A CODING ALGORITHM FOR UNDIRECTED GRAPHS.

Introduction:

This chapter is concerned with the construction of a "code" for undirected graphs. The code which is a string of characters, completely defines the graph so that two graphs have the same code if and only if they are "isomorphic".

The algorithm presented here is based on the work done by Yogesh J. Shah, George I. Davida, and Michael K. McCarthy (19). In section 2.2.1 a counter example is given to show that their (19) algorithm fails to produce the optimum code for certain graphs. A more efficient algorithm for the construction of the code is presented in section 2.2.2. Though the algorithm is based on the earlier work (19), most of the ideas used are different from those presented in the original paper (19). A formal proof of the algorithm is also presented. In section 2.2.3 the coding algorithm is worked out in full for a fairly simple graph.

2.1 Graph Coding Problem and Graph Isomorphism:

Definition 2.1:

Let G be a collection of graphs of certain kind, and let X be a specified set of objects. A "coding procedure" for G is a mapping \( G \rightarrow X \) such that two graphs in G map on to the
same element in X if, and only if, the two are isomorphic. The problem of devising a coding procedure for a given set G is known as "coding problem".

The image of graph g in G under the mapping C is called the "code" of g. An element x of X is called a "valid code" if it is the image of some g in G under C. The set X is usually the set of all strings of some alphabet. We note that there is a one to one correspondence between the isomorphism classes in G and the set of valid codes. The process of obtaining, from a valid code x, a graph whose code is x is called the decoding procedure.

Since two graphs have the same code if, and only if, they are isomorphic, the coding problem is effectively same as the "graph isomorphism problem" (devising an algorithm to test whether two graphs in G are isomorphic or not). But it is not advisable to go for coding procedures, if the problem is just to find out whether two graphs are isomorphic or not. Applying the coding procedure on each graph and then comparing the two codes might be a roundabout method. There are algorithms (18,9,4) available whose input is the pair of graphs to be tested and output is "the two graphs are isomorphic" or "the two graphs are not isomorphic". These algorithms process the two graphs simultaneously and if the two graphs are not isomorphic it may be detected at any stage of the algorithm. Thus if we have a one-shot problem of determining whether two
graphs are isomorphic or not, then algorithms which test this directly are more suitable. On the otherhand, if we are given a new graph and asked to find out whether it is isomorphic to one of the graphs in a list of graphs, then the coding procedure could be more suitable. The list of graphs could be stored in the form of their codes. The code of the new graph is calculated and then we look for a match in the list of graphs. Thus we can replace N (assume there are N graphs in the list) application of the isomorphism program by one application of the coding program and N comparisons to find a match in the list of codes. If the order of complexity of isomorphism program and the coding procedure are same (reasonable assumption) then we save a great deal of time by using the coding procedure. To illustrate this point, let us consider the example given in R.C. Read (16). We are given the task of generating all trees of order p. A method of doing this, that produces no duplicates has been described (12), but is complicated; it is more straightforward to derive these trees from those of order p-1 (which we shall suppose that we have already constructed) as follows. To each of the trees of order p-1 we add, in every possible way, an extra edge, one node of which is already in the tree, while the other is a new node of degree 1. In this way, it is clear that we will get all trees of order p, but there will be duplicates of the same tree. Thus each time we produce a tree, we must search in the list of trees already produced to decide whether to add the currently produced...
tree to the list or not. This is precisely the kind of application in which coding procedure can be advantageously used.

2.2. A Coding Algorithm for Graph Isomorphism

There have been attempts in the past to obtain a coding procedure for a graph (19,16). Our algorithm of finding a code is based on the work done by Yogesh J. Shah, George I. Davida and Michael K. McCarthy (19).

While constructing a coding procedure, we look for suitable graph invariants. An invariant of a graph is a property enjoyed by isomorphic graphs. For example,

(1) number of nodes of degree d (p is arbitrary).
(2) number of edges.
(3) number of circuits of length p (p is arbitrary).
(4) number of components.

are graph invariants. But all these invariants have the defect of being not sufficient to prove isomorphism. Graphs shown in figures 2.1a and 2.1b have the same number of nodes, edges, circuits and components. But still the two are not isomorphic.

