EDGE-PACKING IN PLANAR GRAPHS

Lenwood S. Heath and John Paul C. Vergara

Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061
heath,jpv@cs.vt.edu

October 4, 1995

Abstract

Maximum G Edge-Packing ($EPack_G$) is the problem of finding the maximum number of edge-disjoint isomorphic copies of a fixed guest graph G in a host graph H. This paper investigates the computational complexity of edge-packing for planar guests and planar hosts. Edge-packing is solvable in polynomial time when both G and H are trees. Edge-packing is solvable in linear time when H is outerplanar and G is either a 3-cycle or a k-star (graphs isomorphic to $K_{1,k}$). Edge-packing is NP-complete when H is planar and G is either a cycle or a tree with ≥ 3 edges. A strategy for developing polynomial-time approximation algorithms for planar hosts is exemplified by a linear-time approximation algorithm that finds a k-star edge-packing of size at least 1/2 optimal.

1 Introduction

An *edge-packing* of a graph H is a set of edge-disjoint subgraphs of H. An *edge-partition* of H is an edge-packing of H with every edge of H in some subgraph (i.e., the union of the subgraphs in the packing is exactly H). In a typical edge-packing or edge-partitioning problem, each subgraph in an edge-packing is restricted to have some property. For instance, edge-partitioning a graph into the smallest number of subgraphs, each of which is a matching, is equivalent to the chromatic index problem, the problem of minimally coloring the edges of a graph such that no two adjacent edges have the same color. Edge-packing arises in a variety of other contexts [4, 5, 8, 9, 17, 25, 28, 32].

In this paper, we investigate *Maximum G Edge-Packing* ($EPack_G$), the problem of finding the maximum number of edge-disjoint copies of a fixed guest graph $G = (V_G, E_G)$ in a host graph $H = (V_H, E_H)$.

MAXIMUM G **EDGE-PACKING** ($EPack_G$)

INSTANCE: A host graph H and an integer $K \leq |E_H|/|E_G|$.

QUESTION: Does H contain K edge-disjoint copies of G, i.e., are there K subsets $E_1, E_2, \ldots, E_K$ of E_H such that $E_i \cap E_j = \emptyset$ whenever $i \neq j$, and the subgraph induced by each E_i is isomorphic to G?

As $EPack_G$ is trivial when G is a single edge, we assume that G has at least two edges. Furthermore, we assume that G is connected. Under these assumptions, it suffices to consider only connected hosts. The host class of planar graphs as well as the subclasses of trees and outerplanar graphs are considered in this paper.

Masuyama and Ibaraki [30] show that $EPack_G$ is solvable in linear time when G is a path of length 2 and H is arbitrary. Corneil, Masuyama, and Hakimi [11] investigate $EPack_G$ for arbitrary host graphs. They show that $EPack_G$ is NP-complete when H is arbitrary and G is connected and has at least 3 edges. Their proofs involve reductions from Exact 3-Cover.

When $K = |E_H|/|E_G|$ in the $EPack_G$ instance, we have the special case where all edges are used in the edge-packing. We call this problem *G Edge-Partition* ($EPart_G$), since it aims to partition the edges of H into copies of G.

G **EDGE-PARTITION** ($EPart_G$)

INSTANCE: A host graph H such that $|E_H| = K|E_G|$ for some integer K.

QUESTION: Can the edges of H be partitioned into copies of G?

NP-completeness results for $EPart_G$ apply unchanged to $EPack_G$, since $EPart_G$ is a special case of $EPack_G$.

Some results on $EPart_G$ have already been obtained [1, 6, 7, 12, 13, 22]. $EPart_G$ is NP-complete for connected guest graphs with at least three edges [12]. Dyer and Frieze [13] consider planar host graphs and find that the problem remains NP-complete for guests that are paths or 3-stars but is solvable in polynomial time for guests that are triangles.

Another related problem is the vertex-disjoint counterpart of $EPack_G$, Maximum G Matching (finding the maximum number of vertex-disjoint copies of G in H), a problem that has been studied extensively. Hell and Kirkpatrick [19, 20, 21, 27, 26] have a number of complexity results. They [26] prove NP-completeness for this problem in general (reduction from 3-Dimensional Matching), while Berman, et al. [3] prove NP-completeness for planar hosts (reduction from PLANAR 3-SAT).

In this paper, we show a number of results for edge-packing in planar hosts. When H is a tree, edge-packing is solvable in polynomial time for any G and in linear time for G a path or a star. Edge-packing is solvable in linear time when H is outerplanar and G is either a 3-cycle or a k-star. Edge-packing is NP-complete when H is planar and G is either a cycle or a tree with ≥ 3 edges. (In other work, we [18] provide NP-completeness proofs for edge-packing when H is planar and G is an arbitrary graph containing at least one cycle.) The approximability of $EPack_G$ for planar hosts is considered. A strategy for developing polynomial-time approximation algorithms for planar hosts is exemplified by a linear-time approximation algorithm for guests that are k-stars that finds an edge-packing of size at least 1/2 optimal.

We shall denote particular instances of $EPack_G$ by replacing G with the actual guest graph considered; for example, $EPack_G$ where G is a 3-cycle is alternatively denoted by $EPack_{3-cycle}$. Also, whenever the host graph H is restricted to have some property P, we denote this by $EPack_G(P)$; for example, $EPack_{3-cycle}(planar)$ is $EPack_{3-cycle}$ for planar host graphs. We use the notation $EPack_G(general)$ to signify that the host graph may be an arbitrary, not necessarily planar, graph.

We overload the notation $EPack_G$ in two ways. First, $EPack_G$, when appropriate in the context, corresponds to the obvious optimization problem rather than the decision problem. Second, $EPack_G[H]$ denotes the number of copies of G in a Maximum G Edge-Packing of H.

The paper is organized as follows. Section 2 presents a polyunomial-time algorithm for $EPack_G(tree)$ for any G. In section 3, we provide NP-completeness proofs for several types of planar guest graphs, namely, stars, trees, and cycles. In section 4, we provide linear-time algorithms for $EPack_{3-cycle}(outerplanar)$ and $EPack_{k-star}(outerplanar)$. Section 5 investigates the approximability of $EPack_G(planar)$. In Section 6, we conclude with open problems as well as a table summarizing the current status of $EPack_G$.

2 Tree Guests in Tree Hosts

Maximum G Edge-Packing is solvable in polynomial time for any guest graph G if the host graph H is a tree. Let d be the maximum degree of any vertex in G. Section 2.1 presents an algorithm with time complexity $O(|E_H|^d)$. Section 2.2 identifies special cases where more efficient algorithms can be obtained.

2.1 $EPack_G(tree)$

For $G = (V_G, E_G)$ an arbitrary tree with at least three edges, we obtain a polynomial-time algorithm to solve $EPack_G(tree)$. The degree of the polynomial depends on d. Masuyama and Ibaraki [30] and, independently, Vergara and Heath [31] solve $EPack_{2-path}(tree)$ in linear time and show that the problem extends to arbitrary hosts. The algorithm in this section is a generalization to arbitrary tree guests and utilizes the same general idea of identifying copies of the guest graph G in a rooted host graph H starting at the leaves of H and going up to its root.

We distinguish between unrooted and rooted trees. The guest G is considered to be unrooted. If $v \in V_G$, then G_v denotes the rooted tree obtained by rooting G at v. If $u, v \in V_G$ and G_u is isomorphic to G_v as a rooted tree, then G_u and G_v are considered to be the same tree. The set of rooted trees associated with G is

$$root[G] = \{G_v : v \in V_G\}.$$

If G is as given in Figure 2.1, then $root[G] = \{G_a, G_b, G_c, G_d\}$. Note that $root[G]$ has four elements, not five, since G_d is isomorphic to G_e. If v is deleted from G_v, then one or more rooted subtrees remain. Each subtree (more precisely, its isomorphism class) is denoted by S_i for some integer i. The multiset of these subtrees is denoted by $subtrees[G_v]$. In Figure 2.1, $subtrees[G_b] = \{S_0, S_0, S_1\}$. The set union of all these multisets gives the set of all possible rooted subtrees of G:

$$subtrees[G] = \bigcup_{v \in V_G} subtrees[T_v].$$

In Figure 2.1, $subtrees[G] = \{S_0, S_1, S_2, S_3, S_4\}$. We shall consistently use S_0 to denote the single-node subtree.

We also have corresponding notation for the host graph. We assume that $H = (V_H, E_H)$ is a rooted tree, rooted arbitrarily at some vertex $r \in V_H$. For $v \in V_H$, $c(v)$ denotes the set of children of v. $H[v]$ denotes the subtree in H rooted at v.

Our main algorithm is Pack-tree (see Figure 2.4), which computes the maximum number of edge-disjoint copies of G within a given rooted tree R. We initially call Pack-

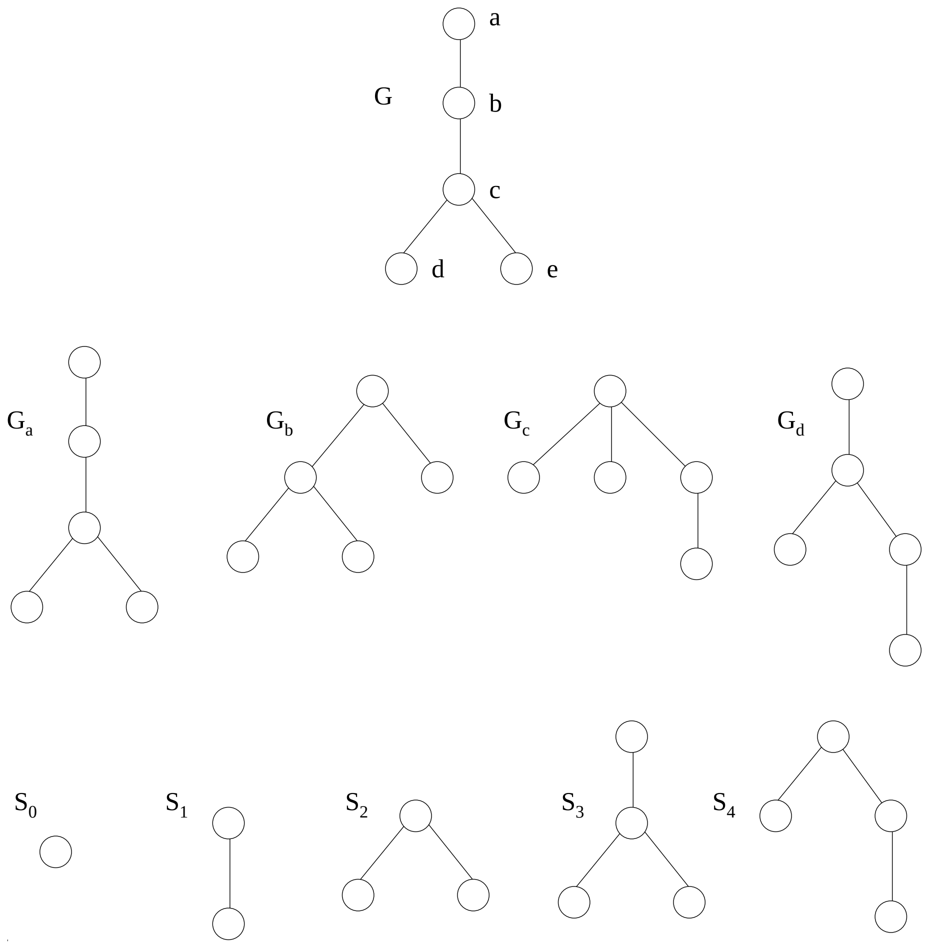

Figure 2.1: Rooted trees and subtrees of G.

TREE(H). In general, PACK-TREE(R) takes a subtree $R = H[v]$ and returns $EPack_G[R]$, the size of a maximum G edge-packing in R, and a subset $available[R]$ of $subtrees[G]$ described in the next paragraph.

Suppose a particular G edge-packing of R with $EPack_G[R]$ copies of G has been identified. There are some edges in R that are not in any of these copies; we call these *available* edges. By choosing some of these edges, a rooted subtree in $subtrees[G]$ can be identified that is rooted at v (recall $R = H[v]$). We call such a subtree an *available* subtree. The set $available[R]$ consists precisely of all available subtrees that occur over all possible maximum G edge-packings of R. Consider the guest G and a tree host R as in Figure 2.2.

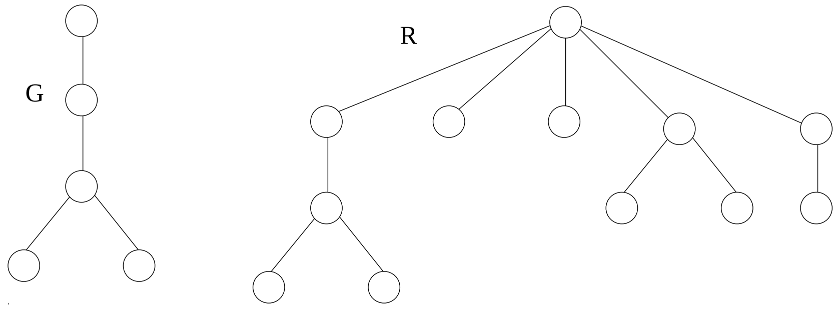

Figure 2.2: Guest G and host R.

