Edge-Packing by Isomorphic Subgraphs

By John Paul C. Vergara and Lenwood S. Heath

TR 91-3

EDGE-PACKING BY ISOMORPHIC SUBGRAPHS

John Paul C. Vergara and Lenwood S. Heath

Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061

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. The problem is primarily considered for several guest graphs (trees and cycles) and host graphs (arbitrary graphs, planar graphs and trees). We give polynomial-time algorithms when G is a 2-path or when H is a tree; we show the problem is NP-complete otherwise. Also, we propose straightforward greedy polynomial-time approximation algorithms which are at least $1/|E_G|$ 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 no edges left over (i.e., the union of the subgraphs in the packing is exactly H). It is often useful to restrict the subgraphs of H to a certain class or property. For instance, edge-partitioning a graph into subgraphs which are all matchings is equivalent to the chromatic index problem, a well-known problem which involves minimally coloring the edges of a graph such that no two adjacent edges have the same color. There are other problems that may be viewed as edge-packings or edge-partitions of graphs.

Colbourn [C1, C2] used edge-packing in the determination of network reliability. Edge-packing the representative graph of a network by spanning trees and then computing separately the reliabilities of these trees provide a lower bound for the reliability of the whole network. Edge-packing the graph by paths whose ends are on two specific terminals of the network provides a bound for the reliability between the two terminals in the same manner.

Klein and Stein [KS] studied edge-packing graphs by cycles. Finding a maximal set of edge-disjoint cycles in a graph aids in solving the minimum-cost circulation problem [GT] which involves repeatedly finding a maximal set of weighted cycles in a weighted directed graph.

Other studies in edge-packing include packing by triangle free graphs (Monochromatic Triangle) or by treshold graphs (Treshold Number) [GJ], packing complete graphs by trees [Y] and packing planar graphs by outerplanar graphs [H2].

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 $H = (V_H, E_H)$. This is the same as determining the largest edge-packing of a graph by subgraphs isomorphic to G. Figure 1, for example, exhibits a maximum G edge-packing of a graph where G is a triangle. Note that selected triangles may share vertices, but never edges. This paper considers $EPack_G$ for several instances of G. The main goal is to provide a sufficient understanding of this problem so that results could be extended to arbitrary guest graphs. In addition, the host graph H is restricted to particular classes of graphs and the effect on the computational complexity of $EPack_G$ is observed. We begin by formally defining $EPack_G$ as a decision problem:

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 $(E_1, E_2, ..., E_K)$ of E_H such that $E_i \cap E_j = \emptyset, \forall i \neq j$ and the graph induced by each E_i is isomorphic to G?

Since $EPack_G$ is clearly trivial when the guest graph G is a single edge, we only consider cases where G is a graph with at least two edges. Of particular interest are the cases where G is a star (graphs isomorphic to $K_{1,k}$), a path or a cycle since these comprise graphs of the simplest types and solving the problem for these simple cases provides possibilities for extension to guest graphs of more general types. It is also useful to consider restricting the

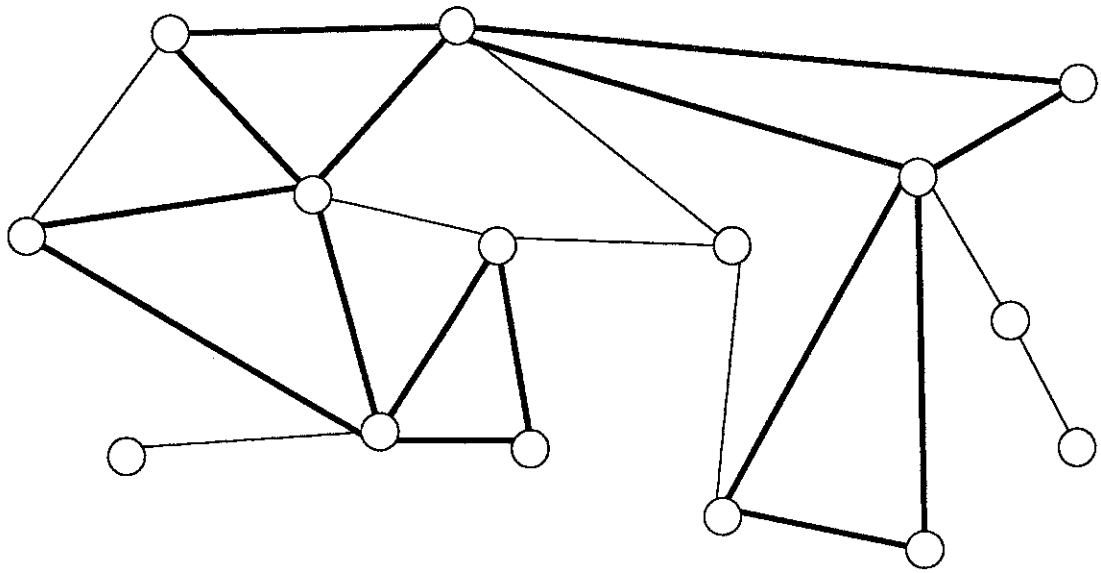

Figure 1: A maximum triangle edge-packing (edges in the packing are in bold).

host graph H to certain classes. Planar graphs are of special importance since they often correspond to suitable applications. The class of planar graphs as well as the subclasses of trees and outerplanar graphs are considered in this paper. Restricting the host graph G to these classes allows us to more precisely determine the computational complexity of $EPack_G$.

It is sufficient to deal only with connected host graphs since the guest graphs which we investigate are themselves connected. The problem (for connected guests and disconnected hosts) clearly reduces to edge-packing each connected component of the host graph. Disconnected guest graphs are a different issue altogether since this means finding copies of different graphs (components of G) in H. We therefore avoid such guest graphs in our discussion since they do not immediately follow from the results obtained for connected guest graphs.

In this paper, we obtain NP-completeness results. NP-complete problems are the class of problems which are conjectured to have no polynomial-time solution but with the characteristic that if one problem from that class is polynomial-time solvable, then the entire class is polynomial-time solvable. Garey and Johnson [GJ] provide an extensive discussion of NP-completeness as well as a list of known NP-complete problems. This paper shows that $EPack_G$ is NP-complete for particular combinations of guest and host graphs. We also investigate the combinations for which the problem is polynomial-time solvable.

Corneil et al [C4] have independently investigated $EPack_G$ for arbitrary host graphs. They define a family of problems called *edge-free packings*, essentially equivalent to the formulation of $EPack_G$ described in this paper. They show the problem is NP-complete for arbitrary hosts where the connected guest graph has at least 3 edges. Their proofs involve reductions from Exact 3-Cover, a known NP-complete problem. They also show

that the problem is polynomial-time solvable when the guest graph is a path of length 2. As $EPack_G$ is NP-complete in general, we mainly concern ourselves with restricted forms of the problem by imposing a particular class of graphs for the host; as mentioned earlier, we specifically investigate planar graphs and its subclasses.

When $K = |E_H|/|E_G|$ in the $EPart_G$ instance, we have the special case where no edges are left over in the packing. We call this problem G *Edge-Partition* ($EPart_G$) since it effectively involves partitioning 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?

$EPart_G$ has been studied by Holyer [H3, J2] and by Dyer and Frieze [DF]. For guest graphs which are either complete graphs or cycles, Holyer proves that $EPart_G$ is NP-complete by reducing from 3-Satisfiability (3-SAT), a known NP-complete problem. Dyer and Frieze, on the other hand, prove $EPart_G$ NP-complete for cases where the guest graph is a path or a star (having three or more edges). Their reductions are from 3-Dimensional Matching (3DM), known NP-complete problem. They show, in addition, that if the host graph is restricted to planar graphs, the problem remains NP-complete for paths and for 3-stars (stars with 3 edges), but becomes polynomial-time solvable for triangles (3-cycles). No conclusions in this case is made for stars or cycles with more than three edges.

Colbourn [C3] studied $EPart_G$ in terms of the construction of latin squares which is in turn useful in experimental design theory. A latin square is an nxn table of entries from the set $\{1, 2, ..., n\}$ such that no number from this set appears more than once in the same row or column of the table. Constructing a latin square corresponds to triangle edge-partitioning the complete tripartite graph formed by viewing the rows, columns and entries as the three sets (or partitions) of vertices. A partial latin square is the situation where not all slots in the table have been occupied. The problem of determining whether a partial latin square can be completed, i.e., the unoccupied slots be assigned appropriate entries, is equivalent to triangle edge-partitioning a corresponding tripartite graph. This problem (completing partial latin squares as well as the corresponding $EPart_G$ problem) has been proven NP-complete in the study.

The results obtained for $EPart_G$, specifically the NP-completeness results, are relevant to $EPack_G$. Since $EPart_G$ is a subproblem of $EPack_G$, it follows that $EPack_G$ is NP-complete whenever $EPart_G$ is NP-complete.

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 quite extensively. In fact, it has been found to be solvable in polynomial time when the guest graph is an edge by regular matching [PL, BM] but NP-complete for other connected guest graphs even if H is planar. Kirkpatrick and Hell [KH] prove NP-completeness for this problem in general (reduction from 3DM) while Berman et al [BJLSS] prove NP-completeness for planar hosts (from Planar 3-SAT). The reduction technique used by Berman et al is to cascade copies of the fixed guest graph such that only alternating

copies may be chosen in a maximum set, thereby corresponding to true-or-false settings for the 3-SAT instance. We use a similar technique in this paper and we elaborate on this later. Much work has been done on G Matching particularly because of its numerous applications in areas such as scheduling, computer network design, and wafer-scale integration [BM, KH, MS, BJLSS]. Applications for G Edge-Packing can be conceivably derived from G Matching since relaxing the vertex-disjoint constraint to edge-disjoint simply allows the existence of repeated or redundant nodes while utilizing as many links (between the nodes) as possible. Of course, the application has to be directly concerned with maximally utilizing such links.

The main results obtained in this paper can be summarized as follows:

$EPack_G$ has a polynomial-time algorithm if:
- G is a path of length 2.
- H is a tree.
- G is a triangle (3-cycle) and H is outerplanar.

$EPack_G$ is NP-complete if:
- G is a tree of 3 or more edges even if H is planar.
- G is a unicyclic graph of 3 or more edges even if H is planar.

In addition, the paper analyzes the approximability of $EPack_G$. Polynomial-time approximation algorithms to solve $EPack_G$ which are at least $1/|E_G|$ optimal are presented. We also investigate the membership of $EPack_G$ to classes of equally approximable problems.

This paper is organized as follows: In chapter 2, definitions necessary for the succeeding discussions are provided. Chapter 3 discusses 2-path edge-packing and presents a polynomial-time algorithm for arbitrary host graphs. Chapters 4 through 7 considers $EPack_G$ in general ($|E_G| \geq 3$) for hosts graphs which are trees, planar graphs, arbitrary graphs and outerplanar graphs, in that order. In chapter 8, we investigate the approximability of $EPack_G$. The paper concludes in chapter 9 where we present related open problems as well as a table summarizing the current status of $EPack_G$.

2 DEFINITIONS

A graph G is represented by (V_G, E_G), an ordered pair of vertices and edges, where V_G and E_G are the vertex set and edge set of G, respectively. We write $n_G = |V_G|$ and $m_G = |E_G|$.

Each problem considered in this paper involves a guest graph and a host graph. These are denoted by G and H, respectively. The guest graphs considered are k-stars, k-paths, k-cycles, forks, and cyclic graphs. The classes of host graphs considered are arbitrary graphs, planar graphs, outerplanar graphs, and trees.

A *k-star* is a graph isomorphic to $K_{1,k}$, a complete bipartite graph on one and k vertices. Let $\{v\}$ and $\{w_1, w_2, ..., w_k\}$ be these vertices; the k-star formed by these partitions is

denoted by $(v : w_1, w_2, ..., w_k)$. A *k-path* is a path of length k and is denoted by the sequence of vertices which form the path. A *k-cycle* is a cycle of length k and is likewise denoted by the sequence of vertices in the cycle. A *fork* is any tree (a connected graph without cycles) which is neither a star nor a path. A *cyclic graph* is any connected graph which contains at least one cycle; cyclic graphs, when refered to as such, are generally those which are not cycles themselves. Forks and cyclic graphs are simply denoted by their edge sets. We may also denote any of the other guest graphs by its edge set (e.g., the path (a, b, c, d) may be denoted by $\{(a, b), (b, c), (c, d)\}$).

A *planar graph* is a graph which can be embedded in a plane in such a way that no two edges intersect except at a common vertex. An *outerplanar graphs* is a planar graph whose vertices can be placed on a circle in the plane while its edges are embedded inside the circle; equivalently, it is a planar graph all of whose vertices lie on one face.

A *tree* is just a connected graph without cycles. A tree may be *rooted*, i.e., a designated vertex is considered the *root* of the tree. The *height* of a rooted tree is the length of the longest path from the root to a vertex in the tree. A rooted tree also imposes a *level* for each vertex of the tree with respect to its distance (length of its connecting path) from the root; all vertices which are farthest from the root have level 0 while the root itself has a level equal to the height of the tree. The parent of a vertex v, $p(v)$, is the vertex in the previous level of the tree which is adjacent to v. The children of v, $c(v)$ are the vertices in the next level of the tree which are adjacent to v. Clearly, the root is the only vertex with no parent. Vertices with no children are called *leaves.*

We may denote particular instances of $EPack_G$ by replacing G with the actual guest graph considered; e.g., $EPack_G$ where G is a 3-cycle is alternatively denoted by $EPack_{3-cycle}$. Also, whenever the host graph H is restricted to some class or property, P, we denote this by $EPack_G(P)$; e.g., $EPack_{3-cycle}(planar)$ is $EPack_{3-cycle}$ for planar host graphs. Analogous notations for $EPart_G$ and $ECover_G$ are likewise used.

3 2-PATH EDGE-PACKING

Maximum G Edge-Packing is solvable in linear time when the guest graph is a 2-path. We first provide an algorithm for the case where the host graph is a tree and show how it extends to arbitrary host graphs.

Theorem 3.1 *$EPack_{2-path}(tree)$ is solvable in $O(m_H)$ time.*

Proof: We describe an algorithm which obtains a maximum set of edge-disjoint 2-paths from a tree H. It uses recursion on the subtrees of the tree being processed and is summarized in algorithm PACK-2-PATH. The algorithm is initially called with the input tree H and its root vertex. Every call to PACK-2-PATH produces a maximum set of 2-paths and a possible left-over edge for the subtree rooted at the given vertex. Both the packing and the left-over edge are carried over to the next higher-level call to PACK-2-PATH. On every call, the algorithm processes the subtrees rooted at the children and the

Algorithm PACK-2-PATH. 2-path edge-packing for trees.

INPUT: A rooted tree $H = (V_H, E_H)$ and a vertex v.
OUTPUT: A 2-path edge-packing of the subtree rooted at v and a left-over edge, if any.

```
1   IF (c(v) = ∅) THEN      /* if v is a leaf, return an empty solution and no left-over edge */
2       RETURN (∅, ∅)
3   ELSE
4       BEGIN
5           A ← ∅;
6           (w_1, w_2, ..., w_n) ← c(v);
7           FOR i ← 1 TO n DO      /* process the subtrees */
8               BEGIN
9                   (RESULT, LEFTEDGE_i) ← PACK-2-PATH(H, w_i);
10                  A ← A ∪ RESULT;
11                  IF (LEFTEDGE_i ≠ ∅) THEN      /* if there is a left-over edge */
12                      A ← A ∪ {{(v, w_i), LEFTEDGE_i}}
13              END;
14          (w'_1, w'_2, ..., w'_m) ← all the w_i such that LEFTEDGE_i = ∅;
15              /* collect remaining edges */
16          FOR j ← 1 to m − 1 STEP 2 DO      /* match these edges */
17              A ← A ∪ {{(v, w'_i), (v, w'_{i+1})}};
18          IF (m is even) THEN      /* all edges were successfully matched */
19              RETURN (A, ∅)
20          ELSE                     /* there was one left over */
21              RETURN (A, (v, w'_m))
22      END;
```

solutions for each $(RESULT, LEFTEDGE_i)$ are collected (lines 7-13). The star formed by the current vertex and its children is then processed in the following manner: Whenever there is a left-over edge for the solutions ($LEFTEDGE_i \neq \emptyset$) of any of the subtrees rooted at the children, it is used to form a 2-path together with an edge of the star (lines 11-12). The remaining edges of the star are then matched to form 2-paths themselves (lines 14-17). Depending on whether there is an odd or even number of edges left in the star, a left-over edge may result (lines 18-21). Figure 2 provides an example of how 2-paths are extracted on a call to PACK-2-PATH. Here, PACK-2-PATH is being processed for the tree rooted at vertex a and has been completed for the subtrees rooted at vertices b through h. Left-over edges have resulted for the subtrees rooted at b and g, hence, the 2-paths (a, b, i) and (a, g, j) are extracted. The remaining star, $(a : c, d, e, f, h)$, has five edges, four of which are extracted as two 2-paths $((c, a, d)$ and $(e, a, f))$. (a, h) is the left-over edge for this call to PACK-2-PATH. The algorithm is reminiscent of a depth-first search in that the children (subtrees) are always processed first, the base cases being the leaves. Leaves are in themselves subtrees with no edges so that they return an empty edge-packing as a solution with no left-over edge (lines 1-2).

The result obtained by algorithm PACK-2-PATH is clearly optimal since at most one edge is left over after the packing algorithm is applied. It is easy to see that algorithm PACK-2-PATH runs in $O(m_H)$ time: an edge is either extracted as part of a 2-path or carried over to the next call until all edges are exhausted. □

It should be noted that since it is always the case that at most one edge is left over in the edge-packing, the corresponding $EPart_{2-path}$ problem is likewise solved in linear time. $ECover_{2-path}$ also has the same complexity since all that needs to be done is add an extra 2-path if a left-over edge results.