![Figure 2.1a](attachment:image.png)

![Figure 2.1b](attachment:image.png)
Hence we must look for an invariant which is more powerful than these.

Theoretically speaking, to find whether two Graphs $G_1$ and $G_2$ of order $p$ are isomorphic or not, one has merely to examine all the $p!$ possible mappings of the vertices of $G_1$ on to those of $G_2$ and see whether there is at least one which preserves adjacency. The same procedure when modified to get a coding algorithm, looks as follows. Obtain all the $p!$ adjacency matrices (by reordering the nodes of the graph) of the given graph. Then select one of these matrices, based on certain criterion, as the code representing the graph. (Note that the above procedure is a coding procedure according to Definition 2.1).

Since the adjacency matrix of an undirected graph is symmetric, one need consider only the upper half of the matrix. Also one can associate with each adjacency matrix a binary number as follows.

The binary number corresponding to the adjacency matrix $A(p \times p)$ is "$a_{12}a_{13}\ldots a_{1p}a_{23}a_{24}\ldots a_{2p}\ldots a_{rr+1}\ldots a_{rp}\ldots a_{p-1\,p}$". We shall denote this as $N_A$.

**Definition 2.2:**

Code of graph $G$ by definition = $\text{Max} \left\{ N_A / A \text{ an adjacency matrix of } G \right\}$

(Among $p!$ binary numbers obtained from the $p!$ adjacency matrices, we choose the largest one as the code of the graph).
2.2.1 A Counter Example:

Yogesh J. Shah, George I. Davida and Michael K. McCarthy (19) have given an algorithm to obtain the above mentioned code (Definition 2.2) without constructing all the p! adjacency matrices. But their algorithm fail to produce the optimum code in (optimum code is an abuse of notation. Code itself means the \( \max \left\{ N_A \mid A \text{ an adjacency matrix of } G \right\} \). But we use it to conform to their (19) terminology.) the case of certain graphs. Here we give a counter example to their algorithm.

The input graph is as shown in figure 2.2. We shall give the different steps and the final code produced by the algorithm.

![Graph C](image)

**Figure 2.2.**
### Step-1: Adjacency Matrix:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

### Minimum Distance Matrix:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

\[ K = 1 \]
Step 2: Flag = 0

\[ A_1 = 1 \]

Step 2: Go to Step 10.

Step 10: \( K = 2 \)

Step 6: Candidates for \( A_2 \)

\[
\begin{array}{c|c}
2 & 1 \\
3 & 1 \\
4 & 1 \\
5 & 1 \\
\end{array}
\]

Step 7: Candidates for \( A_2 \)

\[
\begin{array}{c|c}
2 & 0 \\
3 & 0 \\
4 & 0 \\
5 & 0 \\
\end{array}
\]

Step 8: Candidates for \( A_2 \)

\[
\begin{array}{c|c}
2 & 1 \\
3 & 3 \\
4 & 2 \\
5 & 1 \\
\end{array}
\]

\[ A_2 = 3 \]

Step 2: Go to Step 10.

Step 10: \( K = 3 \)

Step 6: Candidates for \( A_3 \)

\[
\begin{array}{c|c|c}
2 & 1 & 2 \\
4 & 1 & 2 \\
5 & 1 & 2 \\
\end{array}
\]
Step 7: Candidates for $A_3$  

<table>
<thead>
<tr>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

Step 8: Candidates for $A_3$  

<table>
<thead>
<tr>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

$A_3 = 4$

Step 9: Go to step 10.

Step 10: $K = 4$

Now since nodes 2 and 5 are similar, let us continue our example with node 2 as $A_4$ and node 5 as $A_5$.

Step 6: Candidates for $A_6$  

<table>
<thead>
<tr>
<th>B_1</th>
<th>B_2</th>
<th>B_3</th>
<th>B_4</th>
<th>B_5</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Step 7: Candidates for $A_6$  

<table>
<thead>
<tr>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0</td>
</tr>
</tbody>
</table>

Step 8: Candidates for $A_6$  

<table>
<thead>
<tr>
<th>6</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>3</td>
</tr>
</tbody>
</table>

$A_6 = 7$
Step 2: Go to step 10.

