Taking $k = \lceil c \log \log n \rceil$ or $k = \lceil c \log n \rceil$, where *n* is the number of nodes and *c* is some constant, we get polynomial time approximation schemes, *i.e.* algorithms whose solution sizes converge toward optimal as *n* increases. The class of problems for which this approach provides approximation schemes includes maximum independent set, maximum tile salvage, partition into triangles, maximum H-matching, minimum vertex cover, minimum dominating set, and minimum edge dominating set. For these and certain other problems, the proof of solvability on *k*-outerplanar graphs also enlarges the class of planar graphs for which the problems are known to be solvable in polynomial time.

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms; G.2.2. [**Discrete Mathematics**]: Graph theory

General Terms: Algorithms, Theory

Additional keywords: NP-complete, planar graphs, approximation algorithms, approximation schemes, independent set, dominating set, vertex cover, Hamiltonian path, Hamiltonian circuit, partition into triangles, partition into perfect matchings

*A preliminary version of this work was presented at the 24th Annual Symposium on Foundations of Computer Science, Tucson, Arizona, November 7-9, 1983.

Approximation Algorithms for NP-Complete Problems on Planar Graphs*

Brenda S. Baker

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction

This paper describes a general technique that can be used to obtain approximation algorithms for various NP-complete problems on planar graphs. The strategy depends on decomposing a planar graph into subgraphs of a form we call *k*-outerplanar, and combining optimal solutions for the *k*-outerplanar subgraphs. For fixed *k*, the problems of interest are solvable optimally in linear time on *k*-outerplanar graphs by dynamic programming. For general planar graphs, if the problem is a maximization problem, such as maximum independent set, this technique gives for each *k* a linear time algorithm that produces a solution whose size is at least $k/(k+1)$ optimal. If the problem is a minimization problem, such as minimum vertex cover, it gives for each *k* a linear time algorithm that produces a solution whose size is at most $(k+1)/k$ optimal. Taking $k = \lceil c \log\log n \rceil$ or $k = \lceil c \log n \rceil$, where *n* is the number of vertices and *c* is some constant, we get polynomial time approximation schemes, *i.e.* algorithms whose solution sizes converge toward optimal as *n* increases.

The heart of the technique lies in the adaptability of *k*-outerplanar graphs to dynamic programming. Intuitively, a *k*-outerplanar graph has a planar embedding with disjoint cycles properly nested at most *k* deep. (A formal definition is given in the next section.) "Outerplanar" is equivalent to "1-outerplanar" and is the class of all planar graphs that can be laid out with all vertices on the exterior face* [Har]. Every planar graph is *k*-outerplanar for some *k*. Given a planar graph *G*, a *k*-outerplanar embedding of *G* for which *k* is minimal can be found in time polynomial in the number of vertices [BM].

For purposes of discussion, we focus on maximum independent set: Given a graph $G=(V,E)$ and positive integer *K*, does *G* contain an independent set of size at least *K*, *i.e.* a subset V' of *V* with size at least *K* such that no two vertices in V' are joined by an edge in *E*? For planar graphs, finding a maximum independent set is NP-complete [GJ]. At the end of this section, we list a number of other NP-complete problems for which the technique works.

For a planar graph, we show that $O(8^k kn)$ time is sufficient to find an independent set whose size is at least $k/(k+1)$ optimal, where *n* is the number of nodes. Substituting $k = f(n)$ into these expressions, we get approximation schemes that run in time $O(8^{f(n)} f(n) n)$ and produce solutions that are at least $f(n)/(f(n)+1)$ optimal. If $f(n)$ is $O(\log n)$, the running time will be polynomial; there is a tradeoff between the running time and the solution's rate of convergence to optimal.

Several approximation algorithms or schemes have been proposed previously for maximum independent set on planar graphs. Of the polynomial time algorithms that produce solutions that come within some constant times optimal, the best previous result is an $O(n\log n)$ algorithm that achieves at least half optimal [CNS2]. By comparison, for any *k*, we have a linear time algorithm that achieves at least $k/(k+1)$ optimal.

In a very different approach, Lipton and Tarjan applied their planar separator theorem [LT1] to obtain an approximation scheme for maximum independent set [LT2]. They showed that time $O(n \max\{\log n, 2^{f(n)}\})$ is sufficient to obtain an independent set of size at least $(1-O(1/\sqrt{f(n)}))$ optimal [LT2]. Thus, for $f(n)$ = log log *n*, $O(n\log n)$ time is sufficient to obtain a solution whose size is at least $(1-O(1/\sqrt{\log\log n}))$ optimal. Unfortunately, Chiba *et al* [CNS2] calculated that for the solution to have

*The regions defined by a planar embedding are called *faces*. The unbounded face is called the *exterior* face; other faces are *interior* faces.

size at least half optimal, the graph must have at least

$$2^{2^{400}}$$

nodes!*

Both the Lipton-Tarjan approach and our approach have tradeoffs betwen running time and convergence rate. However, for the same running time, the guaranteed convergence rate of our approximation scheme is better. For example, if we allow the Lipton-Tarjan approach to use $O(n(\log n)^4)$ time by setting $f(n)=4\log\log n$, it finds an independent set whose size is at least $(1-O(1/(2\sqrt{\log\log n})))$ optimal. By contrast, if we pick $f(n)=\lceil\log\log n\rceil$ so that our scheme runs in time $O(n(\log n)^3\log\log n)$, it achieves a solution that is at least $\lceil\log\log n\rceil/(1+\lceil\log\log n\rceil)$ optimal. Moreover, the solution will be at least 2/3 optimal for $n>4$ and at least 3/4 optimal for $n>16$.

Another difference between the Lipton-Tarjan approach and ours is that their approach requires at least $\Omega(n\log n)$ time for the worst case. For ours, picking $f(n)$ small enough, *e.g.* $f(n)=\lceil(\log\log n)/6\rceil$, gives running times of $o(n\log n)$ without losing the convergence toward optimal with increasing n.

The power of our approach is due to the adaptability of the structure of k-outerplanar graphs to dynamic programming. The basic idea is as follows. Recursively decompose the graph into subgraphs, and organize a dynamic programming algorithm around the decomposition. The union of the maximum independent sets for two subgraphs with some common boundary nodes is an independent set for the union of the subgraphs only if the two sets are compatible, i.e. if no node of one independent set is adjacent to a node of the other independent set in the union of the subgraphs. A straightforward way of guaranteeing compatibility is take unions only of independent sets that agree on which common nodes are in the set.

The number of boundary nodes for each subgraph will be bounded by $2k$. Thus, for each subgraph, a table with 2^{2k} entries will be needed to keep track of the size of the maximum independent set for each combination of boundary nodes allowed to be in the set.

A k-outerplanar graph can easily be decomposed into two subgraphs with just $2k$ common boundary nodes. In continuing the decomposition, care must be taken to avoid generating more than $2k$ boundary nodes for smaller subgraphs, since table sizes will be exponential in the number of boundary nodes. Figure 1(i) shows a cutting strategy that fails in this regard: if each cut generates at most $2k$ boundary nodes, region A formed by three successive cuts could have $6k$ boundary nodes. Figure 1(ii) shows a more suitable strategy: the graph is sliced like a pie. Unfortunately, the notion of cutting up the graph like a pie is too simplistic. The difficulty arises from nodes embedded within adjoining cycles; there is no single center to cut from. Analysis of the structure of the graphs leads to a decomposition with two inductive steps: one relating adjoining cycles, and the other cutting inward within a cycle.

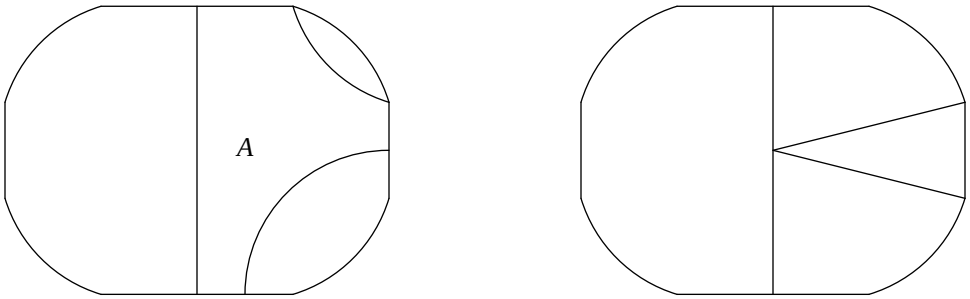
Dolev, Leighton, and Trickey [DLT] studied width k graphs, which are closely related to k-outerplanar graphs. They obtained planar embeddings based on a simple recursive decomposition of width k graphs. The additional machinery developed in this paper is needed to bound the size of the tables in the dynamic programming and to obtain running times of $o(n\log n)$.

Some other problems for which our technique provides approximation schemes are the following problems that are NP-complete for planar graphs: maximum tile salvage [BLS], partition into triangles [GJ, problem GT11], maximum H-matching [BJLSS], minimum vertex cover [GJ, problem GT1], minimum dominating set [GJ, problem GT2], and minimum edge dominating set [GJ, problem GT2]. (Weights may be allowed on the graphs where appropriate.) For each of these problems, there is a linear time algorithm that achieves a solution that is at least $k/(k+1)$ optimal, or at most $(k+1)/k$ optimal, as appropriate, for fixed k, and there is a polynomial time approximation scheme.

For all of the above problems, our results improve on the best previous approximation algorithms or schemes, which are:

1. applications of the Lipton-Tarjan planar separator theorem to minimum vertex cover [BE,CNS1] and to maximum H-matching on planar graphs of bounded degree [L], with bounds similar to those

*Djidjev's improvements [D] in the planar separator theorem bounds can be used to reduce this number substantially, although not to practicality.



(i) (ii)

Figure 1. Schematics showing two possible strategies for decomposing planar graphs.

described above for maximum independent set,

2. for maximum 2×2 tile salvage, a linear time algorithm that achieves at least half optimal [BLS], and
3. for minimum vertex cover, a linear time algorithm that achieves at most 5/3 optimal [BE], an $O(n^2 \log n)$ time algorithm that achieves at most 8/5 optimal [Ho], and a polynomial time algorithm that achieves at most 3/2 optimal [Ho].

All of the problems mentioned above are solvable in linear time on *k*-outerplanar graphs for fixed *k*, as are the following: minimum maximal matching [GJ, problem GT10], 3-colorability [GJ, problem GT4], Hamiltonian circuit [GJ, problem GT37], and Hamiltonian path [GJ, problem GT39]. In addition, the dynamic programming strategy of this paper can also be used to solve partition into perfect matchings [GJ, problem GT16] on *k*-outerplanar graphs in polynomial time for fixed *k*. (For general planar graphs, the problems listed in this paragraph are not amenable to approximation by the techniques described in this paper.) In general, the dynamic programming technique will work for problems that involve local conditions on nodes and edges.

The largest subclasses of planar graphs for which any of the above problems were previously known to be solvable in polynomial time are trees (for maximum independent set [M,MCH], minimum dominating set [CGH], and minimum edge-dominating set [MH,YG]), maximal outerplanar graphs (for maximum independent set [G] and minimum coloring [G]), and series-parallel graphs (for minimum vertex cover and partition into triangles [TNS]). Both trees and maximal outerplanar graphs are subclasses of outerplanar (1-outerplanar) graphs; series-parallel graphs include outerplanar graphs but are a proper subclass of planar graphs.