Clearly, $EPack_G[R] = 2$. Figure 2.3 illustrates two ways of packing 2 copies of G in R (available edges are in dashed lines). The available set is $available[R] = \{S_0, S_1, S_2, S_3, S_4\}$ because each S_i exists as an available subtree in at least one of these two ways. Specifically any of S_0, S_1, S_2, and S_4 are available in the first packing, while any of S_0, S_1, S_2, and S_3 are available in the second packing.

When PACK-TREE($H[v]$) is invoked, it makes the recursive call PACK-TREE($H[w_i]$) for each child w_i of v (lines 7-10). Respective solutions ($p_i = EPack_G[H[w_i]]$ and $L_i = available[H[w_i]]$) are then collected. The solution for the current call to PACK-TREE is the sum (line 13) of all packings plus any remaining copies of G that can be formed from the available sets. The latter is obtained by a call to algorithm EXTRACT-TREES (line 11-12), which returns the number of copies of G and a resulting set of available subtrees. The base case in this recursive algorithm occurs when it is called with a leaf of H, in which case, it returns an edge-packing of size zero and available set $\{S_0\}$, the single-node tree (lines 2-4).

Algorithm EXTRACT-TREES (see Figure 2.5) takes as input a set A of sets of available subtrees. Its job is to determine how many copies of G can be constructed using v as root. It returns the maximum number of copies of G that can be formed from A and a subset of $subtrees[G]$, representing the set of possible available subtrees after a maximum packing is obtained. Note that this time a copy of G is formed by combining these subtrees with a root vertex. The algorithm involves partitioning the input set A (equivalently, the children

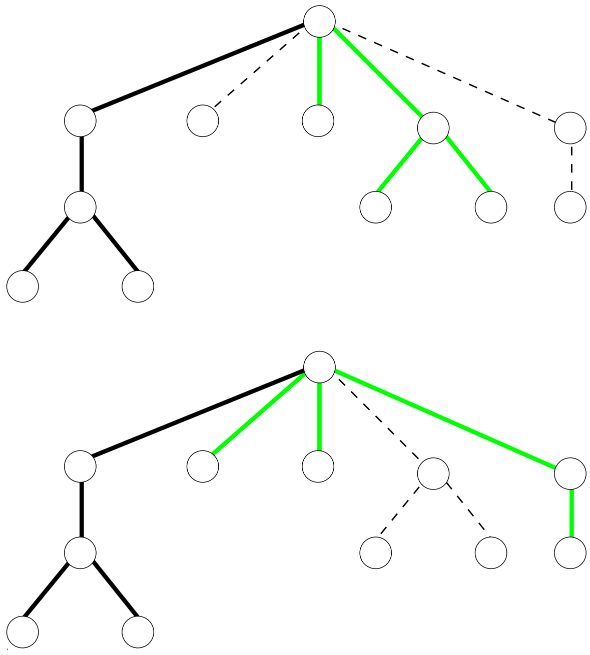

Figure 2.3: Two ways of packing two copies of G in R.

of v from PACK-TREE) into sets of size at most d (3 in the example of Figure 2.1). Recall that d, the maximum degree of a vertex of G is the maximum number of subtrees that combine to form a copy of G. All possible ways to partition A are considered (lines 1-2). For each possible partition (lines 3-5), EXTRACT-TREES calls TRY-PARTITION, a procedure that tests whether a copy of G can be constructed in each part of the partition using any of the rooted forms of G ($root[G]$). Algorithm TRY-PARTITION (see Figure 2.6) returns the number of copies of G obtained if a particular partition is used and a resulting available set of subtrees. The partitions (there could be more than one) in which the most copies of G are detected are then considered (lines 6-7). Algorithm EXTRACT-TREES returns the cardinality of a best partition as well as the union of all possible available sets.

Figure 2.7 illustrates how the partitions are handled as well as how solutions propagate up the tree. The guest remains the example of Figure 2.1. Here, the solutions for vertices w, x, y, and z have been gathered and PACK-TREE is being processed for vertex v. There

Algorithm Pack-tree(R). G edge-packing for trees.

INPUT: A rooted tree R.
OUTPUT: $EPack_G[R]$ and $available[R]$.

$v \leftarrow$ root of R;
if $(c(v) = \emptyset)$ **then**
  $\triangleright$ v is a leaf; return 0 and the tree with a single vertex
  return $(0, \{S_0\})$;
else
  $\triangleright$ process the children of v recursively
  begin
    $(w_1, w_2, \ldots, w_n) \leftarrow c(v)$;
    for $i \leftarrow 1$ **to** n **do**
      $(p_i, L_i) \leftarrow$ Pack-tree($R[w_i]$);
    $\triangleright$ extract copies of G from available sets
    $(q, S) \leftarrow$ Extract-trees($\{L_1, L_2, \ldots, L_n\}$);
    return$((\sum_{i=1}^{n} p_i) + q,\ S)$;
  end;

Figure 2.4: Algorithm Pack-tree.

are fourteen possible partitions of $\{w, x, y, z\}$ in which each part contains at most three elements:

$$\Big\{\{w\},\{\mathbf{x}\},\{\mathbf{y}\},\{z\}\Big\},\Big\{\{\mathbf{w},\mathbf{x}\},\{\mathbf{y}\},\{z\}\Big\},\Big\{\{\mathbf{w},\mathbf{y}\},\{\mathbf{x}\},\{z\}\Big\},\Big\{\{\mathbf{w},\mathbf{z}\},\{\mathbf{x}\},\{\mathbf{y}\}\Big\},$$
$$\Big\{\{\mathbf{x},\mathbf{y}\},\{w\},\{z\}\Big\},\Big\{\{\mathbf{x},\mathbf{z}\},\{w\},\{\mathbf{y}\}\Big\},\Big\{\{\mathbf{y},\mathbf{z}\},\{w\},\{\mathbf{x}\}\Big\},$$
$$\Big\{\{\mathbf{w},\mathbf{x}\},\{\mathbf{y},\mathbf{z}\}\Big\},\Big\{\{\mathbf{w},\mathbf{y}\},\{\mathbf{x},\mathbf{z}\}\Big\},\Big\{\{\mathbf{w},\mathbf{z}\},\{\mathbf{x},\mathbf{y}\}\Big\},$$
$$\Big\{\{\mathbf{w},\mathbf{x},\mathbf{y}\},\{z\}\Big\},\Big\{\{\mathbf{w},\mathbf{x},\mathbf{z}\},\{\mathbf{y}\}\Big\},\Big\{\{\mathbf{w},\mathbf{y},\mathbf{z}\},\{\mathbf{x}\}\Big\},\Big\{\{\mathbf{x},\mathbf{y},\mathbf{z}\},\{w\}\Big\}.$$

Note that a part in a partition has at most $d = 3$ elements. The parts in bold are those that can form a copy of G. For example, the part $\{w, z\}$ forms G_b because the set of available subtrees of the solution for w includes S_2 while the set for z includes S_0. S_0 and

Algorithm Extract-trees(A). Used in G edge-packing for trees.

INPUT: A set of sets of subtrees $A = \{U_1, U_2, \ldots, U_n\}$.
OUTPUT: The size of a maximum set of copies of G that can be formed from A and a set of available subtrees.

(partition[1],partition[2],...,partition[a]) $\leftarrow$ all partitions of $\{1, 2, ..., n\}$
    such that each part of the partition has size at most d;
    $\triangleright$ there are $a = O(n^{d-1})$ such partitions
for $i \leftarrow 1$ **to** a **do**
    $\triangleright$ obtain an edge-packing for each possible partition
    $(q_i, L_i) \leftarrow$ Try-partition(A, partition[i]);
$p \leftarrow \max_{1 \leq i \leq a} q_i$;
return(p,$\cup_{q_i = p} L_i$);

Figure 2.5: Algorithm Extract-trees.

S_2, when connected by a 2-path, form a copy of G_b. In the example, the partition which exhibits the most copies of G is $\{\{w, z\}, \{x\}, \{y\}\}$, which has 3 copies. This, combined with 1, 0, 0, and 2, the edge-packing solutions for w, x, y, and z, respectively, makes 6 copies of G for the tree rooted at v. Since all edges in the star $\{(v, w), (v, x), (v, y), (v, z)\}$ were used up in this maximum G edge-packing, S_0 is the only available subtree left.

We now formally state our result:

Theorem 2.1 *$EPack_G(tree)$, where G is a fixed tree, is solvable in $O(|E_H|^d)$ time.*

Proof: We perform induction on the subtrees of H. It is easy to verify that the base case, a call to Pack-tree($H[v]$) where v is a leaf of H, return a packing of size 0 and an available set consisting of the tree with a single vertex (S_0). Assume that we have obtained the maximum edge-packings (and available sets) for the subtrees of H rooted at the children of a given vertex v. Using the maximum edge-packings of these subtrees to compute the maximum edge-packing of the tree rooted at v is appropriate because at most

Algorithm Try-partition$(A, parts)$. Used in G edge-packing for trees.

INPUT: Sets of subtrees $A = \{U_1, U_2, \ldots, U_n\}$ and a partition $parts$ of $\{1, 2, \ldots, n\}$.

OUTPUT: The number of copies of G that can be formed from A using the partition $parts$ and a resulting available set of subtrees.

$p \leftarrow 0$;
$available \leftarrow \emptyset$;
$\triangleright$ obtain all m parts of the partition
$\{x_1, x_2, \ldots x_m\} \leftarrow parts$;
for $i \leftarrow 1$ **to** m **do**
    $\triangleright$ for each part determine if a copy of G exists
    begin
        $\{y_1, y_2, \ldots, y_l\} \leftarrow x_i$;
        $\triangleright$ $y_1, y_2, \ldots, y_l$ are integers from $\{1, 2, \ldots, n\}$
        if $\exists Z_1 \in U_{y_1} \exists Z_2 \in U_{y_2} \ldots \exists Z_l \in U_{y_l}$ such that
                $Z_1, Z_2, \ldots, Z_l$ form some $T \in root[G]$ **then**
            $p \leftarrow p + 1$;
        else
            $available \leftarrow available \cup x_j$
    end;
$L \leftarrow$ all rooted subtrees $S_k \in subtrees[G]$ formed from
        any combination $Z_1, Z_2, \ldots, Z_m$ such that
        $\{j_1, j_2, \ldots, j_m\} \subset available$ and $Z_1 \in U_{j_1}, Z_2 \in U_{j_2}, \ldots, Z_m \in U_{j_m}$;
return(p, L);

Figure 2.6: Algorithm Try-partition.

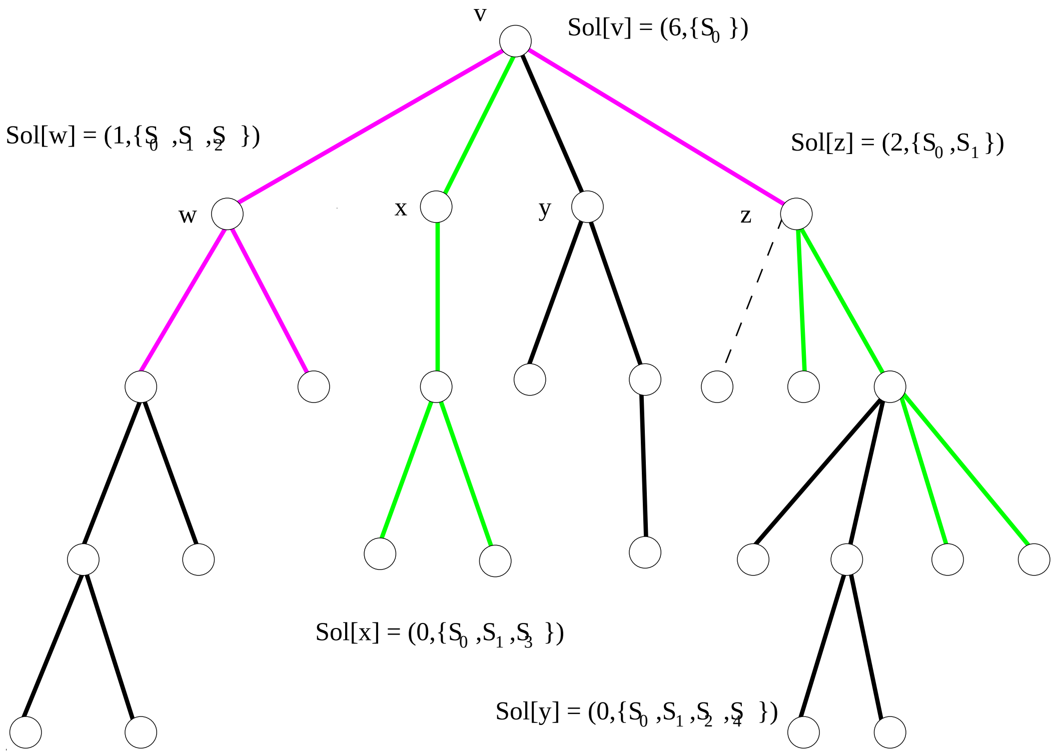

Figure 2.7: G-edge-packing for trees.

one other copy of G can be (partially) contained in each of the subtrees. This is handled by the fact that all possible available sets of edges that can be part of a copy of G using the current root and all relevant ways to partition the subtrees are considered and exhausted in the algorithm. Selecting the partition that yields the most additional copies of G and then combining this with the maximum edge-packings of the subtrees clearly results in a maximum edge-packing for the tree rooted at v. The number of possible partitions of a set of size n such that each part is at most some size k is equivalent to the number of partitions of the set into k parts [15]. There are $O(n^{k-1})$ such partitions. Therefore, there are $\mathrm{O}(|E_H|^{d-1})$ possible partitions at each call to EXTRACT-TREES. Since the number of T_i and the number of S_i are fixed, EXTRACT-TREES takes $O(|E_H|^{d-1})$ time and it follows that PACK-TREE takes $O(|E_H|^d)$ time. $\square$

2.2 Special Cases: Stars and Paths

We can exploit the structure of special kinds of trees such as a k-star ($K_{1,k}$, a complete bipartite graph where $k \geq 3$) and a k-path (a path with $k \geq 2$ edges). If algorithm PACK-TREE is followed, time complexities of $O(|E_H|^k)$ for k-stars and $O(|E_H|^2)$ for k-paths are obtained. However, improvements can be made for these cases, as stated in the following results.