Step 10: \( K = 7 \)
\[ A_7 = 6 \]

Step 2: Go to Step 10.

Step 10: \( K = 8 \)
\[ A_8 = 8 \]

Step 2: Go to Step 10.

Step 10: \( K = 9 \)

Step 6: Candidates for \( A_9 \)  
<table>
<thead>
<tr>
<th></th>
<th>( B_1 )</th>
<th>( B_2 )</th>
<th>( B_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>3</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

\[ A_9 = 10 \]

Step 2: Go to Step 10.

Step 10: \( K = 10 \)
\[ A_{10} = 9 \]

Now the adjacency matrix according to the above ordering is,
<table>
<thead>
<tr>
<th></th>
<th>A_1</th>
<th>A_2</th>
<th>A_3</th>
<th>A_4</th>
<th>A_5</th>
<th>A_6</th>
<th>A_7</th>
<th>A_8</th>
<th>A_9</th>
<th>A_10</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A_2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A_3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A_4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A_5</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A_6</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>A_7</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>A_8</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>A_9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A_10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Hence the corresponding code is

"1111000000001100000001000000000000000011001100".

But if we interchange A_9 and A_10,

We will get,

"1111000000001100000001000000000000000011010010" as the code.

This is certainly larger than the code obtained using the algorithm.

2.2.2 Description of the Algorithm:

Now we shall propose an algorithm which results in the optimum code for a graph.

Our method aims at getting a total ordering of the nodes such that the adjacency matrix of the graph when written in this order results in the optimum code.
Definition 2.3: Ordered m-partition.

Let V be the set of nodes of a graph G. Then \( \{ P_1, P_2, P_3, \ldots, P_m \} \) is an ordered m-partition \( P \) of the nodes of \( G \) if

(a) \( P_i \) is a subset of \( V \) for \( 1 \leq i \leq m \).

(b) The \( P_i \)'s are mutually exclusive.

ie \( P_i \cap P_j = \emptyset \) for all \( i, j \in \{ 1, 2, \ldots, m \} \) and \( i \neq j \).

(c) The \( P_i \)'s are exhaustive ie

\[
\bigcup_{i=1}^{m} P_i = V
\]

We shall call \( m \) as the rank of the partition \( P \).

Definition 2.4: The class index of \( P_k \) of an ordered partition \( \{ P_1, P_2, \ldots, P_k, \ldots, P_m \} \) is the subscript \( k \).

Definition 2.5: In an ordered m-partition \( P = \{ P_1, P_2, \ldots, P_m \} \)

\( P_k < P_1 \) if and only if \( k < 1 \).

Thus our task is to find an ordered n-partition (\( n \) is the cardinality of the set of nodes of the graph) which will result in the optimum adjacency matrix. Let us label the node in \( P_1 \) of the above, final, ordered n-partition as \( A_1 \).

We shall first find \( A_1 \) and then the other \( A_i \)'s inductively.

2.2.2.1 Method of finding \( A_1 \)

Definition 2.6 Valency of a node

The valency of a node \( V_i \) of a graph \( G \) is the number of edges in \( G \) which are incident on \( V_i \).
Definition 2.6 a: Neighbour: Node $V_i$ of a graph $G$ is a neighbour of node $V_j$ if there is an edge in $G$ connecting $V_i$ and $V_j$.

(Note: Neighbour relation is symmetric).

Definition 2.6 b: U Number: U number of a pair of nodes $(V_i, V_j)$ is the cardinality of the set of nodes which are adjacent to both $V_i$ and $V_j$ if $A(V_i, V_j) = 1$. i.e $V_i$ is a neighbour of $V_j$. It is zero otherwise.

Definition 2.6 c: MW Number: The MW number of node $V_i$ is the largest of all U numbers of the pair $(V_i, V_j)$ where node $V_j$ is a neighbour of node $V_i$.

Definition 2.6 d: VN Number: Let the set

$$S = \left\{ s \mid U(V_i, s) = MW(V_i) \right\}$$

Now VN number of node $V_i = \max (\text{valency } s / s \in S)$

Lemma 1:

Consider two adjacency matrices $A$ and $A'$ of a graph $G$. Let the first $K$ rows of the two matrices be identical. If the length of the string of 1's following the diagonal in the $(K+1)^{th}$ row of $A$ is greater than that of $A'$, then $N_A$ is greater than $N_{A'}$.