Corollary 3.1 *$EPart_{2-path}(tree)$ and $ECover_{2-path}(tree)$ are solvable in $O(m_H)$ time.*

The algorithm described where the host graph is a tree easily extends to connected host graphs in general. As long as the host graph can be modeled as a tree, the algorithm can be applied accordingly. Any connected graph can be modeled as a tree as follows: Designate a vertex of the graph as the root. Starting with this root, mark all unmarked edges incident to it and designate all vertices adjacent to it through these edges as its children. Recursively apply this step to each child until all edges are exhausted. What results is a tree where some of the leaves may be duplicates of other nodes in the tree (see Figure 3). With this, we have the following result:

Theorem 3.2 *$EPack_{2-path}$ is solvable in $O(m_H)$ time.*

Proof: Given an arbitrary graph, a corresponding tree model can be obtained by the procedure described above and thereby algorithm PACK-2-PATH can be applied. The possibility of repeated vertices does not affect how the algorithm works since we are looking

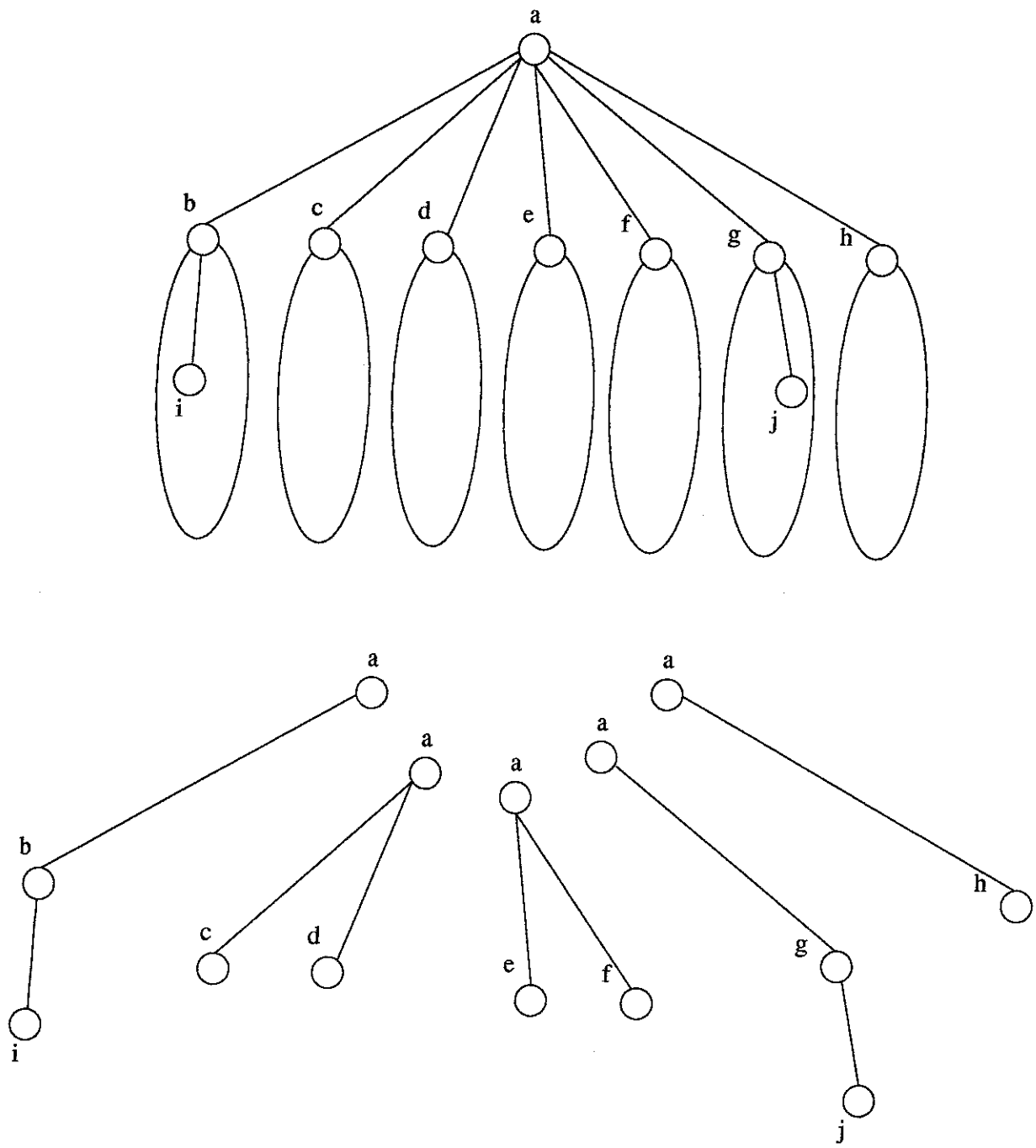

Figure 2: Recursively extracting 2-paths from a tree.

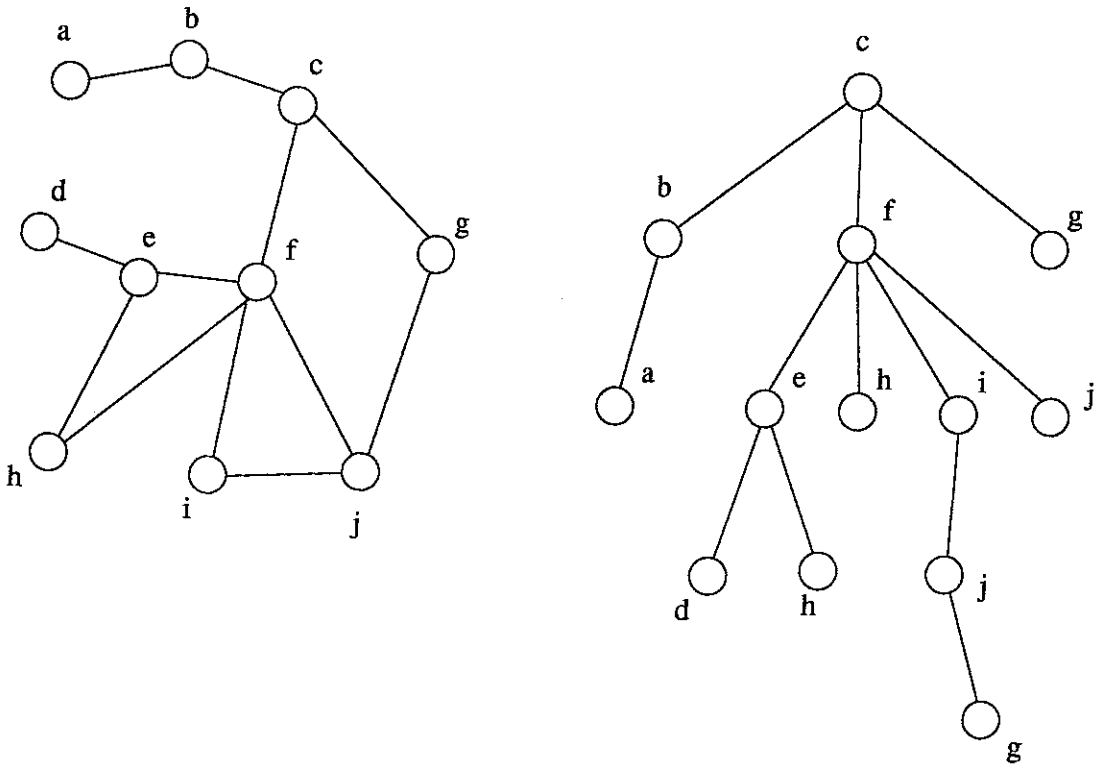

Figure 3: A tree model of a graph.

for edge-disjoint copies of 2-paths so that such repetition is allowed. The process takes $O(m_H)$ time since transforming an arbitrary graph to its tree model likewise takes $O(m_H)$ time (edges are just exhausted). □

As in the case for trees, at most one edge is left over in the packing so that the corresponding $EPart_G$ and $ECover_G$ problems are also solved in linear time.

Corollary 3.2 *$EPart_{2-path}$ and $ECover_{2-path}$ are solvable in $O(m_H)$ time.*

The results in this section provide one more interesting observation: if we consider the *edge graph* of the host (i.e., edges in the original graph become vertices which in turn are adjacent if their corresponding edges are adjacent in the original graph—sometimes called *line graphs* in other literature), the equivalent problem is to find a maximum set of independent (vertex-disjoint) edges in this edge graph. This is the regular matching problem which, for arbitrary host graphs, are solved in $O(n_H^{1/2} m_H)$ at best [PL]. We have shown in this section that the matching problem is simpler when the host graph is an edge graph.

Corollary 3.3 *The matching problem for edge graphs is solvable in $O(m_H)$ time.*

4 TREE HOSTS

When considering tree hosts, only guest graphs which are themselves trees are relevant. This section is divided into three parts, the first two of which deal with special classes of trees, namely, stars and paths. The third part addresses tree guests in general (with $|E_G| \geq 3$), where a less efficient but nevertheless polynomial-time algorithm is presented. The results in this section are extensions of the algorithm provided for 2-path edge-packing with tree hosts.

4.1 k-stars as guests

The algorithm to solve $EPack_{2-path}$ for trees immediately extends to $EPack_{k-star}$ for any $k > 2$.

Theorem 4.1 *$EPack_{k-star}(tree)$ is solvable in $O(m_H)$ time.*

Proof: Our revised algorithm is called PACK-K-STAR. We use the same recursive technique where each call to PACK-K-STAR produces a maximum set of k-paths and a set of left-over edges. However, instead of detecting pairs of edges to form 2-paths, k-tuples of edges are matched to form k-stars (lines 16-18). Of course, left-over edges are useful only when there are $k-1$ of them. If this is indeed the case (lines 19-20), they are used to form a k-star in the next higher-level call to the algorithm (lines 11-12). When there are less than $k-1$ left-over edges (lines 21-22), these are simply discarded since these edges can not possibly form another k-star unless a previously formed one is sacrificed. Figure 4 illustrates what occurs during a call to PACK-K-STAR. Here, the extracted 4-stars are $(c : a, i, j, k)$ and $(a : b, d, e, f)$ and the left-over edges form the star $(a : g, h)$.

Algorithm PACK-K-STAR likewise takes $O(m_H)$ time since, at each call, edges are either extracted, discarded, or carried over to the next call. Note that unlike in the case for 2-path edge-packing where at most one edge is left over (there is no way another 2-path can be extracted), the result obtained for k-stars presents a possibility that m_G or more edges may be discarded in the process so a counting argument does not suffice in showing that we indeed obtain a maximum edge-packing. Instead, we reason that the recursive method of collecting solutions of subtrees rooted at the children of the current vertex is appropriate because at most one k-star can be added through any of the children and this is provided for since left-over edges are carried over to the next level whenever possible. □

The corresponding $EPart_G$ problem is at least as easy since all that needs to be detected is the existence of discarded edges.

Corollary 4.1 *$EPart_{k-star}(tree)$ is solvable in $O(m_H)time$.*

Algorithm PACK-K-STAR. k-star edge-packing for trees.

INPUT: A rooted tree $H = (V_H, E_H)$ and a vertex v.
OUTPUT: A k-star edge-packing of the subtree rooted at v and left-over edges, if any.

```
1   IF (c(v) = ∅) THEN      /* if v is a leaf, return an empty solution and no left-over edges */
2       RETURN(∅, ∅)
3   ELSE
4       BEGIN
5           A ← ∅;
6           (w_1, w_2, ..., w_n) ← c(v);
7           FOR i ← 1 TO n DO      /* process the subtrees */
8               BEGIN
9                   (RESULT, LEFTEDGES_i) ← PACK-K-STAR(H, w_i);
10                  A ← A ∪ RESULT;
11                  IF (LEFTEDGES_i ≠ ∅) THEN
                        /* if there are k − 1 left-over edges from the subtree */
12                      A ← A ∪ {{(v, w_i)} ∪ LEFTEDGES_i}
13              END;
14          (w'_1, w'_2, ..., w'_m) ← all the w_i such that LEFTEDGES_i = ∅;
15              /* collect edges which were not used to form k-stars in previous step */
16          left ← m mod k;
17          FOR j ← 1 to m − k + 1 STEP k DO      /* match these edges */
18              A ← A ∪ {{(v, w'_i), (v, w'_{i+1}), ..., (v, w'_{i+k−1})}};
19          IF (left = k − 1) THEN      /* if there were k − 1 edges left over */
20              RETURN (A, {(v, w'_{m−k+1}), (v, w'_{m−k+2}), ..., (v, w'_m)})
21          ELSE                        /* if less than k − 1 edges are left-over, discard */
22              RETURN (A, ∅)
23      END;
```

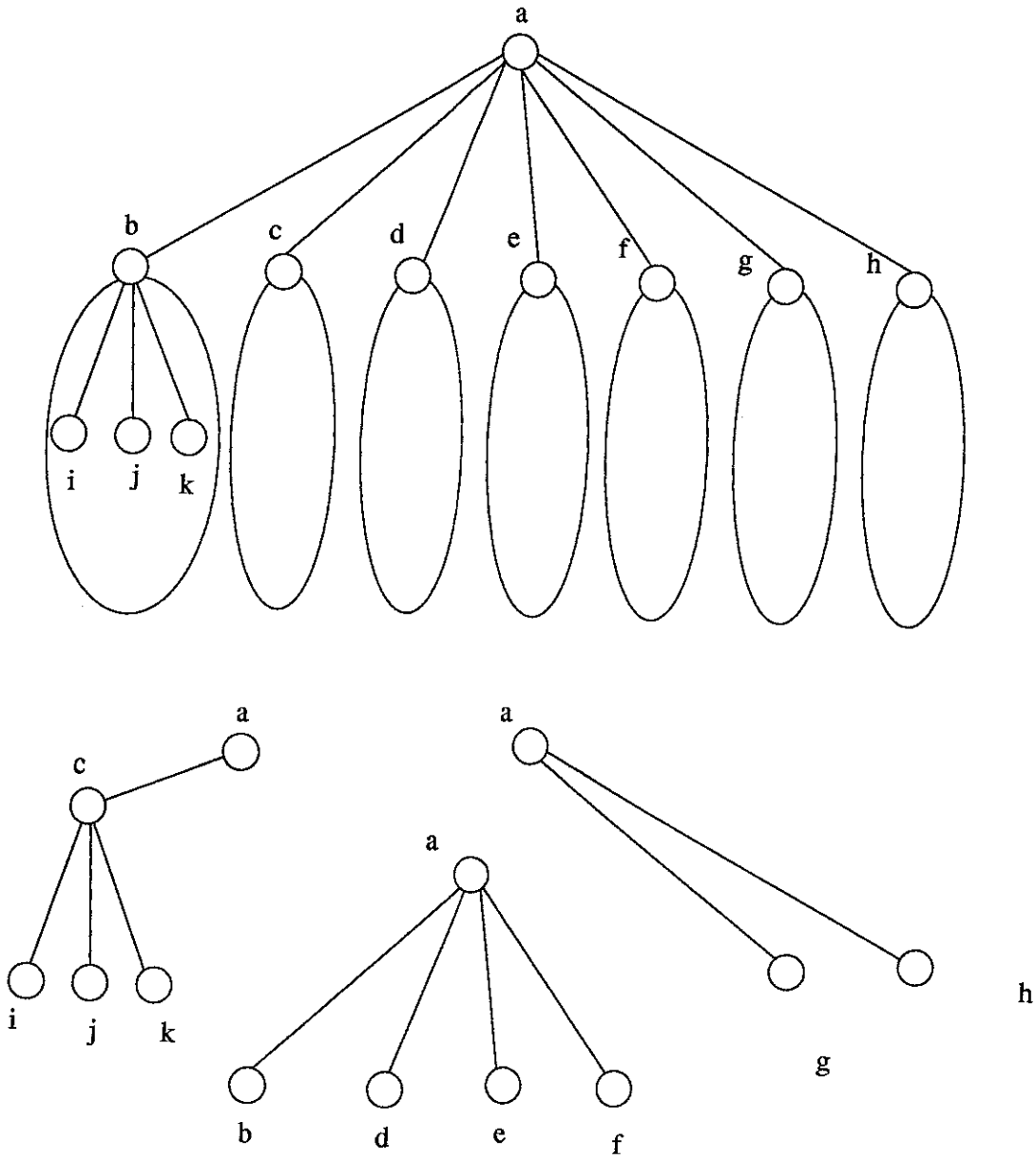

Figure 4: Recursively extracting k-stars ($k = 4$).

Algorithm PACK-K-PATH. k-path edge-packing for trees.

INPUT: A rooted tree $H = (V_H, E_H)$ and a vertex v.
OUTPUT: A k-path edge-packing of the subtree rooted at v and a left-over path (set of edges).

```
1   IF (c(v) ≠ ∅) THEN
2       RETURN (∅, ∅)      /* if v is a leaf, return an empty solution and no left-over path */
3   ELSE
4       BEGIN
5           A ← ∅;
6           (w_1, w_2, ..., w_n) ← c(v);
7           FOR i ← 1 TO n DO      /* process the subtrees */
8               BEGIN
9                   (RESULT, LEFTPATH_i) ← PACK-K-PATH(H, w_i);
10                  A ← A ∪ RESULT;
11                  IF (|LEFTPATH_i| = k − 1) THEN
12                      A ← A ∪ {{(v, w_i)} ∪ LEFTPATH_i};
13              END;
14          (w'_1, w'_2, ..., w'_m) ← all the w_i such that |LEFTPATH_i| < k − 1;
15          (LEFTPATH'_1, LEFTPATH'_2, ..., LEFTPATH'_m) ←
                all the LEFTPATH_i such that |LEFTPATH_i| < k − 1;
16          (TEMP, LONGESTLEFT) ← EXTRACT-K-PATHS
                    ((v, w'_1) ∪ LEFTPATH'_1, (v, w'_2) ∪ LEFTPATH'_2, ...,
                    (v, w'_m) ∪ LEFTPATH'_m);
17          RETURN(A ∪ TEMP, LONGESTLEFT)
18      END;
```

4.2 k-paths as guests

For guest graphs which are k-paths ($k > 2$), we present algorithm PACK-K-PATH. This is likewise a revision of our 2-path edge-packing algorithm for tree hosts. Although the revision is not a simple one, the same general concept is retained.

Theorem 4.2 *$EPack_{k-path}(tree)$ is solvable in $O(m_H)$ time.*

Proof: A call to algorithm PACK-K-PATH still produces an edge-packing for the subtree and a set of remaining edges which, this time, form a path down the subtree. As in our previous algorithms, a call to PACK-K-PATH collects the solutions $(RESULT, LEFTPATH_i)$ for the subtrees rooted at the children of the current vertex

Algorithm EXTRACT-K-PATHS. used in k-path edge-packing for trees.

INPUT: A set of paths $\{PATH_1, PATH_2, ..., PATH_n\}$.
OUTPUT: A maximum k-path edge-packing for this set of paths and a longest left-over path.

```
1   {PATHLIST_1, PATHLIST_2, ..., PATHLIST_k} ←
        BIN-SORT({PATH_1, PATH_2, ..., PATH_n}) according to |PATH_i|;
2   A ← ∅;
3   FOR i ← 1 TO k DO
4       BEGIN
5           MATCHED ← TRUE;
6           j ← i;
7           WHILE (|PATHLIST_i| > 0 AND MATCHED) DO
8               BEGIN
9                   WHILE (|PATHLIST_j| = 0 AND j < k) DO
                        /* look for a matching path */
10                      j ← j + 1;
11                  IF (j = k + 1) THEN
12                      MATCHED ← FALSE;
13                  ELSE
14                      BEGIN     /* if a match is found, extract the k-path. */
15                          PATH_x ← HEAD(PATHLIST_i);
16                          PATH_y ← HEAD(PATHLIST_j);
17                          A ← A∪ TRIM(PATH_x ∪ PATH_y, k)
                                /* TRIM the path to be of length k;
                                   extra edges are discarded */
18                          PATHLIST_i ← DELETEHEAD(PATHLIST_i);
19                          PATHLIST_j ← DELETEHEAD(PATHLIST_j);
20                      END;
21              END;
22      END;
23  i ← k;
24  WHILE (i > 0 AND |PATHLIST_i| = 0) DO
        /* look for the longest possible left-over path */
25      i ← i − 1;
26  IF (i = 0) THEN      /* all paths were used */
27      RETURN(A, ∅)
28  ELSE
29      RETURN(A, HEAD(PATHLIST_i));
```

(lines 7-13). Whenever $|LEFTPATH_i| = k-1$ for a subtree, the left-over path is used to form a k-path with a corresponding edge of the star formed by the current vertex and its children. The remaining edges of the star as well as the other left-over paths are then processed by the procedure EXTRACT-K-PATHS (line 16).

EXTRACT-K-PATHS takes a set of paths as input. These paths are exactly the paths formed by connecting the edges of the star with their corresponding left-over paths. The idea is to find as many k-paths as possible by combining pairs of these paths while leaving a longest possible left-over path. This is done by first sorting the lengths of the paths being processed (line 1) and then repeatedly matching the shortest path(s) with another shortest possible path such that the two can form a k-path (lines 9-20). The process stops when no more k-paths exist in the subtree. It is always guaranteed that the longest path is retained when possible (line 29) because the longest ones are the last to be used in the formation of new k-paths.