In related work, Bui and Peck [BP] have applied the dynamic programming techiques of this paper to the graph bisection problem. They consider *s*-partitions of a graph, where an *s*-partition is a partition of the vertices of *G* into disjoint sets of size *s* and $n - s$; an optimal *s*-partition is one for which the size of the cut-set is minimal over all *s*-partitions. They show that given an *n*-vertex planar graph *G*, an integer *s*, $0 \le s \le n$, and a planar embedding of *G* such that each biconnected component is at most *m*-outerplanar, $O(m^2 n^3 2^{3m})$ time is sufficient to find the optimal *s*-partition of the graph. Also, Bui and Jones [BJ] have used the dynamic programming techniques of this paper to show that optimal vertex separators for planar graphs with vertex separators of size $O(\log n)$ can be computed in polynomial time.

In the remainder of the paper, we prove the results for maximum independent set. The overall strategy is the same for the other problems listed. Differences arise for the various problems in how to decompose general planar graphs into *k*-outerplanar graphs and in what to keep in the tables used in the dynamic programming. Some general comments about the differences are given in the last section, and the individual problems are discussed in the appendix.

2. Definitions and overall strategy for the approximation algorithms

First, we define level k nodes in a planar embedding E of a planar graph G. (See Figure 2.) A node is at *level 1* if it is on the exterior face. Call a cycle of level i nodes a *level i face* if it is an interior face in the subgraph induced by the level i nodes. For each level i face f, let G_f be the subgraph induced by all nodes placed inside f in this embedding. Then the nodes on the exterior face of G_f are at *level* $i+1$.

Throughout the paper we assume that the planar embedding is represented by an appropriate data structure such as that used by Lipton and Tarjan [LT1]. Thus, levels of nodes can be computed in linear time.

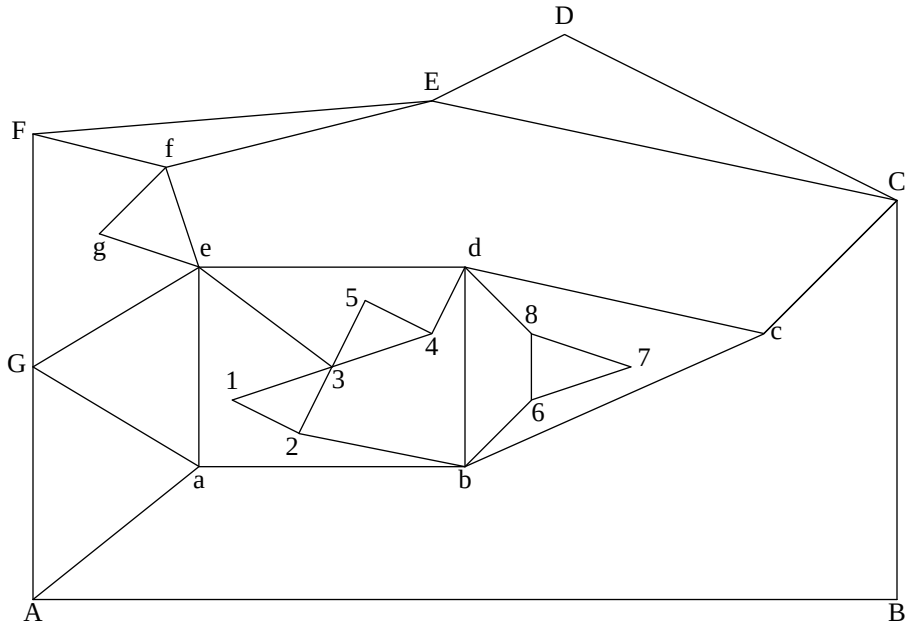


Figure 2. A 3-outerplanar embedding. Nodes $A-G$ are at level 1, nodes $a-g$ are at level 2, and nodes 1-8 are at level 3.

A planar embedding is *k-level* if it has no nodes of level $>k$. A planar graph is *k-level* if it has a *k*-outerplanar embedding.

Now, let us assume the following theorem, whose proof will be given in the next section.

THEOREM 1. Let k be a positive integer. Given a k-outerplanar embedding of a k-outerplanar graph, an optimal solution for maximum independent set can be obtained in time $O(8^k n)$, where n is the number of nodes.

For any positive integer k, the linear time algorithm of Theorem 1 can be incorporated as follows into a linear time algorithm that achieves solutions at least $k/(k+1)$ optimal for general planar graphs. First, given a planar graph G, we generate a planar embedding using the linear time algorithm of Hopcroft and Tarjan [HT]. Let S_{OPT} be a maximum independent set for G. For at least one r, $0 \le r \le k$, at most $1/(k+1)$ of the nodes in S_{OPT} are at a level that is congruent to r (mod $k+1$). Thus, for each i, $0 \le i \le k$, we do the following. Let G_i be the graph obtained by deleting all nodes of G whose levels are congruent to i (mod $k+1$). The remaining graph is composed of components with k-outerplanar embeddings. Thus, we may apply the linear time algorithm to each component to obtain a maximum independent set, and the union of these solutions is the maximum independent set for G_i. From above, the solution for G_r is at least $k/(k+1)$ as large as the optimal solution for G. By taking the largest of the solutions for the G_i's as the solution for G, we get an independent set whose size is at least $k/(k+1)$ optimal. Thus, we have the following theorem.

T**HEOREM** 2. For fixed k, there is an $O(8^k kn)$ time algorithm for maximum independent set that achieves a solution of size at least $k/(k+1)$ optimal for general planar graphs. Choosing $k = \lceil c \log\log n \rceil$, where c is a constant, yields an approximation scheme that runs in time $O(n(\log n)^{3c} \log\log n)$ and achieves a solution of size at least $\lceil c \log\log n \rceil / (1 + \lceil c \log\log n \rceil)$ optimal. In each case, n is the number of nodes in the graph.

3. A linear time algorithm for outerplanar graphs

In this section, we show how to solve the maximum independent set problem in linear time for outerplanar graphs. The techniques used here will be generalized in the next section to handle k-outerplanar graphs.

Given a connected outerplanar graph, replace each bridge† (x,y) by two edges between x and y. This will allow us to treat a bridge as a face rather than as a special case. Call the resulting graph G. Define an edge of G to be *exterior* if it lies on the exterior face, and *interior* otherwise.

The maximum independent set for G will be computed by recursively processing an ordered, rooted tree $\overline{G}$ that represents the structure of G. Each leaf of $\overline{G}$ will represent an exterior edge of G; every other vertex will represent a face of G and will be called a *face* vertex. $\overline{G}$ is constructed as follows.

First, suppose there are no cutpoints* in G. Place a vertex** in each interior face and on each exterior edge, and draw an edge from each vertex representing a face f to each vertex representing either an adjacent face (*i.e.* a face sharing an edge with f) or an exterior edge of f. (This tree is closely related to the dual of the graph; however, the dual would lack vertices for exterior edges and would have an additional vertex for the exterior face.) An example is shown in Figure 3. The planar embedding induces a cyclic ordering on the edges of each vertex in the tree. Choosing a face vertex v as the root and choosing which child of v is to be its leftmost child determine the parent and ordering of children for every other vertex of $\overline{G}$. (See Figure 4.)

Label the vertices of $\overline{G}$ recursively as follows. Label each leaf of the tree with the oriented exterior edge it represents. Label each face vertex with the first and last nodes in its children's labels.

If a face vertex is labeled (x,y), the leaves of its subtree represent a directed walk of exterior edges in a counterclockwise direction from x to y. For the root, $x = y$ and the directed walk covers all the exterior edges. For any other face vertex v, $x \neq y$, and (x,y) is an interior edge shared by the face represented by v and the face represented by its parent in the tree. For example, consider Figure 4. The leaves of the node labeled $(3,7)$ represent a walk along nodes $3,4,5,6,7$. The leaves of the root in Figure 4 represent a counterclockwise walk around the exterior edges beginning and ending at node 1. The vertex labeled $(1,3)$ represents the face containing nodes 1-3, its parent represents the face containing nodes 1,3,7,9, and $(1,3)$ is the interior edge shared by these faces.

Now, suppose G has cutpoints. Since we have eliminated bridges by turning them into faces with two edges, each cutpoint is a vertex at which two or more faces meet (without sharing an edge). As before, place a vertex in each interior face and on each exterior edge, create an edge between each pair of vertices representing faces sharing an edge, and create an edge between each face vertex and the vertices on its exterior edges. This gives a tree for each biconnected component; faces that meet at a cutpoint have their vertices in different trees. Now, create an edge between any pair of vertices representing faces that share a vertex and lie in different trees, and continue to do this until all the trees have been joined into a single tree $\overline{G}$. For example, in Figure 5, node 9 is a cutpoint, and an edge has been created between the vertices for two faces sharing this cutpoint, namely the faces with nodes 7,8,9 and 9,10,11, respectively; an edge joining the vertices for the faces with nodes 9,1,3,7 and 9,10,11 could equally well have been chosen. The rooting and ordering of the tree are again determined by choosing a face vertex as the root and choosing one of its children as its leftmost child, as illustrated in Figure 6.

Label the vertices recursively as before. The label of each leaf again represents an oriented exterior

---

†A *bridge* is an edge whose removal disconnects the graph.

*A *cutpoint* is a node whose removal disconnects the graph.

**To avoid confusion, we will use "node" for the original graph and "vertex" for trees.

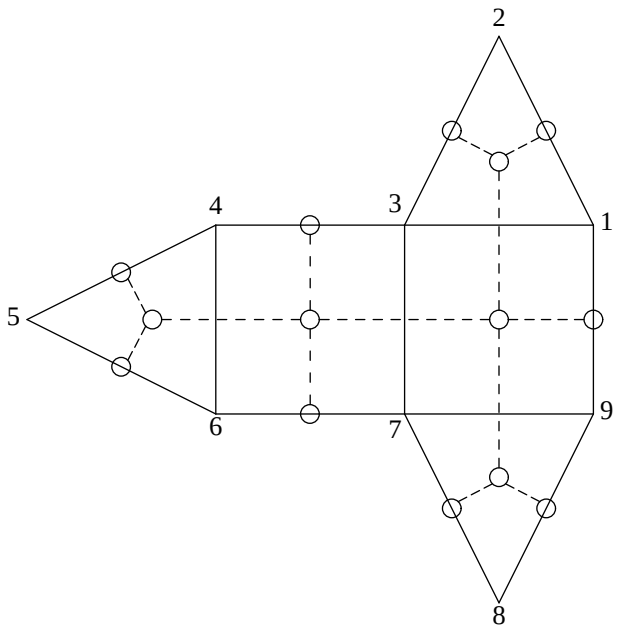


Figure 3. An outerplanar graph (with no cutpoints) and a tree (drawn with dashed lines).