Proof: The result follows directly from the definition of $N_A$.

Lemma 2:

The node corresponding to the first row of the optimum adjacency matrix is a node of maximum valency.
Proof: Let $A$ be the optimum adjacency matrix and $v$ be the node associated with the first row of $A$. Let the valency of $v$ be $d$. Then, the binary number $N_A$ corresponding to $A$.

$$
\sum_{i=1}^{d+1} \binom{n}{2} \cdot 2^{i-1} < \binom{n}{2} \cdot 2^{d+1-1}
$$

Let us assume that there exists a node $v'$ whose valency $d' > d$. Now construct an adjacency matrix $A'$ whose first $(d+1)$ nodes are $v'$ and $d$ of its neighbors. The binary number $N_{A'}$, corresponding to $A'$

$$
\sum_{i=1}^{d+1} \binom{n}{2} \cdot 2^{i-1} \geq \binom{n}{2} \cdot 2^{d'+1-1}
$$

This implies that $N_{A'} > N_A$ contradicting the assumption on $A$.

Q.E.D.

Algorithm I

Step 1: Find the node(s) of maximum valency. If there is a single node $M$ of maximum valency, assign $M$ as $A_1$ and stop.

Else go to Step 2.

Step 2: Let there be $j$ nodes $(M_1, M_2, \ldots, M_j)$ of maximum valency. For each node, find its MW number. If there is a single node $M$ having maximum MW number, assign $M$ as $A_1$ and stop. Else, go to Step 3.

Step 3: Let there be $j'$ nodes $(M_1', M_2', \ldots, M_j')$ having maximum MW number. For each node find its VN number. If there is a
single node with maximum VN number assign it as $A_1$ and stop. Else go to Step 4.

**Step 4:** Let there be $\ell$ nodes $(M_1, M_2, \ldots, M_{\ell})$ having maximum VN number. Then the candidates for $A_1$ are $M_1, M_2, \ldots, M_{\ell}$ stop.

The input to Algorithm I is an adjacency matrix of the given graph $G$. The algorithm either finds $A_1$ or it gives a list of candidates for $A_1$. We can show that $A_1$ is indeed one of the nodes in the list of candidates given by the algorithm.

From definition 2.2 of optimum code, it is evident that we should look for $A_1$ along the following lines.

a. According to Lemma 2 the first row of the optimum adjacency matrix must correspond to a node of maximum valency. This is guaranteed by Step 1.

b. If constraint 'a' is satisfied by more than one node, then we see from Lemma 1 that $A_1$ should be so chosen, such that the string of 1's following the diagonal in the second row of the adjacency matrix is of maximum length. This is guaranteed by Step 2 of the algorithm.

c. If there are more than one nodes competing for $A_1$ satisfying constraints 'a' and 'b' then $A_1$ should be so chosen such that the second row of the adjacency matrix has maximum number of 1's. This is guaranteed by Step 3 of the algorithm.
For each candidate for $A_1$ given by the algorithm, we shall construct an ordered 2-partition as follows.

$$P_1 = (\text{candidate for } A_1)$$
$$P_2 = V - (\text{candidate for } A_1)$$

For the sake of simplicity we shall call the candidate for $A_1$ in each partition as $A_1$ itself. The node qualified by the name $A_1$ is interpreted from the context.

**Definition 2.7: Refinement of an ordered m-partition**

An ordered partition $P' = (P_1', P_2', \ldots, P_r')$ is said to be a refinement of an ordered partition $P = (P_1, P_2, \ldots, P_m)$ if

1) for every $i \ (1, 2, \ldots, r)$
   $$P'_i \subseteq P_j \text{ for some } j \in (1, 2, \ldots, m).$$

2) for every $K, i \in (1, 2, \ldots, r)$
   $$P'_i \subseteq P'_i, \ P'_i \subseteq P'_j \text{ and } i < j \Rightarrow P'_K \subseteq P'_i.$$

$P'$ is said to be a proper refinement of $P$ if the rank of $P'$ is greater than that of $P$. (Note that the rank of $P'$ is never less than that of $P$). In such a case $P'$ is said to be finer than $P$.