Figure 5 illustrates how k-paths are extracted during a call to PACK-K-PATH (and EXTRACT-K-PATHS). In the figure, the k-path first extracted is (a, h, p, q, r, s) because (h, p, q, r, s) is a $(k-1)$-path. The other extracted k-paths are (b, a, g, m, n, o) and (i, d, a, f, k, l). The retained path is (a, e, j) since it is the longest left-over.

Packing as many k-paths as possible in a subtree is appropriate because we can add at most one more k-path which is not entirely contained in this subtree to this maximum set. It is guaranteed that this possibility is provided for by leaving the longest path down the subtree for use by the next higher-level call to PACK-K-PATH. The sorting step in EXTRACT-K-PATHS takes $O(m_H)$ time since the elements being sorted are integral and bounded (by k), hence, a bin sort is appropriate. The same bound results in a linear time complexity even for the loop in lines 7-21 of EXTRACT-K-PATHS. Since k is fixed, the number of actual iterations performed in this loop is dependent on the number of edges of H.

□

The problem's $EPart_G$ counterpart has, of course, a similar complexity.

Corollary 4.2 *$EPart_{k-path}(tree)$ is solvable in $O(m_H)$ time.*

4.3 Tree guests in general

When the guest graph is an arbitrary tree, the revision becomes even more complicated although we still end up with a polynomial time algorithm.

First, we consider the fixed guest graph G and enumerate all possible ways (up to isomorphism) that G can be rooted. We then enumerate all the possible rooted subtrees that result from all such rooted trees. Figure 6 exhibits an example of a tree G, its possible forms as a rooted tree (T_0, T_1, T_2, T_3), and the resulting possible rooted subtrees $(S_0, S_1, S_2, S_3, S_4)$. Since G is fixed, there is a fixed number of possible rooted trees and subtrees for G. Furthermore, each T_i (or S_i) corresponds to a collection of subtrees which are exactly the subtrees that results if the root of T_i (or S_i) is deleted. In the example, T_2

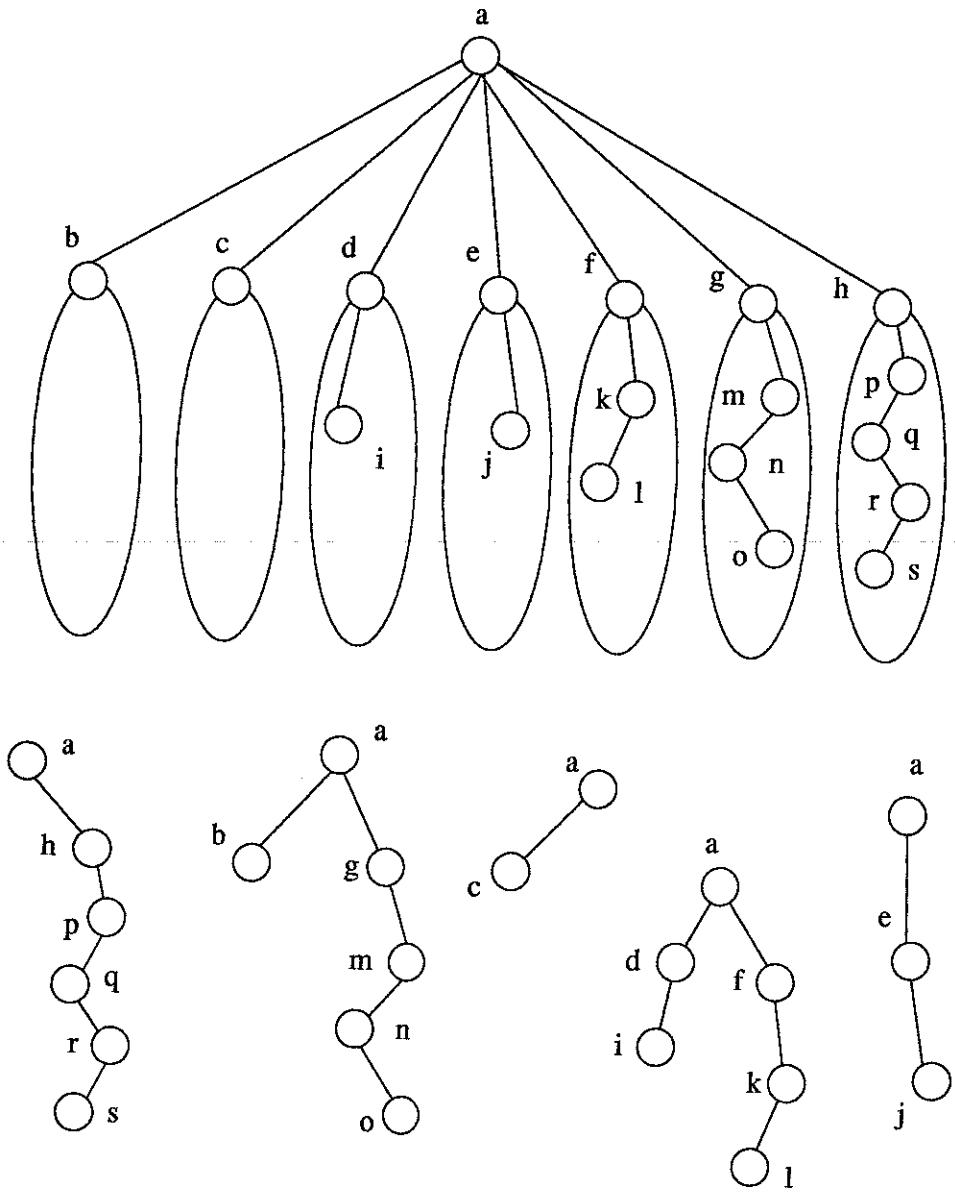

Figure 5: Recursively extracting k-paths ($k = 5$).

corresponds to the collection of subtrees (S_0, S_0, S_1) while S_4 corresponds to the collection (S_0, S_1). These rooted trees and subtrees of G as well as the collection of subtrees associated with them are used by our edge-packing algorithm.

As in our previous algorithms, algorithm PACK-TREE provides a solution for the subtree rooted at a given vertex v of the host tree H. This time, for simplicity, a solution consists of a number which corresponds to the size of a maximum edge-packing (instead of the packing itself) and a set of possible left-over subtrees. The second part may be any subset of the set of rooted subtrees of G and pertains to all possible parts of G that can be left over while retaining the maximum G edge-packing. The result of a call to PACK-TREE on a vertex v depends on the sets of possible left-over parts gathered by calling PACK-TREE on each of its children (lines 7-11). Algorithm EXTRACT-TREE is a procedure which collects as many copies of G from these sets of possible parts. Instead of presenting EXTRACT-TREE in algorithm format, we give a sketch on how it works. First, note that although there may be more than one possible part (subtree) that may be retained in a call to PACK-TREE on one of the children of v, for instance, only one part can actually be used on the current call to PACK-TREE. EXTRACT-TREE therefore involves partitioning the children of v into sets of size at most some fixed number (3, in our example). This fixed number is the maximum degree of a vertex of G, equivalently, the maximum number of subtrees that combine to form a copy of G. All possible ways to partition $c(v)$ are considered. For each possible partition, the algorithm tests whether a copy of G may be detected in each part of the partition using any of the rooted forms of G that were previously enumerated. The partitions where the most copies of G are detected are then considered. This number of copies of G is then kept track of as well as the set of possible left-over subtrees that can occur given that any of these maximum solutions were used. Of course, all this requires is, for each case, inspect those children of v which were not used to form a copy of G.

Figure 7 illustrates how the partitions are handled as well as how solutions are carried over to the next level; we use the example (for G) that we had earlier. Here, the solutions for vertices b, c, and d have been gathered and PACK-TREE is being processed for vertex a. There are five possible partitions for $\{b, c, d\}$ and they are $\{\{b\}, \{\mathbf{c}\}, \{d\}\}$, $\{\{\mathbf{b}, \mathbf{c}\}, \{d\}\}$, $\{\{\mathbf{b}, \mathbf{d}\}, \{\mathbf{c}\}\}$, $\{\{\mathbf{c}, \mathbf{d}\}, \{b\}\}$, and $\{\{\mathbf{b}, \mathbf{c}, \mathbf{d}\}\}$. The parts which are in bold are those which can form some T_i (or a copy of G). For example, the part $\{b, c\}$ form T_1 because the set of possible left-over subtrees of the solution for b includes S_2 while the set for c includes S_0. S_0 and S_2, in turn, forms a copy of T_1. In the example, the partition which exhibits the most copies of G is $\{\{b, d\}, \{c\}\}$ which has 2 copies. This, combined with 1, 0, and 2, the edge-packing solutions for b, c, and d, respectively, makes 5 copies of G for the tree rooted at a. Since all edges in the star $(a : b, c, d)$ were used up in this maximum G edge-packing, S_0 is the only possible left-over subtree.

We now formally state our result:

Theorem 4.3 *$EPack_G(tree)$, where G is a fixed tree, is solvable in polynomial time.*

Proof: The exhaustive nature of EXTRACT-TREE guarantees that the result obtained

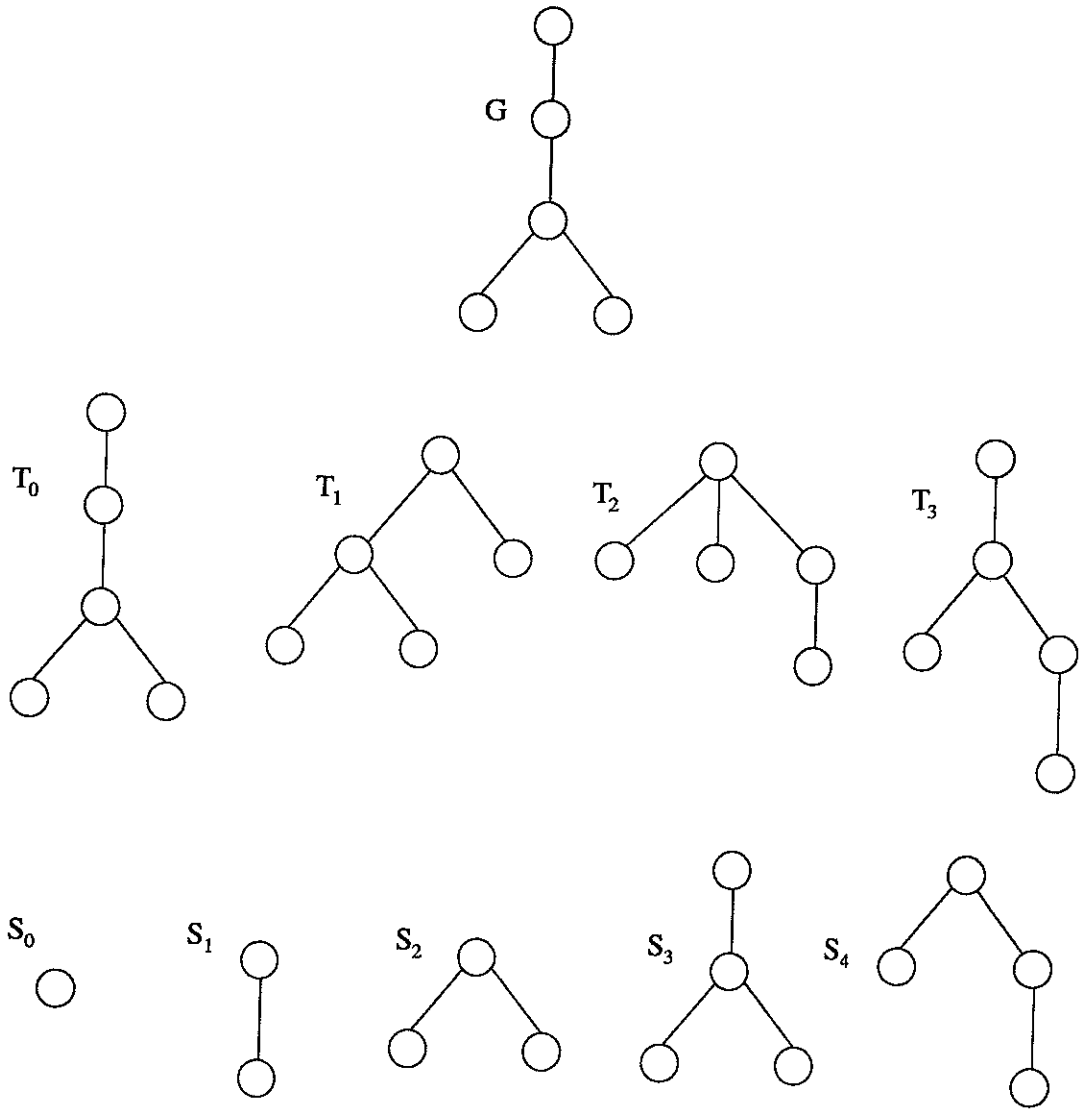

Figure 6: Rooted subtrees of G.

Algorithm PACK-TREE. G edge-packing for trees.

INPUT: A rooted tree $H = (V_H, E_H)$ and a vertex v.
OUTPUT: The size of a maximum G edge-packing of the subtree rooted at v
and a set of possible left-over subtrees.

```
1   IF (c(v) ≠ ∅) THEN
2       RETURN (0, {S_0});      /* if v is a leaf, return 0 and an empty left-over subtree*/
3       ELSE
4       BEGIN
5           COUNT ← 0;
6           (w_1, w_2, ..., w_n) ← c(v);
7           FOR i ← 1 TO n DO      /* process the subtrees */
8               BEGIN
9                   (RESULT, LEFTSUBS_i) ← PACK-TREE(H, w_i);
10                  COUNT ← COUNT + RESULT
11              END;
12          (TEMP, SUBTREES) ← EXTRACT-TREES
                (LEFTSUBS_1, LEFTSUBS_2, ..., LEFTSUBS_n);
13          RETURN(COUNT + TEMP, SUBTREES);
14      END;
```

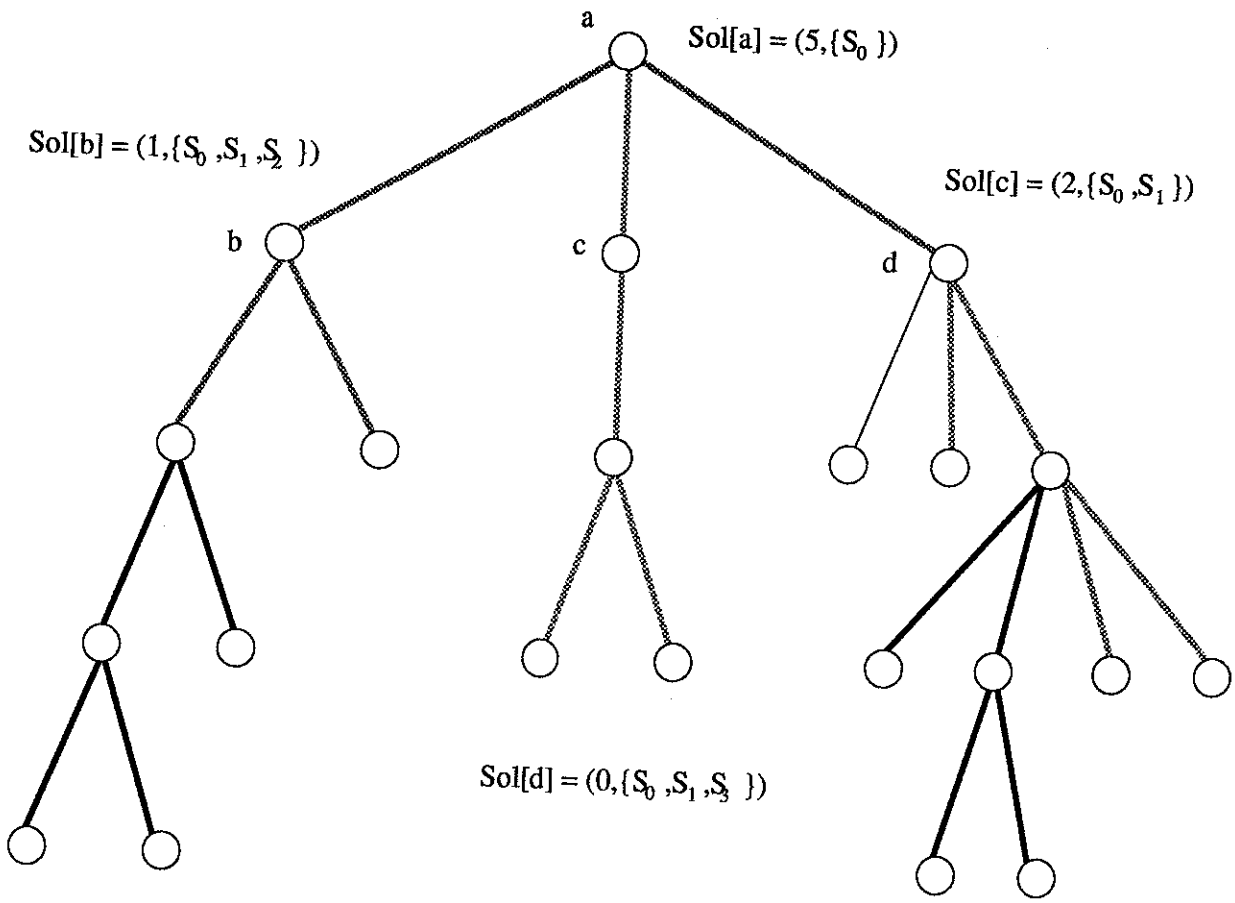

Figure 7: G-edge-packing for trees.

by algorithm PACK-TREE is indeed optimal. Using the maximum edge-packings of the subtrees of a given tree to compute the maximum edge-packing of the tree is appropriate because at most one other copy of G can be (partially) contained on each of the subtrees. This is handled by the fact that all possible left-over sets of edges which can be part of a copy of G are considered and exhausted in the algorithm. 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 [H1, NW]. This, in turn, is equivalent to an expression which is of order $O(n^{k-1})$. Therefore, there are $O(m_H^{p-1})$ possible partitions at each call to EXTRACT-TREE where p is the maximum degree of a vertex in G (the maximum size of a part in the partition). Since the number of T_i's and S_i's are fixed, EXTRACT-TREE takes $O(m_H^{p-1})$ time and it follows that PACK-TREE takes $O(m_H^p)$ time. □

Stars and paths are special cases because of their unique structure. For a k-star, there are only two ways to root such a graph; moreover, there are only two possible subtrees (the one-vertex tree and the $(k-1)$-star). A k-path is also a special case because its subtrees are all paths and a longest path always contains the other shorter subtrees so only the longest needs to be kept track of in the computation. In addition, the maximum degree for a vertex in a k-path is 2; recall that this corresponds to the maximum size of a part in a partition. As a result, and as seen in the previous sections, edge-packing with these guest graphs is a lot simpler and takes less time.

5 PLANAR HOSTS

Unfortunately, the extension technique used for 2-path edge-packing from tree hosts to arbitrary hosts does not apply to guest graphs which are trees in general. This is mainly because the algorithm for 2-paths always guaranteed an edge-packing that is maximum through a counting argument (at most one edge is left over) so that any tree model of a graph will work. However, for trees in general (stars, paths, or otherwise) as guest graphs, the edge-packing obtained usually involves leaving over m_G or more edges, thereby suggesting that the tree model of a graph may not produce the actual maximum set of guest graphs. Figure 8 is an example of a graph where there are 2 edge-disjoint 3-stars $(\{(b:a,c,e),(d:b,c,e)\})$ as well as 2 edge-disjoint 3-paths $(\{(a,b,d,e),(d,c,b,e)\})$, but its corresponding tree model fails to produce either of these sets. $EPack_G$ where G is a star or a path is, in fact, NP-complete for arbitrary hosts. This follows from a more restricted NP-completeness result which we prove in this section. The result is for planar hosts, a subproblem of the arbitrary case.

