AN ABSTRACT OF THE THESIS OF

Amanda D. Leegard for the degree of Master of Science in Computer Science presented on November 27, 2002.

Title: A Fast Algorithm for Determining the Primitivity of an $n \times n$ Nonnegative Matrix

Abstract approved

Redacted for privacy

Paul Cull

Nonnegative matrices have a myriad of applications in the biological, social, and physical genres. Of particular importance are the primitive matrices. A nonnegative matrix, $M$, is primitive exactly when there is a positive integer, $k$, such that $M^k$ has only positive entries; that is, all the entries in $M^k$ are strictly greater than zero. This method of determining if a matrix is primitive uses matrix multiplication and so would require time $\Omega(n^\alpha)$ where $\alpha > 2.3$ even if fast matrix multiplication were used. Our goal is to find a much faster algorithm. This can be achieved by viewing a nonnegative matrix, $M$, as the adjacency matrix for a graph, $G(M)$. The matrix, $M$, is primitive if and only if $G(M)$ is strongly connected and the greatest common divisor of the cycle lengths in $G(M)$ is 1. We devised an algorithm based in breadth-first search which finds a set of cycle lengths whose
gcd is the same as that of G(M). This algorithm has runtime $O(e)$ where $e$ is the number of nonzero entries in $M$ and therefore equivalent to the number of edges in $G(M)$. A proof is given shown the runtime of $O(n + e)$ along with some empirical evidence that supports this finding.
A Fast Algorithm for Determining the Primitivity of an $n \times n$ Nonnegative Matrix

by
Amanda D. Leegard

A THESIS
submitted to
Oregon State University

in partial fulfillment of
the requirements for the
degree of
Master of Science

Presented November 27, 2002
Commencement June 2003
Master of Science thesis of Amanda D. Leegard presented on November 27, 2002.

APPROVED:

Redacted for privacy

Major Professor, representing Computer Science

Redacted for privacy

Head of the Department of Computer Science

Redacted for privacy

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Redacted for privacy

Amanda D. Leegard, Author
ACKNOWLEDGEMENTS

Thanks to Dr. Cull for all the much needed help.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Introduction</td>
<td>2</td>
</tr>
<tr>
<td>1.1 Problem</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Definitions and Notations</td>
<td>3</td>
</tr>
<tr>
<td>1.3 Background</td>
<td>5</td>
</tr>
<tr>
<td>1.4 Overview</td>
<td>7</td>
</tr>
<tr>
<td>2 Algorithm</td>
<td>7</td>
</tr>
<tr>
<td>2.1 Example</td>
<td>12</td>
</tr>
<tr>
<td>2.2 Proof of Correctness</td>
<td>15</td>
</tr>
<tr>
<td>2.3 Runtime Analysis</td>
<td>22</td>
</tr>
<tr>
<td>3 Experiments</td>
<td>26</td>
</tr>
<tr>
<td>4 Conclusions</td>
<td>32</td>
</tr>
<tr>
<td>4.1 Results</td>
<td>32</td>
</tr>
<tr>
<td>4.2 Future Work</td>
<td>33</td>
</tr>
<tr>
<td>References</td>
<td>34</td>
</tr>
<tr>
<td>Appendix</td>
<td>37</td>
</tr>
<tr>
<td>Appendix: C Implementation of Algorithm</td>
<td>38</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Example adjacency list</td>
<td>4</td>
</tr>
<tr>
<td>1.2</td>
<td>Pseudo-code for computing $M^{n^2-2n+2}$ using $O(n^2)$ matrix multiplications</td>
<td>6</td>
</tr>
<tr>
<td>1.3</td>
<td>Pseudo-code for computing $M^{n^2-2n+2}$ using repeated squares</td>
<td>6</td>
</tr>
<tr>
<td>2.1</td>
<td>Algorithm pseudo-code</td>
<td>11</td>
</tr>
<tr>
<td>2.2</td>
<td>Adjacency Lists</td>
<td>12</td>
</tr>
<tr>
<td>2.3</td>
<td>Example of Backward-Breadth-First search starting at vertex 1, filling in $P$</td>
<td>13</td>
</tr>
<tr>
<td>2.4</td>
<td>Example of Forward-Breadth-First search starting at vertex 1, filling in $S$ and $C$</td>
<td>14</td>
</tr>
<tr>
<td>2.5</td>
<td>Forward breadth-first search pseudo-code with runtimes</td>
<td>23</td>
</tr>
<tr>
<td>3.1</td>
<td>Breakdown of runtime for determining primitivity of random primitive graphs</td>
<td>28</td>
</tr>
<tr>
<td>3.2</td>
<td>Breakdown of runtime for determining primitivity of random non-primitive graphs</td>
<td>30</td>
</tr>
<tr>
<td>3.3</td>
<td>Graph comparing runtime of determining primitivity of primitive and non-primitive matrices</td>
<td>31</td>
</tr>
<tr>
<td>3.4</td>
<td>Graph comparing runtime of computing gcd of primitive and non-primitive matrices</td>
<td>32</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>11</td>
</tr>
</tbody>
</table>