**Definition 2.8: Breaker node**

We are given an ordered m-partition $P$ and a node $v$ of a graph $G$. We refine the partition $P$ by splitting each class $P_i$ into two classes (one class could be empty) $P'_i$ and $P''_i$ where
\[ P'_i = (x / x \in P_i \text{ and } x \text{ is a neighbour of } v) \]

\[ P''_i = P'_i - P_i. \]

The resulting new partition is ordered as follows. \( P'_1, P''_1, P'_2, P''_2, \ldots, P'_m, P''_m \). All empty classes should be ignored. We shall call the above refined partition as \( P' \). Here node \( v \) is said to be used as "Breaker node" in refining the partition \( P \).

**Definition 2.9: Position vector:**

We associate an \( m \)-dimensional vector with each node of an ordered \( m \)-partition. The \( k^{\text{th}} \) co-ordinate of the \( m \)-dimensional vector of a node \( v \) is the number of nodes of the \( k^{\text{th}} \) class which are adjacent to \( v \). To be more mathematical, with each node \( v \), we associate \( \bar{v} = (v_1, v_2, \ldots, v_m) \). Where \( v_j \) = no. of neighbours of \( v \) in \( P'_j \). The above-defined vector is called the "Position Vector" of a node of an ordered partition.

We can define a mapping \( M \) from the set of position vectors to the set of integers such that, position vector \( \bar{v}_1 \) is lexicographically greater than position vector \( \bar{v}_2 \) if and only if \( M(\bar{v}_1) > M(\bar{v}_2) \).

**Definition 2.10: Position Number:**

Let \( \bar{v} \) be the position vector of a node \( v \). Then \( M(\bar{v}) \) is called the "position number" of the node \( v \). (Assume that \( M \) is defined).

Now we apply Algorithm II to all the partitions produced by Algorithm I. Along with each partition, we also input a pointer which points to one of the classes of the partition. Initially the pointer points to Class 1.
Algorithm II.

Step 1: Use the member of the class indicated by the pointer as the "Breaker node" to refine the partition. (Note: At this point, the class indicated by the pointer has exactly one member). Advance the pointer by one unit.

Step 2: If each class of the partition has exactly one element, then stop. Else find the position number of each node of the class indicated by the pointer. Let the pointer point to class $i$. If there is a single node $V_j$ of maximum position number then split class $i$ into two classes $P'_i$ and $P''_i$ where

$$P'_i = (V_j) \quad \text{and} \quad P''_i = \text{Class } i - P'_i.$$ 

The resulting partition is $(P_1, P_2, ..., P_{i-1}, P'_i, P''_i, P_{i+1}, ..., P_m)$. Stop. Else go to step 3.

Step 3: Let $M_1, M_2, ..., M_k$ be the nodes of maximum position number. Then construct $K$ partitions, one for each $M_j \in \{M_1, M_2, ..., M_k\}$ as follows. The partition corresponding to $M_j$ is $(P_1, P_2, ..., P_{i-1}, (M_j), P_i - (M_j), P_{i+1}, ..., P_m)$ Stop.

Algorithm II is applied simultaneously to all the partitions. Only those partitions which result in maximum position number in Step 2, are maintained for further processing. Other partitions are ignored.

Each of the output partitions of Algorithm II corresponding to an input partition is either
exit (a):  A partition with each class containing exactly one element  
or
exit (b):  A partition with less than $n$ classes, but the pointer advanced by one unit. If Algorithm II takes exit 'a' then we move to Algorithm III. Else each partition corresponding to exit 'b' is given as input to Algorithm II. Ultimately we will reach exit 'a' since the input graph is finite.

**Algorithm III.**

**Step 1:** Choose any partition determined by Algorithm II. Let $A$ be the adjacency matrix corresponding to this partition.

**Step 2:** Find $N_A$. The code of the graph is $N_A$. Stop.

**Definition 2.11:**

A class $C$ of a partition $P$, having exactly one element is said to be a "fixed class" if in each class of the partition, either all the members are adjacent to the unique member of class $C$ or none of them is adjacent to that member.