Theorem 2.2 *$EPack_{k-star}(tree)$ is solvable in $O(|E_H|)$ time.*

Proof: There are only two ways to root a k-star: one with exactly k vertices as subtrees (the center of the star is its root) and another with exactly one subtree which in turn has exactly $k-1$ vertices as subtrees. The only relevant subtrees are a single vertex and a $(k-1)$-star rooted at its center. This makes the partitioning process in EXTRACT-TREES much simpler. Instead of partitioning into parts with sizes at most k, we simply detect the presence of $(k-1)$-star subtrees and use them to form one k-star each. The remaining subtrees are single vertices. For these, we collect k-tuples of vertices to form k-stars. If there are exactly k-1 single vertices left, these are returned as as a left-over subtree. Notice that this alternate process to EXTRACT-TREES involves a simple scan and takes linear time (instead of $O(|E_H|^{d-1})$) with respect to the number of children of the vertex being processed. Since PACK-TREE calls EXTRACT-TREES once per vertex, it is easy to verify that edge-packing in this case runs in $O(|E_H|)$ time. □

The following is a result previously obtained by Masuyama and Ibaraki [30].

Theorem 2.3 *$EPack_{k-path}(tree)$ is solvable in $O(|E_H|)$ time.*

Proof: In this case, the only relevant subtrees are paths themselves. Also, we take advantage of the fact that paths are subgraphs of longer paths. Therefore, instead of keeping track of all possible available subtrees, we just remember the longest available path since this represents all the other possible paths. In extracting k-paths (EXTRACT-TREES), instead of inspecting partitions with parts of size at most 2, we first detect $k-1$ paths since these immediately form k-paths. Now the only way other k-paths can be formed is to pair up two paths (a part of size 2 from a partition in the original EXTRACT-TREES algorithm) of appropriate lengths. The problem therefore reduces to finding as many pairs of paths (from a set of available paths) that form a k-path. The set of paths can be sorted by length in linear time using bin-sort and then the pairs can be selected making sure the shorter paths are used first (a straight-forward scan of the sorted list). This guarantees that a maximum set of paths is obtained and that the longest possible path is retained. EXTRACT-TREES therefore runs in linear time with respect to the children of the vertex being processed, as in the k-star case. Consequently, PACK-TREE takes $O(|E_H|)$ time. $\square$

3 Planar Hosts

In this section, we restrict the guest graph G to be a cycle or a tree with at least 3 edges. For such a G, we show that $EPack_G(planar)$ is NP-complete.

It is clear that $EPack_G(general)$ is in NP. To verify that K subgraphs of G are isomorphic to a fixed graph G and are edge-disjoint is a polynomial-time task. Our proofs that $EPack_G$ is NP-hard for the considered guests G are reductions from the known NP-complete problem PLANAR 3-SAT (Lichtenstein [29]). PLANAR 3-SAT is a variant of the problem 3-SAT (3-SATISFIABILITY) that involves boolean variables and clauses over the variables. A *literal* is a variable (positive literal) or the negation of a variable (negative literal). In a 3-SAT instance, a clause is a set containing at most three literals. A *truth assignment* is an assignment of *true* or *false* values to the variables. The clauses are said to

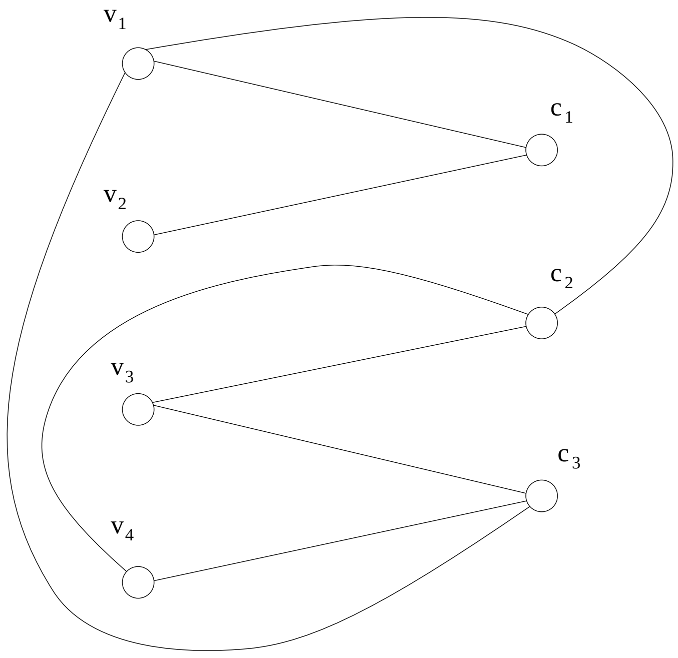

Figure 3.8: The graph of $(v_1 + v_2)(\overline{v_1} + v_3 + \overline{v_4})(v_1 + \overline{v_3} + v_4)$.

be *satisfied* by a truth assignment if for each clause, at least one of the literals in the clause evaluate to true under the assignment. For a set V of variables and a set C of clauses over V, define $G_p = (V \cup C, E)$, where $E = \{(v, c) : v \text{ or } \overline{v} \text{ belongs to the clause } c\}$. PLANAR 3-SAT has the added restriction that the graph G_p is planar.

PLANAR 3-SATISFIABILITY (PLANAR 3-SAT)

INSTANCE: A set V of m variables $\{v_1, v_2, \ldots, v_m\}$, and a set C of n clauses $\{c_1, c_2, \ldots, c_n\}$ over V where $|c_i| \leq 3$ for each c_i in C such that the bipartite graph G_p is planar.

QUESTION: Is there a satisfying truth assignment for C?

Figure 3.8 exhibits a PLANAR 3-SAT instance, where, $c_1 = (v_1 + v_2)$, $c_2 = (\overline{v_1} + v_3 + \overline{v_4})$, and $c_3 = (v_1 + \overline{v_3} + v_4)$.

We first investigate tree guests. Dyer and Frieze [13] prove NP-completeness for $EPack_G(planar)$ where G is a path. In Section 3.1, we prove that $EPack_{k-star}$ is NP-complete if $k \geq 3$. In Section 3.2, we extend the proof to tree guests that are forks (trees that are neither stars nor paths). Next, we investigate guests that are *cyclic graphs* (graphs containing cycles). In Section 3.3, we prove that $EPack_{k-cycle}$ is NP-complete, and in Section 3.4, we describe other cyclic graphs where our previous proof techniques apply.

3.1 k-Stars as Guests

In this section, we show that k-star edge-packing in planar graphs is NP-complete.

Theorem 3.1 *$EPack_{k-star}(planar)$, where $k \geq 3$, is NP-complete.*

Proof: We first consider the case of G a 3-star (Dyer and Frieze [13] have a proof for this case) and then explain how to extend the proof to k-stars in general.

We transform a PLANAR 3-SAT instance to an $EPack_G$ instance, a host graph H composed of *gadgets*. There are m variable gadgets V_i, where $1 \leq i \leq m$. Each V_i contains $4n$ vertices $u_i[j]$, $\overline{u_i}[j]$, $v_i[j]$ and $\overline{v_i}[j]$, where $0 \leq j \leq n-1$. The edges in V_i are those contained in the cycle $u_i[0]$, $\overline{u_i}[0]$, $u_i[1]$, $\overline{u_i}[1]$,..., $u_i[n-1]$, $\overline{u_i}[n-1]$, plus two sets of spikes: *positive spikes*, the edges $(u_i[j],\ v_i[j])$, and *negative spikes*, the edges $(\overline{u_i}[j],\ \overline{v_i}[j])$, where $0 \leq j \leq n-1$. Figure 3.9 shows what a variable gadget looks like if $n = 3$.

For each clause c_j, there is a clause gadget C_j $(1 \leq j \leq n)$. It contains $3 + |c_j|$ vertices, all but two of which, $c_j[0]$ and $c_j[1]$, are shared (identified) with vertices of the variable gadgets. The edges of C_j are those of a $(2 + |c_j|)$-star, the formation of which depend on the literals contained in the clause c_j. For each positive or negative literal in c_j, a positive or negative spike from the corresponding variable gadget is included in the clause gadget. As in Figure 3.10, suppose $c_j = (v_1 + v_2 + \overline{v_3})$. Then, the other vertices involved in C_j are $u_1[j_1]$, $u_2[j_2]$, $\overline{u_3}[j_3]$, and a vertex (the center of the star) identified with all of $v_1[j_1]$,

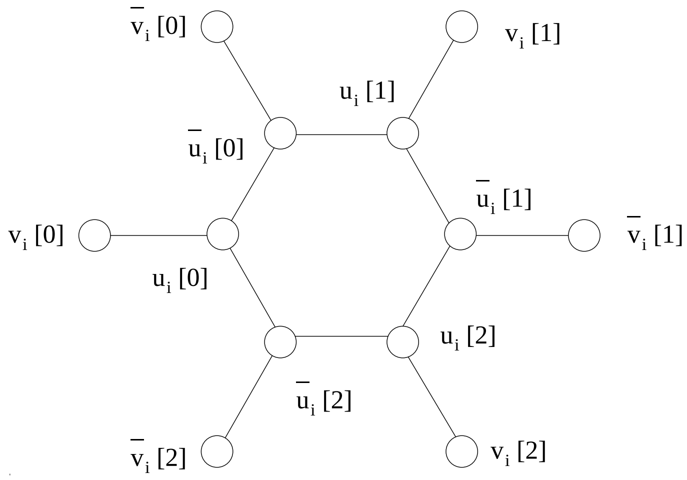

Figure 3.9: A variable gadget for 3-stars

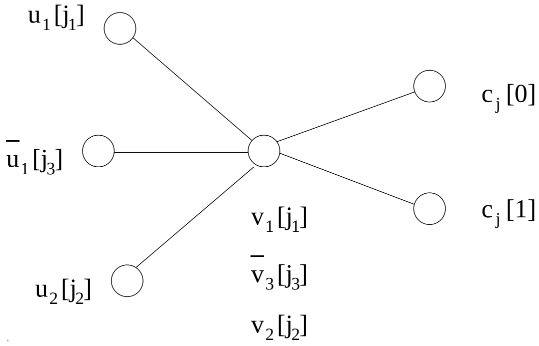

Figure 3.10: A clause gadget for 3-stars; assume $c_j = (v_1 + v_2 + \overline{v_3})$.)

$v_2[j_2]$ and $\overline{v_3}[j_3]$. The edges between these vertices are particular spike edges of the variable gadgets V_1, V_2 and V_3. j_1, j_2 and j_3 refer to the actual index appropriate for the clause c_j for each of the variable gadgets. These indices depend on the cyclic ordering of adjacent clause vertices to variable vertices in the PLANAR 3-SAT instance. The intention is to preserve the planarity of the graph in the transformation. Since the clause gadget is a $(2 + |c_j|)$-star and since $|c_j| \leq 3$, at most one 3-star can be extracted from it.