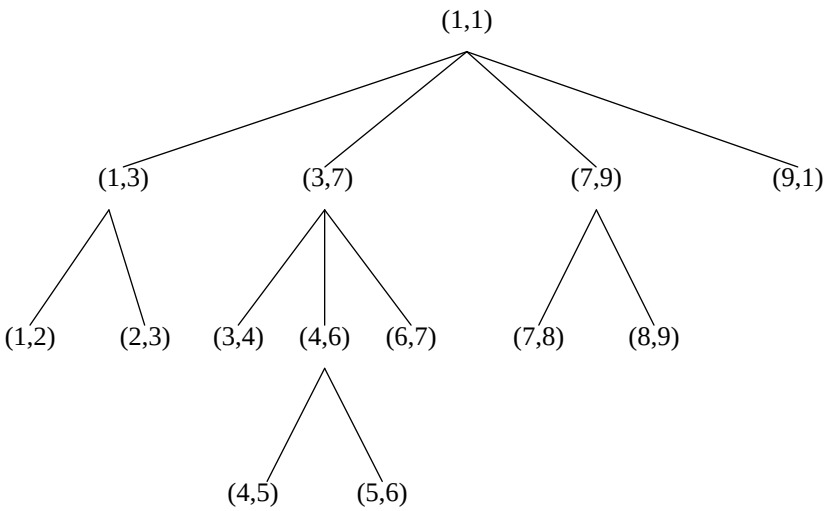


Figure 4. A rooted, ordered, labeled tree $\overline{G}$ for the graph G of Figure 3.

edge, and as before, the leaves of any subtree represent a directed walk of exterior edges. The root is again labeled (r,r) for some node r. For any other face vertex labeled (x,y), if $x \neq y$, the label again represents an interior edge shared by two faces. However, if $x = y$, the label represents a cutpoint shared by two faces, as for the vertex labeled $(9,9)$ in Figure 6.

Computation of the maximum independent set proceeds as follows. The result of processing a vertex v labeled (x,y) is a table that gives the sizes of maximum independent sets for the subgraph represented by the subtree rooted at v, according to whether the nodes x and y are in the set. That is, for each of the four possible bit pairs representing whether each of x and y is in the set, the table contains the size of the maximum independent set for the subgraph. The procedure that computes a table for vertex v is given in Figure 7.

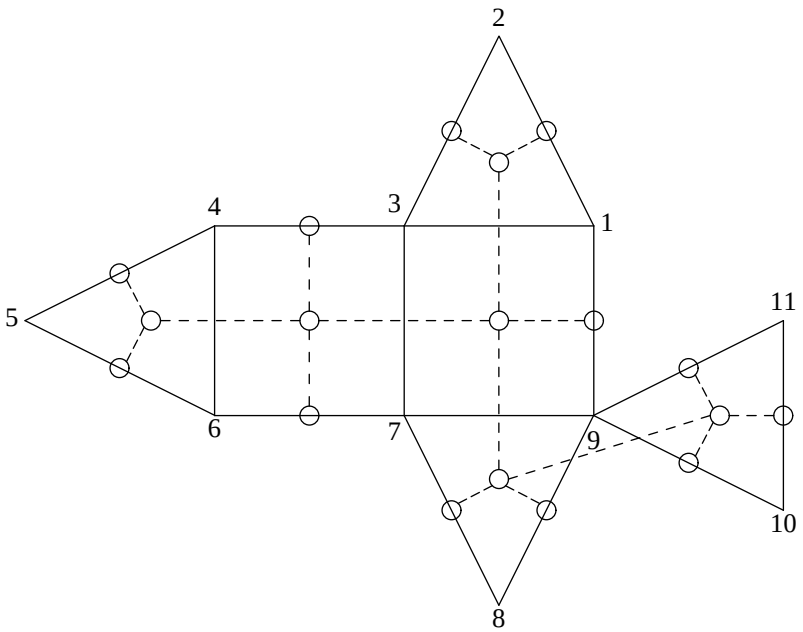


Figure 5. An outerplanar graph with a cutpoint (node 9) and a corresponding tree (shown by dashed lines).

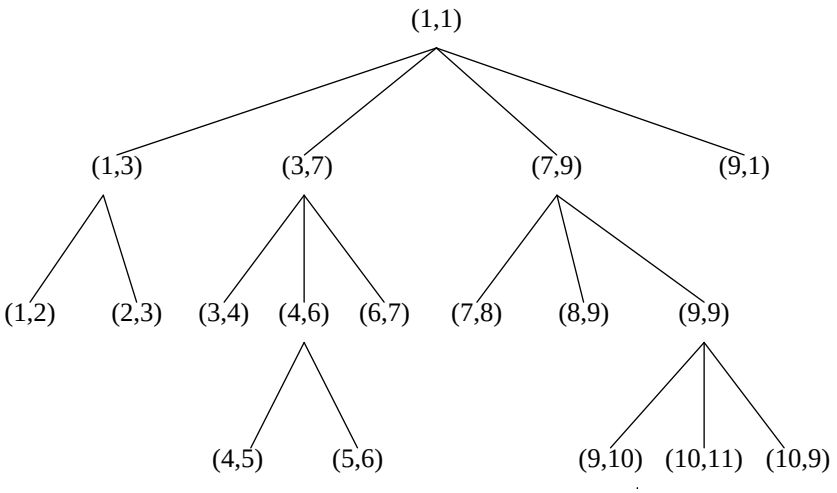


Figure 6. A rooted, ordered, labeled tree $\overline{G}$ for the graph G of Figure 5.

The details not given in Figure 7 are as follows. The table for a leaf v representing an edge (x,y) specifies that the size of a maximum independent set for the subgraph is 1 if exactly one endpoint of (x,y) is in the set, 0 if neither is in it, and undefined (illegal) if both are in it. The procedure *merge* works as follows. For some z, the current table T has a value for each bit pair representing x and z, the child c has label (z,w) for some w, and $table(c)$ has a value for each bit pair representing z and w. The updated table T has a value for each bit pair representing x and w. This updated value is found for bit pair (b_1,b_3) (representing x and w) by taking the maximum over 0-1 values of bit b_2 of $V(T) + V(table(c)) - b_2$, where $V(T)$ is the value of T for (b_1,b_2) (representing x and z) and $V(table(c))$ is the value of $table(c)$ for (b_2,b_3) (representing z and w). Subtracting b_2 prevents counting z twice if z is in the maximum independent set. When

```
procedure table(v)
begin
if v is a level 1 leaf with label (x,y)
        then return a table representing the edge (x,y);
else   /* v is a face vertex */
        begin
        T= table (u), where u is the leftmost child of v;
        for each other child c of v from left to right
                T = merge (T, table (c));
        return (adjust(T));
        end
end
```

Figure 7. Procedure for computing a table for vertex v.

all of the children's tables have been processed, T has a value for each bit pair representing x and y. Finally, the label (x,y) is incorporated into T by the procedure *adjust*. In particular, if $x=y$, the table entry is set to "undefined" for each bit pair requiring exactly one of x and y to be in the maximum independent set, and the value representing the inclusion of both x and y in the set is decremented by one to avoid counting $x=y$ twice. If $x\neq y$, the table entry for the bit pair specifying that both x and y are in the set is set to "undefined," because (x,y) represents an edge.

When *table* is called on the root (r,r) of the tree, it results in a table with entries for four bit pairs. Thanks to the procedure *adjust*, two of the values are undefined because they represent inconsistency as to whether r is in the maximum independent set. The larger of the two remaining values is the size of the maximum independent set for the graph.

4. A linear time algorithm for k-level graphs

In this section, we prove Theorem 1 by developing a linear time algorithm for maximum independent set on k-outerplanar graphs. The algorithm is based on finding a recursive decomposition in which each subgraph has at most $2k$ boundary nodes.

We will assume that the graph is connected and that the level i nodes within every level $i-1$ face induce a connected subgraph. (If not, add some fake edges to obtain connectivity, but in computing values in tables, ignore these edges.) We refer to this connected induced subgraph as a *level i component*. Note that a level i component is an outerplanar graph. We assume that every bridge in a level i component has been replaced by two edges, so that we can treat it as a face with exactly two edges, as in the preceding section.

Trees.

Each level i component is outerplanar, and hence we can use the method of the preceding section to construct a tree for it. As before, the leaves of a level i tree represent edges exterior to the level i component, the other vertices represent faces of the level i component, and the leaves from left to right represent a counterclockwise walk around the exterior edges of C.

The only additional constraints are on how the root and its leftmost child are selected for a tree at level i, $i>1$. Suppose the level $i-1$ trees have already been constructed. Consider a level i component C enclosed by a level $i-1$ face f. Let T be a triangulation of the region between C and f obtained by adding edges as necessary between nodes of C and nodes of f; such a triangulation can be generated in linear time by scanning the nodes of C and f in parallel.

Suppose the vertex representing f has label (x,y). The tree for C will have a root with label (z,z), where z is adjacent to x in T. In particular, if $x=y$, pick z to be any node adjacent to x in T; if $x\neq y$, pick z to be the third node in the triangle with x and y in T.

Figure 8 illustrates possible triangulations for two regions of the graph of Figure 2. Suppose C is the

level 2 component with nodes 1-5, and *f* consists of nodes *a*, *b*, *d*, *e*. If the vertex representing *f* has label (e,e), then *z* could be node 1, 3, or 5; if the vertex representing *f* has label (d,b), then *z* would be node 4.

If *C* has just the one node *z*, then its tree will consist of just the root. Otherwise, the leftmost child of the root will have label (z,u), where (z,u) is the first exterior edge of *C* counterclockwise around *z* from (z,x) in *T*. The root will represent the face of *C* containing (z,u). For example, in Figure 8, let *C* again be the level 2 component containing nodes 1-5. If *z* is node 4 and *x* is node *d*, then the leftmost child will represent exterior edge $(4,5)$ of *C*; if *z* is node 3 and *x* is node *e*, then the leftmost child will represent exterior edge $(3,1)$ of *C*.

Finally, the remainder of the tree is constructed as in Section 3, so that the leaves from left to right represent a counterclockwise walk around the exterior edges of *C*. Figure 9 shows a possible set of trees for the 3-outerplanar graph of Figure 2.

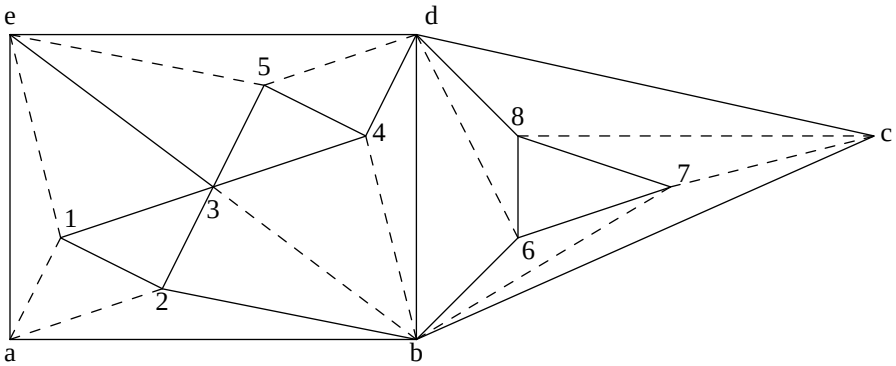


Figure 8. Triangulating regions between levels 2 and 3 for the graph of Figure 2. Edges added in the triangulation are shown as dashed lines.