**Definition 2.12:**

An ordered $m$-partition of the nodes of a graph $G$ of order $n$ is said to possess the order preserving property if, the partition can be refined to an ordered $n$-partition which results in the optimum code.
Lemma 3: Let $P = (P_1, P_2, \ldots, P_m)$ be an ordered $m$-partition of the nodes of a graph $G$ of order $n$, having the order preserving property. If $K$ is the largest integer such that any class whose index is not greater than $K$ is a fixed class, then, the $(K+1)^{th}$ node of the optimum adjacency matrix must be a member of class $(K+1)$ having the maximum position number in that class.

Proof: The $(K+1)^{th}$ node of the optimum adjacency matrix must be a member of class $(K+1)$ because of the order preserving property of the partition.

Since the first $K$ classes are "fixed classes", the first $K$ rows of all the adjacency matrices corresponding to the $n$-partitions refining $P$ coincide with those of the optimum adjacency matrix i.e. any refinement of $P$ results in a code whose first $K$ segments (the portion of the code contributed by the first $K$ rows) coincide with those of the optimum one.

Now the $(K+1)^{th}$ node must be suitably chosen to maximise the $(K+1)^{th}$ segment. Let $V_{K+1}$ be a member of class $(K+1)$ having the maximum position number in that class. Let $l_i$ be the number of elements in class $i$.

Since the nodes can be reordered only at the class level (only nodes within a class can be reordered) the maximum value for the $(K+1)^{th}$ segment of the code with $v$ as the $(K+1)^{th}$ node is 

$$s_{K+1} s_{K+2} \cdots s_m$$

Where $s_j$ is a bit string of $\bar{v}_j$'s followed by $(l_j - \bar{v}_j)$ 0's and $\bar{v}_j$ is the $j^{th}$ co-ordinate of the position vector of node $v$. 
We now observe that the \((K+1)^{th}\) segment of the code with \(v_{K+1}\) as the \((K+1)^{th}\) node is greater than or equal to the \((K+1)^{th}\) segment of the code with any other node as the \((K+1)^{th}\) node because the position vector of \(v_{K+1}\) is lexicographically greater than or equal to that of any other node.

**Lemma 4:**

If the input to Algorithm II possesses the order preserving property then there exists at least one partition with order preserving property in the list of output partitions produced by Algorithm II.

**Proof:**

Let \(P\) be the input partition and \(K\) be the index of the class indicated by the pointer at entry to Algorithm II. When we enter Step 1 of Algorithm II, the following is true for each class \(C\) whose index is less than \(K\).

In each class of the partition, either all the members are adjacent to class \(C\) or none of them is adjacent to class \(C\). (Class \(C\) is a fixed class)

Hence the first \((K-1)\) rows of all the adjacency matrices corresponding to the \(n\)-partitions refining \(P\) coincide with those of the optimum adjacency matrix i.e. any refinement of the input partition results in a code whose first \((K-1)\) segments (the portion of the code contributed by the first \(K-1\) rows) coincide with those of the optimum one.
Now to maximise the segment of the code, contributed by the $k^{\text{th}}$ row, we are forced to do the following. In each class members which are adjacent to the node corresponding to the $k^{\text{th}}$ row must be preferred to those which are not. This is guaranteed by Step 1. Thus step 1 retains the order preserving property of the input partition.

When we enter step 2, we would have assigned nodes from $A_1$ to $A_k$ where $k+1$ is the index of the class indicated by the pointer. From Lemma 3, we note that $A_{k+1}$ must be a member of Class $k+1$ having the maximum position number in that class. This is guaranteed because we put in the list of candidates for $A_{k+1}$, all the nodes in Class $k+1$ which have the maximum position number. Thus at least one of the output partitions possesses the order preserving property.

Lemma 5:

All the partitions given as input to Algorithm III result in the same adjacency matrix.

Proof:

When we exit Algorithm II with pointer value $K$, we can associate with each preserved partition a partial adjacency matrix whose first $K$ rows correspond to the first $K$ classes of the partition.

(Note that the first $K$ rows do not change with the refinement of the partition).
From the arguments given in the proof of Lemma 4 and because of the fact that we preserve only those partitions with maximum position number, we observe that all these partial adjacency matrices are identical.

When we enter Algorithm III, the pointer value is $n$, the order of the graph. Hence all the partitions given as input to Algorithm III result in the same adjacency matrix.