Recall that to prove NP-completeness, it has to be shown that $EPack_G$ is in NP and that it is NP-hard. Clearly, $EPack_G$ 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. It remains to show that $EPack_G$ is NP-hard, that is, it reduces from all problems in NP. Of course, it suffices to perform a reduction from a known NP-complete problem. We choose Planar 3-SAT since

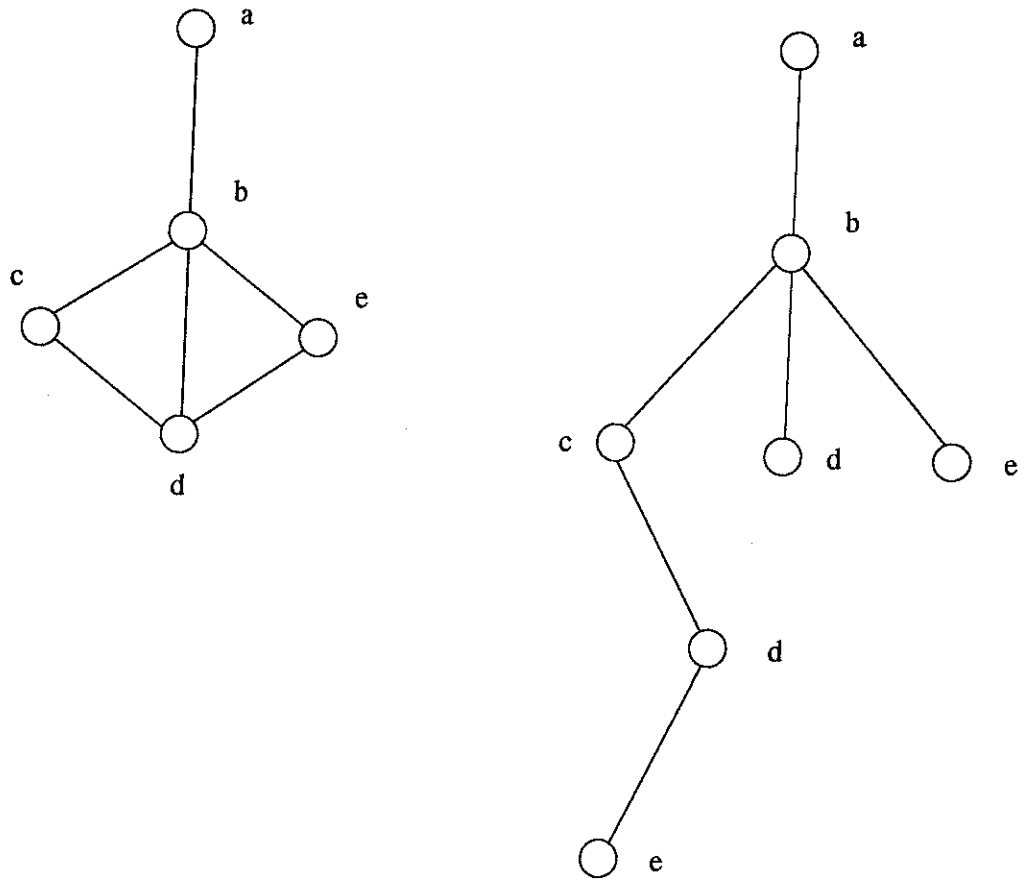

Figure 8: Counterexample: a graph where its tree model does not produce the correct set of 3-stars or 3-paths.

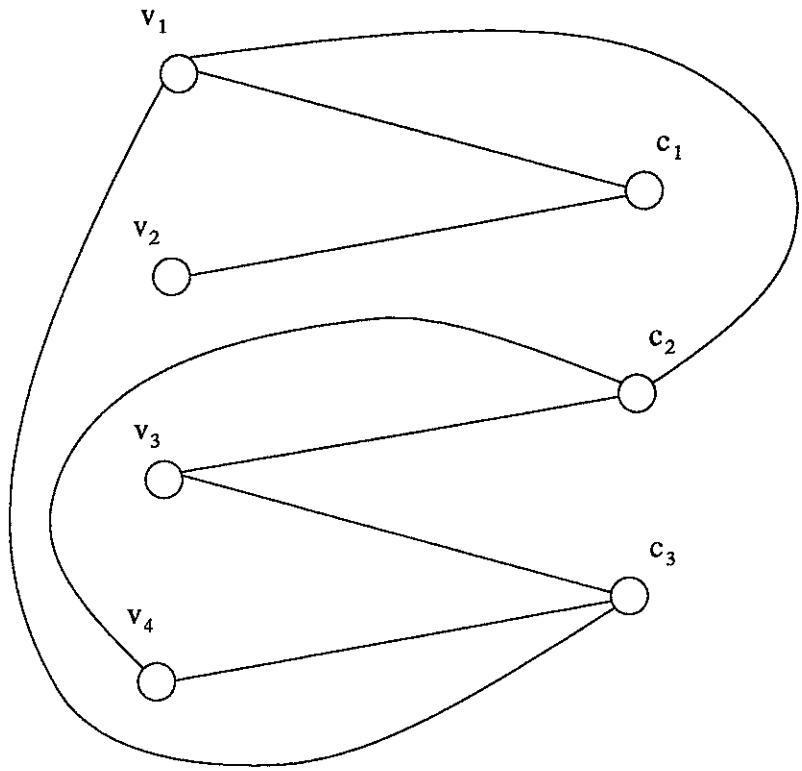

Figure 9: The graph of $(v_1 + v_2)(\bar{v_1} + v_3 + \bar{v_4})(v_1 + \bar{v_3} + v_4)$.

we deal with planar host graphs. Planar 3-SAT is the satisfiability problem with the added restrictions that each clause contain at most 3 literals and that the graph formed by the variables and clauses is planar. It is formally defined as follows [L]:

PLANAR 3-SATISFIABILITY (PLANAR 3-SAT).

INSTANCE: A set V of m variables $\{v_1, v_2, ..., v_m\}$, and a set C of n clauses $\{c_1, c_2, ..., c_n\}$ over V where $|c_i| \leq 3$ for each c_i in C such that the bipartite graph $G_p = (V \cup C, E)$, where E contains those pairs $\{(v, c) : v \text{ or } \bar{v} \text{ belongs to the clause } c\}$, is planar.
QUESTION: Is there a satisfying truth assignment for C?

Figure 9 exhibits a Planar 3-SAT instance. For each particular guest graph G where we intend to show the NP-completeness of $EPack_G(planar)$, we need to reduce an arbitrary Planar 3-SAT instance to an $EPack_G(planar)$ instance. Equivalently, we need to represent the variables and clauses described above in a graph H and show that for some K, there is a satisfying truth assignment for C if and only if there are K edge-disjoint copies of G in H. The method used is similar to that used by [BJLSS] in their generalized planar-matching NP-completeness proofs. The graph H (the $EPack_G$ instance) consists of gadgets (subgraphs) representing the variables and clauses of the Planar 3-SAT instance. These gadgets contain cascaded copies of G so that choosing particular copies of G means excluding others. In fact, the following requirements must be met:

- For each variable gadget, $V_i, 1 \leq j \leq m$, there must be exactly two ways to choose a maximum set of edge-disjoint copies of G which correspond to the true-or-false assignment (mode) of a variable. Furthermore, different alternating sets of edges must be available (not contained in a copy of G) for each mode.

- Every clause gadget $C_j, 1 \leq j \leq n$ must contain edges (and therefore, vertices) which are shared (identified) with the variable gadgets – corresponding to literal (v or $\bar{v}$) membership in a clause. These edges must be independently contained in a copy of G within the gadget and must be essential in the formation of that copy – the objective being that a copy of G is formed if and only if the clause is satisfied.

- The guest graphs which are cascaded in a gadget must occur in a circular (cyclic) fashion so that the planarity of the graph is retained.

Provided these requirements are satisfied, it directly follows that C is satisfiable if and only if there are K edge-disjoint copies of G in the constructed graph H.

In the following subsubsections, we first provide separate NP-completeness results for guest graphs which are stars and paths. We then extend these results to trees in general by addressing guests which are forks (trees which are neither stars or paths). Also, since we now investigate planar hosts, there are other possible guest graphs, namely, those containing cycles; we provide some results for these as well.

For each of these guest graphs, we show how the reduction is performed from the Planar 3-SAT instance, that is, what the graph H and the gadgets V_i and C_j look like for the different guest graphs.

5.1 k-stars as guests

Theorem 5.1 *$EPack_{k-star}(planar)$ is NP-complete.*

Proof: Let us first consider the case when G is a 3-star and then explain how to extend the proof to k-stars in general.

For 3-stars, a variable gadget, V_i, which corresponds to the variable v_i, contains $4n$ vertices, namely, $u_i[j]$, $\bar{u}_i[j]$, $v_i[j]$ and $\bar{v}_i[j]$, where $1 \leq j \leq n$. The edges for this variable gadget are those contained in the cycle ($u_i[1]$, $\bar{u}_i[1]$, $u_i[2]$, $\bar{u}_i[2]$,..., $u_i[n]$, $\bar{u}_i[n]$), and those of the form ($u_i[j]$, $v_i[j]$) and ($\bar{u}_i[j]$, $\bar{v}_i[j]$), which we call spike edges (or spikes). Figure 10 shows what a variable gadget looks like if $n = 3$. There are exactly two ways to choose a maximum set of 3-stars from such a gadget. One way is to extract all stars (there are n of them) of the form ($\bar{u}_i[j]$: $\bar{v}_i[j]$, $u_i[j]$, $u_i[j+1]$); the other is to choose those of the form ($u_i[j]$: $v_i[j]$, $\bar{u}_i[j]$, $\bar{u}_i[j-1]$). Note that the addition (or subtraction) within the indices wrap modulo n. These two choices correspond to true-or-false modes, as shown in the figure (chosen edges are in bold). In both cases, all cycle edges and half of the spike edges are chosen. The chosen spike edges are precisely one of two sets of alternating spikes; in either case, one set is chosen and the other is made available. Also, as will be seen when

we describe a clause gadget, some of the vertices in V_i may be identified with vertices from a clause gadget and from other variable gadgets.

A clause gadget, C_j, which corresponds to the clause c_j, on the other hand, contains $3 + |c_j|$ vertices, all but 2 of which ($c_j[1]$ and $c_j[2]$) 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 which variables are contained in the clause c_j. As in figure 11, suppose $c_j = (v_1 + v_2 + \bar{v}_3)$. Then, the other vertices involved in C_j are $u_1[j_1]$, $u_2[j_2]$, $\bar{u}_3[j_3]$, and a vertex (the center of the star) identified with all of $v_1[j_1]$, $v_2[j_2]$ and $\bar{v}_3[j_3]$. The edges between these vertices are 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 of the clause c_j with respect to its cyclic ordering over the respective variables (to preserve the planarity of the graph). Since the gadget is a $(2 + |c_j|)$-star and since $|c_j| \leq 3$, at most one 3-star can be extracted from it.

Figure 12 illustrates how H is constructed from our Planar 3-SAT instance in figure 9.

Since sharing a spike edge with a clause gadget signifies variable membership in a clause, it is clear that the only way a 3-star can be extracted from a clause gadget is when at least one of these spikes is not being used 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.

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 gadgets, the edges that are added stem from the vertices in the cycle ($u_i[j]$ and $\bar{u}_i[j]$). For the clause gadgets, they stem from the center of the star. Figures 13 and 14 show what these gadgets look like for 5-stars.

□

5.2 k-paths as guests

Theorem 5.2 *$EPack_{k-path}(planar)$ is NP-complete.*

Proof: Dyer and Frieze [DF] have proven that $EPart_G$ (and thus, $EPack_G$) is NP-complete for planar host graphs when G is a k-path. Nevertheless, for consistency and completeness, we show the variable and clause gadgets of an alternate proof.

We first deal with guest graphs which are 3-paths and later extend our results to k-paths in general.

For G a 3-path, a variable gadget, V_i, contains $10n$ vertices: $u_i[j]$, $\bar{u}_i[j]$, $v_i[j]$, $\bar{v}_i[j]$, $w_i[j]$, $x_i[j]$, $y_i[j]$, $\bar{w}_i[j]$, $\bar{x}_i[j]$, and $\bar{y}_i[j]$, where $1 \leq j \leq n$. Edges of a gadget include those contained in the paths: $(w_i[j], u_i[j], y_i[j], \bar{w}_i[j], \bar{u}_i[j], \bar{y}_i[j], w_i[j+1])$, for all $j, 1 \leq j \leq n$. They also include edges of the following form: $(w_i[j], x_i[j])$, $(\bar{w}_i[j], \bar{x}_i[j])$, $(u_i[j], v_i[j])$ and

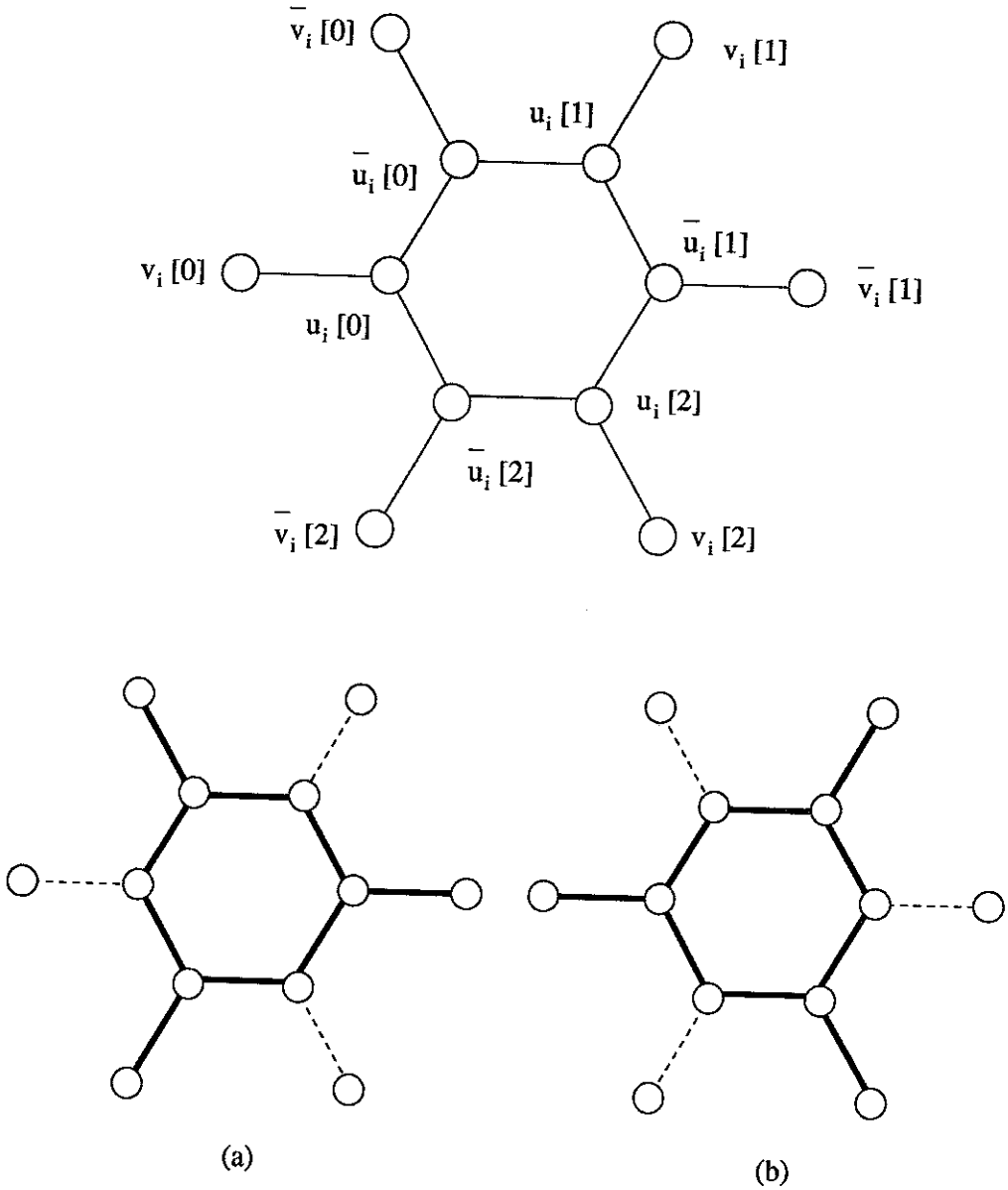

Figure 10: A variable gadget for 3-stars with its true and false modes.

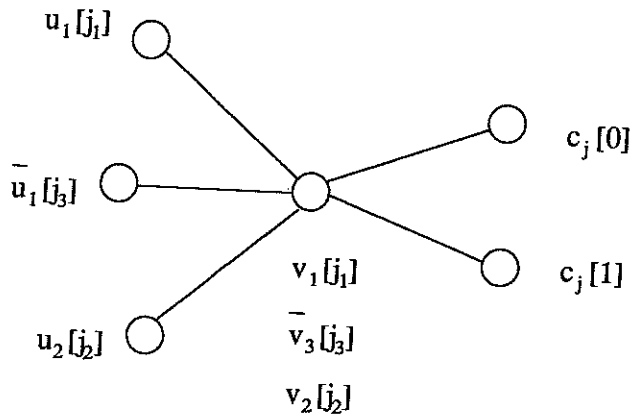

Figure 11: A clause gadget for 3-stars; assume $c_j = (v_1 + v_2 + \bar{v}_3)$.)

$(\bar{u}_i[j], \bar{v}_i[j])$. We call the last two edges spikes in the same manner that we did with k-stars. Figure 15 exhibits a variable gadget when $n = 3$. For simplicity, only one part of the gadget is identified and labeled. This subgadget, which corresponds to a clause containing the variable v_i, is in fact essential in our explanation.

Note that cascaded copies of the subgadget compose the entire gadget. Furthermore, the edges of the form $(w_i[j], x_i[j])$ are shared by neighboring subgadgets, with each subgadget having two shared edges, the left and the right. We claim that there are at most $3n$ 3-paths that can be edge-packed in the entire gadget. Each subgadget consists of eleven edges and therefore contains at most 3 edge-disjoint 3-paths. Also, the subgadget needs to use at least one of the shared edges to obtain these 3 3-paths. It does not help to extract 3-paths across subgadgets because an edge $(w_i[j], x_i[j])$ is just discarded when it could have been used as part of that 3-path. For example, the path $(\bar{y}_i[1], w_i[2], u_i[2], v_i[2])$ is one which spans across two sub-gadgets but extracting it causes the edge $(w_i[2], x_i[2])$ to be discarded when the path $(x_i[2], w_i[2], u_i[2], v_i[2])$ could have been extracted instead. Hence, we can assume that 3-paths are always wholly contained within a subgadget. And since a subgadget can contain 3 3-paths while using only one shared edge, the maximum number of 3-paths for the entire gadget is $3n$.