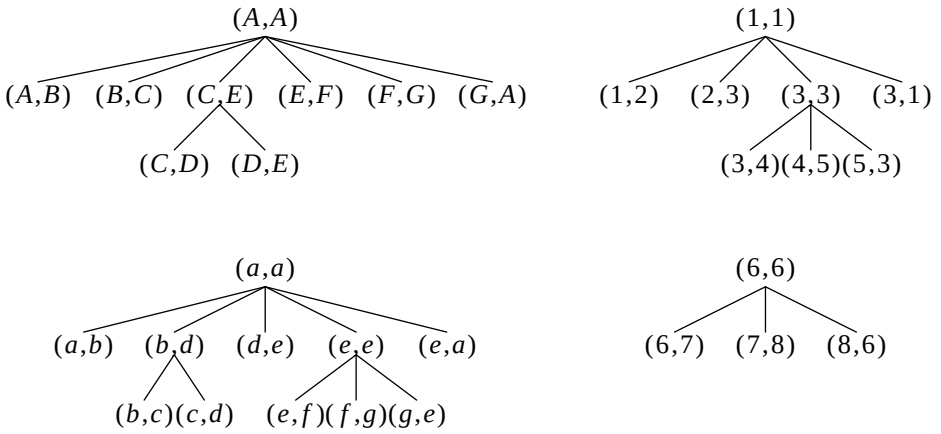


Figure 9. Trees for the graph of Figure 2.

Slices

The dynamic programming will involve repeatedly merging tables of subgraphs into tables for the union of the subgraphs. The relevant subgraphs will be called "slices" by analogy with cutting a pie into slices. Each vertex of each tree will have a corresponding slice, and the tables for the slices will be computed recursively. Each boundary of a slice for a level *i* vertex contains *i* nodes, one for each level from 1 to *i*, listed in order of decreasing level. However, a slice for a level *i* vertex may include higher-level nodes nested within the slice in the embedding. In particular, the slice for a level *i* face vertex includes the nodes enclosed by the face, and is obtained by recursing on the root of the tree of the enclosed level $i+1$

component. The table for the face vertex will include the higher-level nodes in its counts, but the merger of tables for adjoining slices can proceed as if there were no higher-level nodes, because the slices have a common boundary of length i. This is the trick that handles adjoining faces with embedded higher-level nodes.

Three slices for the graph of Figure 2 are shown in Figure 10. These are associated with the vertices for level 3 edges $(6,7)$, $(7,8)$, and $(8,6)$, respectively, and do not contain any higher-level nodes. The boundaries of the slice for $(6,7)$ are $6,b,A$ and $7,b,A$, the boundaries of the slice for $(7,8)$ are $7,b,A$ and $8,c,B$, and the boundaries of the slice for $(8,6)$ are $8,c,B$ and $6,d,C$.

The level 2 face vertex labeled (b,d) in Figure 9 represents a face that encloses the level 3 nodes 6, 7, and 8. The slice for this face vertex consists of the union of the slices shown in Figure 10 plus the edge (b,d). Hence it contains the level 3 nodes 6, 7, and 8. Its boundaries are b,A and d,C.

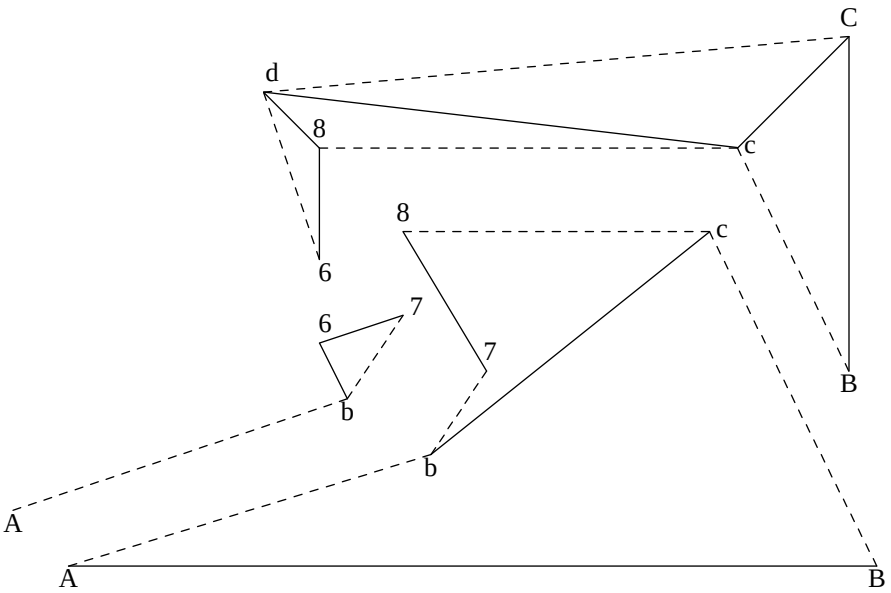


Figure 10. Three slices for the graph of Figure 2. Boundary nodes are connected with dashed lines where no edge exists.

The structure of the algorithm for computing the dynamic programming tables is based on the recursive definition of slices. Informally, slices are defined recursively, as follows, beginning with the slice of the root of the level 1 tree. Let v be a tree vertex labeled (x,y).

(1) If v represents a level i face with no enclosed nodes, $i \geq 1$, its slice is the union of the slices of its children, plus (x,y).

(2) If v represents a level i face enclosing a level $i+1$ component C, its slice is that of the root of the tree for C plus (x,y). (However, as noted above, the boundaries only run from level i to level 1 instead of from level $i+1$ to 1).

(3) If v represents a level 1 leaf, its slice is the subgraph consisting of (x,y).

(4) If v represents a level i leaf, $i > 1$, then its slice includes (x,y), edges from x and y to level i nodes, and the slices computed recursively for appropriate level i trees. Here, "appropriate" is determined by slice boundaries placed along edges in a triangulation of the region between level i and level $i+1$.

Formalizing these ideas depends on defining how boundaries are laid out between levels. Essentially, triangulations are used to draw boundaries so that no edges cross slice boundaries.

Let C be a level i component enclosed by a level $i-1$ face f whose tree vertex is labeled (x,y). Let $TRI(C,f)$ be the triangulation of the region between C and f already constructed in defining the trees. For any pair of successive edges (x_1,x_2), (x_2,x_3) in a counterclockwise walk around the exterior edges of C, there is at least one node y of f that is adjacent to x_2 in $TRI(C,f)$ such that the edges (x_2,x_1), (x_2,y),

(x_2,x_3) occur in counterclockwise order around x_2. Call such a node y a *dividing point* for (x_1,x_2) and (x_2,x_3).

For example, according to the triangulations shown in Figure 8, a and b are dividing points for $(1,2)$ and $(2,3)$, b is a dividing point for $(2,3)$ and $(3,4)$, and e is a dividing point for $(5,3)$ and $(3,1)$. Note that node 3 is a cutpoint of the component containing nodes 1-5 and occurs between two pairs of successive exterior edges. (The definition of dividing point was chosen to be with respect to two edges because of cutpoints.)

We wish to use the dividing points to construct slice boundaries running from one level to the next. If there were no cutpoints at any level, it would be possible to define a boundary from a function assigning a level $i-1$ node to each level i node. Because of cutpoints, however, we must define functions relating tree vertices rather than just graph nodes.

Therefore, to define how left and right boundaries of slices run from our level i component $\overline{C}$ to the enclosing face f, we use the dividing points to define functions from vertices of $\overline{C}$ to $1,2,...,r$, where r is the number of children of the tree vertex corresponding to f. These functions are called *LB* and *RB* for Left Boundary and Right Boundary, respectively.

(F1) Let the leaves of $\overline{C}$ be $v_1,v_2,\cdots,v_t$ from left to right, and let v_j have label (x_j,x_{j+1}), for $1\le j\le t$. Let the children of *vertex*(f) be $z_1,z_2,...,z_r$ from left to right, where z_j has label (y_j,y_{j+1}), for $1\le j\le r$. Define $LB(v_1)=1$ and $RB(v_t)=r+1$. For $1<j\le t$, define $LB(v_j)=q$ if q is the least $p\ge LB(v_{j-1})$ for which y_p is a dividing point for (x_{j-1},x_j) and (x_j,x_{j+1}). For $1\le j<t$, define $RB(v_j)=LB(v_{j+1})$.

(F2) If v is a face vertex of $\overline{C}$, with leftmost child c_L and rightmost child c_R, define $LB(v)=LB(c_L)$ and $RB(v)=RB(c_R)$.

Note that if the edges of C are taken in counterclockwise order, successive dividing points also occur in counterclockwise order around f. Therefore, in rule F1, there will always be a value of p satisfying $p\ge LB(v_{j-1})$. This condition is needed only for the case where the vertex for f is labelled (a,a) for some a and some dividing point is a. For this case, the condition ensures that the values of *LB* and *RB* are nondecreasing for sibling vertices from left to right.

Figure 11 indicates the values of *LB* and *RB* for the level i trees given in Figure 9, $i>1$. For each vertex v, $LB(v)$ and $RB(v)$ are printed to its left and right, respectively. (The inductive definition implies that for two successive vertices v_j and v_{j+1} at the same level, $RB(v_j)=LB(v_{j+1})$, so that only one value needs to be given between each such pair of vertices.) For example, if v is the vertex labeled (b,d), $LB(v)=1$ and $RB(v)=3$. The values for the level 3 trees are consistent with the triangulations given in Figure 8; the triangulation giving the values for the tree of the level 2 component containing nodes $a-g$ is not shown.

The boundaries of slices follow the values of *LB* and *RB* from one level to the next. More precisely, the boundaries of slices are defined inductively as follows for a vertex v labeled (x,y).

(B1) If v is a level 1 leaf, the left boundary of *slice*(v) is x and the right boundary is y.

(B2) Suppose v is a vertex of a tree for a level i component enclosed by the level $i-1$ face f. Let s be the number of children of *vertex*(f). In the following, define the right boundary of the 0th child of *vertex*(f) to be the same as the left boundary of the first child of *vertex*(f), and the left boundary of the $(s+1)$st child to be the same as the right boundary of the sth child. If $LB(v)=q$, the left boundary of *slice*(v) is x plus the left boundary of the slice of the qth child of *vertex*(f). If $RB(v)=t$, the right boundary of *slice*(v) is y plus the right boundary of the slice of the $(t-1)$st child of *vertex*(f).

By rules B1 and B2, each boundary (left and right) of a level i vertex has exactly one node of each level less than or equal to i, for $i\ge 1$.

From these rules and the values of *LB* given in Figure 11, it may be verified that the left boundary of the slice for the vertex labeled (A,B) is A, the left boundary for the slice of the vertex labeled (b,c) is b,A, and the left boundary for the slice of the vertex labeled $(7,8)$ is $7,b,A$, as shown in Figure 10. Similarly, the right boundary of the vertex labeled (A,B) is B, the right boundary of the vertex labeled (b,c) is c,B, and the right boundary of the vertex labeled $(7,8)$ is $8,c,B$.