**Theorem:**

Algorithm I, II and III together result in the optimum code of the graph.

**Proof:**

When we invoke Algorithm II for the first time, one of the input partitions has the order preserving property because $A_1$ is indeed one of the nodes in the list of candidates for $A_1$ produced by Algorithm I. From lemma 4, we see that one of the $n$-partitions produced by Algorithm II is indeed an optimum one. Further on applying lemma 5 we note that all the $n$-partitions produced by Algorithm II are optimal.

Q.E.D.

2.2.3 Example:

Here we shall work out the coding algorithm in full for a reasonably simple graph shown in figure 2.3.
**Algorithm I**

**Step 1:**

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Valency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
</tbody>
</table>

Candidates for $A_1$ are nodes 1 and 9.

**Step 2:**

<table>
<thead>
<tr>
<th>Candidates for $A_1$</th>
<th>MW number</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
</tr>
</tbody>
</table>

**Step 3:**

<table>
<thead>
<tr>
<th>Candidates for $A_1$</th>
<th>VN number</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
</tr>
</tbody>
</table>

Node 9 is assigned as $A_1$

**Algorithm II**

The input partition to Algorithm II is,

(9) (1, 2, 3, 4, 5, 6, 7, 8, 10)

Pointer points to (9)
Step 1:

Breaker node is 9.

The refined partition is

\( (9) \ (6,7,8,10) \ (1,2,3,4,5) \)

Pointer points to \((6,7,8,10)\).

Step 2:

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Position Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>010100</td>
</tr>
<tr>
<td>7</td>
<td>010101</td>
</tr>
<tr>
<td>8</td>
<td>010001</td>
</tr>
<tr>
<td>10</td>
<td>010000</td>
</tr>
</tbody>
</table>

New partition is

\( (9) \ (7) \ (6,8,10) \ (1,2,3,4,5) \)

Step 1:

Breaker node is 7.

The refined partition is

\( (9) \ (7) \ (6) \ (8,10) \ (4) \ (1,2,3,5) \)

Step 2: Node 6 is the only node in the class indicated by the pointer. Hence no refinement.

Step 1: Breaker node is 6.

The new partition is

\( (9) \ (7) \ (6) \ (8,10) \ (4) \ (1,2,3,5) \)

Step 2:

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Position Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0100000001</td>
</tr>
<tr>
<td>10</td>
<td>0100000000</td>
</tr>
</tbody>
</table>
The refined partition is

(9) (7) (6) (8) (10) (4) (1,2,3,5)

**Step 1:** Breaker node is 8

The new partition is

(9) (7) (6) (8) (10) (4) (5) (1,2,3)

**Step 2:** Node 10 is the only node.

Hence no refinement.

**Step 1:** Breaker node is 10.

No refinement.

**Step 2:** Node 4 is the only node.

Hence no refinement.

**Step 1:** Breaker node is 4.

The new partition is

(9) (7) (6) (8) (10) (4) (5) (1) (2,3)

**Step 2:** Node 5 is the only node.

Hence no refinement.

**Step 1:** Breaker node is 5. No refinement.

**Step 2:** Node 1 is the only node. Hence no refinement.

**Step 1:** Breaker node is 1.

No refinement.

**Step 2:**

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Position Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0000000000000000101</td>
</tr>
<tr>
<td>3</td>
<td>0000000000000000101</td>
</tr>
</tbody>
</table>

The output partition of Algorithm II are
Now if we input these two partitions to Algorithm III, we get the optimum adjacency matrix as

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

The code of the given graph is

"1111000000100100000000000000100000000100100111"

One can intuitively feel that in general, the complexity of working of the algorithm grows with the number of edges of the graph. Hence if the number of edges of the given graph of order n is greater than n (n-1)/4, then one can construct the code of the complement of the graph. Also we can attach a special symbol to the code to indicate that the code is that of the complement.
REFERENCES


5. HARRY, F., Graph Theory Addison-Wesley (1969).


***
2.3 CONCLUSION:

The algorithm discussed in this chapter was implemented in IBM 370/155 using PL/I as the source language. The flow chart of the program is given in Appendix I. Some features which are not discussed in this chapter are introduced into the program to reduce the computation time considerably in some special cases.