True or false modes correspond to whether the left or right shared edge is used when extracting the 3-paths (the choice of which shared edge is used propagates to all subgadgets). Let us suppose that the left shared edge is used. Doing so causes all $(u_i[j], v_i[j])$ – alternating spike edges – to be used. In fact, there are only two possible ways (per subgadget) to choose 3 3-paths, namely, the set of paths

$$\{(x_i[j], w_i[j], u_i[j], v_i[j]), (u_i[j], y_i[j], \bar{w}_i[j], \bar{x}_i[j]), (\bar{w}_i[j], \bar{u}_i[j], \bar{y}_i[j], w_i[j+1])\},$$

or the set of paths

$$\{(x_i[j], w_i[j], u_i[j], v_i[j]), (u_i[j], y_i[j], \bar{w}_i[j], \bar{x}_i[j]), (\bar{v}_i[j], \bar{u}_i[j], \bar{y}_i[j], w_i[j+1])\}.$$

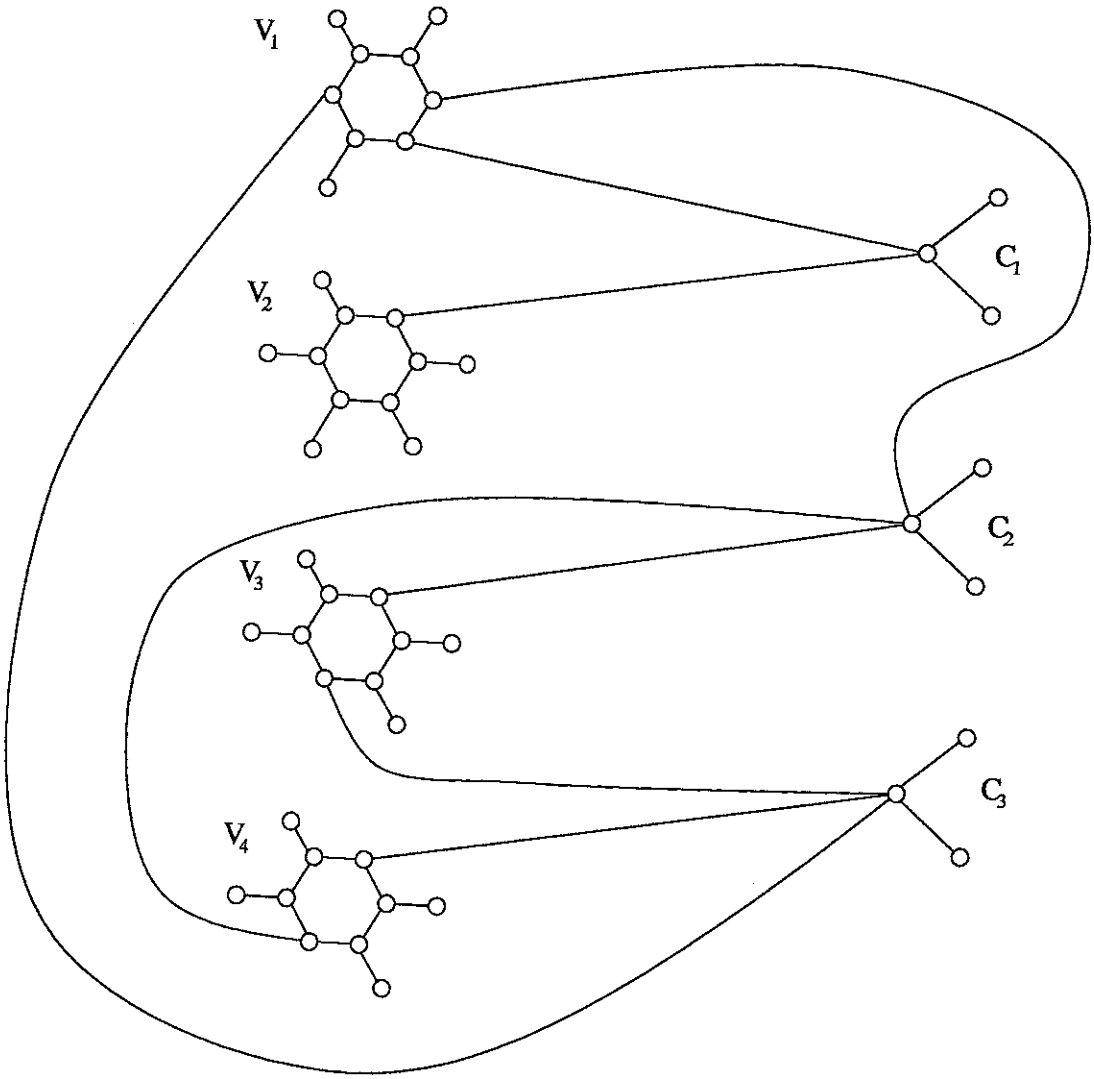

Figure 12: Reduction to an $EPack_{3-star}(planar)$ instance.

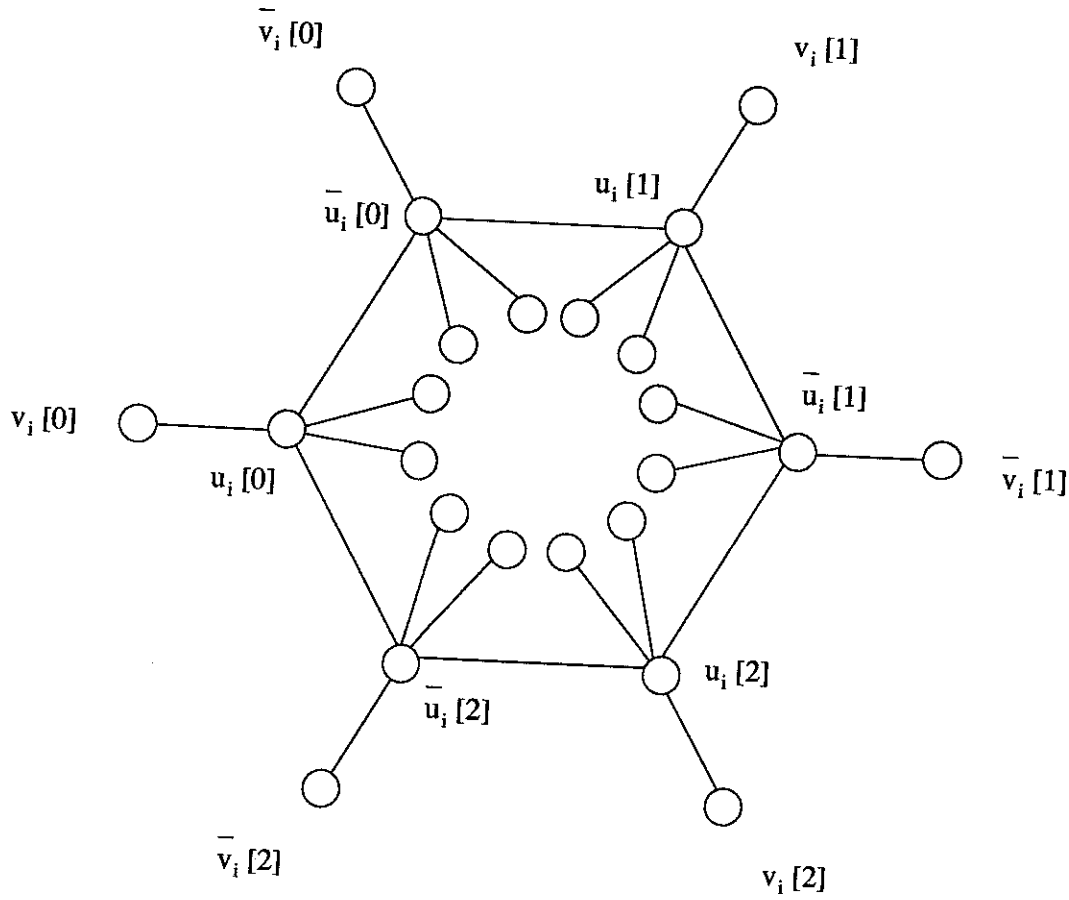

Figure 13: A variable gadget for 5-stars.

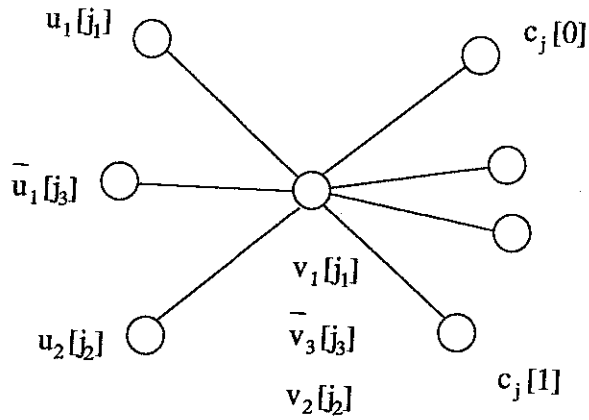

Figure 14: A clause gadget for 5-stars.

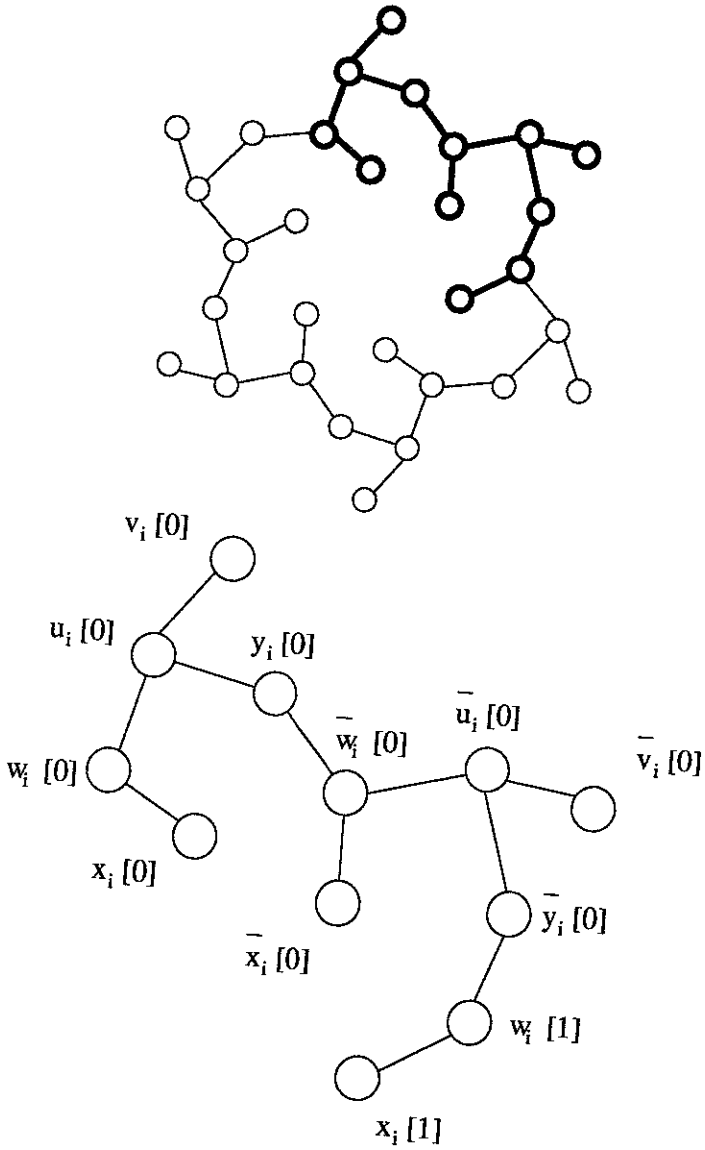

Figure 15: A variable gadget for 3-paths.

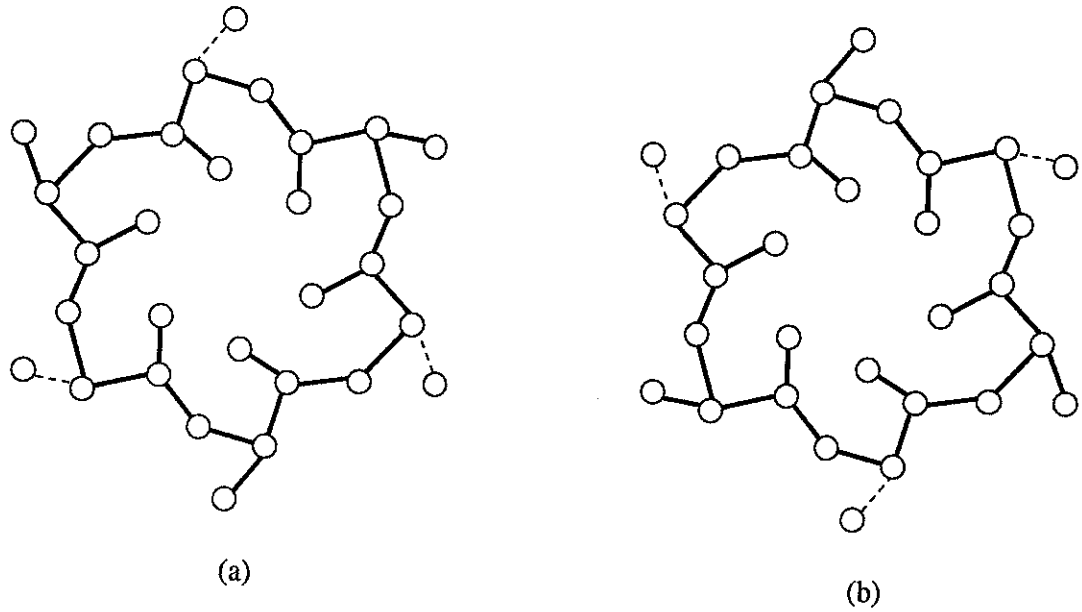

Figure 16: True and false modes for 3-paths.

It can be verified that these are the only possibilities. Moreover, since it is intended that the spike edges (either the edge $(u_i[j], v_i[j])$ or the edge $(\bar{u}_i[j], \bar{v}_i[j])$) are shared with clause gadgets, the second possibility is not relevant (both of the spikes are used). Using a similar argument for the case where the right shared edge of a subgadget is used, we end up with two possible ways to choose $3n$ 3-paths which correspond to true or false modes as shown in figure 16.

Again, as in the case when we handled 3-stars, it should be noted that the vertices of a variable gadget may be identified with vertices of other variable gadgets if the variables appear together in a clause.

A clause gadget, C_j, as shown in figure 17, contains vertices which include $c_j[1]$, $c_j[2]$,

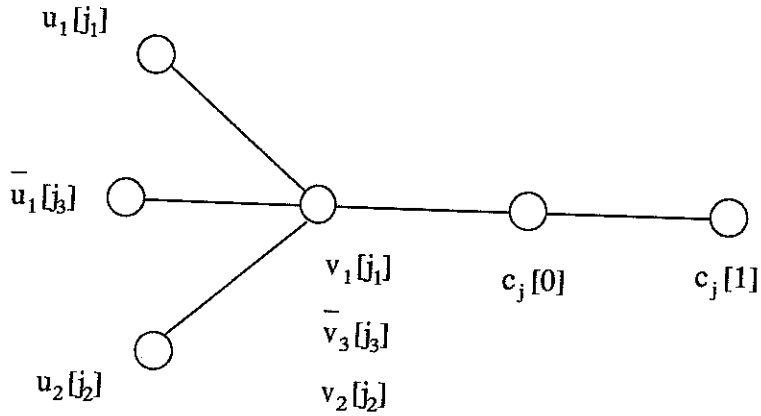

Figure 17: A clause gadget for 3-paths; assume $c_j = (v_1 + v_2 + \bar{v}_3)$.

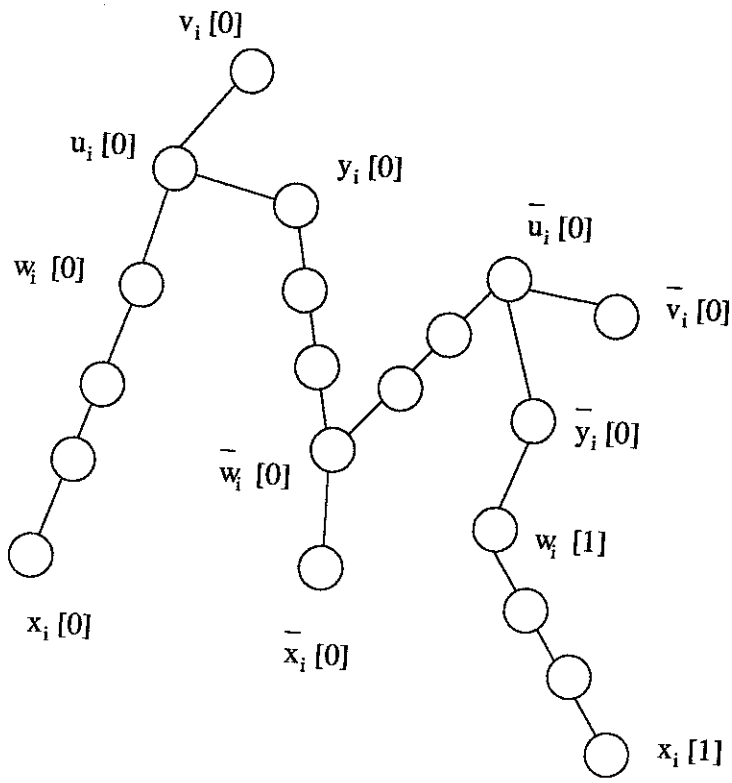

Figure 18: A variable subgadget for 5-paths.

and $|c_j| + 1$ other vertices identified with the vertices of variable gadgets associated with the variables contained in c_j. The edges of C_j are those of a 2-path formed by $c_j[1]$, $c_j[2]$ and the outer vertex ($v_i[j]$ or $v_i[\bar{j}]$) of the variable gadgets involved, and those of a $|c_j|$-star formed by the spikes of the variable gadgets. The figure provides an example where we use the same clause that we have used with 3-stars ($c_j = (v_1 + v_2 + \bar{v_3})$). The objective is that a 3-path can be extracted from the clause gadget if and only if at least one of the edges (a spike in the variable gadget) in the star is not being used by a variable gadget. The way planarity is preserved is the same as that used when dealing with stars: the cyclic ordering over the respective variables is maintained.

There are $3n$ 3-paths that can be extracted per variable gadget; there is at most one 3-path per clause gadget and this occurs only when at least one of the edges shared with the variable gadgets is available. Thus, the value of K in our reduction is $3mn + n$. Extension to k-paths for $k > 3$ simply involves lengthening the parts of the gadgets to accommodate k-paths instead of 3-paths. In particular, the edges of the form $(w_i[j], x_i[j])$, $(y_i[j], \bar{w_i}[j])$, $(\bar{w_i}[j], \bar{u_i}[j])$ and $(c_j[1], c_j[2])$ are magnified accordingly. As an example, figure 18 shows the variable subgadget for 5-path edge-packing while figure 19 shows the clause gadget. □

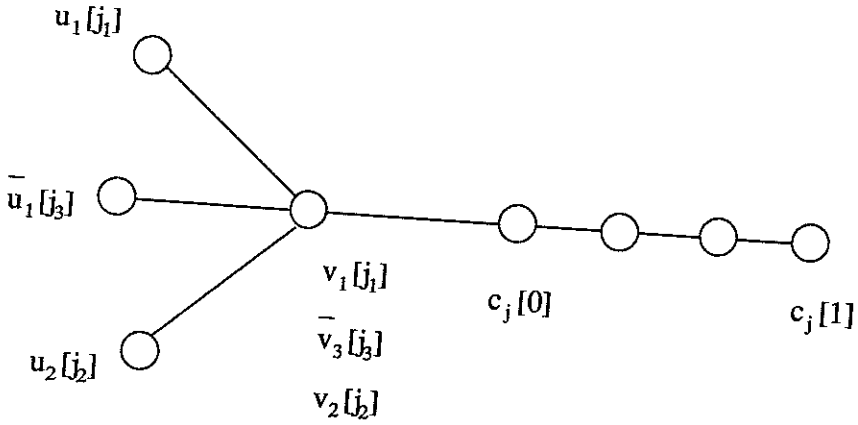

Figure 19: A clause gadget for 5-paths.