There are exactly two ways to choose a maximum set of 3-stars from a variable gadget. One way is to extract all n stars $(\overline{u_i}[j]\colon \overline{v_i}[j], u_i[j], u_i[j+1 \pmod n])$; the other is to choose $(u_i[j]\colon v_i[j], \overline{u_i}[j], \overline{u_i}[j-1 \pmod n])$. Note that the addition (or subtraction)

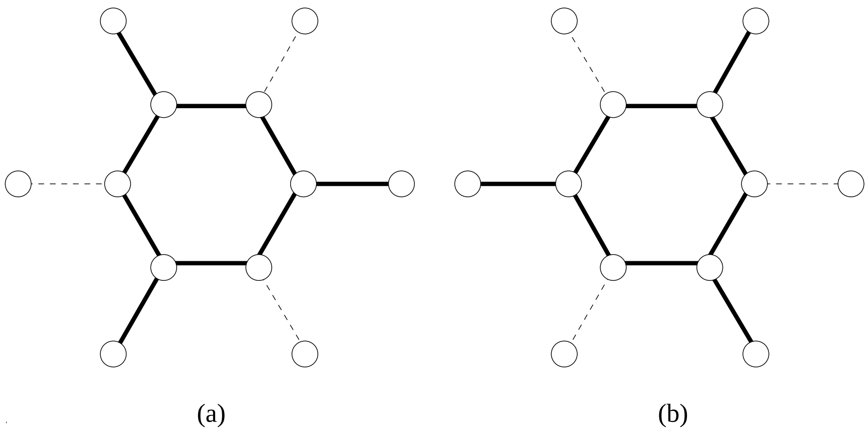

Figure 3.11: A variable gadget for 3-stars in its (a) true and (b) false modes.

within the indices wrap modulo n. These two choices correspond to true-or-false modes, as shown in Figure 3.11 (chosen edges are in bold). In both cases, all cycle edges and half of the spike edges are chosen. The chosen edges are precisely one of two sets of alternating spikes (positive or negative spikes); in either case, one set is chosen and the other is made available to a clause gadget.

Figure 3.12 illustrates the host H that corresponds to the PLANAR 3-SAT instance in Figure 3.8.

Since sharing a spike edge with a clause gadget signifies variable membership in a clause, the only way a 3-star can be extracted from a clause gadget is when at least one of these spikes is made available by a variable gadget, or equivalently, when an appropriate true-or-false assignment is made to a variable to satisfy the clause. There are n 3-stars that can be extracted from the m variable gadgets; there are n clause gadgets which can have at most one 3-star only if the situation described above occurs. This means that the value of K in our $EPack_G$ instance is $mn + n$; i.e., in our reduction from PLANAR 3-SAT, C is satisfiable if and only if there are $mn + n$ edge-disjoint 3-stars in the resulting $EPack_G$ instance.

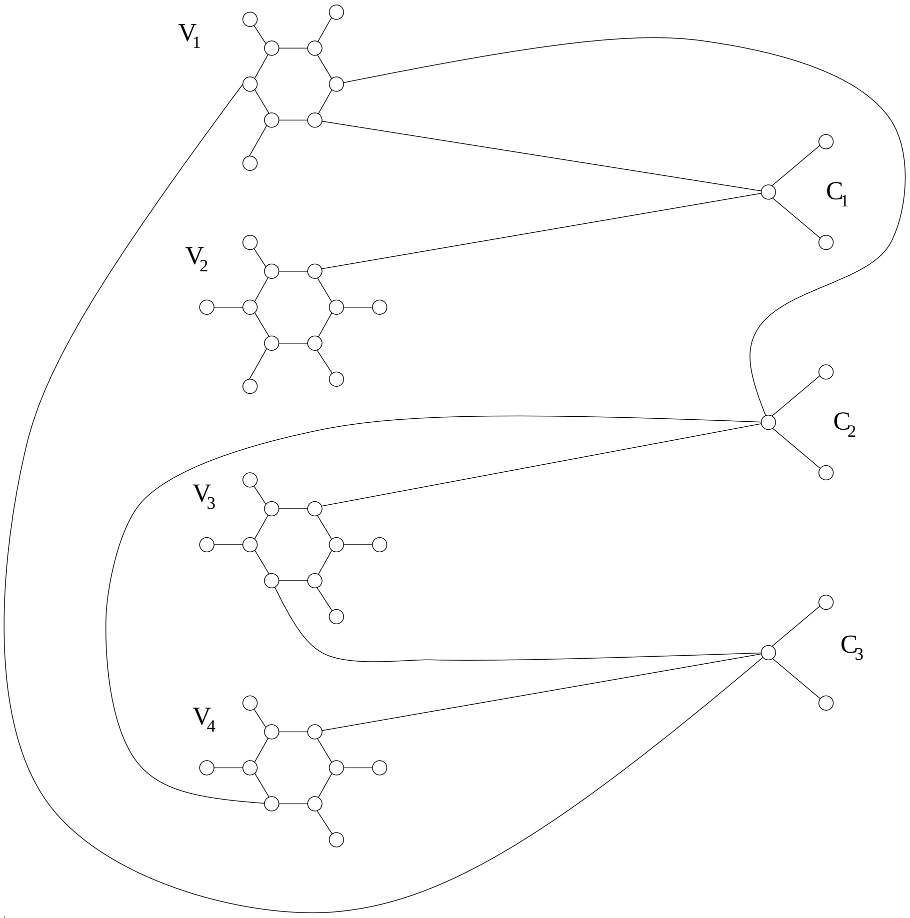

Figure 3.12: Reduction to an $EPack_{3-star}(planar)$ instance.

This reduction method immediately extends to k-stars, for $k > 3$. We simply add edges to the gadgets to form k-stars instead of 3-stars. For the variable gadget V_i, the edges that are added are incident on the vertices in the cycle ($u_i[j]$ and $\overline{u_i}[j]$). For the clause gadgets, they stem from the center of the star. Figures 3.13 and 3.14 show what these gadgets look like for 5-stars. $\square$

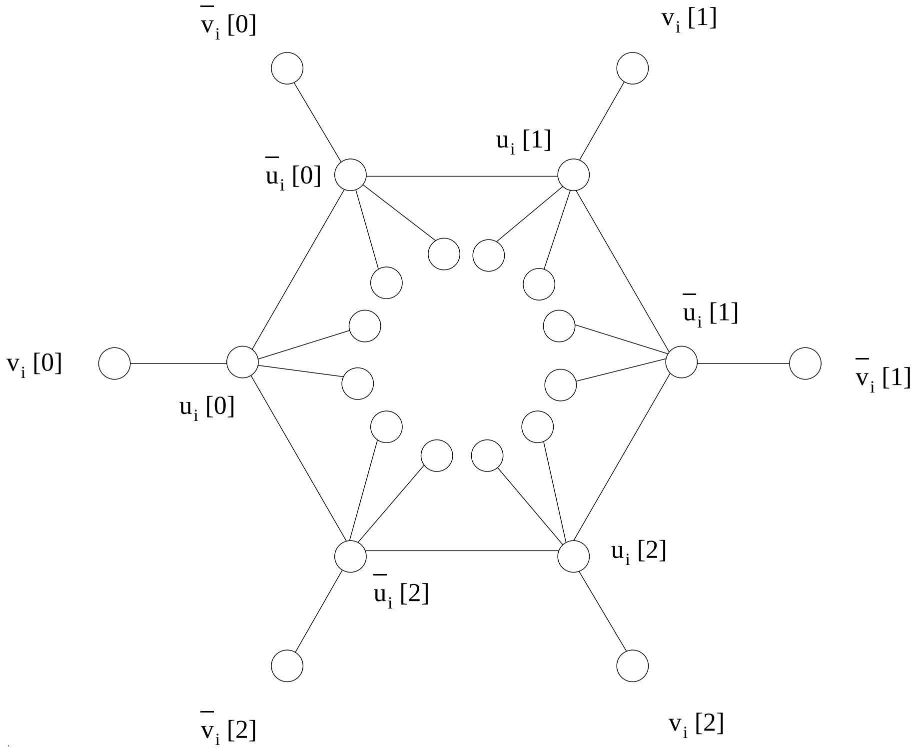

Figure 3.13: A variable gadget for 5-stars.

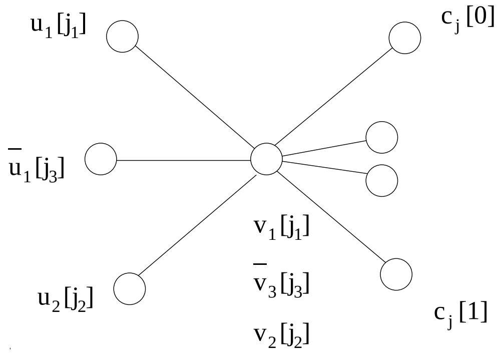

Figure 3.14: A clause gadget for 5-stars.

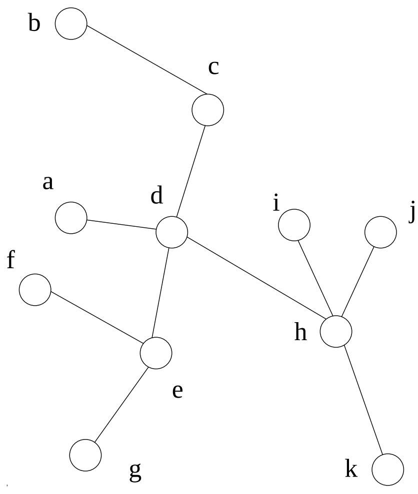

Figure 3.15: An example of a fork.

3.2 Forks as Guests

In this section, the guest graph G is a fork, a tree that is neither a star nor a path. We show that $EPack_G(planar)$ is NP-complete. We first examine the structure of a fork. A *pendant edge* is an edge where one of its endpoints has degree one. A *pendant connector* is a vertex where all but one of its incident edges are pendant edges. In Figure 3.15, for instance, $(a, d), (b, c), (f, e), (g, e), (i, h), (j, h)$, and (k, h) are the pendant edges while c, e, and h are the pendant connectors of the given fork.

It is straightforward to prove the following lemma:

Lemma 3.1 *A fork has at least three pendant edges and at least two pendant connectors.*

This lemma allows us to partition the edge set of a fork into parts. Let $G = (V_G, E_G)$ be a fork, let u be a pendant connector of G of minimum degree, and let $\delta \geq 2$ be its degree. An A-B-C-D partition of a fork is defined as follows. Arbitrarily select a pendant edge e_1 incident to u, and let $A = \{e_1\}$. Next select two pendant edges e_2 and e_3 that are not incident to u. The existence of e_2 and e_3 is implied by Lemma 3.1 and by our choice of u. Let $B = \{e_2\}$ and $C = \{e_3\}$. The remaining edges of G form the set D. By construction,

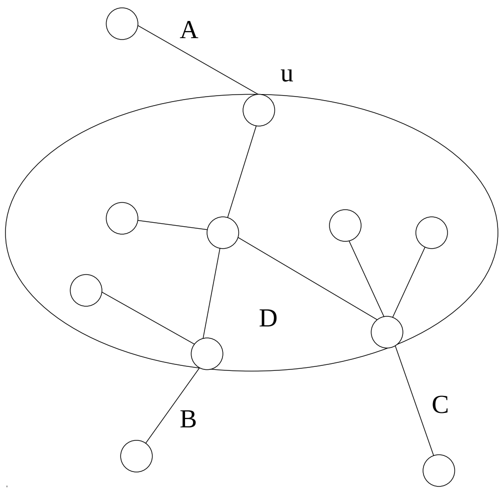

Figure 3.16: Dividing a fork into parts.

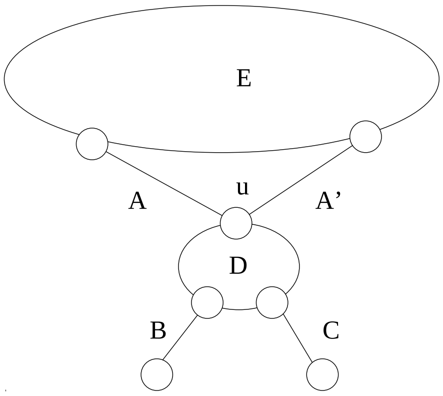

Figure 3.17: A fork in a graph.

each of the sets induces a connected subgraph of G. Figure 3.16 illustrates an A-B-C-D partition of the fork in Figure 3.15.

We are now ready to prove our theorem.

Theorem 3.2 *If $G = (V_G, E_G)$ is a fork, then $EPack_G(planar)$ is NP-complete.*

Proof: Let V and C constitute an instance of PLANAR 3-SAT. Transform the graph G_p into an instance of $EPack_G(planar)$ by substituting for each variable node in G_p a variable gadget and for each clause node in G_p a clause gadget.