The graph shown in figure 2.4 was given as test input to the program. This is a carefully "cooked up" graph to cause as much trouble as possible. (It does not mean that this is the worst problem. The author is yet to find a tight upper bound for the time taken by the algorithm and the corresponding graph which results in this maximum time.) The program took about 10 seconds to construct the code for this graph which has an automorphism group of large order. (It should be noted that the number of partitions produced by Algorithm II is generally proportional to the order of the automorphism group of the input graph. Hence the time taken to produce the code, also grows with the order of the automorphism group). Code construction for several other graphs (including those of higher order) with a smaller automorphism group took a much lesser time.
GRAPH e

Figure-2.4.

Though the algorithm proposed in this chapter is for the construction of a code, with a little modification it could also be used for checking directly whether two graphs are "isomorphic" or not.

***

Note: In actual implementation we do some more work to reduce computation:

1. All the isolated vertices are not considered since they come at the end in the ordering of the nodes and also they can be put in any order. Removing isolated nodes and taking complement if necessary can be done immediately while it is not possible any more.

2. When we are in step 3 of algorithm II if the max position number of the node is still that clan can be ordered in an arbitrary way.
The adjacency matrix $A$ of a graph $G$ (and hence $G$) can be uniquely recovered from the string of 0's and 1's obtained by sequential concatenation of the rows of the upper triangle of $A$. This string represents a number in binary form and among all the $n!$ graphs (re-labellings of $G$) obtained by permuting the rows and columns of $A$ the one whose string represents a number of greatest magnitude may be taken as a canonical representative of this isomorphism class of graphs. The string of this graph is called the optimum code of the graphs of this class. It is obvious that two (or more) graphs can be tested for isomorphism by the generation and comparison of such codes. This paper presents an interesting algorithm to generate the optimum code of a graph. Two examples are given, worked out in considerable detail.

Starting with the given adjacency matrix $A$ with nodes $M_1$, $M_2$, ..., $M_n$, the algorithm proceeds to relabel the nodes as $A_1$, $A_2$, ..., $A_n$, sequentially. The following concepts are used: (i) If $i$ and $j$ are adjacent in $G$, the $U$ number of $i$ with respect to $j$ is the number of nodes adjacent to both $i$ and $j$ (ii) $B_d$ number of an unassigned node $j$ is the minimum distance between $j$ and the assigned node $A_d$.

The algorithm is based on the observation that between two candidates for the choice of $A_k$ the one with maximum valency, maximum $U$ number and minimum $B_d$ number(s) should be preferred. In presenting the details, however, the authors appear to have given a wrong stipulation...
(in step 6 of the algorithm) as to the appropriate $B_d$ comparisons. The reviewer feels that this error can be rectified by restricting the appropriate comparisons to only those $B$'s for which $r \leq d \leq k - 1$ where $r$ is the minimum index of $A_i$ such that $d(A_i, A_i) = d(A_i, A_k) - 1$. In fact adjacency, rather than distance seems to play the crucial role.

With step 6 as it is, the algorithm does not always lead to the optimum code as, for example for the graph with initial code 

```
1111000000000000000011000000100000000010011010.
```

Here the algorithm (as it stands) selects mode 10 as $A_9$ and mode 9 as $A_{10}$ whereas 9 as $A_9$ and 10 as $A_{10}$ leads to a higher binary number. The corresponding $B_d$'s are

```

<table>
<thead>
<tr>
<th>$B_1$</th>
<th>$B_2$</th>
<th>$B_3$</th>
<th>$B_4$</th>
<th>$B_5$</th>
<th>$B_6$</th>
<th>$B_7$</th>
<th>$B_8$</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>
```

The authors have erred by observing $B_3$ and this could have been avoided by restricting the relevant $B_d$'s to $B_6$, $B_7$ and $B_8$.

The defect in the algorithm mentioned above was first observed by K.N. Venkataraman in his M.Tech. thesis 'Some Graph Theoretic Algorithms' (Indian Institute of Technology, Madras - 1975) The counter-example given above is due to him. The thesis provides a corrected version of the algorithm using different notation, terminology and concepts.

Sd/-

(K.R. Parthasarathy)