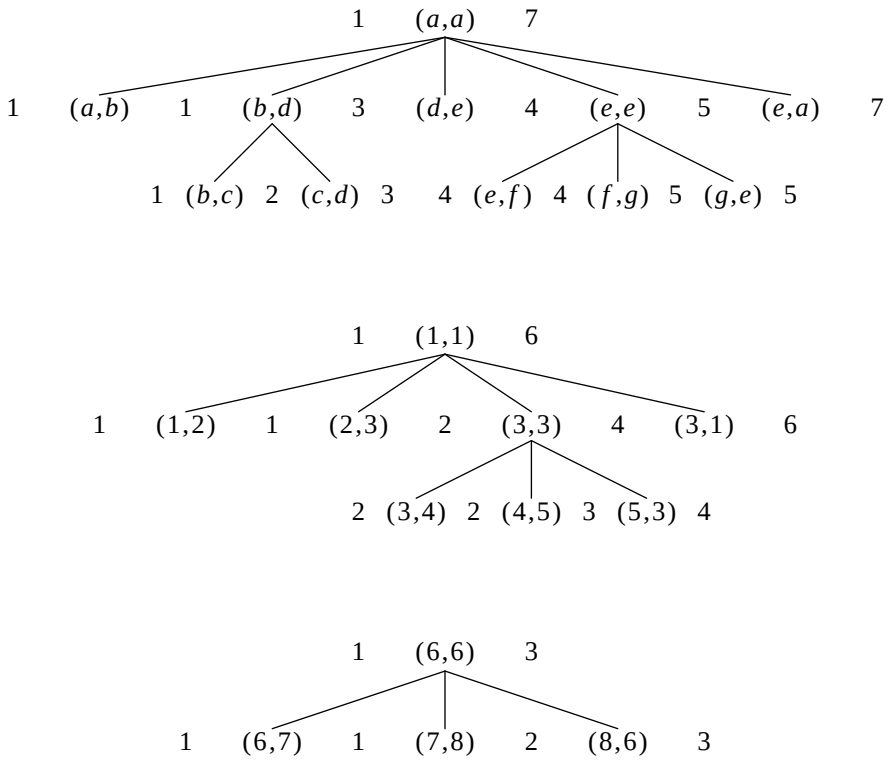


Figure 11. Values of LB and RB for the level 2 and 3 trees of Figure 9.

PROPOSITION 1. For any face vertex v, the left boundary of v's slice is the same as the left boundary of the slice of v's leftmost child, and the right boundary of v's slice is the same as the right boundary of the slice of v's rightmost child.

Proof. By rule B2, the first node in the boundary of a vertex v is the first node in v's label and the remainder is determined by $LB(v)$. The labeling of trees causes the first node in the label of v to be the same as the first node in the label of its leftmost child c_L. By rule F2, $LB(v)$ is defined to be the same as $LB(c_L)$. A similar analysis applies to right boundaries. □

An easy induction argument shows that for any pair of successive siblings v_1 and v_2 in a level i tree, the right boundary of the slice of v_1 is the same as the left boundary of the slice of v_2.

Slices are defined formally as follows. Let v be a level i vertex, $i \geq 1$, with label (x,y). Again, if a vertex u has s children, define the left boundary of the $(s+1)$st child to be the same as the right boundary of the sth child.

(S1) If v is a face vertex, and $face(v)$ encloses no level $i+1$ nodes, then $slice(v)$ is the union of the slices of the children of v, together with (x,y) if (x,y) is an edge (*i.e.* $x \neq y$).

(S2) If v is a face vertex and $face(v)$ encloses a level $i+1$ component C, then $slice(v)$ is the subgraph containing $slice(root(\overline{C}))$ plus (x,y) (if (x,y) is an edge).

(S3) If v is a level 1 leaf, then $slice(v)$ is the subgraph consisting of (x,y).

(S4) Suppose v is a level i leaf, $i > 1$. Suppose the enclosing face is f, and $vertex(f)$ has children u_j, $1 \leq j \leq t$, where u_j has label (z_j, z_{j+1}). If $LB(v) \neq RB(v)$, then $slice(v)$ is the subgraph containing (x,y), any edges from x or y to z_j, for $LB(v) \leq j \leq RB(v)$, and $slice(u_j)$, for $LB(v) \leq j < RB(v)$. If $LB(v) = RB(v) = r$, then $slice(v)$ is the subgraph containing (x,y), any edges from x or y to z_r, the left boundary of $slice(u_r)$, and any edges between boundary nodes of successive levels.

Using rules S3 and S4, it is easily verified that the slices in Figure 10 are correct for the vertices labeled $(6,7)$, $(7,8)$, and $(8,6)$. Using rule S1, it may be seen that the slice for the level 3 vertex labeled $(6,6)$ is the union of the slices in Figure 10. Using rule S2 in addition, reveals that the slice for the vertex labeled (b,d) consists of edge (b,d) plus the union of the slices in Figure 10, as claimed earlier.

Figure 12 shows the slices for the leaves of the level 3 component containing nodes 1-5. By rule S4, the slice for the vertex labeled $(4,5)$ includes the slice of the vertex labeled (b,d) plus edges $(4,5)$ and $(4,d)$, and therefore includes nodes 6-8 of the other level 3 component. The slice for the vertex labeled $(3,1)$ includes the slice for the level 2 vertices labeled (e,e) and (e,a) by rule S4; the slice for (e,e) includes those of (e,f), (f,g), and (g,e) by rule S2, and by rules S4 and S3, these in turn include the level 1 edge (E,F), while the slice for (e,a) includes (F,G) and (G,A).

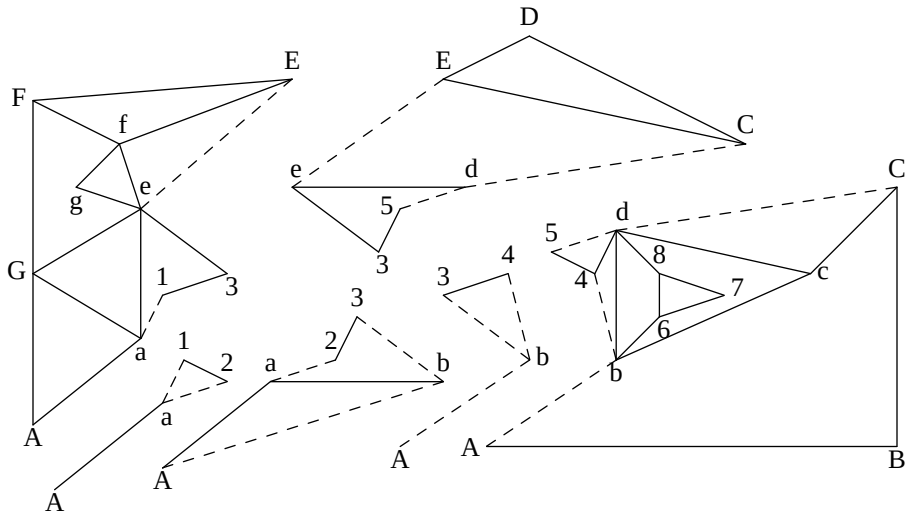


Figure 12. The slices for the leaves of the tree for the component with nodes 1-5.

PROPOSITION 2. The slice for any vertex v of a tree T includes the slices of all its descendants in T plus the slices of all vertices for components enclosed by faces corresponding to descendants of v.

Proof. The proof is by double induction starting with level k. For level k, rule S1 applies to each face vertex and consequently each level k face vertex includes the slices of its descendants. (No components are enclosed by level k faces.) Assume that the result holds for level j, and consider a vertex v in a level $j-1$ tree. If v is a leaf, the statement is vacuously true. Assume that the result holds for all descendants of v. Then either rule S1 applies, implying that the slice for v is the union of the slices of v's children, or rule S2 applies, implying that the slice for v includes the slice of the root of $\overline{C}$, where C is the component enclosed by $face(v)$. In the latter case, by the induction hypothesis, the slice of the root of $\overline{C}$ includes the slices of the leaves of $\overline{C}$, as well as the slices of vertices for components enclosed by faces of C. By the way LB and RB were defined in rule F1, the slices of the leaves of $\overline{C}$ include the slices of all the children of v. Thus, in either case, the slice of v includes the slices of its descendants (including the slices of any vertices for enclosed components) and the slices of any vertices corresponding to the component (if any) enclosed by $face(v)$. By induction, the result holds for all vertices at level $j-1$. □

PROPOSITION 3. Every edge of the graph is included in the slice of at least one tree vertex.

Proof. Every edge between two level i nodes, $i \geq 1$, is the label of some tree vertex and is included in the slice for that vertex by rules S1-S4. So consider an edge e between a level i node u and a level $i-1$ node of

the enclosing face. By planarity, e lies between two successive exterior edges e_1 and e_2 incident on u. By rules $F1$, $B2$, and $S4$, e is included in the slice for one of e_1 and e_2. □

The dynamic programming

In this subsection, we show how to construct and manipulate tables by dynamic programming to compute the maximum independent set. A table will be constructed for each slice. Since a level i slice S has $2i$ boundary nodes, there are 2^{2i} possible subsets of boundary nodes (ignoring possible duplications of nodes between the two boundaries). The table for S has 2^{2i} entries, one for each subset. The entry for a subset is either the size of a maximum independent set of S containing exactly that subset of boundary nodes, or undefined if that subset cannot be part of an independent set (*e.g.* because two of the boundary nodes of the subset are adjacent). In this computation, duplicate nodes on the two boundaries are treated as distinct except for nodes at level i.

The recursive definition of slices determines the organization of the dynamic programming. As rule S2 suggests, the table for the whole graph, which is the slice for the level 1 root, is obtained from the table for the slice of the level 2 root labeled (a,a), which in turn is obtained from the table for the slice of the level 3 root labeled $(1,1)$. As rule S1 suggests, the slice of this level 3 root is obtained by merging the tables of its children, and so forth. This strategy yields the main procedure *table* of the dynamic programming algorithm. The procedure is given in Figure 13. (The procedures *merge*, *adjust*, *contract*, *extend*, and *create* are described below.)

The table manipulations at each step are accomplished by several basic operations. The operations corresponding to rules S1-S3 are straightforward. Corresponding to rule S1, which takes the union of slices, is an operation called *merge*, which merges the tables of the slices. Corresponding to rule S2, which obtains a level i slice from a level $i+1$ slice, is a procedure called *contract*, which modifies the level $i+1$ table to reflect the shorter boundary of the level i slice. Since rules S1 and S2 incorporate (x,y) into the slice, where (x,y) is the label of the level i vertex, a procedure called *adjust* is used to incorporate (x,y) into the table: either to reflect the existence of an edge (x,y), or to reflect the fact that both boundaries contain a common node $x=y$. For rule S3, a table is computed directly for the level 1 edge.

The table processing corresponding to rule S4 is tricky. In the second case of rule S4. where $LB(v)=RB(v)$, the desired table is constructed directly for the desired slice by the procedure *create*. But consider the first case of rule S4, with $LB(v)\neq RB(v)$, which must take level $i-1$ slices and add level i edge (x,y) plus edges from x or y to the level $i-1$ nodes. The basic idea of the table manipulation is to use *create* to create an initial level i table, *extend* to turn each level $i-1$ table into a level i table, and *merge* to merge these tables. The tricky part is that no edge may cross a slice boundary in the course of doing the table manipulation. In particular, in order to handle the edges from x and y to the next lower level, a point must be found between the edges from x and the edges from y. Fortunately, because of the planarity constraint, there is some level $i-1$ node z_p such that all the nodes other than z_p adjacent to x are clockwise from z_p, while all the nodes other than z_p adjacent to y are counterclockwise from z_p. Only z_p can be adjacent to both x and y. Thus, the approach is to find z_p, use *create* to construct an initial level i table for a subgraph containing z_p, and then to extend the tables on one side using x and the tables on the other side using y.