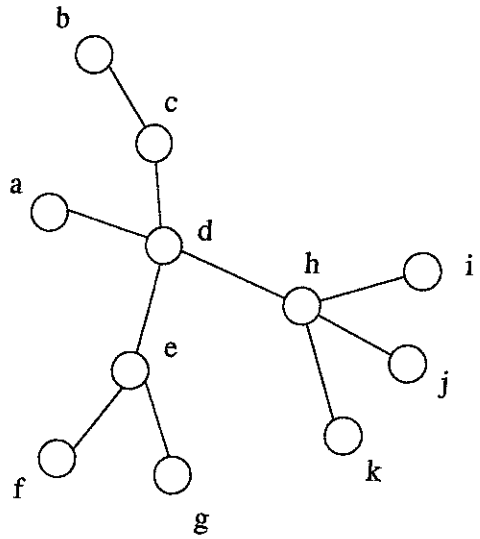

Figure 20: An example of a fork.

5.3 Forks as guests

In this section, we consider tree guests which are neither stars nor paths. We call these graphs forks and extend our results for stars and paths to these guest graphs. We first examine the structure of a fork.

We begin with some definitions. 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 20, 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.

Given the above definitions, it is easy to verify the following lemma:

Lemma 5.1 *A fork has at least three pendant edges and at least two pendant connectors.*

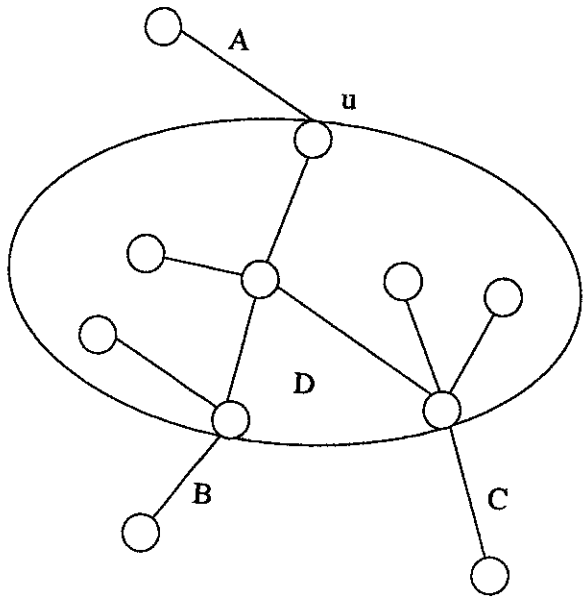

Figure 21: Dividing a fork into parts.

Our proof depends on the above lemma and involves dividing a given fork into four parts. First, we identify the smallest pendant connector of the fork, that is, the one with minimum degree. Let u be this pendant connector. We then arbitrarily select one of the pendant edges incident to u and denote this as edge A. Next, we select two other pendant edges in the fork which are not incident to u. These edges exist as implied by lemma 5.1 and by our choice of u. We denote these as edges B and C. The rest of the graph is denoted as part D. Figure 21 illustrates how such a division into parts is made using the same fork used in the previous example.

Before we describe the gadgets, consider figure 22, where we show a graph containing a copy of a fork G with its parts labeled accordingly. Note that there are two choices, A and A', for the first part of G. Also, parts E and F refer to the rest of the graph connected to A and A', respectively. We claim that when packing copies of G in this graph, the only significant 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 part of F to form two copies of G. In other words, either part D is wholly contained in a single copy of G or not at all. This claim is easy to verify as the following discussion shows. Suppose two edge-disjoint copies of G exists through D (or vertex u) with one copy containing A and the other containing A'. Let k be the degree of the smallest pendant branch of G. In the figure, the degree of u is clearly $k+1$. Extracting two copies of G through u will cause one copy to have a pendant branch of degree less than k, which is a contradiction, and thus verifies our claim that only one copy of G may be extracted using part D.

We are now ready describe our gadgets. We omit the explicit enumeration of vertices and edges for simplicity.

The principle behind the construction is similar to our previous results for stars and

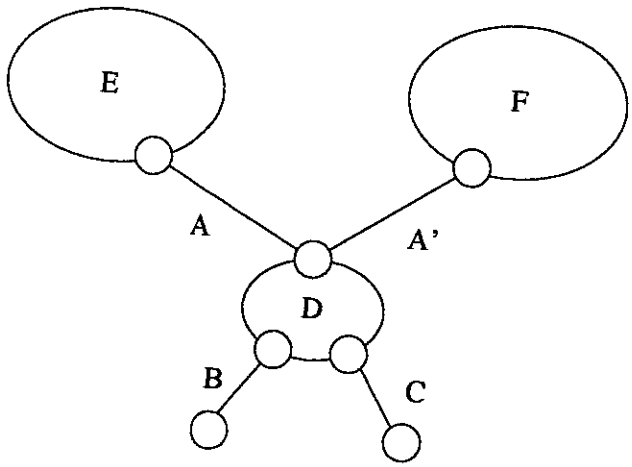

Figure 22: A fork in a graph.

paths. Eight copies of D are linked together as shown in figure 23 and sets of these are cascaded to form the variable gadget, V_i. Note that at most seven edge-disjoint copies of G may be extracted from each set. We have labeled only one set or subgadget for simplicity. Although there are several ways to pack these seven copies in a subgadget, the only significant ones are those which alternately use and make available edges of the form $(u_i[j], v_i[j])$ and $(\bar{u}_i[j], \bar{v}_i[j])$. These edges again serve as the "spikes" of our variable gadget. Essential in the selection of copies of G is the decision on 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 a variable gadget are demonstrated in figure 24.

A clause gadget, C_j, is simply a copy of the fork G with c_j alternatives for edge A, as exemplified by figure 25. These are precisely the edges that are identified with the variable gadgets. Of course, the motivation is that whenever at least one of these edges is free, a copy of G is formed.

As in our previous proofs, the planarity of the graph is retained. This essentially completes our construction.

Theorem 5.3 *$EPack_G(planar)$ where G is a fixed fork, is NP-complete.*

Proof: The proof follows from the reduction discussed above. There are $7n$ copies of G in a variable gadget and at most one copy of G may exist 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 in the graph if and only if there is a corresponding truth assignment in the Planar 3-Sat instance.

□

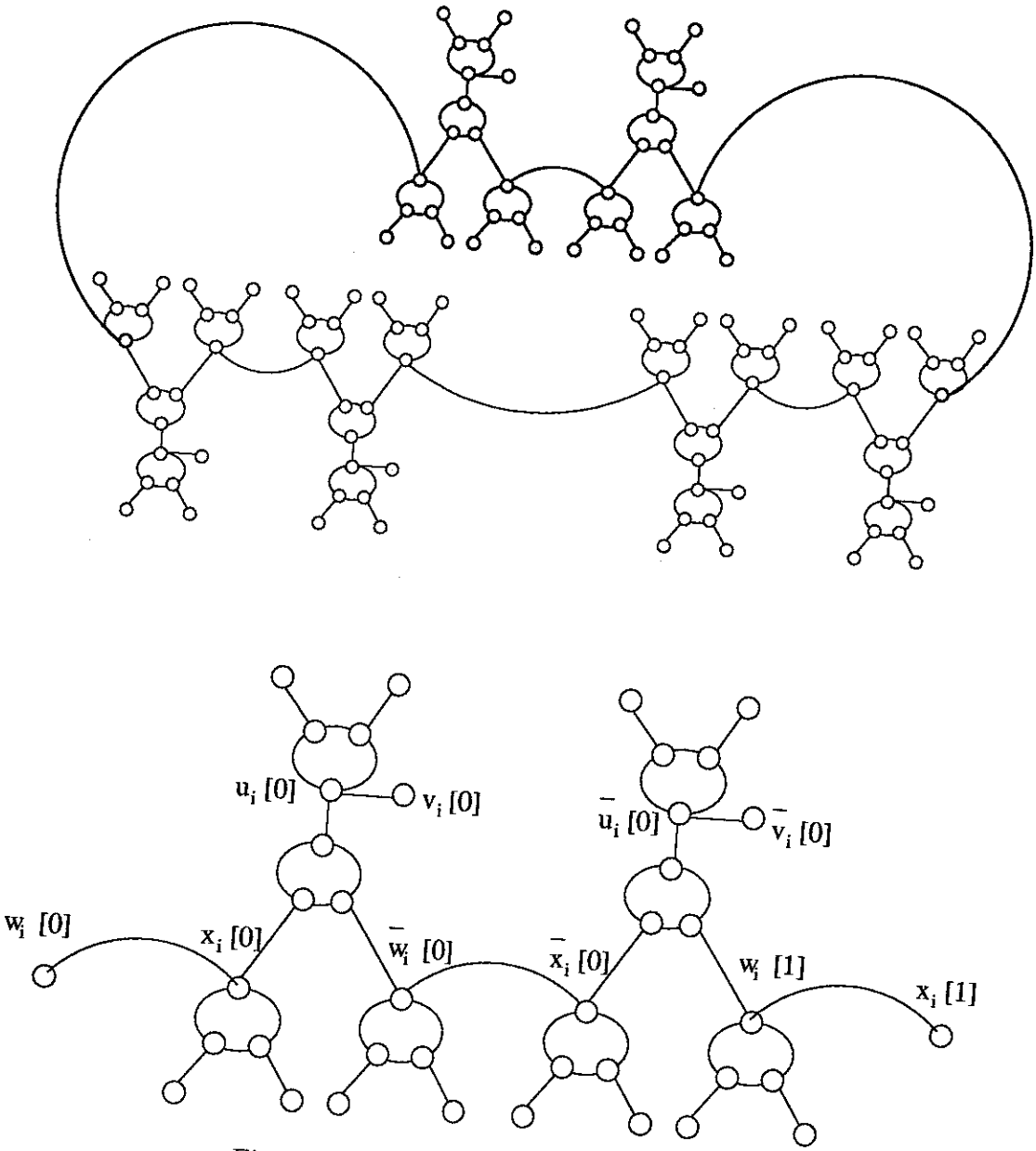

Figure 23: A variable gadget for forks.

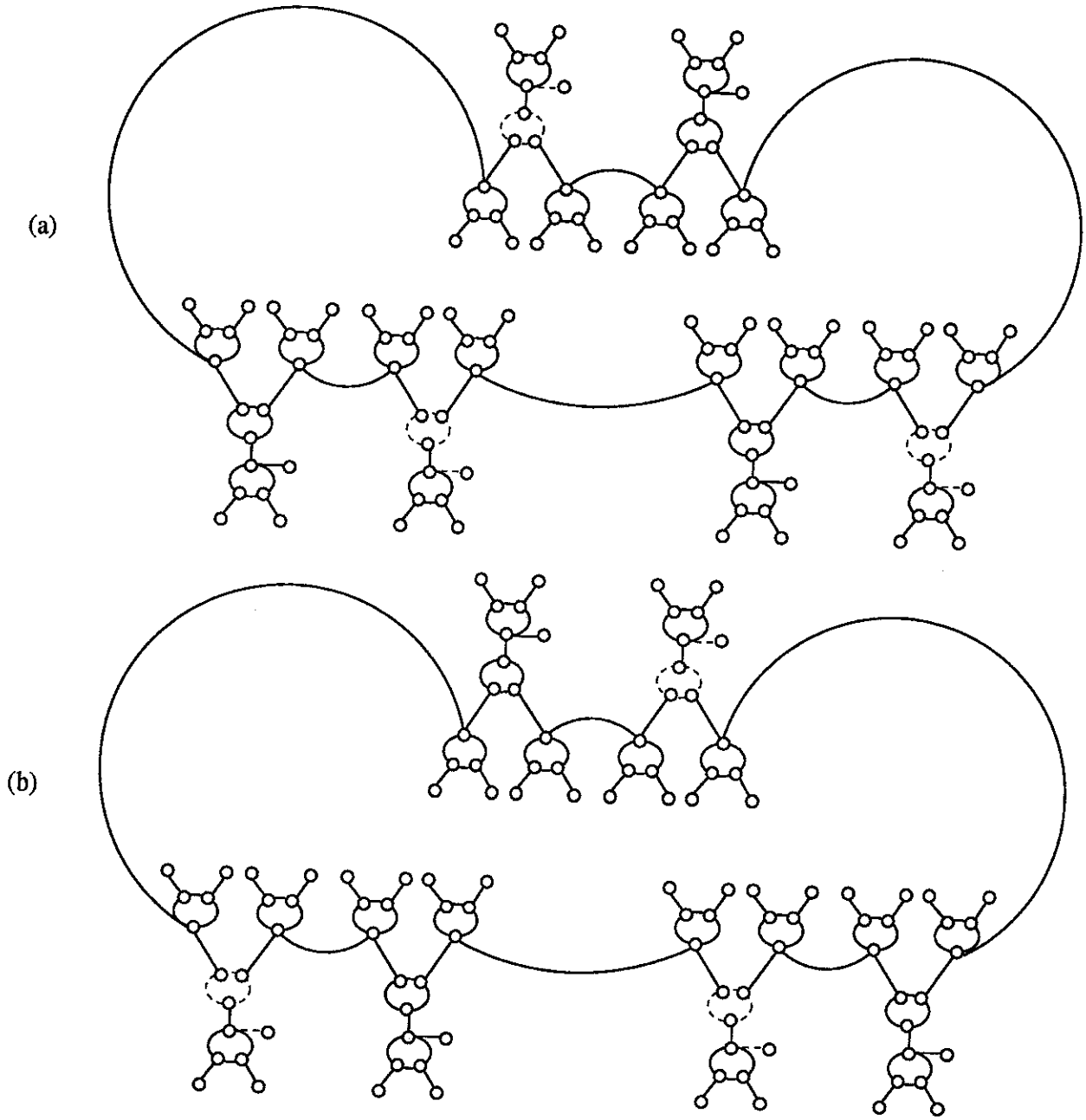

Figure 24: True and false modes for forks.

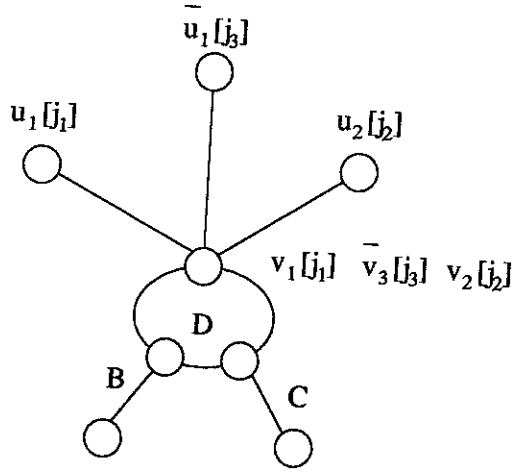

Figure 25: A clause gadget for forks.

5.4 k-cycles as guests

k-cycle edge-packing is likewise NP-complete for planar host graphs. Similar constructions for gadgets are made with a slight difference when dealing with clause gadgets. We formally state our result as a theorem:

Theorem 5.4 *$EPack_{k-cycle}(planar)$ is NP-complete.*

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

A variable gadget, V_i, for 3-cycles contains $2n+1$ vertices which are: $v_i[j]$ and $\bar{v}_i[j]$, where $1 \leq j \leq n$, and an extra vertex u_i. The edges for this variable gadget are those contained in the cycle $(v_i[1], \bar{v}_i[1], v_i[2], \bar{v}_i[2], \ldots, v_i[n], \bar{v}_i[n])$, and those contained in the star formed by the edges $(u_i, v_i[j])$ and $(u_i, \bar{v}_i[j])$, $1 \leq j \leq n$.

Figure 26 shows the gadget for $n = 3$. The figure also shows the gadget in its true and false modes. These correspond to the two ways that n 3-cycles can be extracted. In each case, alternating cycle edges are used and this is significant because these are the edges that are shared with the clause gadgets. It can be verified that n is the maximum number of 3-cycles that can be extracted 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 27. As in our previous cases, the construction of a clause gadget depends on the variables contained in a particular clause. Each C_j contains 10 vertices whenever there are 3 variables in the corresponding clause, as in the figure. Edges in a clause gadget form a structure similar to the variable gadget: they form edges of a cycle and a star. Taking the specific clause example that we have been using $(c_j = (v_1 + v_2 + \bar{v}_3))$, the vertices are $c_j[0]$, $v_1[j_1]$, $\bar{v}_1[j_1]$, $c_j[1]$, $\bar{v}_3[j_3]$, $v_3[j_3+1]$, $c_j[2]$, $v_2[j_2]$, $\bar{v}_2[j_2]$, and $c_j[3]$. Clearly, some of these vertices are shared with the corresponding variable gadgets, specifically, those vertices which are part of a cycle edge in the variable gadget. The maximum number of 3-cycles that can be extracted from this particular gadget is 4

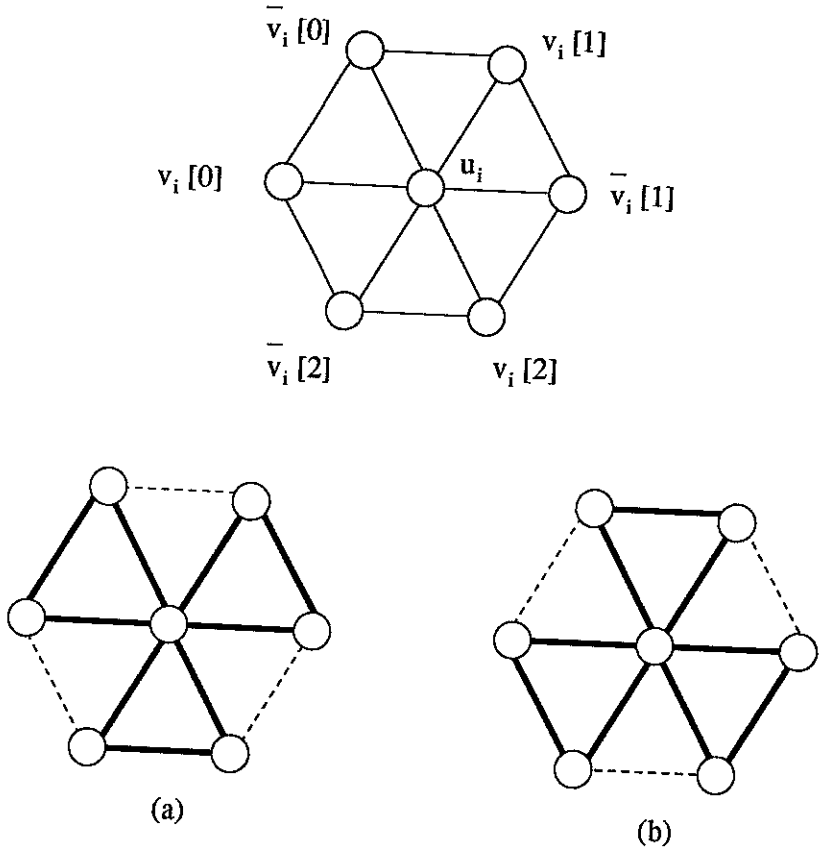

Figure 26: A variable gadget for 3-cycles with its true and false modes.