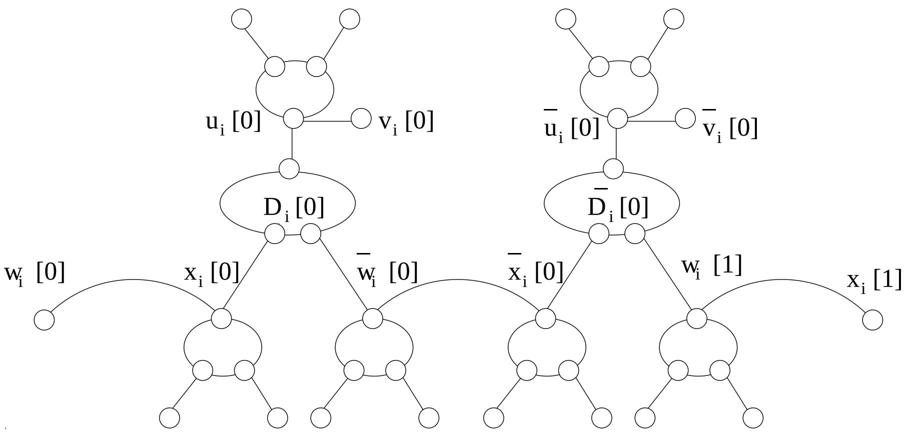

Figure 3.18: A variable subgadget for forks.

First choose an A-B-C-D partition of the edge set of G. Consider Figure 3.17, which shows a graph containing a copy of G with its parts labeled accordingly. Note that there are two choices, A and A', for the first pendant edge of G. Part E refers to the remainder of the graph connected to A and A'. Observe that the degree of u in this graph is $\delta + 1$. We claim that when packing copies of G in this graph, either part D is wholly contained in a single copy of G or is not used. Equivalently, the only choices are to combine A, B, C and D or A', B, C and D to form a single copy of G or to use A with some part of E and A' with some other part of E to form two copies of G. To obtain a contradiction, suppose that two edge-disjoint copies of G exist that use edges in D (or vertex u) with one copy containing A and the other containing A'. Because all edges incident to u within D are leaves except for one edge that connects u to the rest of D, it follows that u is a pendant connector of at least one of the two copies of G. Only one copy of G can use the edge that connects u to the rest of D. This implies that the other copy of G only uses leaves connected to u. Since u has degree $\delta + 1$ in the entire graph, the degree of u in the copy of G where it is a pendant connector, is less than δ—a contradiction. This contradiction establishes the claim that only one copy of G may use edges from D.

We now describe the gadgets. We omit the explicit naming of vertices and edges for

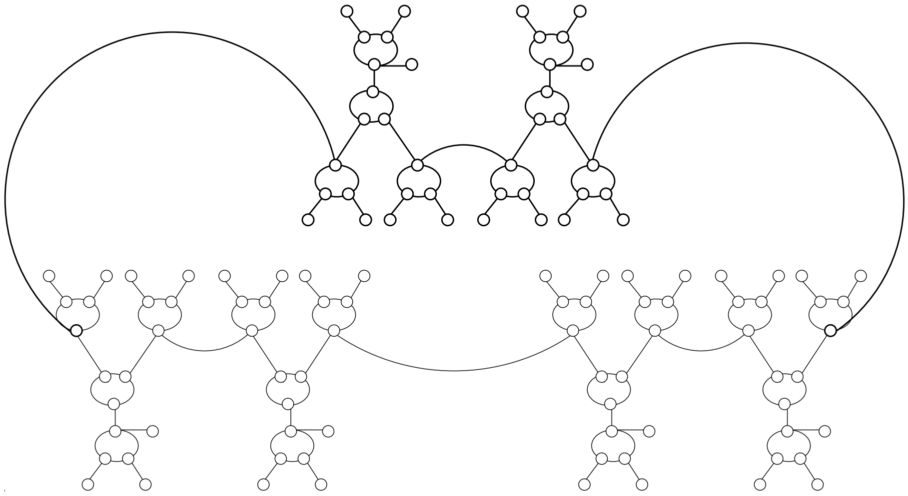

Figure 3.19: A variable gadget for forks.

simplicity. The construction is similar to that for stars. Eight copies of D are linked together, as shown in Figure 3.18, to obtain a subgadget, and n of these subgadgets are cascaded to form the variable gadget V_i as shown in Figure 3.19 ($n = 3$ in the figure). We have labeled only one subgadget for simplicity. In a subgadget, we label two of the eight copies of D, $D_i[j]$ and $\overline{D_i}[j]$. By the previous claim, at most seven edge-disjoint copies of G (using whole copies of D) may be packed in each subgadget. Although there are several ways to pack these seven copies in a subgadget, we are only interested in those which alternately use and make available D-copies and edges of the form $D_i[j], (u_i[j], v_i[j])$ and $\overline{D_i}[j], (\overline{u_i}[j], \overline{v_i}[j])$. Note that these edges may be identified with edges in clause gadgets depending on corresponding adjacencies in the PLANAR 3-SAT instance. Essential in the selection of copies of G is the decision of which $(w_i[j], x_i[j])$ edge to use in a subgadget. Using the right or left $(w_i[j], x_i[j])$ edge in a subgadget propagates around the entire gadget, hence causing a corresponding truth assignment. The consequence is an alternating set of edges made available to applicable clause gadgets. The true or false modes of a variable gadget are demonstrated in Figure 3.20 (available edges and unused D-copies are shown by dotted lines).

(a)

(b)

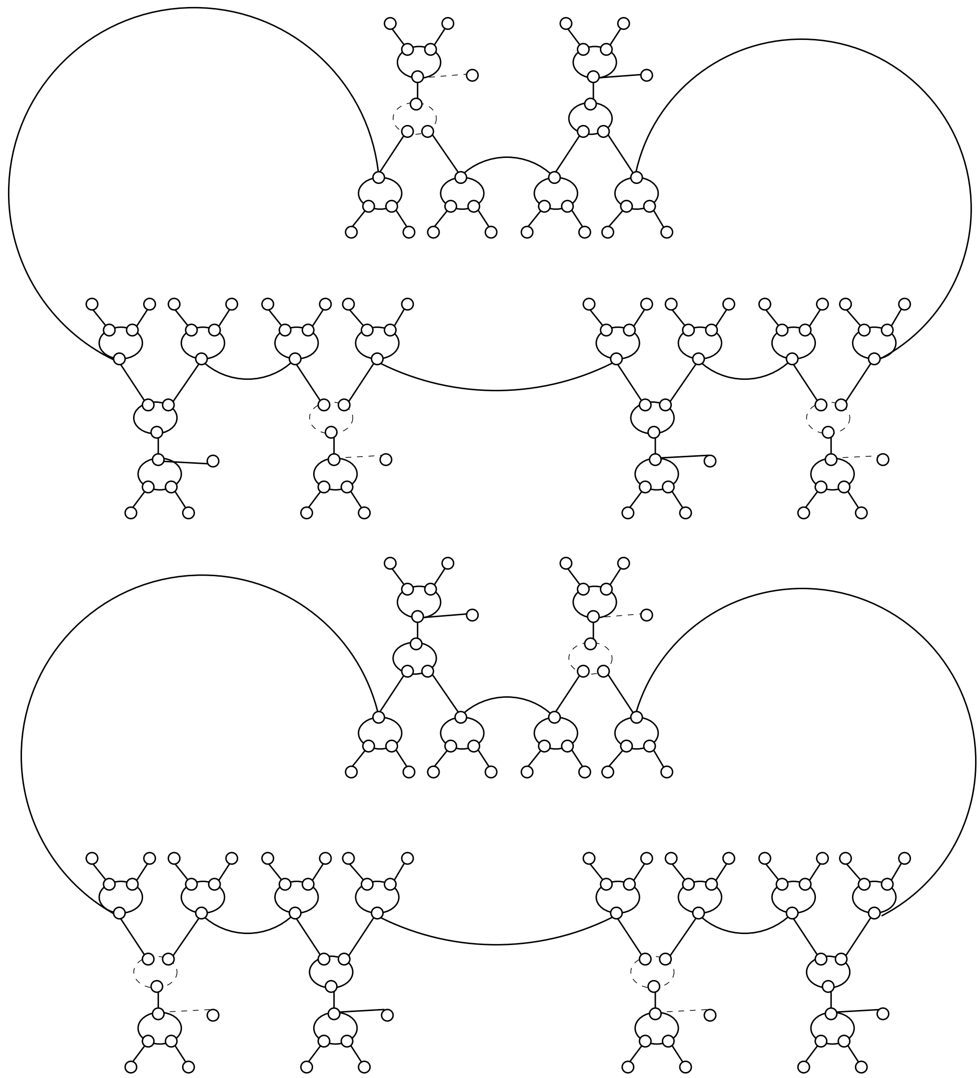

Figure 3.20: (a) True and (b) false modes for forks.

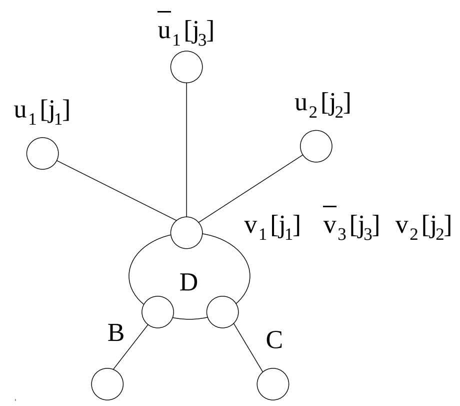

Figure 3.21: A clause gadget for forks.

A clause gadget C_j is simply a copy of the fork G with c_j alternatives for edge A, illustrated in Figure 3.21. These are precisely the edges that are identified with the variable gadgets (the vertex joining these edges is identified with the corresponding vertices in the variable gadgets). Of course, the motivation is that whenever at least one of these edges is free, a copy of G is formed.

It is important to note that the adjacencies between variables and clauses retain the same cyclic ordering in the PLANAR 3-SAT instance so that the planarity of the graph is preserved. This completes our construction.

There are $7n$ copies of G in a variable gadget and at most one copy of G can be packed in a clause gadget. This means that the value of K in our reduction is $7mn + n$; i.e., $7mn + n$ edge-disjoint copies of G exists for the graph if and only if there is a satisfying truth assignment in the PLANAR 3-SAT instance. $\square$

3.3 k-Cycles as Guests

Edge-packing by k-cycles is also NP-complete for planar host graphs. Analogous constructions for the gadgets are provided here.

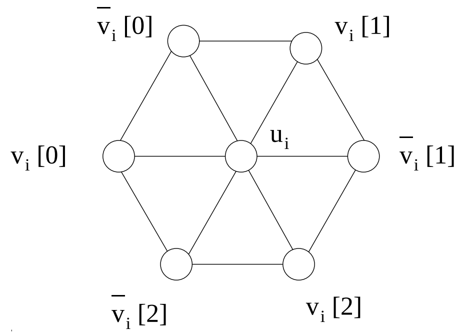

Figure 3.22: A variable gadget for 3-cycles.

Theorem 3.3 *$EPack_{k-cycle}(planar)$ is NP-complete.*

Proof: We first show this for 3-cycles and later extend our results to arbitrary k-cycles.

Let V and C constitute an instance of PLANAR 3-SAT. for each variable v_i, where $1 \leq i \leq m$, there is a variable gadget V_i that contains $2n+1$ vertices which are: $v_i[j]$ and $\overline{v_i}[j]$, where $0 \leq j \leq n-1$, and an extra vertex u_i. The edges for this variable gadget are those contained in the cycle ($v_i[0]$, $\overline{v_i}[0]$, $v_i[1]$, $\overline{v_i}[1]$,..., $v_i[n-1]$, $\overline{v_i}[n-1]$), and those contained in the star formed by the edges (u_i, $v_i[j]$) and (u_i, $\overline{v_i}[j]$), $0 \leq j \leq n-1$.

Figure 3.22 shows the gadget for $n = 3$, while Figure 3.23 shows the gadget in its true and false modes. These correspond to the two ways that n 3-cycles can be packed. In each case, alternating cycle edges are used. This is significant because these are the edges that are shared with the clause gadgets. It is easy to see that n is the maximum number of 3-cycles that can be packed in a variable gadget and that the two ways to do so are exactly those presented in the figure.

A clause gadget for 3-cycles is shown in Figure 3.24. The clause gadget depends on the variables contained in a particular clause. Each clause gadget C_j, $1 \leq j \leq n$, contains 10 vertices, $c_j[k], 0 \leq k \leq 9$, where $c_j[0]$ is a designated center vertex. If there are 3 variables in a clause c_j, edges in the clause gadget form a structure similar to the variable gadget: they form edges of a cycle and a star and but this time they comprise 9 cascaded 3-cycles.