For example, consider the slice shown in Figure 14(i) for the level 2 vertex labeled (e,a). The algorithm finds that $z_p=G$ is a point between the level 1 nodes adjacent to e and the level 1 nodes adjacent to a. *Create* constructs a level 2 table T for the middle subgraph of Figure 14(ii). This subgraph is not a slice according to rules S1-S4 but may be considered to have boundaries e,G and a,G. The table for the level 1 leaf labeled (F,G) is extended to include the level 2 node e and the edge (e,G) and is then merged with T. Next, the table for the level 1 leaf labeled (G,A) is extended to include the level 2 node a and the edges from a to G and A. Finally, the last two tables are merged to give a table for the whole slice shown in Figure 14(i).

We describe procedures *adjust*, *merge*, *contract*, *create*, and *extend* informally since the details of the table manipulations are tedious and straightforward.

(1) *adjust(T)*. This operation checks the relationship between the two highest level nodes in the boundaries of the slice represented by T and modifies T accordingly. In particular, suppose the highest

```
procedure table(v)
begin
let (x,y) be the label of v;
let i be the level of v;
if v is a face vertex and face(v) encloses no level i+1 component
        then begin
        T= table (u), where u is the leftmost child of v;
        for each other child c of v from left to right
                T = merge (T, table (c));
        return (adjust(T));
        end
else if v is a face vertex and face(v) encloses a level i+1 component C
        then return ( adjust( contract ( table ( root (C̄)))));
else if v is a level 1 leaf
        then return a table representing the edge (x,y);
else /* v is a level i leaf, i>1 */
        begin
        let f be the level i face enclosing the component for v;
        let the labels of the children of vertex( f ) be (z1,z2),
                (z2,z3), ..., (zm,zm+1)
        if y is adjacent to some zr, LB (v) ≤ r ≤ RB (v),
                then let p be the least such r
        else p = RB (v);
        /* note zp is a point between nodes adjacent to x and nodes adjacent to y */
        T = create (v,p);

        /* extend the leftmost p tables to include x and edges from x to zr, r ≤ p,
                and merge with T */
        j = p − 1;
        while j ≥ LB (v) do begin
                T = merge ( extend (x, table (uj ), T)),
                        where uj is the jth child of vertex( f );
                j = j − 1;
                end

        /* extend the remaining tables to include y and edges from y to ur, r ≥ p,
                and merge with T */

        j = p;
        while j < RB (v) do begin
                T = merge ( T , extend (y, table (uj ) )),
                        where uj is the jth child of vertex( f );
                j = j + 1;
                end
        return(T);
        end
end
```

Figure 13. The dynamic programming algorithm.

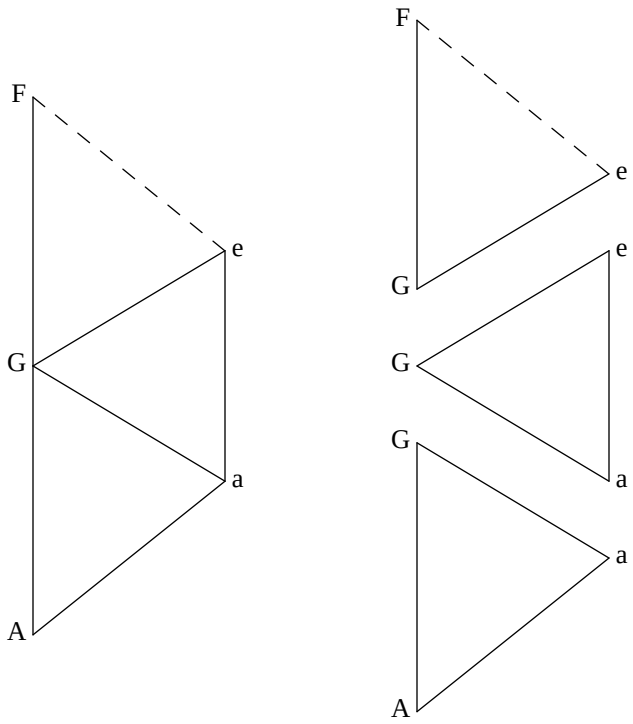


Figure 14. Computing a table for the slice of the vertex labeled (e,a).

level nodes in the left and right boundaries are x and y, respectively. If $x=y$, then any entry requiring exactly one of x and y to be in the maximum independent set is set to "undefined," and for any entry with both in the independent set, the count of nodes in the independent set is corrected to avoid counting $x=y$ twice. If $x\neq y$ and (x,y) is an edge in the graph, any entry that requires both x and y to be in the set is set to "undefined."

(2) *merge*$(\mathrm{T}_1,\mathrm{T}_2)$. This operation merges two tables T_1 and T_2 for level i slices S_1 and S_2, respectively, that share a common boundary, *i.e.* the right boundary of S_1 is the same as the left boundary of S_2. The resulting table will be for a slice whose first boundary is the left boundary of S_1 and whose second boundary is the right boundary of S_2. For each pair of vectors $\overline{u}$, $\overline{v}$ representing whether each vertex in the boundaries of the new slice is in the set, the new value will be the maximum over all $\overline{z}$ of the value in T_1 for $\overline{u}$, $\overline{z}$ plus the value in T_2 for $\overline{z}$, $\overline{v}$ minus the number of 1's in $\overline{z}$ (to avoid counting any vertex twice).

(3) *contract(T)*. This operation changes a level $i+1$ table T into a level i table T' (with a shorter boundary). Here, T is the table for $root(\overline{C})$, where $\overline{C}$ is the tree of a level $i+1$ component C within a level i face f, and T' is the table for $vertex(f)$. Let $S=slice(root(\overline{C}))$ and $S'=slice(vertex(f))$. For some $\overline{u}$ and $\overline{v}$, the left and right boundaries, respectively, for $slice(vertex(f)$ are $x,\overline{u}$ and $y,\overline{v}$. By construction, the label of the root of $\overline{C}$ is (z,z) for some z, and the boundaries of S are $z,x,\overline{u}$ and $z,y,\overline{v}$. For each pair of (0,1)-valued vectors representing whether each node of x, $\overline{u}$, y, and $\overline{v}$ is in the independent set, T has two values: one for z in the set, and one for z not in the set. "Contract" picks the larger of these two values as the new value for $x,\overline{u}$ and $y,\overline{v}$. The resulting table has 2^{2i} entries, reflecting the boundaries of length i of S'.

(4) *create(v,p)*. In this case, v is a leaf of a tree for a level $i+1$ component enclosed by a face f, and $p\leq t+1$, where the children of $vertex(f)$ are u_1, u_2, ..., u_t. This operation creates a table for the

subgraph including (i) the edge (x,y) represented by v, (ii) the subgraph induced by the left boundary of u_p, if $p \le t$, or the right boundary of u_p if $p = t+1$, and (iii) any edges from x or y to the level i node of this boundary. Since at most $i+2$ nodes and $i+2$ edges can occur in this slice, each entry of the table can be computed in $O(i)$ time.

(5) *extend(z,T).* Given a table T for a level i slice and a level $i+1$ node z, this operation computes a table for a level $i+1$ slice as follows. The boundaries of the new slice will be the old boundaries plus z. For any $\overline{u},\overline{v}$ representing whether each of the boundary points of the level i slice is in the maximum independent set, the new table has two entries: one for z in the set, and one for z not in the set. For z not in the set, the value in the new table will be the same as the old value for $\overline{u}$, $\overline{v}$. For z in the set, the new value is undefined if z and a level i boundary node are both in the set and are adjacent, and one more than before otherwise.

We claim that the above algorithm produces a correct table for the slice of the root of the level 1 tree, this slice includes the whole graph, and the boundaries of this slice are both a, where (a,a) is the label of the root of the level 1 tree. By definition of tables, the table includes four values, according to whether each of the boundary nodes is in the independent set. Two of the values are undefined since they represent inconsistency as to whether a is in the set. Taking the best of the remaining two values gives the solution for the maximum independent set.

Proof of Correctness

First, we show that the recursion is finite.

LEMMA 1. Calling the main procedure *table* on the root of the level 1 tree leads to exactly one recursive call on every other vertex of each tree of each level.

Proof. First, we show by contradiction that *table* cannot be called more than once on the same tree vertex. For if so, list in order the successive vertices on which *table* is called, and let u be the first vertex that appears for the second time in the list. If u is the root of a tree, *table* is called on u only from within a call on *vertex*(f), where f is the face enclosing the component corresponding to u; since no second call has occurred on *vertex*(f) by the time the second call occurs for u, u cannot be the root of a tree. Let v be the parent of u. If $face(v)$ does not enclose a higher level component, then *table* is called on u only from within a call on v; as before, since v has not had a second call, this case cannot apply. Therefore, $face(v)$ encloses a higher level component C, and *table* is called on u only while processing a leaf of $\overline{C}$. Suppose u is the jth child of v. If *table* is called on u while processing a leaf y of $\overline{C}$, then according to the algorithm, $LB(y) \le j < RB(y)$. But the definitions of LB and RB imply that each is nondecreasing for leaves taken from left to right, and $RB(y) = LB(z)$, where z is the right sibling (if any) of y. Therefore, $j < RB(y) = LB(z)$ and *table* cannot be called on u while processing any leaf of $\overline{C}$ to the right of y. We conclude that *table* is called at most once on u, contradicting the choice of u.

Next, we show that the main procedure is called on every vertex. We do this inductively by showing that for every vertex v, a call on v results in a call on every descendant of v and on every vertex of a tree for a component enclosed by a face corresponding to v or a descendant of v. For a level k vertex v, there are no enclosed components and the structure of the algorithm causes recursive calls on descendants. Assume that the statement is true for level $j \le k$ and for descendants of a level $j-1$ node v. If v is a leaf, the statement is vacuous. So suppose v is not a leaf. If $face(v)$ doesn't enclose any higher-outerplanar component, we need only apply the induction hypothesis to obtain the desired result. If $face(v)$ encloses a higher-outerplanar component, a call on v results in a call on the root of the tree for this component, and by the induction hypothesis, this call results in calls on all the leaves of this tree. But calls on the leaves result in calls on the children of v; because of the way LB and RB are defined, a call is made on every child of v. By the induction hypothesis applied to the children of v, the desired result holds. □

LEMMA 2. A call of *table* on the root of the level 1 tree results in a correct table for the corresponding slice.

Proof. We show that a call on a vertex v results in a correct table for the slice of v. The proof is by induction on the number of recursive calls caused by a call on v. By Lemma 1, the number of recursive calls is always finite. We assume that the procedures *adjust*, *merge*, *extend*, *create*, and *contract* are implemented correctly, and that the algorithm computes a table correctly for a level 1 leaf.

If a call on v does not generate any recursive calls, either v is a level 1 leaf, or v is a level i leaf, $i>1$, $LB(v)=RB(v)$, and a table is computed by *create*. In either case, the resulting table is correct by assumption.

Assume for j that whenever at most j recursive calls are made a correct table is computed. Suppose the call on a level i vertex v generates $j+1$ recursive calls. We show that the table computed for v must also be correct. Let (x,y) be the label of v.