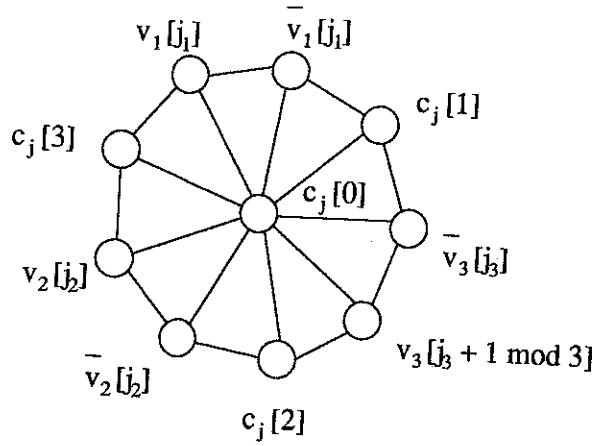

Figure 27: A clause gadget for 3-cycles; assume $c_j = (v_1 + v_2 + \bar{v_3})$.

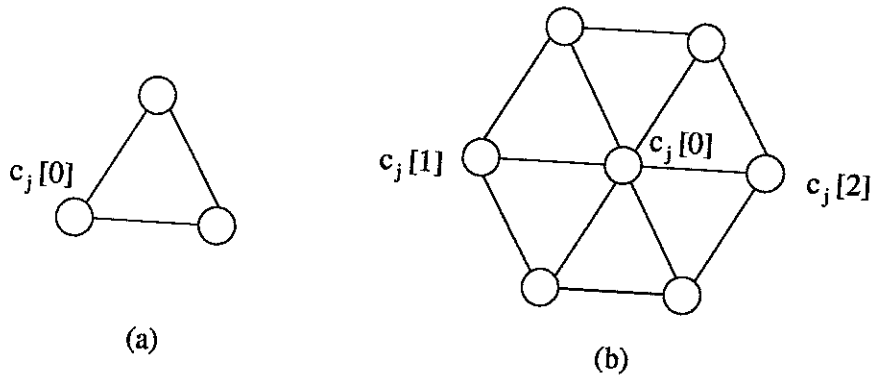

Figure 28: Other clause gadgets for 3-cycles: (a) $|c_j| = 1$; (b) $|c_j| = 2$.

and this is possible only if at least one of the shared edges is not being used by a variable gadget.

For clauses containing 1 or 2 variables, the gadgets look like those in figure 28 and the maximum number of 3 cycles that can be extracted are 1 and 3, respectively. The value for K in our reduction is therefore $mn + n_1 + 3n_2 + 4n_3$, where n_i is the number of clauses which contain exactly i variables.

Extension to k-cycles in general requires magnifying the cycle edges of the gadgets (those which are in the outer cycle) to accommodate for k-cycles instead of 3-cycles. Figure 29 and 30 illustrate how this is done for 5-cycles.

□

5.5 Arbitrary guest graphs

There are other possible planar guest graphs which are neither trees nor cycles. We conjecture that $EPack_G(planar)$ is NP-complete for these guest graphs as well.

Conjecture 5.1 *$EPack_G(planar)$, where G is a fixed planar graph with 3 or more edges, is NP-complete .*

A possible approach that may be taken to prove the above conjecture is to "model" the arbitrary guest graph in terms of a guest graph where the NP-completeness of $EPack_G$ is already known. For example, if we can model some arbitrary planar guest graph by a cycle and pattern the reduction accordingly, the NP-completeness of $EPack_G$ for that planar guest graph will follow provided that the conditions necessary for the gadgets involved are retained. This generalization technique was used by Kirkpatrick and Hell [KH] and Berman et al [BJLSS] in proving the completeness of Maximum G Matching. The proofs they had are both divided into two parts: guest graphs with unique maximum two-connected components and guest graphs which have more than one maximum two-connected component. The base cases for these two parts are 3-cycles and 2-paths, respectively. After showing

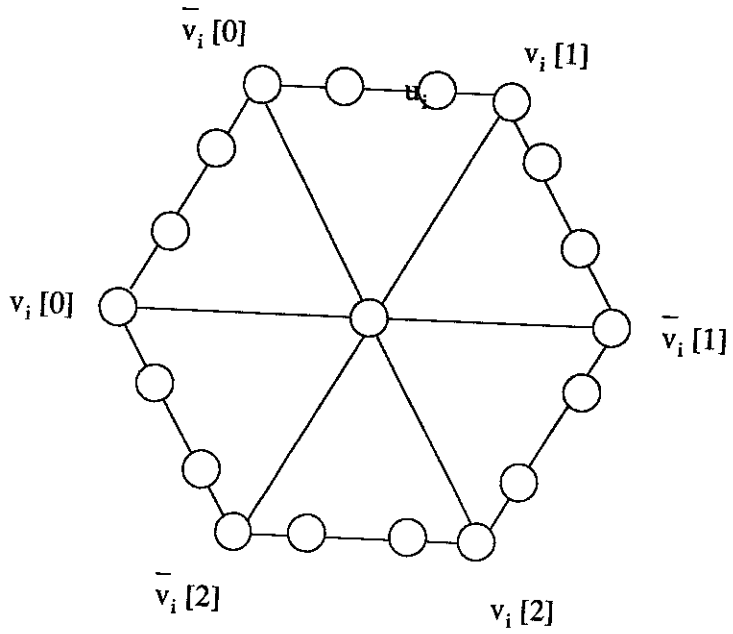

Figure 29: A variable gadget for 5-cycles.

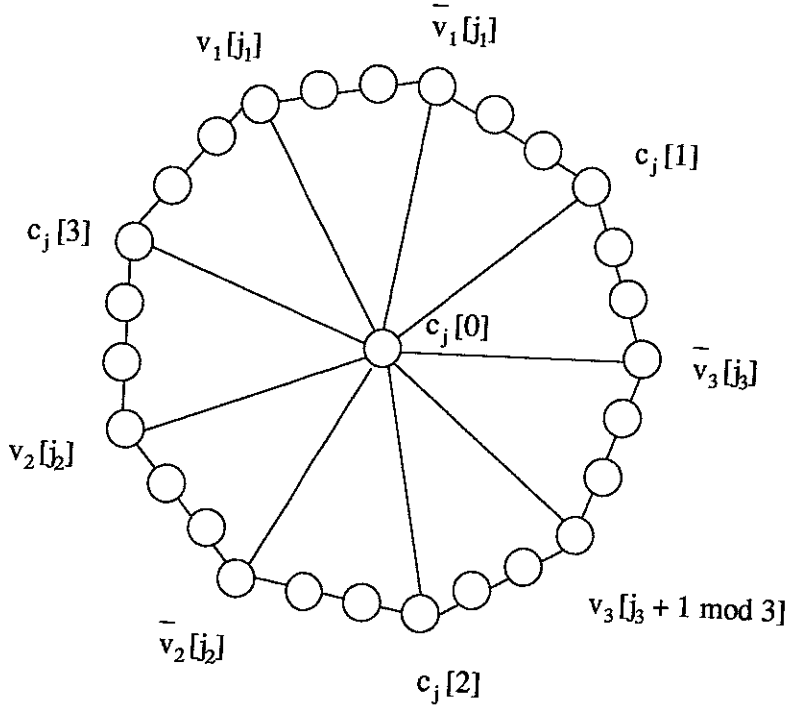

Figure 30: A clause gadget for 5-cycles.

NP-completeness for these base cases, they modeled arbitrary guest graphs with these graphs and concluded NP-completeness for these graphs as well. However, the gadgets used were copies of G cascaded through single vertices thereby controlling the possibility of creating other copies of G within and across the original copies. Since there is instead edge-disjointedness in $EPack_G$, copies are cascaded through edges and the existence of other copies of G is generally less predictable. It is in this respect that the technique used in Maximum G Matching does not immediately translate to Maximum G Edge-Packing.

We could, however, generalize to some restricted subclasses of cyclic graphs because they directly follow from the ideas of our previous results. For example, unicyclic guest graphs (graphs with one cycle) are trivial consequences of our proof for cycles since the additional edges are simply attached to the cycles in the gadgets and the proof immediately applies. Also, if the cyclic guest graph has enough pendant edges and pendant connectors, then our proof for fork guest may apply as well.

6 ARBITRARY HOSTS

The NP-completeness results obtained with planar hosts carry over to arbitrary hosts since the planar case is just a subproblem of the arbitrary case.

Corollary 6.1 *$EPack_G$, where G is a fixed tree or cycle, is NP-complete.*

In fact, this result has been proven by previous work [C4, H3, DF] as discussed in the introductory section. In fact, this arbitrary case has been completely resolved by Corneil et al [C4].

Theorem 6.1 *$EPack_G$, where G is a fixed graph with 3 or more edges, is NP-complete .*

7 OUTERPLANAR HOSTS

Conjecture 7.1 *$EPack_G(outerplanar)$, where G is a fixed outerplanar graph, is solvable in polynomial time.*

We conjecture that $EPack_G$ is solvable in polynomial time when both the guest and the host graph are restricted to outerplanar graphs. Many NP-complete graph problems which remain NP-complete if the graph involved is planar have been found to be solvable in polynomial time when the graph is outerplanar [B, J3].

We can, in fact, present a simple algorithm to solve $EPack_G(outerplanar)$ where G is a triangle (3-cycle). Consider the *dual* of an outerplanar graph which we define as follows: Each face (except for the outer face) in the graph is a vertex in the dual. The outer face corresponds to several vertices in the dual, depending on the number of edges which lie on the outer face of the graph. Each edge in the outerplanar graph is an edge in the dual in

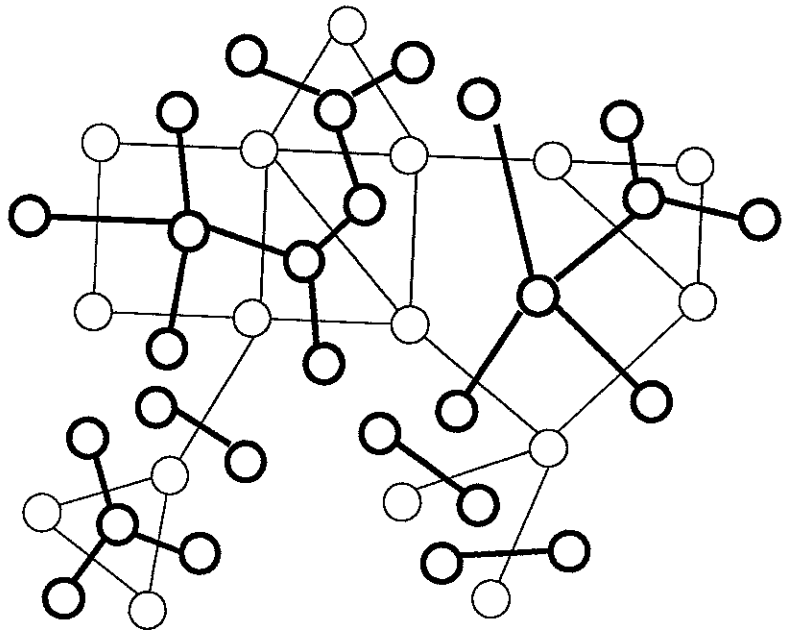

Figure 31: An outerplanar graph (white vertices, black edges) and its dual (black vertices, dotted edges).

the following way: whenever faces in the graph are adjacent (separated by an edge), so are the vertices in the dual. Figure 31 is an example of an outerplanar graph and its dual.

Observe that the dual of an outerplanar graph is simply a collection of trees [FGH]. Also, detecting a maximum set of edge-disjoint triangles in an outerplanar graph is equivalent to finding a maximum set of independent (non-adjacent) degree-3 vertices in the dual. This is in turn done by treating each tree of the dual separately. It therefore suffices to present an algorithm which does exactly that—detects a maximum set of independent degree-3 vertices in a tree. We choose the tree to be rooted at a vertex of degree one (all trees have at least two of these [BM]).

The algorithm (PACK-DUAL), like our previous algorithms, performs recursion on the subtrees of a tree rooted at the current vertex. The call is initially made with the root vertex which can never be in the independent set to begin with (it has degree 1). The solution at each vertex (subtree) is an independent set of vertices and a flag which 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 7-11). If the current vertex is of degree 3 (a possible candidate in the set), the children are checked for membership in their respective solutions (lines 12-13). If neither is in its respective solutions, then v is included in the set (line 14); otherwise, it is not (line 15). Deciding on the latter is appropriate since choosing to add v causes one or both of its children to be removed implying no increase in the current independent set. Thus, the result at every call to PACK-DUAL is indeed optimal.

Algorithm PACK-DUAL. Used in 3-cycle edge-packing for outerplanar graphs.

INPUT: A rooted tree $H = (V_H, E_H)$ (dual of graph) and a vertex v.
OUTPUT: A maximum independent set of degree-3 vertices for the tree rooted at v, and a flag which indicates whether v is in that set.

```
1   IF (c(v) = ∅) THEN
2       RETURN (∅, FALSE);      /* if v is a leaf, return an empty solution */
3   ELSE
4       BEGIN
5           A ← ∅;
6           (w_1, w_2, ..., w_n) ← c(v);
7           FOR i ← 1 TO n DO      /* process the subtrees */
8               BEGIN
9                   (RESULT, USED_i) ← PACK-DUAL(H, w_i);
10                  A ← A ∪ RESULT;
11              END;
12          IF (n = 2) THEN      /* if v is a degree-3 vertex*/
13              IF (NOT (USED_1 OR USED_2)) THEN
14                  RETURN(A ∪ v, TRUE);
15          RETURN(A, FALSE);
16      END;
```

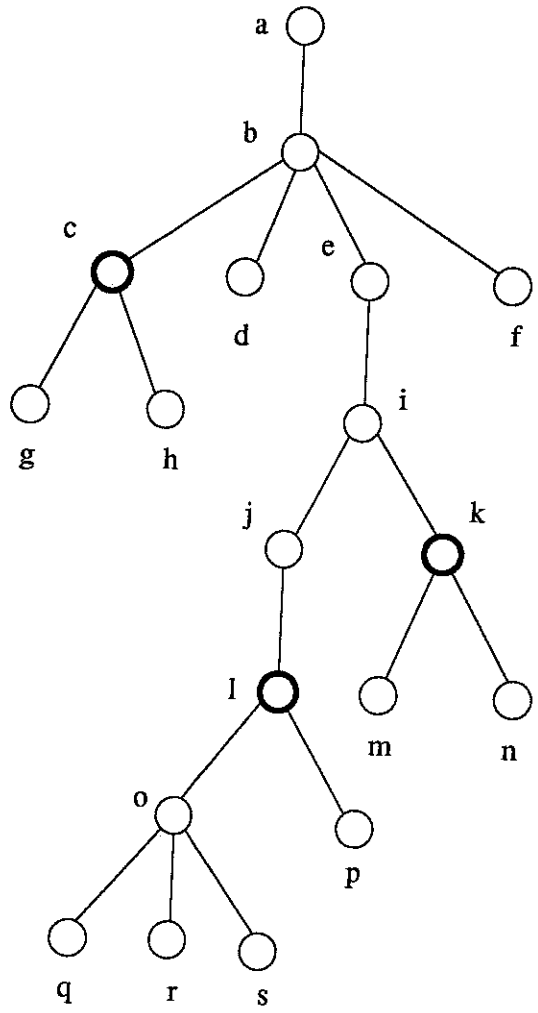

Figure 32: Finding an independent set of degree-3 vertices in a tree.

As an example, consider figure 32 which is, incidentally, the largest component of the dual in our previous example. Here, the independent set of degree-3 vertices is in bold: $\{c, k, l\}$. When PACK-DUAL is called with vertex i, the independent sets for its subtrees are computed; they are $\{l\}$ and $\{k\}$ which, combined, produces $\{k, l\}$. 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 7.1 *$EPack_{3-cycle}(outerplanar)$ is solvable in $O(m_H)$ time.*

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

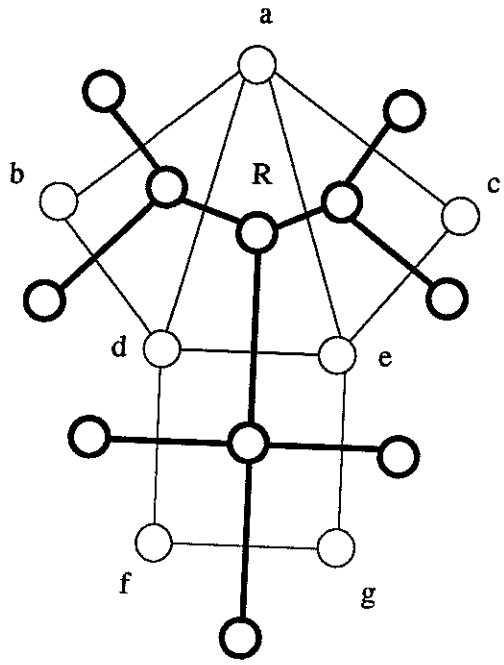

Figure 33: 5-cycles in an outerplanar graph.

The algorithm unfortunately does not extend to $EPack_G$ where G is any non-trivial outerplanar graph other than a 3-cycle. Only 3-cycles can independently correspond to single regions (vertices in the dual) in an outerplanar graph. For 5-cycles, for example, as in figure 33, the guest graphs may occur within multiple regions as shown. Moreover, two 5-cycles may share the same regions even though they are edge-disjoint as in the figure where there are two edge-disjoint 5-cycles ((a, b, d, e, c) and (a, d, f, g, e)) which share a region (R).

A different approach will be necessary for such guest graphs.

8 THE APPROXIMABILITY OF $EPack_G$

In those cases where $EPack_G$ is NP-complete, an approximation algorithm which runs in polynomial time is appropriate. In this section, we propose such an algorithm and show that it always produces at least a fixed fraction of the optimal solution. We also examine $EPack_G$ and its relation to the class Max SNP, a class of maximization problems which have precisely that approximability characteristic.

8.1 A greedy algorithm

We present algorithm PACK-G, a greedy polynomial-time algorithm for finding edge-disjoint copies of a guest graph G in a host graph H.

Algorithm PACK-G. An approximation algorithm for $EPack_G$.

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

```
1   {G_1, G_2, ..., G_n} ← ALL-COPIES-OF-G(H);
        /* identify and obtain all possible copies of G in H */
2   S ← {G_1, G_2, ..., G_n};
3   A ← ∅;
4   WHILE (S ≠ ∅) DO
5       BEGIN
6           G_k ← the copy of G in S such that (|NEIGHBORS(G_k)|) is minimum;
7           A ← A + G_k;
8           FOREACH G_i such that G_i is in NEIGHBORS(G_k) DO
9               S ← S − G_i;
10          S ← S − G_k
11      END;
12  RETURN(A);
```

The first step in the algorithm (line 1) is an exhaustive search of all copies of G in H. The algorithm proceeds by iteratively selecting copies of G and collecting them into a set A (lines 4-11) which is precisely the obtained edge-packing. Edge-disjointedness is enforced since every G selected at an iteration never shares an edge with any other copy of G in the current packing. $NEIGHBORS(G_k)$ refers to the set of other copies of G in S which share an edge with G_k (this set has to be re-calculated at each iteration). This set is used in the exclusion of neighboring copies of a selected copy of G (lines 8-9) and also in the actual choice of a copy of G (line 6).

It is worthwhile to note that the algorithm exhibits $EPack_G$'s resemblance to 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 [J1] proposed 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).

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

Theorem 8.1 *Algorithm PACK-G runs in polynomial time and is at least* $1/m_G$ *optimal.*

Proof: PACK-G runs in polynomial time because G is fixed. There are at most $O(n_H^p)$ copies of G in an arbitrary host graph, where $p = n_G$. In fact, detecting such copies will take $O(n_H^p)$ time since it will require testing isomorphism (to G) for each possible set of p vertices in H. Again, note that G, and therefore p, is fixed. $NEIGHBORS(G_k)$ is obtained by simply maintaining a list of copies of G for every edge—where copies which include an edge are in the list of 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 $O(m_H)$ time, so the entire algorithm runs in $O(n_H^p)$ time.