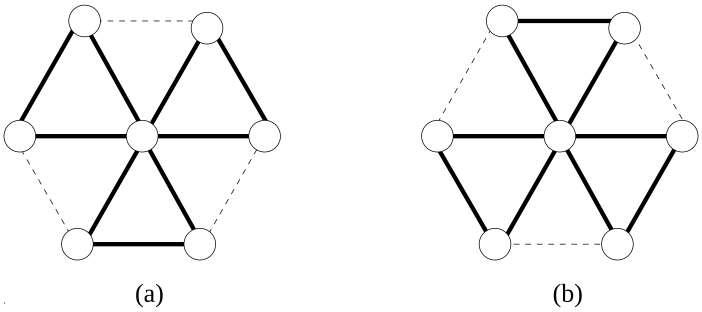

Figure 3.23: A variable gadget for 3-cycles in its (a) true and (b) false modes.

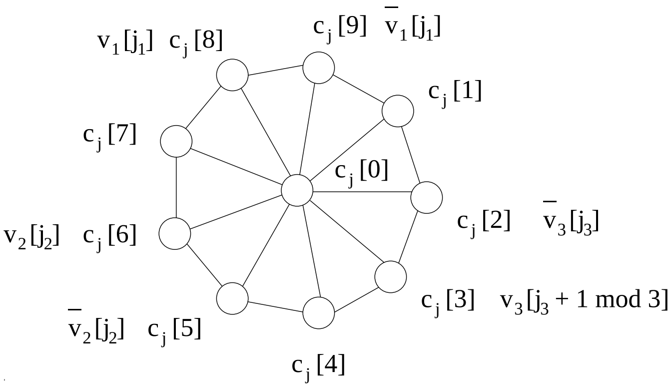

Figure 3.24: A clause gadget for 3-cycles; assume $c_j = (v_1 + v_2 + \overline{v_3})$.

We also identify some of the $c_j[k]$ vertices with vertices from the variable gadgets. Taking the specific clause example that we have been using ($c_j = (v_1 + v_2 + \overline{v_3})$), the vertices are $c_j[0]$ (the center vertex), $c_j[1]$, $\overline{v_3}[j_3]$ ($c_j[2]$), $v_3[j_3 + 1 \pmod{n}]$ ($c_j[3]$), $c_j[4]$, $v_2[j_2]$ ($c_j[5]$), $\overline{v_2}[j_2]$ ($c_j[6]$), $c_j[7]$. $v_1[j_1]$ ($c_j[1]$) ($c_j[8]$), and $\overline{v_1}[j_1]$ ($c_j[9]$), The shared vertices are connected by edges which are also cycle edges in the variable gadget. The maximum number of 3-cycles that can be extracted from this particular gadget is 4 and this is possible only if at least one of the shared edges is made available by a variable gadget.

Figure 3.25(a) and 3.25(b) illustrate clause gadgets when clauses contain just 1 or 2 variables. Note that for $|c_j| = 1$, we eliminate the edge $(c_j[5], c_j[6])$ and $(c_j[8], c_j[9])$ from the gadget, so that 4 copies may be packed only if the edge $(c_j[2], c_j[3])$ is made available by the corresponding variable gadget. We eliminate the edge $(c_j[8], c_j[9])$ when $|c_j| = 2$ for

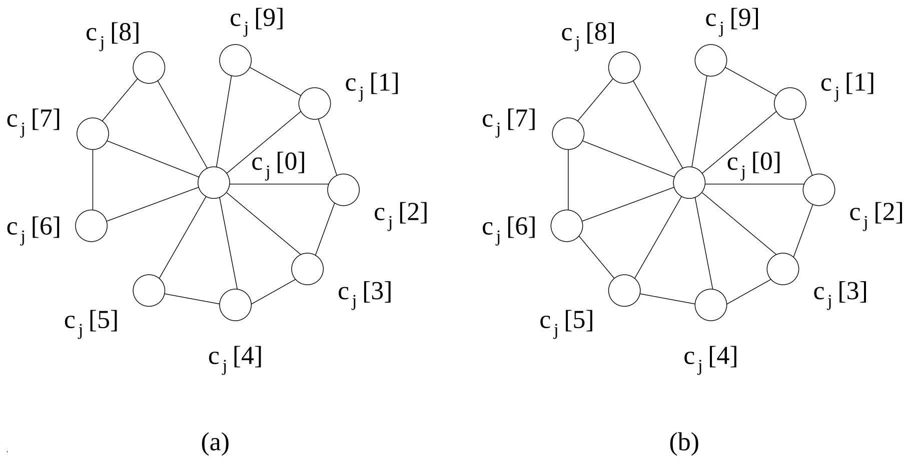

Figure 3.25: Other clause gadgets for 3-cycles: (a) $|c_j| = 1$; (b) $|c_j| = 2$.

the same reason.

The value for K in our reduction is therefore $mn + 4n$, since we pack n copies of G per variable gadget and 4 copies of G per clause gadget.

Extension to k-cycles in general requires magnifying the cycle edges of the gadgets (those which are in the outer cycle) to accommodate k-cycles instead of 3-cycles. Figure 3.26 illustrates how this is done a variable gadget for 5-cycle guests. □

3.4 Arbitrary Guest Graphs

There are still other planar guest graphs that have not been investigated: cyclic graphs in general (graphs containing cycles). NP-completeness results for these cases are currently in preparation [18]. With that, we have the following general theorem.

Theorem 3.4 *$EPack_G(planar)$, where G is a fixed planar graph with 3 or more edges, is NP-complete.*

General cyclic graphs require more careful gadget constructions since we have to guarantee that only particular ways to pack copies are possible. Cascading these graphs in the

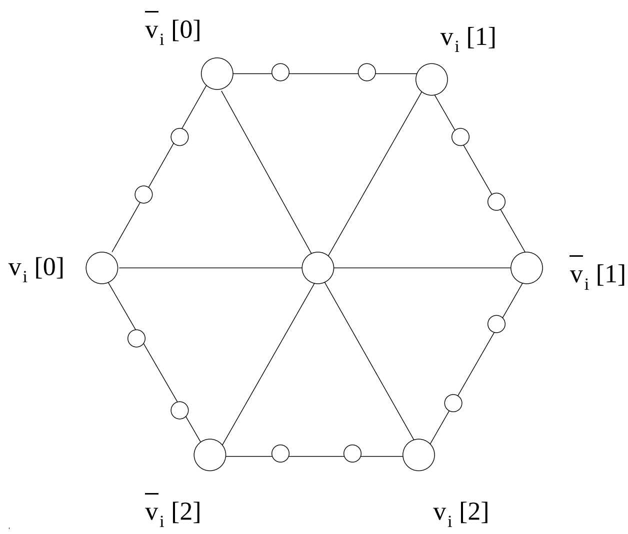

Figure 3.26: A variable gadget for 5-cycles.

same straightforward manner we have used for cycles may introduce stray copies of G.

Corneil, Masuyama and Ibaraki [11] and Dor and Tarsi [12] have investigated edge-packing by cyclic graphs but the hosts they consider are arbitrary (not necessarily planar) graphs. Their proofs involve classifying cyclic graphs in terms of their connectivity and then appropriately constructing the gadgets.

We can, however, generalize our results here to some restricted subclasses of cyclic graphs, because they directly follow from the ideas of our previous results.

Theorem 3.5 *If G contains exactly one cycle, then $EPack_G$(planar) is NP-complete.*

This follows from our proof for cycles (Theorem 3.3) since the edges in G that are not in the cycle are simply attached to the cycles in the gadgets and the proof immediately applies. Also, if the cyclic guest graph satisfies the conclusions of Lemma 3.1 with respect to pendant edges and pendant connectors, then an adaptation of the NP-completeness proof for a fork guest applies. The same definition for pendant edges (one of its endpoints has degree one) and pendant connectors (a vertex where all but one of its incident edges are pendant edges) apply. We summarize this result in the following theorem.

Theorem 3.6 *If G is any connected graph with at least 2 pendant conectors and 3 pendant edges, then $EPack_G(planar)$ is NP-complete.*

4 Outerplanar Hosts

We conjecture that $EPack_G$ is solvable in polynomial time when both the guest and the host graph are restricted to outerplanar graphs.

Conjecture 4.1 *$EPack_G(outerplanar)$, where G is a fixed outerplanar guest, is solvable in polynomial time.*

Many NP-complete graph problems that remain NP-complete if the graph involved is planar have been found to be solvable in polynomial time when the graph is outerplanar [2, 24]. In this section, we give linear time algorithms for edge-packing 3-cycles and k-stars into an outerplanar host.

4.1 $EPack_{3-cycle}(outerplanar)$

We present an algorithm to solve $EPack_G(outerplanar)$ where G is a triangle (3-cycle). We use the notion of the *weak dual* of an outerplanar graph (see Fleishchner, Geller, and Harary [14]). To obtain the weak dual of an outerplanar graph H, first construct the usual dual H^* of H as a planar graph: for each face f of the planar embedding, there is a vertex f^* of H^*, and, for each edge in H incident on the faces f_1 and f_2, there is a corresponding edge (f_1^*, f_2^*) in H^*. Let f_0 be the exterior face of the planar embedding of H. (Note that any loops and multiple edges in H^* are incident to f_0^*.) If (f_0^*, f_i^*) is an edge of H^* and $f_i \neq f_0$, then substitute edge (g, f_i^*) for (f_0^*, f_i^*), where g is a new vertex that appears incident to no other edge in T. If (f_0^*, f_0^*) is one of (possibly) multiple loops in H^*, then substitute edge (g, h) for this loop, where g and h are new vertices that appear incident to

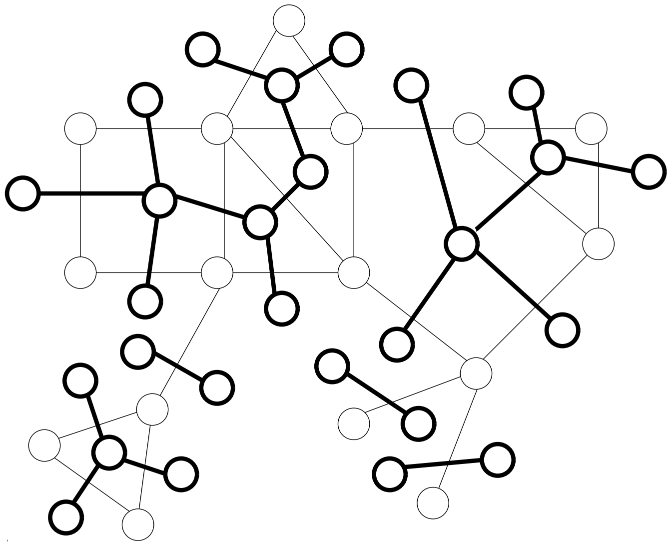

Figure 4.27: An outerplanar graph, H, and its weak dual, H^* (in bold vertices and edges).

no other edge in T (the edge (g, h) is then a connected component of T that corresponds to a cut edge in H). Figure 4.27 is an example of an outerplanar graph and its weak dual.

Fleischner, Geller, and Harary [14] observe that the weak dual of an outerplanar graph is a forest. Also, detecting a maximum set of edge-disjoint triangles in an outerplanar graph is equivalent to finding a maximum set of non-adjacent degree-3 vertices in the weak dual. This is in turn done by treating each tree of the weak dual separately. To solve $EPack_G(outerplanar)$, it suffices to give an algorithm that finds a maximum independent set consisting only of degree-3 vertices in a tree. We choose the tree to be rooted at a vertex of degree one.

Algorithm Pack-dual (see Figure 4.28) is recursive on the subtrees of a tree rooted at the current vertex. PACK-DUAL is initially called from the root vertex, which can never be in the independent set, as it has degree 1. The solution at each vertex (subtree) is the maximum independent set consisting of degree-3 vertices in the subtree rooted at that vertex and a flag that indicates whether that vertex is in the set or not. At each call to PACK-DUAL, the results from the children of the current vertex are collected (lines 7-10).

Pack-dual(T, v). Used in 3-cycle edge-packing for outerplanar graphs.

INPUT: A rooted tree $T = (V_T, E_T)$ (weak dual of an outerplanar graph) and a vertex v.
OUTPUT: A maximum independent set of degree-3 vertices for T rooted at v.

```
1   if (c(v) = ∅) then
2       ▷ if v is a leaf, return an empty solution
3       return (∅)
4   else
5       begin
6           (w_1, w_2, ..., w_n) ← c(v);
7           for i ← 1 to n do
8               ▷ process the subtrees
9               (S_i) ← Pack-dual(T, w_i)
10          A ← ∪_{i=1}^{n} S_i
11          if (n = 2) and (not (w_1 ∈ S_1 or w_2 ∈ S_2)) then
12              ▷ if v is of degree-3 and its children are not part of the solution
13              return(A ∪ {v})
14          else
15              return(A);
16      end;
```

Figure 4.28: Algorithm Pack-dual.