First, suppose v is a face vertex and $face(v)$ encloses no level $i+1$ component. By rule S1, the slice for v is the union of the slices of v's children plus (x,y) if (x,y) is an edge. Also, the left (right) boundary of the slice of v is the same as the left boundary of v's leftmost (rightmost) child by Proposition 1 above. After the tables for the children are merged in pairs, the resulting table is for a slice consisting of the union of the children's slices and the left (right) boundary of the slice is derived from the leftmost (rightmost) child. By Proposition 1, the left boundary includes x and the right boundary includes y. However, at this point the table assumes that $x\neq y$ and (x,y) is not an edge. It also assumes that all lower level boundary nodes are distinct, even though they might be duplicated on the left and right boundaries. However, our definition of a level i table requires that any duplicate boundary nodes not at level i be treated as if they are distinct. Therefore, the table is correct except for the treatment of x and y. If $x=y$, *adjust* sets to "undefined" any table entry requiring exactly one of them to be in the independent set and corrects the count in any table entry requiring both to be in the independent set. If $x\neq y$ and (x,y) is an edge, *adjust* sets to "undefined" any entry that requires both x and y to be in the independent set. Thus, the final table computed for v is correct.

Second, suppose v is a face vertex and $face(v)$ encloses a level $i+1$ component C. By rule S2, the slice for v is the subgraph consisting of the slice of $root(\overline{C})$ plus (x,y) if (x,y) is an edge. Also, the left (right) boundary of v's slice is the same as the left (right) boundary of v's leftmost (rightmost) child by Proposition 1. Now, the left (right) boundary of the slice of $root(\overline{C})$ is the same as the left (right) boundary of the slice of the leftmost (rightmost) leaf of $\overline{C}$ by Proposition 1 applied inductively. Let u_L and u_R be the leftmost and rightmost leaves of $\overline{C}$, respectively. Let (c,d) be the label of the root of $\overline{C}$. By the definition of tree labeling, c is the first node in the label of u_L and d is the second node in the label of u_R. By rule B1 of the definition of *LB* and *RB*, $LB(u_L)=1$ and $RB(u_R)=r+1$, where r is the number of children of v. By rule B2 of the definition of boundaries, the left boundary of u_L is c plus the left boundary of the leftmost child of v, and the right boundary of u_R is d plus the right boundary of the rightmost child of v. Therefore, the boundaries of v's slice are the same as those of the slice of the root of $\overline{C}$ except for the extra level $i+1$ nodes c and d for the latter slice. Consequently, the table computed by the algorithm for the root of $\overline{C}$ is correct for v except for (i) the extra nodes in the boundaries (ii) the nonincorporation of (x,y) if (x,y) is an edge, and (iii) ignorance of the duplication of x if $x=y$ (since duplication is taken into account only for level $i+1$ nodes). The procedure *contract* corrects for (i). By Proposition 1 and the definition of boundaries, x and y are on the left and right boundaries, respectively of the slices of $\overline{C}$ and v. Hence, *adjust* corrects for (ii) and (iii). Therefore, the final table is correct for v.

Finally, suppose v is a leaf and $LB(v)<RB(v)$. By rule S4, $slice(v)$ is the subgraph consisting of (x,y), any edges from x or y to z_j, for $LB(v)\leq j\leq RB(v)$, and $slice(u_j)$, for $LB(v)\leq j<RB(v)$, where u_j is the jth child of the vertex for the enclosing face f. By rule B2 of the definition of boundaries, its left boundary is x plus the left boundary of the qth child of $vertex(f)$, where $LB(v)=q$, and its right boundary is y plus the right boundary of the $(t-1)$st child of $vertex(f)$, where $RB(v)=t$. Let the label of the jth child of $vertex(f)$ be (z_j,z_{j+1}) as in the algorithm, for $1\leq j\leq m$, where m is the number of children of $vertex(f)$. The algorithm finds a p such that if x is adjacent to z_j, then $j\leq p$, and if y is adjacent to z_j, then $y\geq p$. It applies *create* to construct a table T for a slice including the edge (x,y), edges from x and y to z_p, and the left boundary of u_p (if $p\leq m$) or the right boundary of u_{p-1} (if $p=m+1$). By applying $extend(x,table(u_j))$ for $j<p$, these tables are made compatible for merging with T on the left; similarly, $extend(y,table(u_j))$ applied to $j>p$ makes these tables compatible for merging with T on the right. Since the table mergers/extensions include precisely the tables for u_j, where $LB(v)\leq j<RB(v)$, and incorporate the edges

from x and y to z_j, where $LB(v) \le j \le RB(v)$, the final table for v is correct. □

We have shown that the table computed for the root R of the level 1 tree is correct for its slice. From Proposition 2, R's slice includes the slices of all tree vertices. By Proposition 3, every edge is included in the slice for some tree vertex. Therefore, the slice for R includes the whole graph. The table has four entries, according to whether the left and right boundary nodes are in the set. From Proposition 1, the boundaries of the slice are both a, where the label of the root is (a,a). The definition of tables requires that the entries be undefined where the corresponding subset would imply inconsistency as to whether a is in the set. Thus, there are just two meaningful values, representing the size of the maximum independent set for a in the set and for a not in the set. The maximum of these is obviously the size of the maximum independent set as claimed.

Performance bounds

Building the trees and computing boundaries of slices requires linear time. From above, the algorithm is called recursively at most once for each tree vertex. Since each leaf in a tree represents an oriented exterior edge, the number of vertices in trees is at most the number of edges in the k-outerplanar graph. For a planar graph, the number of edges is linear in the number of nodes. Therefore, the number of calls on the main procedure is linear in the number of nodes. Each call on *adjust*, *extend*, or *contract* requires $O(4^k)$ time, each call on *merge* requires $O(8^k)$ time, and each call on *create* requires $O(k4^k)$ time. The time used for these five operations dominates the time for other bookkeeping operations. Each of these five operations is performed at most once per recursive call on the main procedure. Therefore, the algorithm uses $O(8^k n)$ time, where n is the number of nodes in the k-outerplanar graph.

The space required is $O(4^k n)$. If only the size of the maximum independent set is desired, the tables can be deleted after they are used. If the actual solution set is needed, the tables can be kept and the set computed by retracing the steps of the computation.

This completes the proof of Theorem 1.

5. Adapting the above algorithm to handle problems other than maximum independent set

The appendix contains comments about algorithms for the other problems mentioned in the introduction. The following are some general comments.

First, we discuss how to solve the problems optimally on k-outerplanar graphs. The structure of each algorithm is the same; the only difference is in the tables used in the dynamic programming. For some r determined by each problem, each table for a slice of level at most k has at most r possible values for each of the k nodes in the boundary. (These values encode information such as whether the node or an edge incident on the node is in the solution set, or in the more complex case of H-matching, what part of H each node and boundary edge represent and what part of H must lie on each side of the boundary.) Thus, the table has at most r^{2k} entries, and if the encoding is chosen carefully, tables can be merged in $O(r^{3k})$ time or $O(r^{4k})$ time. For independent set, vertex cover, dominating set, and edge dominating set, $r = 2$; for minimum maximum matching and 3-coloring, $r = 3$. For H-matching, r depends on H. (See the appendix for the definition of H-matching.) For Hamiltonian path and cycle, $r = O(k)$.

To incorporate the above algorithms on k-outerplanar graphs into approximation algorithms on general planar graphs, the general planar graph must be decomposed into k-outerplanar graphs. The details of the decomposition differ for the various problems listed in the first section, but are essentially like that described for maximum independent set. (The greatest difference is for maximum H-matching, for which to get a solution at least $(k-1)/k$ optimal, kh-outerplanar graphs must be used, where h is the diameter of H. More details are given in the appendix.)

Acknowledgment. The author would like to thank Tom Leighton and Bob Tarjan for helpful discussions and David Johnson and Eric Grosse for comments on a draft of this paper.

References

[BE] R. Bar-Yehuda and S. Even, "On approximating a vertex cover for planar graphs," *Proc. of 14th ACM Symposium on Theory of Computing* (1982), 303-309.

[BLS] F. Berman, F. T. Leighton, and L. Snyder, "Optimal tile salvage," VLSI Memo No. 82-119, M.I.T. (1982).

[BJLSS] F. Berman, D. S. Johnson, F. T. Leighton, P. W. Shor, and L. Snyder, "Generalized planar matching," *J. Algorithms* 11,2 (1990),153-184.

[BM] D. Bienstock and C. L. Monma, "On the complexity of embedding planar graphs to minimize certain distance measures," *Algorithmica 5* (1990), 93-109.

[BJ] T. N. Bui and C. Jones, Finding optimal vertex separators in planar graphs with small separators, draft (1992), Computer Science Department, Pennsylvania State University, University Park, Pennsylvania.

[BP] T. N. Bui and A. Peck, "Partitioning planar graphs," *SIAM J. on Comput. 21,2* (1992), 203-215.

[CNS1] N. Chiba, T. Nishizeki, and N. Saito, "Applications of the Lipton and Tarjan's planar separator theorem," *J. Info. Proc. 4,4* (1981), 203-207.

[CNS2] N. Chiba, T. Nishizeki, and N. Saito, "An approximation algorithm for the maximum independent set problem on planar graphs," *SIAM J. on Comput. 11,4* (1982), 663-675

[CGH] E. Cockayne, S. Goodman, and S. Hedetniemi, "A linear algorithm for the domination number of a tree," *Info. Proc. Letters 4* (1975), 41-44.

[D] H. N. Djidjev, "On the problem of partitioning planar graphs," *SIAM J. on Alg. and Disc. Meth. 3,2* (1982), 229-240.

[DLT] D. Dolev, F. Leighton, and H. Trickey, "Planar Embedding of Planar Graphs," *Advances in Computing Research, Vol. II: VLSI Theory* (ed. F. Preparata), JAI Press, Inc., Greenwich, CT (1984), 147-161.

[GJ] M. R. Garey and D. S. Johnson, *Computers and Intractability, A Guide to the Theory of NP-Completeness*, W. H. Freeman and Co., San Francisco (1979).

[G] F. Gavril, "Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph," *SIAM J. Comput. 1,2* (1972), 180-187.

[Har] F. Harary, *Graph Theory*, Addison-Wesley, Reading, Massachusetts (1971).

[Ho] D. S. Hochbaum, "Efficient bounds for the stable set, vertex cover and set packing problems," W.P. #50-80-81, GSIA, Carnegie-Mellon University, May, 1981.

[HT] J. Hopcroft and R. E. Tarjan, "Efficient planarity testing," *J. ACM 21* (1974), 549-568.

[L] F. T. Leighton, personal communication (1983).

[LT1] R. J. Lipton and R. E. Tarjan, "A separator theorem for planar graphs," *SIAM J. Appl. Math. 36,2* (1979), 177-189.

[LT2] R. J. Lipton and R. E. Tarjan, "Applications of a planar separator theorem," *SIAM J. Comput. 9,3* (1980), 615-627.

[M] S. Mitchell, *Algorithms on trees and maximal outerplanar graphs: design, complexity analysis, and data structures study*, Ph.D. thesis, Univ. of Virginia, 1977.