It remains to show that the algorithm produces at least $1/m_G$ 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 m_G copies of G from the optimal solution are discarded (this is because the choice will have to share at least one edge with each discarded copy). Upon completion of the algorithm, in the worst case, we end up with $1/m_G$ of the optimal. □

Note that the "minimum" criterion used in line 6 is not at all used in this proof which means there is a possibility that R is actually higher than what we claim.

8.2 $EPack_G$ is in Max SNP

There have been several attempts to classify problems in NP in terms of their approximability [PY, PR]. The first attempt was by Papadimitriou and Yannakakis [PY] where define the class Max SNP which consists of maximization versions of decision problems in NP. They proved, using the characterization of Max SNP, that problems in this class can always be approximated within a bounded ratio—that is, there exists an algorithm which produces a fixed fraction of the actual optimal solution. The results in the previous section implies that even a random algorithm for approximating $EPack_G$ (choosing an arbitrary G_k in line 6) is at least $1/m_G$ of the optimal. This leads one to conjecture that $EPack_G$ is in Max SNP. Problems in Max SNP are those which can be logically expressed as follows: $max_S|\{x : p(x, S, H)\}|$, where H is the instance of the problem, S is a structure from H, x is an element of the set which we intend to maximize, and p is a quantifier-free predicate. The expression means that we look for the structure S that produces the set which contains the most number of elements such that each element x satisfies the predicate $p(x, S, H)$. The notation is derived from the logical representation of NP problems first introduced by Fagin [F].

As an example, consider the maximization version of 3-SATISFIABILITY (MAX-3-SAT) where instead of finding the truth assignment which satisfies all the clauses, we look for one which satisfies the most number of clauses. Here, H is the set of variables and clauses, S is the set of variables that we assign a true value to (the truth assignment), x is a clause, and p asserts that at least one of the literals of x is true under the truth

assignment S. Since MAX-3-SAT restricts a clause to having at most three literals, p can easily be written in a finite expression without using quantifiers.

Note that not all problems in NP can be expressed in the way described above. However, $EPack_G$ can be expressed in that way which gives us our next result.

Theorem 8.2 *$EPack_G$ is in Max SNP.*

Proof: It suffices to express $EPack_G$ in the general format defined above. We simply describe the correspondence of H,S,x, and P with our problem. Particularly, H is the input (host) graph, S is a set of sets of m_G edges of H, x represents a set of m_G edges, and $p(x, S, H)$ captures the following assertion: (x induces a graph which is isomorphic to G) and (all edges of x occur exactly once in S). It is not hard to see that such an assertion can be finitely expressed without quantifiers. The first part (isomorphism) is a finite, quantifier-free expression since G is fixed. The second part (membership and edge-disjointedness) involves a test for each of the edges of x (we enforce that no other copy of G in S has that edge), and, since x has the same cardinality as G, this pasrt is also a finite, quantifier-free expression. Finally, it is easily verifiable that the resulting expression is indeed a restatement of $EPack_G$. □

We can go further and show that $EPack_G$ is MAX SNP-complete. This means that all Max SNP problems reduce to it while preserving approximability to a constant ratio—that is, we need to guarantee that the ratio between the fractions of the optimals obtained stays within a constant bound during the reduction. The significance of Max SNP-hardness is that whenever there is an improvement that can be made with approximating a Max SNP-hard problem, then corresponding improvements can be made to all problems in Max SNP as well. This is more precisely discussed in terms of polynomial time approximation schemes (PTAS) by Papadimitriou and Yanakakis. In proving Max SNP-completeness for $EPack_G$, a reduction from Bounded Independent Set (BIS_b), a known Max SNP-complete problem (a problem which is both in Max SNP and Max SNP-hard), is probably most appropriate. (BIS_b is just the Independent Set problem with a constant bound b on the maximum degree of the graph instance). We, in fact, show that $EPack_{k-star}$ is Max SNP-complete in the following result:

Theorem 8.3 *$EPack_{k-star}$ is Max SNP-complete.*

Proof: Given a BIS_b instance with degree bound b, we attach edges to those vertices with degree less than b so that all original vertices will have degree equal to this bound. Figure 34 illustrates how this is done for $b = 4$. Choosing edge-disjoint b-stars in the resulting instance corresponds to choosing the centers of the stars to obtain an independent set for the original instance. More importantly, whenever there is an approximation algorithm for the resulting $EPack_{b-star}$ instance which obtains a fraction of the optimal solution,the same fraction (the constant ratio, in this case, is 1) of the optimal solution can be obtained for the BIS_b instance. This completes the reduction. □

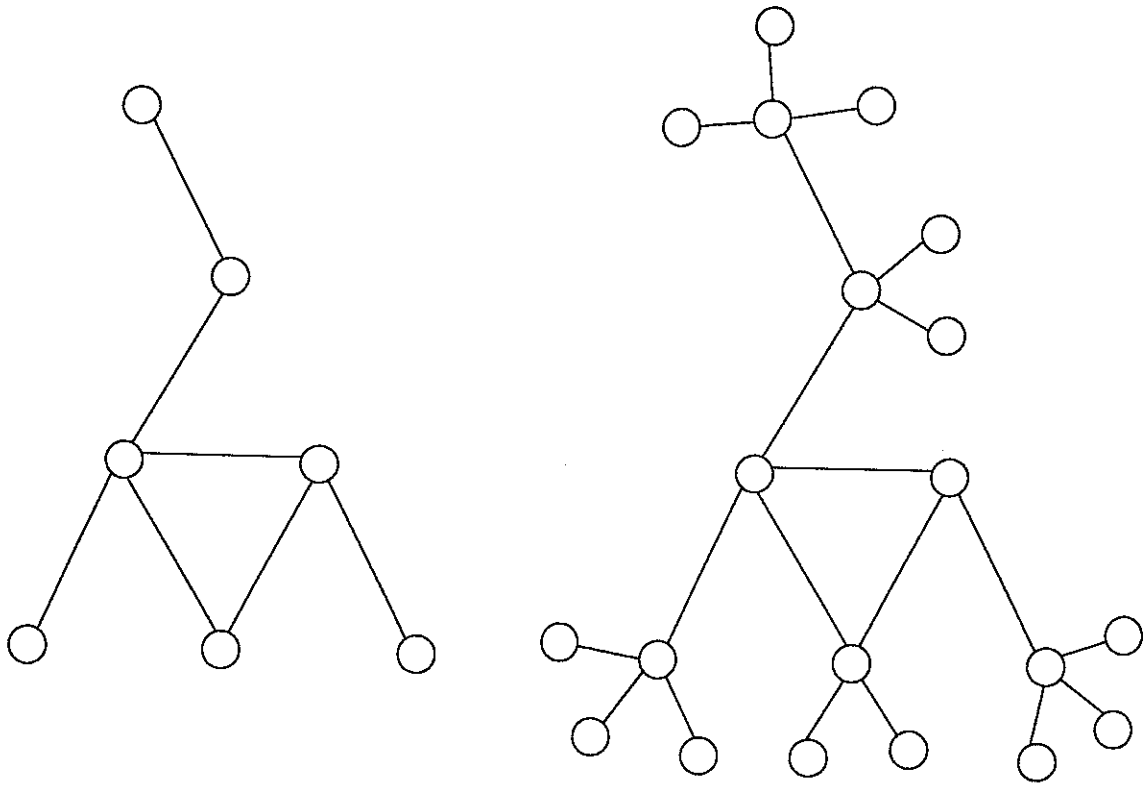

Figure 34: Reduction example from BIS_4 to $EPack_{4-star}$.

There is an analogous situation for k-cycles although there is no direct easy generalization for arbitrary guest graphs.

8.3 Another greedy algorithm

We can investigate other ways to approximate $EPack_G$ by simply changing the criterion used at line 6 of algorithm PACK-G when selecting a copy of G to be included in the maximum set. For instance, instead of selecting the copy which has the least number of neighbors in S, the algorithm can select the copy which shares the least number of its edges to other copies of G in S. This would seem appropriate specially if we consider the proof on why the algorithm performs $1/m_G$ of the actual optimal. Choosing such copies of G avoids the exclusion of too many independent copies of G so that it is reasonable to predict that this new approximation scheme is an actual improvement. Take the example in figure 35 where the guest graph is a 6-cycle. Clearly, there are three 6-cycles that can be extracted from the figure, in particular, those which surround the regions X, Y, and Z. These 6-cycles are indeed chosen first because they share the least number of edges to the other 6-cycles (only 2 edges). However, using our first algorithm will cause a wrong first choice (particularly, the 6-cycle surrounding region C) and will exclude the actual optimal set because although the choice has the least number of neighbors, it shares more edges.

Unfortunately, there are cases where this new algorithm performs poorly, as in figure 36, where choosing the 6-cycle surrounding region C (the one with the least number of edges shared) is incorrect since it will exclude the actual optimal set of 6-cycles (3 of them). Incidentally, our first algorithm finds an optimal solution in this example.

9 CONCLUSIONS AND FURTHER RESEARCH

Maximum G Edge-Packing ($EPack_G$) is NP-complete for most of the non-trivial guest graphs studied. When the guest graph is a 2-path or when the host graph is a tree, the problem is solvable in polynomial time. It remains NP-complete, however, for most of the guest graphs studied (trees and cycles), even if the host is restricted to planar graphs. The straightforward polynomial-time algorithms studied which approximate $EPack_G$ exhibit solutions which are at least $1/m_G$ of actual optimal solutions. The status of $EPack_G$ is summarized in table 1.

Still open for research is how to extend these results to other instances of $EPack_G$, such as for cyclic guests in general. The generalization technique used in proving Maximum G-Matching [KH, BJLSS] may serve as a suitable starting approach. 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. Although this has been shown for the case where the guest graph is a triangle, the reasoning does not immediately extend to other guest graphs.

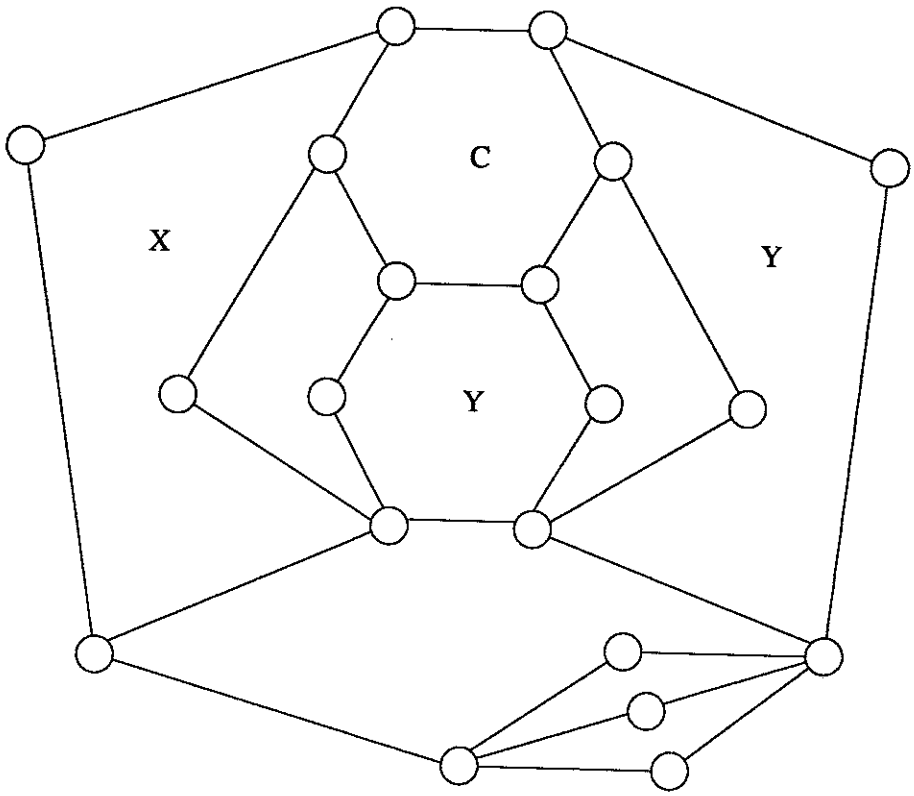

Figure 35: Finding 6-cycles in a graph where the second selection criterion performs better.

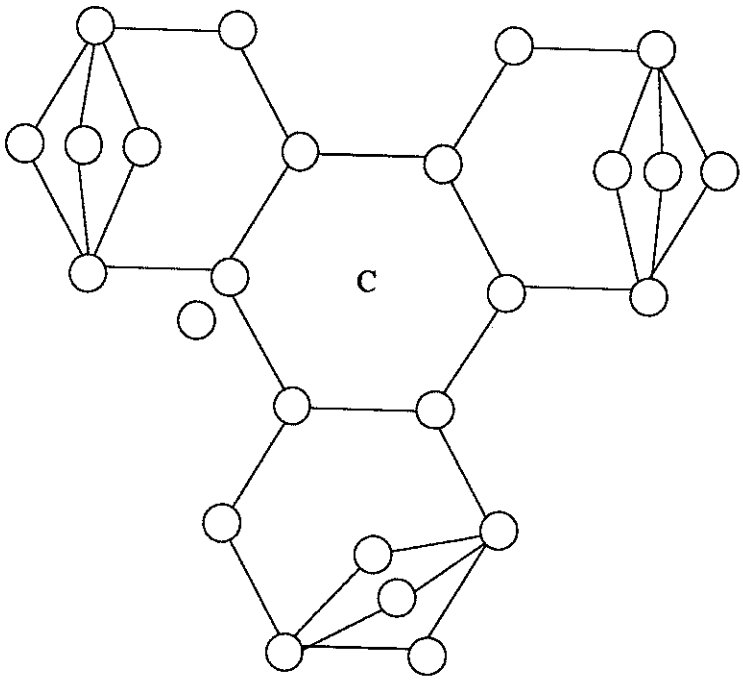

Figure 36: Finding 6-cycles in a graph where the first selection criterion performs better.

Table 1: The current status of $EPack_G$. P: polynomial-time solvable, NPC: NP-complete, ?: conjectured, (): results are incomplete.

guest graphs	arbitrary	planar	outerplanar	tree
2-path	P	P	P	P
star	NPC [DF]	NPC	P?	P
path	NPC [DF]	NPC [DF]	P?	P
arbitrary tree	NPC [C4]	NPC	P?	P
cycle	NPC [H3]	NPC	(P,k=3)	P
arbitrary cyclic	NPC [C4]	NPC?	P?	P

Also, we still have to prove whether $1/m_G$ is indeed a tight bound for the approximation algorithms studied. In addition, determining the effects on this bound when the host graph is restricted to planar graphs, for instance, is certainly helpful.

Although $EPack_G$ is not in Max SNP, we have shown, for some cases, that it is Max SNP-hard. This stems from its resemblance to the Bounded Independent Set problem which has been shown to be both in Max SNP and Max SNP-hard. Approximability-preserving reductions for stars and cycles as guests are straightforward. Reductions for other guest graphs remains to be done.

Another approach is to classify host graphs in a different manner. In this paper, we chose to take the hierarchy of trees, outerplanar graphs and planar graphs. We can investigate other classes within and without this hierarchy such as grid graphs, bipartite graphs, genus-k graphs, etc. Restricting the graphs involved to as many classes as possible clearly aids in completely understanding the complexity our problem. This approach has been used in numerous other graph-theoretic problems [J3].

Disconnected guest graphs is a situation which is also interesting. The NP-completeness results obtained in this paper probably extend to disconnected guest graphs (we first select a non-trivial connected component of G and use that in the reduction). Complications may arise, however, in that 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. This becomes a problem specially when we impose connectedness in the host graph. The polynomial-time algorithms that were achieved in this paper for simple classes of host graphs such as trees do not easily extend to disconnected guests graphs for the same reason.

Another direction we can take is to look for guest graphs of a fixed size (in terms of the number of edges) but are not necessarily isomorphic to a given graph. Similar situations have been studied by Dyer and Frieze [DF]. Of course, we can, on the other hand, restrict the guest graphs to particular types such as paths or cycles but relax the restriction on size. These are slight deviations from our original problem because the guest graph is not fixed in either structure or size but they are clearly related.

References

[B] Baker B. (1983), "Approximation algorithms for NP-complete problems on planar graphs", in 24th Symposium on Foundations of Computer Science.

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

[BM] Bondy J.A. and U.S. Murty (1976), Graph Theory with Applications, North Holland.

[C1] Colbourn C.J. (1987), The Combinatorics of Network Reliability, Oxford U. Press.

[C2] Colbourn C.J. (1988), "Edge-packings of graphs and network reliability", Discrete Mathematics, 72, 49-61.

[C3] Colbourn C.J. (1984), "The complexity of completing partial latin squares", Discrete Applied Mathematics, 8, 25-30.

[C4] Corneil D.G., S. Masuyama and S.L. Hakimi (1991), "Edge-free packings of graphs", submitted to Discrete and Applied Mathematics.

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

[F] Fagin R. (1974), "Generalized first-order spectra and polynomial-time recognizable sets", in Karp R. (ed.), Complexity of Computations, SIAM-AMS Proceedings, 7, 43-73.

[FGH] Fleischner H.J., D.P. Geller and F. Harary (1974), "Outerplanar graphs and weak duals", J. Indian Math. Soc., 38, 215-219.

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

[GT] Goldberg A. and R. Tarjan (1988), "Finding the minimum-cost circulations by canceling negative cycles", in 20th Ann. ACM Symposium on Theory of Computing.

[H1] Hall M. (1986), Combinatorial Theory, John Wiley and Sons, Inc.

[H2] Heath L.S. (1990), "Edge coloring planar graphs with two outerplanar subgraphs", accepted to 2nd Annual ACM-SIAM Symposium on Discrete Algorithms.

[H3] Holyer I. (1981),"The NP-Completeness of some edge-partition problems", SIAM J. Computing 10(4), 713-717.

[J1] Johnson D.S. (1974), "Approximation algorithms for combinatorial problems", J. Computer and System Science, 9, 256-278.

[J2] Johnson D.S. (1982), "The NP-completeness column: an ongoing guide", J. of Algorithms, 3, 182-195.

[J3] Johnson D.S. (1985), "The NP-completeness column: an ongoing guide", J. of Algorithms, 16, 434-451.

[KH] Kirkpatrick D.G. and P. Hell (1978), "On the completeness of a generalized matching problem", in 10th Ann. ACM Symposium on Theory of Computing.

[KS] Klein P. and C. Stein (1990), "A Parallel algorithm for eliminating cycles in undirected graphs", to appear in Information Processing Letters.

[L] Lichtenstein D. (1982), "Planar satisfiability and its uses", SIAM J. Computing, 11, 329-343.

[MS] Mirzaian A. and K. Steiglitz (1981), "A note on the complexity of the star-star concentrator problem", IEEE Transactions on Communications, 39(10), 1549-1552.

[NW] Nijenhuis A. and H.S. Wilf (1978), "Combinatorial Algorithms", 2nd ed., Academic Press.

[PR] Panconesi A. and D. Ranjan (1990), "Quantifiers and approximation", in 22nd Ann. ACM Symposium on Theory of Computing.

[PY] Papadimitriou C.H. and M. Yannakakis (1988), "Optimization, approximation, and complexity classes", in 20th Ann. ACM Symposium on Theory of Computing.

[PL] Peterson P.A. and M.C. Loui (1988), "The general maximum matching algorithm of Micali and Vazirani", Algorithmica, 3, 511-513.

[Y] Yap H.P. (1988), "Packing of graphs—a survey", Discrete Mathematics, 72, 395-404.