If the current vertex is of degree 3 (a possible candidate in the set), its two children are checked for membership in the respective solutions (lines 11). If neither is in the respective solution, then v is included in the set (line 13); otherwise, it is not (line 15). Deciding on the latter is appropriate since choosing to add v to the independent set causes one or both of its children to be removed from the set, implying no increase in size of the current independent set. Thus, the result at every call to PACK-DUAL is indeed optimal and guarantees that the resulting independent set for the graph is maximum.

As an example, consider Figure 4.29, the largest component of the weak dual in the example of Figure 4.27. The maximum independent set of degree-3 vertices is in bold:

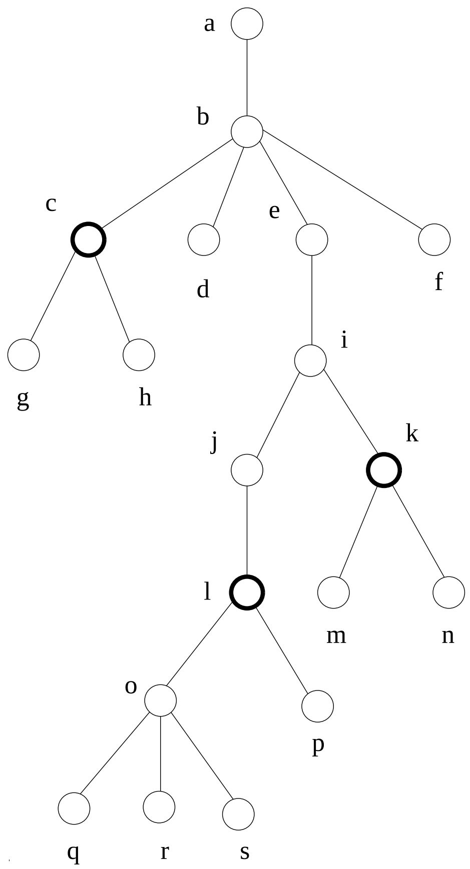

Figure 4.29: Finding an independent set of degree-3 vertices in a tree.

$\{c, k, l\}$. When PACK-DUAL is called with vertex i, the independent sets for its subtrees are computed. These are $\{l\}$ and $\{k\}$ and their union is $\{k, l\}$. Since k is a child of i but is included in the maximum set, i is not added even though it is a degree-3 vertex.

Theorem 4.1 *$EPack_{3-cycle}(outerplanar)$ is solvable in $O(|E_H|)$ time.*

Proof: Applying algorithm PACK-DUAL to every tree of the weak dual of an outerplanar graph indirectly obtains a maximum 3-cycle edge-packing for the graph as explained above. Since edges in the weak dual correspond to edges in the outerplanar graph, the algorithm takes $O(|E_H|)$ time: the edges (or vertices) in the weak dual are each visited once during the search. □

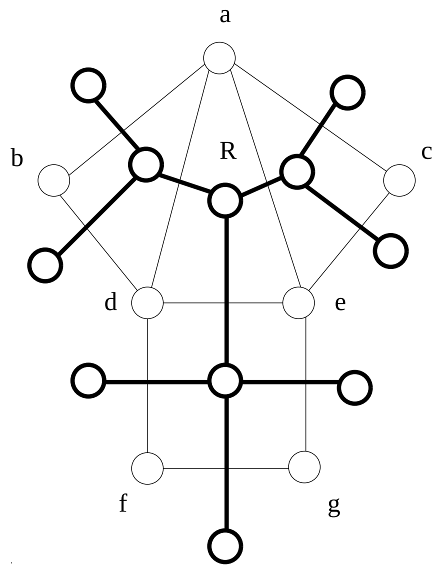

Figure 4.30: 5-cycles in an outerplanar graph.

The algorithm does not immediately extend to $EPack_G$ where G is another outerplanar graph. Note that 3-cycles correspond to single faces (vertices in the weak dual) in an outerplanar graph. A 3-cycle is the only graph where such a correspondence can be made. For 5-cycles, for example, as shown in Figure 4.30, the guest graphs may span multiple faces. Moreover, two 5-cycles may enclose the same face even though they are edge-disjoint; in Figure 4.30, there are two edge-disjoint 5-cycles (a, b, d, e, c) and (a, d, f, g, e) that enclose a face. However, since G is fixed, there are a fixed number of possibilities (correspondences with the dual) that can be enumerated in advance. This suggests a more complex version of PACK-DUAL where every possible correspondence is accomodated. The algorithm, which is beyond the scope of this paper, still runs in polynomial-time since G is fixed.

4.2 $EPack_{k-star}(outerplanar)$

In this section, we present a linear-time dynamic programming algorithm to edge-pack a guest that is a k-star, $k \geq 3$, in an arbitrary outerplanar host H (for more on dynamic programming, see Cormen, Leiserson, Rivest [10]). To simplify the dynamic programming equations, transform the problem as follows. First, add sufficient edges to the host graph H to obtain a maximal (hence biconnected) outerplanar graph H'; because it is biconnected, H' has a unique outerplanar embedding. Then, assign 0-1 weights to each edge (v_s, v_t) in H', giving it weight $w(v_s, v_t) = 1$ if (v_s, v_t) is also in H and weight $w(v_s, v_t) = 0$ otherwise. The transformed problem is to pack as many stars of *weight exactly* k into H' as possible.

The purpose for transforming the problem is revealed when we consider the weak dual H^* of H'. Since every interior face of H' is a triangle, every vertex of H^* has degree either 1 or 3. An edge in H^* is labeled (v_s, v_t), if it is the dual edge of (v_s, v_t) in H'. Label the vertices of H' $v_1, v_2, \ldots, v_n$ around the exterior face in counterclockwise order. Choose the root of H^* to be the degree-1 vertex that is incident to the edge labeled (v_n, v_1); the remaining degree-1 vertices are leaves of H^*. Let $H'(v_s, v_t)$ be the subgraph of H' consisting of all edges of H' that are strictly below the edge (v_s, v_t) in H^*. The general strategy of the algorithm is to perform dynamic programming on H^* from its leaves up to its root, utilizing a *worth function* W associated with the edges of H^* and hence with the edges of H'.

The worth function associated with the edge (v_s, v_t) in H' takes into account the maximum number of weight-k stars that can be packed into $H'(v_s, v_t)$, as well as the possibility that a packing in $H'(v_s, v_t)$ may omit edges incident to v_s and edges incident to v_t. If P is a packing of weight-k stars in $H'(v_s, v_t)$, then we write $P \in H'(v_s, v_t)$. We further write $P(c_s, c_t) = y$ if P is a packing of exactly y weight-k stars such that there are available edges of total weight c_s incident to v_s and available edges of total weight c_t incident to v_t. This leads to the following definition of the worth function

$$W[(v_s, v_t), c_s, c_t] = \max_{P \in H'(v_s, v_t)} P(c_s, c_t)$$

where (v_s, v_t) in H', $0 \le c_s \le k-1$, and $0 \le c_t \le k-1$. In words, $W[(v_s, v_t), c_s, c_t]$ is the maximum number of weight-k stars that can be packed into $H'(v_s, v_t)$ while omitting edges of total weight c_s incident to v_s and edges of total weight c_t incident to v_t. For an edge (v_i, v_{i+1}), $1 \leq i \leq n-1$, that is incident to a leaf of H^*, we have

$$W[(v_i, v_{i+1}), 0, 0] = 0$$

and

$$W[(v_i, v_{i+1}), c_i, c_{i+1}] = -\infty,$$

whenever $c_i \geq 1$ or $c_{i+1} \geq 1$. This gives the base cases for the dynamic programming algorithm.

The general case occurs at a vertex of degree 3 in H^*. Suppose (v_r, v_s, v_t) is a triangle in H' such that (v_r, v_t) is above (v_r, v_s) and (v_s, v_t) in H^*. This is the last time at which vertex v_s is considered, so we need to identify the circumstances under which a k-star centered at v_s may be part of a packing. Such a k-star can be created by available edges from $H'(v_r, v_s)$ and $H'(v_s, v_t)$ together with (v_r, v_s) or (v_s, v_t) or both. If we use, for example, (v_r, v_s) in a k-star centered at v_s, it is unavailable to any k-star centered at v_r. For the computation of each $W[(v_r, v_t), c_r, c_t]$, where $0 \leq c_r \leq k-1$ and $0 \leq c_t \leq k-1$, we get 4 major possibilities depending on whether (v_r, v_s) is or is not used at v_s and on whether (v_s, v_t) is or is not used at v_s.

To simplify the discussion of the computation of $W[(v_r, v_t), c_r, c_t]$, we consider the allocations of (v_r, v_s) and of (v_s, v_t) separately. We define an intermediate quantity $B[(v_a, v_b), x, y]$, for $0 \leq x \leq k-1$ and $0 \leq y \leq k-1$, with the meaning that $B[(v_a, v_b), x, y]$ is the maximum number of weight-k stars that can be formed below (v_a, v_b) with (v_a, v_b) allocated to use either v_a or v_b and with edges of weight x available at v_a and with edges of weight y available at v_b. This quantity is the maximum of five quantities:

- If $w(v_r, v_s) = 0$, then $B[(v_a, v_b), x, y] = W[(v_a, v_b), x, y]$.

- If $w(v_a, v_b) = 1$ and $x = 0$, then $1 + W[(v_a, v_b), k-1, y]$ is a possibility, coresponding to allocation of (v_a, v_b) to v_a to form a weight-k star centered at v_a.

- If $w(v_a, v_b) = 1$ and $x > 0$, then $W[(v_a, v_b), x-1, y]$ is a possibility, coresponding to allocation of (v_a, v_b) to v_a.

- If $w(v_a, v_b) = 1$ and $y = 0$, then $1 + W[(v_a, v_b), x, k-1]$ is a possibility, coresponding to allocation of (v_a, v_b) to v_b to form a weight-k star centered at v_b.

- If $w(v_a, v_b) = 1$ and $y > 0$, then $W[(v_a, v_b), x, y-1]$ is a possibility, coresponding to allocation of (v_a, v_b) to v_b.

We may summarize the above in the expression:

$$B[(v_a, v_t), x, y] \quad = \quad \max \begin{cases} W[(v_a, v_b), x, y] & \text{if } w(v_a, v_b) = 0, \\ 1 + W[(v_a, v_b), k-1, y] & \text{if } w(v_a, v_b) = 1 \text{ and } x = 0, \\ W[(v_a, v_b), x-1, y] & \text{if } w(v_a, v_b) = 1 \text{ and } x > 0, \\ 1 + W[(v_a, v_b), x, k-1] & \text{if } w(v_a, v_b) = 1 \text{ and } y = 0, \\ W[(v_a, v_b), x, y-1] & \text{if } w(v_a, v_b) = 1 \text{ and } y > 0 \end{cases}$$

The computation of $W[(v_r, v_t), c_r, c_t]$ is now expressed in terms of B. There are two subexpressions, corresponding to whether an additional weight-k star is formed at v_s.

$$W[(v_r, v_t), c_r, c_t] \quad = \quad \max \begin{cases} \max\limits_{0 \le y \le k-1} B[(v_r, v_s), c_r, y] + \max\limits_{0 \le x \le k-1} B[(v_s, v_t), x, c_t], \\ \max\limits_{\substack{0 \le x \le k-1 \\ 0 \le y \le k-1}} 1 + B[(v_r, v_s), c_r, x] + B[(v_r, v_s), y, c_t] \end{cases}$$

Once the worth function has been calculated for the last edge, (v_n, v_1), the number of weight-k stars in a maximum edge packing is calculated by the expression

$$\max_{\substack{0 \le i \le k-1 \\ 0 \le j \le k-1}} B[(v_n, v_1), i, j]$$

As is usual with dynamic programming, an algorithm that actually finds a maximum edge packing is a simple modification of the above algorithm that merely finds the optimal value. The linear time complexity follows from the fact that H' and H^* are linear in the size of H together with the fact that the dynamic programming algorithm requires a constant amount of effort per edge of H^* (though the constant does depend on k). We have shown the following.

Theorem 4.2 *$EPack_{k-star}(outerplanar)$, $k \geq 3$, is solvable in $O(|V_H|)$ time.*

Dynamic programming on the weak dual of an outerplanar host graph may be the basis for efficient edge-packing algorithms for other outerplanar guests as well, though we have not characterized these guests.

5 The Approximability of $EPack_G$

For an edge-packing problem that is NP-complete, an approximation algorithm that runs in polynomial time is desirable. In this section, we first consider a greedy approximation algorithm for general $EPack_G$ problems and show that it always produces at least a fixed fraction of the optimal solution. Then we propose a more sophisticated approximation algorithm that applies to general planar hosts and obtains better approximations for most (outerplanar) guests.

5.1 A Greedy Algorithm

We present Algorithm Pack-G (see Figure 5.31), a straightforward greedy polynomial-time algorithm for finding edge-disjoint copies of a guest graph G in a host graph H. The algorithm identifies all copies of G (lines 1-3), iteratively selects a copy that has the least number of neighboring copies (lines 7-8), and excludes these neighbors (lines 9-10).