[MCH] S. L. Mitchell, E. J. Cockayne, and S. T. Hedetniemi, "Linear algorithms on recursive representations of trees," *J. Comp. Syst. Sci. 18* (1979), 76-85.

[MH] S. L. Mitchell and S. T. Hedetniemi, "Edge domination in trees," *Proc. 8th Southeastern Conf. on Combinatorics, Graph Theory, and Computing*, Winnipeg (1975).

[PY] C. H. Papadimitriou and M. Yannakakis, "Worst-case ratios for planar graphs and the method of induction on faces," *Proc. of 22nd Symposium on Foundations of Computer Science* (1981), 358-363.

[TNS] K. Takamizawa, T. Nishizeki, and N. Saito, Linear-time computability of combinatorial problems on series-parallel graphs, *J. ACM* 29,3 (1982), 623-641.

[YG] M. Yannakakis and F. Gavril, "Edge dominating sets in graphs," *SIAM J. on Appl. Math.* 38,3 (1980), 364-372.

Appendix

The introduction lists a number of problems besides maximum independent set as being solvable in polynomial time on k-outerplanar graphs for fixed k. Several of these have approximation schemes for planar graphs as well. For each problem, the overall strategy is similar to that for maximum independent set, but the details differ. In the following, we sketch for each problem whatever new ideas are needed for that problem.

Problems that have approximation schemes for planar graphs.

1. **Minimum vertex cover**

   Given a graph $G=(V,E)$ and a positive integer $K \le |V|$ is there a vertex cover of size K or less for G, *i.e.* a subset $V' \subseteq V$ with $|V'| \le K$ such that for each edge $(u,v) \in E$ at least one of u and v belongs to V' [GJ, problem GT1]?

   Minimum vertex cover has a linear time algorithm on k-outerplanar graphs that is similar to that of maximum independent set except in the details of bookkeeping. Each table contains the minimum size of a vertex cover for the slice covered by the table, subject to which boundary nodes are in the set.

   For fixed k, a linear time algorithm that finds a vertex cover of size at most $(k+1)/k$ optimal proceeds as follows, given a planar graph. For each i, $0 \le i < k$, it uses the above algorithm for $(k+1)$-outerplanar graphs to obtain optimal solutions for the overlapping $(k+1)$-outerplanar graphs induced by levels $jk+i$ to $(j+1)k+i$, $j \ge 0$. For each i, the union over j of the solutions gives a vertex cover for the whole graph. The algorithm picks the best of these as its approximation to optimal. To see that this approximation is at most $(k+1)/k$ optimal, consider any optimal solution S. For some t, $0 \le t < k$, at most $|S|/k$ nodes in S are in levels congruent to t (mod k). For $j \ge 0$, let S_j be the set of nodes in S in levels $jk+t$ through $(j+1)k+t$, and $\overline{S}_j$ the optimal solution for the subgraph induced by these levels. Clearly, $|\overline{S}_j| \le |S_j|$ for each j. The sum over j of the $|S_j|$'s is at most $((k+1)/k)|S|$ since only nodes in levels congruent to t (mod k) are counted twice. But the algorithm produces a vertex cover of size no larger than the sum of the $|\overline{S}_j|$'s.

   A polynomial time approximation scheme is obtained from the above approximation algorithm as in the case of maximum independent set.

2. **Minimum dominating set**

   Given a graph $G=(V,E)$ and a positive integer $K \le |V|$ is there a dominating set of size K or less for G, *i.e.* a subset $V' \subseteq V$ with $|V'| \le K$ such that for all $u \in V - V'$ there is a $v \in V'$ for which $(u,v) \in E$ [GJ, problem GT2]?

   Again, the linear time algorithm to solve the problem on k-outerplanar graphs differs from that of maximum independent set only in the bookkeeping operations. Each table contains the size of a minimum dominating set for the slice covered by the table, according to which boundary nodes are in the set. The approximation algorithms and schemes are similar to those of minimum vertex cover in using $(k+1)$-outerplanar graphs induced by levels $jk+i$ through $(j+1)k+i$, $j \ge 0$.

3. **Minimum edge dominating set**

   Given a graph $G=(V,E)$ and a positive integer $K \le |V|$ is there a set $E' \subseteq E$ of K or fewer edges such that every edge in E shares at least one endpoint with some edge in E' [GJ, problem GT2]?

   The structure of the linear time algorithm for minimum edge dominating set on k-outerplanar graphs is similar to that of maximum independent set, but the bookkeeping operations differ because a dominating set contains edges rather than vertices. Each table contains the size of a minimum edge dominating set according to which boundary nodes are endpoints of edges in the set for the slice covered by the table. The approximation algorithms and schemes are similar to those of minimum vertex cover in using $(k+1)$-outerplanar graphs induced by levels $jk+i$ through $(j+1)k+i$, $j \ge 0$.

4. **Maximum triangle matching**

   Given a planar graph $G=(V,E)$ and integer $K \le |V|/3$, are there at least K node-disjoint induced subgraphs that are triangles? (This is a variation on partition into triangles [GJ, problem GT11].)

The overall strategy on k-outerplanar graphs is similar to that for maximum independent sets, but the table mergers are more complicated. Each table entry contains the maximum number of disjoint triangles in the slice according to which boundary nodes are in triangles. Each triangle is either within a single level or within two successive levels. Therefore, when a boundary node is in a triangle, the triangle includes at least two edges not on the boundary. In merging two tables, when a node of the common boundary is in a triangle in both slices, the triangles are distinct but not disjoint and this value must be ignored in computing the value in the merged tables. By examining all pairs of vectors representing which nodes in the common boundary are in triangles in the two slices, a table merger can be accomplished in $O(16^k)$ time, and the overall algorithm runs in $O(16^k|V|)$ time.

For fixed k, to obtain a solution that is at least $(k-1)/k$ optimal, the approximation algorithm solves the problem for each k-outerplanar subgraph induced by deleting edges between levels congruent to i and $i+1$ (mod k), for each i, $0 \le i < k$, and takes the best of the solutions as its approximation. For some i, $0 \le i < k$, at most $1/k$ of the triangles in an optimal solution for the whole graph include edges between levels congruent to i and $i+1$ (mod k), since every triangle lies either entirely within one level or within two successive levels. Hence, the approximation is at least $(k-1)/k$ optimal.

An approximation scheme is obtained from the approximation algorithms in the same manner as for maximum independent set.

5. **Maximum H-matching and maximum tile salvage**

Let H be a connected graph with 3 or more nodes. Given $G=(V,E)$ and a positive integer k, does G contain k or more induced node-disjoint subgraphs isomorphic to H [BJLSS]? (Maximum triangle matching and maximum tile salvage [BLS] are subproblems.)

To solve H-matching on a k-outerplanar graph, each table T must encode the number of copies or partial copies of H in the slice covered by the table, according to what subgraphs of H occur along the boundaries. Since H is fixed and finite, there are a bounded number of possibilites for how each boundary node or edge can occur in H (including which of its neighbors in H are in the slice). Thus, for some r, each table has size $O(r^k)$. Table mergers can be done in time $O(r^{2k})$, and the overall time is linear in $|V|$.

Let h be the diameter of H. The approximation algorithm on planar graphs finds a solution that is at least $(k-1)/k$ optimal as follows. It applies the above algorithm to kh-outerplanar graphs obtained by deleting edges between levels congruent to $hi-1$ and hi (mod kh), where $0 \le i < k$, and takes the best of the k sets. For some i, $0 \le i < k$, at most $1/k$ of the copies of H in an optimal solution S have their highest level nodes in a level congruent to $hi+j$ (mod kh) for some j, $0 \le j < h$, since there are k such possible groups of levels. Because a copy of H spans at most h levels, removing edges between levels congruent to $hi-1$ and hi (mod kh) can break only copies of H whose highest levels are congruent to $hi+j$ (mod kh), $0 \le j < h$, *i.e.* at most $1/k$ of the copies in S. Therefore, the algorithm produces a solution whose size is at least $(k-1)/k$ optimal.

An approximation scheme is obtained from the approximation algorithms in the same manner as for maximum independent set.

Problems solvable in linear time on k-outerplanar graphs, but not amenable to approximation on general planar graphs.

1. **Minimum maximal matching**

Given a graph $G=(V,E)$ and a positive integer $K \le |E|$ is there a subset $E' \subseteq E$ with $|E'| \le K$ such that E' is a maximal matching, *i.e.* no two edges in E' share a common endpoint and every edge in $E-E'$ shares a common endpoint with some edge in E' [GJ, problem GT10]?

The algorithm on k-outerplanar graphs is similar to that for maximum independent set except in the bookkeeping. Each table contains the size of a minimum maximal matching according to whether each boundary node is on an edge in the set, not on an edge in the set but sharing a common endpoint with an edge in the set, or neither. In table mergers, each node on the common boundary must either be on an edge in exactly one set or must share a common endpoint with an edge in the merged set.

The usual approximation approach does not work for minimum maximal matching because a subset of an optimal solution may not satisfy the requirement that every edge in $E-E'$ share a common endpoint

with an edge in E'.

2. **Three-colorability**

Given a graph $G=(V,E)$, is G 3-colorable, *i.e.* can the nodes be colored with three colors such that adjacent nodes are always assigned different colors [GJ, problem GT4]?

The algorithm on k-outerplanar graphs is similar to that for maximum independent set except in the bookkeeping. A table encodes whether the slice can be 3-colored for each possible coloring of the boundary nodes. Thus, there are 3^{2k} entries in each table, and a table merger takes time $O(3^{3k})$.

3. **Hamiltonian circuit and Hamiltonian path**

Does G contain a Hamiltonian circuit [GJ, problem GT37]? Does G contain a Hamiltonian path [GJ, problem GT39]?

The overall strategy of the algorithms on k-outerplanar graphs is similar for both problems to that for maximum independent set, but the bookkeeping is quite different. Each table encodes sets of disjoint paths covering all nodes in the slice; the endpoints of each path must be boundary nodes. A boundary node may be either an endpoint of a path or an intermediate point on a path. For a given set of paths, a labeling of the nodes specifies for each node either that it is an intermediate node or the other endpoint (*i.e.* which boundary the other endpoint is on and its level). For each labeling of the boundary nodes, the table contains 1 if there is a set of paths consistent with the labeling, and 0 otherwise. A merger requires joining paths in the two slices. Since k is fixed, the overall running time is linear in $|V|$.

A problem solvable in polynomial time on k-outerplanar graphs.

Partition into perfect matchings

Given a graph $G=(V,E)$ and a positive integer $K \leq |V|$, can the vertices of G be partitioned into $r \leq K$ disjoint sets $V_1, V_2, ..., V_r$ such that the subgraph induced by each V_i is a perfect matching (consists entirely of vertices of degree one) [GJ, problem GT16]?

The algorithm on k-outerplanar graphs is similar to that of maximum independent set except in the bookkeeping. For each slice, a label for a boundary node specifies a number from 1 to K and whether the node is adjacent to a vertex with the same number in the slice. For each labeling of the boundary nodes of a slice, a table contains 1 if there is a partition of the slice consistent with the labeling and consistent with being extended into a solution for the whole graph, and 0 otherwise. Since $K \leq |V|$ and k is fixed, the size of the tables and the overall running time are polynomial in $|V|$.