PACK-$G(H)$. An approximation algorithm for $EPack_G$.

INPUT: A host graph $H = (V_H, E_H)$.
OUTPUT: An approximate G edge-packing of H.

```
▷ identify and obtain all possible copies of G in H
{G_1, G_2, ..., G_n} ← ALL-COPIES-OF-G(H);
S ← {G_1, G_2, ..., G_n};
A ← ∅;
while (S ≠ ∅) do
    begin
        G_k ← the copy of G in S such that (|NEIGHBORS(G_k)|) is minimum;
        A ← A + G_k;
        foreach G_i where G_i is in NEIGHBORS(G_k) do
            S ← S − G_i;
        S ← S − G_k
    end;
return(A);
```

Figure 5.31: Algorithm PACK-G.

Neighboring copies of a G-copy G_k ($NEIGHBORS(G_k)$) are other G-copies in the host that share an edge with G_k (this set has to be re-calculated at each iteration).

It is worthwhile to note that the algorithm exhibits the connection between $EPack_G$ and the Independent Set problem (finding the largest subset of vertices in a graph such that no two vertices are adjacent to each other). Vertices in the Independent Set instance correspond to copies of G in the $EPack_G$ instance while edges correspond to the "neighbor" relation. In fact, Johnson [23] proposes an analogous algorithm for independent set and showed that there is no finite ratio R such that the result produced by the algorithm is $\geq R * OPT$ (where OPT is the actual optimal solution), assuming no restrictions on the graph are made.

The above seems to imply that the algorithm in Figure 5.31 is just as bad. However, the nature of the $EPack_G$ problem imposes a certain restriction (G is fixed) so that at most $|E_G|$ neighbors in any independent set can exist for a particular copy of G. It is this restriction that allows us to state the following:

Theorem 5.1 *Algorithm PACK-G runs in polynomial time and produces an edge-packing within* $1/|E_G|$ *of the optimal solution.*

Proof: PACK-G runs in polynomial time because G is fixed. There are $O(n_H^p)$ copies of G in an arbitrary host graph, where $p = |V_G|$. Indentifying all copies of G requires $O(n_H^p)$ time since it can be implemented by testing isomorphism (to G) for each possible set of p vertices in H. $NEIGHBORS(G_k)$ is obtained by simply maintaining a list of copies of G for every edge, where copies that include an edge are in the list for that edge. Initially, this can be done simultaneously with the exhaustive search described above and can be updated in $O(1)$ time at each iteration. Lines 3-12 therefore take $O(|E_H|)$ time, so the entire algorithm runs in $O(n_H^p)$ time.

It remains to show that the algorithm produces at least $1/|E_G|$ of the optimal solution. Consider an optimal solution for an instance of $EPack_G$. For every choice made within each iteration of the loop in lines 4–11, at most $|E_G|$ copies of G from the optimal solution are discarded (this is because the choice must share at least one edge with each discarded copy). Upon completion of the algorithm, in the worst case, the greedy edge-packing is within $1/|E_G|$ of the optimal. □

5.2 Approximation Via Leveling

We illustrate a general technique for constructing edge-packing algorithms for planar hosts with an approximation algorithm for $EPack_{k-star}(planar)$ that produces an edge-packing of size at least 1/2 the optimal. The technique may be seen as a special case of the work of Baker [2]. The technique depends on processing a planar graph in levels, an approach employed by Heath [16], as well as others.

The first phase of the algorithm adds sufficient edges to a planar host $H = (V_H, E_H)$ to obtain a maximal planar graph $H' = (V_H, E_{H'})$. Assign 0-1 edge weights to E_H, so that weight $w(u, v) = 1$ exactly when $(u, v) \in E_H$. Fix a planar embedding of H'; every face of H' is a triangle. Partition the vertex set V_H of H' into levels $V_0, V_1, \ldots$ as follows. V_0 consists of the three vertices on the exterior face. For notational convenience, $V_{-1} = \emptyset$. Inductively, V_i, for $i > 0$, is given by

$$V_i = \{v \in V \mid \exists (u, v) \in E_{H'} \text{ and } u \in V_{i-1}\} - V_{i-2}.$$

In other words, V_i consists exactly of those vertices whose shortest path to the exterior face has length i.

Now define level graphs L_i derived from H' as follows. Begin with the minimal subgraph of H' that contains every edge incident to some vertex in V_i. If (u, v) is such an edge and $u, v \in V_i$, then put (u, v) in L_i without modification. If (u, v) is such an edge and $u \in V_i$ while $v \in V_{i-1} \cup V_{i+1}$, then put $(u, v[u])$ in L_i, where $v[u]$ is a new vertex that is incident to no other edges of L_i. This last modification breaks any cycles involving vertices in V_{i-1} or in V_{i+1}. Edge weights carry over to L_i. It is easy to show the following

Lemma 5.1 *Each level graph L_i is outerplanar. Furthermore, any k-star in H is a weight-k star in some L_i, and vice versa.*

This completes the first phase of the algorithm.

The second and final phase of the algorithm finds a maximum edge-packing in each L_i by weight-k stars, using the algorithm of Section 4.2. It is clear that the weighted problem is the one to consider here, since H' in general contains edges that are not in H. Suppose the maximum edge-packing for L_i contains Y_i weight-k stars. Then H contains at least

$$Y = \max\left\{\sum_{j=0}^{\infty} Y_{2j}, \sum_{j=0}^{\infty} Y_{2j+1}\right\}$$

edge-disjoint k-stars, because two k-stars whose centers are in levels ≥ 2 apart cannot share an edge. Let X be the size of a maximum k-star edge-packing in H. We claim that $Y \geq X/2$. The claim follows from the observation that

$$X \leq \sum_{j=0}^{\infty} Y_j \leq 2Y.$$

We have proved

Theorem 5.2 *$EPack_{k-star}(planar)$, $k \geq 3$, can be approximated within at least $1/2$ the optimal value in $O(|E_H|)$ time.*

The linear time complexity is straightforward.

The technique illustrated by this approximation algorithm can be adapted to other planar guest graphs. If G is a planar graph of diameter δ, then each L_i is induced by $\delta + 1$, rather than 1, consecutive levels of H'. One expects an approximation within a factor of $1/\delta$, rather than $1/2$. (For $\delta > 2$, the full approach of Baker [2] promises a better approximation at the cost of a high time complexity for large δ.) Efficiency depends on the existence of a fast, optimal algorithm for edge-packing G in a planar host of $\delta + 1$ levels. The technique will be most successful for planar guests of small diameter.

6 Conclusions

$EPack_G(tree)$ is solvable in polynomial time for any G, and in linear time for G a star or a path. $EPack_G(planar)$ is NP-complete for G a cycle or a tree with ≥ 3 edges. The status of the computational complexity of $EPack_G$ is summarized in the following table (P: polynomial-time solvable, NPC: NP-complete, P?: conjectured polynomial-time, (): results are incomplete).

Guest	Arbitrary Host	Planar Host	Outerplanar Host	Tree Host
2-path	P [30, 31]	P [30, 31]	P [30, 31]	P [30, 31]
Star	NPC [13]	NPC	P	P
Path	NPC [13]	NPC [13]	P?	P
Tree	NPC [11]	NPC	P?	P
Cycle	NPC [22]	NPC	(P,k=3)	P
Cyclic	NPC [11, 12]	NPC [18]	P?	P

$EPack_G(planar)$ where G contains a cycle is addressed in a separate paper [18]. There also remains the problem of determining classes of host graphs between planar graphs and trees where $EPack_G$ is solvable in polynomial time (given P $\neq$ NP). Outerplanar graphs are the most likely candidates. While this has been shown for the case where the guest graph is a triangle or a k-star, the proofs do not immediately extend to other guest graphs.

The case of disconnected guest graphs is also of interest. The NP-completeness results obtained in this paper extend to disconnected guest graphs (we first select a non-trivial connected component of G and use that in the reduction). Complications arise, however, with our algorithmic results. When the guest is disconnected, we cannot deal with the connected components of the guest graph G independently because these have to be disconnected in a selection of a copy of G.

References

[1] N. Alon, *A note on decomposition of graphs into isomorphic matchings*, Acta Mathematica Hungaria, 42 (1983), pp. 221–223.

[2] B. S. Baker, *Approximation algorithms for NP-complete problems on planar graphs*, Journal of the Association for Computing Machinery, 41 (1994), pp. 153–180.

[3] F. Berman, D. Johnson, T. Leighton, P. W. Shor, and L. Snyder, *Generalized planar matching*, Journal of Algorithms, 11 (1990), pp. 153–184.

[4] Y. Caro, *The decomposition of trees into subtrees*, Journal of Graph Theory, 8 (1984), pp. 471–479.

[5] V. Chvátal and P. L. Hammer, *Aggregation of inequalities in integer programming*, Discrete Mathematics, 1 (1977), pp. 145–162.

[6] E. Cohen and M. Tarsi, *NP-completeness of graph decomposition problems*, Journal of Complexity, 7 (1991), pp. 200–212.

[7] C. J. Colbourn, *The complexity of completing partial Latin squares*, Discrete Applied Mathematics, 8 (1984), pp. 25–30.

[8] ——, *The Combinatorics of Network Reliability*, Oxford University Press, 1987.

[9] ——, *Edge-packings of graphs and network reliability*, Discrete Mathematics, 72 (1988), pp. 49–61.

[10] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, McGraw-Hill Book Company, 1990.

[11] D. G. Corneil, S. Masuyama, and S. L. Hakimi, *Edge-disjoint packings of graphs*, Discrete Applied Mathematics, 50 (1994), pp. 135–148.

[12] D. Dor and M. Tarsi, *Graph decomposition is NPC — a complete proof of Holyer's conjecture*, in Proceedings of the 24th Annual ACM Symposium on Theory of Computing, 1992, pp. 252–263.

[13] M. E. Dyer and A. M. Frieze, *On the complexity of partitioning graphs into connected subgraphs*, Discrete Applied Mathematics, 10 (1985), pp. 139–153.

[14] H. J. Fleischner, D. P. Geller, and F. Harary, *Outerplanar graphs and weak duals*, Journal of the Indian Mathematical Society, 38 (1974), pp. 215–219.

[15] M. Hall, *Combinatorial Theory*, John Wiley and Sons, 1986.

[16] L. S. Heath, *Embedding planar graphs in seven pages*, in Proceedings of the 25th Annual IEEE Symposium on Foundations of Computer Science, 1984, pp. 74–83.

[17] ——, *Edge coloring planar graphs with two outerplanar subgraphs*, in Proceedings of the Second Annual ACM-SIAM Symposium on Discrete Algorithms, 1991, pp. 195–202.

[18] L. S. Heath and J. P. C. Vergara, *Edge packing cyclic graphs in planar graphs*. In preparation, 1995.

[19] P. Hell and D. G. Kirkpatrick, *On generalized matching problems*, Information Processing Letters, 12 (1981), pp. 33–35.

[20] ——, *Packings by cliques and by finite families of graphs*, Discrete Mathematics, 49 (1984), pp. 45–59.

[21] ——, *Packings by complete bipartite graphs*, SIAM Journal on Algebraic and Discrete Methods, 7 (1986), pp. 199–209.

[22] I. Holyer, *The NP-completeness of some edge-partition problems*, SIAM Journal on Computing, 10 (1981), pp. 713–717.

[23] D. S. Johnson, *Approximation algorithms for combinatorial problems*, Journal of Computer and System Science, 9 (1974), pp. 256–278.

[24] ——, *The NP-completeness column: an ongoing guide*, Journal of Algorithms, 6 (1985), pp. 434–451.

[25] M. Jünger, G. Reinelt, and W. R. Pulleyblank, *On partitioning the edges of graphs into connected subgraphs*, Journal of Graph Theory, 9 (1985), pp. 539–549.

[26] D. G. Kirkpatrick and P. Hell, *On the completeness of a generalized matching problem*, in Proceedings of the 10th Annual ACM Symposium on Theory of Computing, 1978, pp. 240–245.

[27] ——, *On the complexity of general graph factor problems*, SIAM Journal on Computing, 12 (1983), pp. 601–609.

[28] P. Klein and C. Stein, *A parallel algorithm for eliminating cycles in undirected graphs*, Information Processing Letters, 34 (1990), pp. 307–312.

[29] D. Lichtenstein, *Planar formulae and their uses*, SIAM Journal on Computing, 11 (1982), pp. 329–343.

[30] S. Masuyama and T. Ibaraki, *Chain packing in graphs*, Algorithmica, 6 (1991), pp. 826–839.

[31] J. P. C. Vergara and L. S. Heath, *Edge-packing by isomorphic subgraphs*, Tech. Rep. TR 91-3, Department of Computer Science, Virginia Polytechnic Institute and State University, 1991.

[32] H. P. Yap, *Packing of graphs—a survey*, Discrete Mathematics, 72 (1988), pp. 395–404.