<p>| 2.1  | Algorithm Input | 8  |
| 2.2  | Algorithm Data Structures | 11 |</p>
<table>
<thead>
<tr>
<th>Equation</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 $P[i] = $ number of edges needed to be followed to reach vertex $i$ from starting vertex, $v$, in the backward graph (i.e. the number of edges needed to be followed to reach starting vertex, $v$, from vertex $i$ in the forward graph)</td>
<td>8</td>
</tr>
<tr>
<td>2.2 $S[i] = $ the set of vertices that can be reached from starting vertex, $v$, by following exactly $i$ edges excluding those vertices that can be reached by following less than $i$ edges</td>
<td>9</td>
</tr>
<tr>
<td>2.3 Calculating cycle lengths in graph</td>
<td>10</td>
</tr>
</tbody>
</table>
A Fast Algorithm for Determining the Primitivity of an $n \times n$ Nonnegative Matrix

Amanda D. Leegard
leegard@cs.orst.edu

Department of Computer Science
Oregon State University
Corvallis, OR 97331 U.S.A.
1 Introduction

The notion of primitivity has been attributed to the German mathematician, Ferdinand Georg Frobenius [Mey00, Her54, Min88]. In 1912 he stated that an $n \times n$ nonnegative matrix, $M$, is primitive if and only if there exists a positive integer, $k$, such that every entry in the matrix $M^k$ is strictly greater than zero. The minimum $k$ value for which this holds true is known as the exponent of primitivity or the index of primitivity [BaR97, BeP96, LaT85]. More recent work in the theory of primitive matrices has used graph theory [KOR00, CFR02, Vit75, She95, Lew71, DuM62, HeL66, Lil94, Liu90, Jia85, HoJ85]. This paper applies graph theory to discover a fast algorithm that determines whether or not a matrix is primitive.

1.1 Problem

Let $M$ be a nonnegative $n \times n$ matrix; meaning that, every entry, $M_{i,j}$ where $0 \leq i, j < n$, in $M$ is greater than or equal to zero. There are instances where knowing the primitivity of such matrices can prove to be useful. For example, in dealing with input-output economic models or population age distribution models [Mey00]. Several methods for determining primitivity are widely known [BaR97, CFR02, Mey00]. The goal of this paper is to find a fast algorithm for determining whether or not a nonnegative matrix is primitive.
1.2 Definitions and Notations

The following definitions and notations will be used throughout this paper. A nonnegative matrix, $M$, can be represented by a directed graph. The first step in doing this is converting $M$ to a $(0,1)$-matrix. In a $(0,1)$-matrix all the entries are only zeros and ones. Note that this will have no effect on the primitivity of the matrix since the property of primitivity only depends on the distribution of zero and non-zero entries—the actual values have no affect. To convert a nonnegative matrix to a $(0,1)$-matrix, simply replace all the positive entries with ones. The graph representation of the resulting matrix is defined as follows:

**Definition 1.1:** The forward directed graph associated with an $n \times n$ $(0,1)$-matrix, $M$, consists of a vertex set, $V = \{0, 1, \ldots, n-1\}$, and an edge set, $E$. An edge from $i$ to $j$, represented by $(i, j)$, is in $E$ exactly when $M_{i,j} = 1$. This graph will be denoted by $G(M)$.

**Definition 1.2:** The backward directed graph associated with an $n \times n$ $(0,1)$-matrix, $M$, consists of a vertex set, $V = \{0, 1, \ldots, n-1\}$, and an edge set, $E$. An edge from $i$ to $j$, represented by $(i, j)$, is in $E$ exactly when $M_{j,i} = 1$. This graph will be denoted by $G^b(M)$. 

For this paper it will be assumed that all graphs will be represented as adjacency lists. An adjacency list, $Adj$, consists of $n$ lists, one corresponding to each vertex in the graph, $G$. For each vertex, $v \in G$, $Adj[v]$ is a list that contains all the vertices pointed to by the edges coming out of $v$ [CLR90].

**Example 1.1:** The matrix $M = \begin{bmatrix} 2 & 0 \\ 3 & 1 \end{bmatrix}$ can be converted to a graph by first converting it to a $(0,1)$-matrix. This would result in the matrix $\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$. Now by Definitions 1.1, 1.2 and the above description of an adjacency list, the graph representations for the matrix would be that seen in Figure 1.1. Note that in this paper vertices are indexed using 0 through $n-1$.

![Figure 1.1 Example adjacency list.](image)

With these definitions tying matrices to graphs, it now makes sense to give some definitions relating graph properties and matrix properties.
Definition 1.3: A nonnegative, $n \times n$ matrix, $M$, is irreducible if and only if $G(M)$ is strongly connected [DuM62, HeL66].

Definition 1.4: A directed graph, $G$, is strongly connected if every two vertices are reachable from each other [CLR90].

Theorem 1.1: A nonnegative, $n \times n$ matrix, $M$, is primitive if and only if $M$ is irreducible and the greatest common divisor of the lengths of cycles in the graph $G(M)$ is one [CFR02, DuM62, LiL94, Liu90, Jia85, HoJ85].

Definition 1.5: A path $(v_0, v_1, \ldots, v_k)$ in a directed graph forms a cycle if $v_0 = v_k$ and the length of the cycle is at least one. The length of a cycle is equal to the number of edges in the cycle. If all the vertices in the cycle are distinct, it is said to be a simple cycle [CLR90].

1.3 Background

There are many different methods for determining whether or not a matrix is primitive. It has been widely proven, initially by Wielandt, that the upper bound on the exponent of primitivity is $n^2 - 2n + 2$ [GKP95, HoV58, Lew71, Sch66]. This means that computing $M^{n^2 - 2n + 2}$ is sufficient for determining primitivity. Using this method would require performing $O(n^2)$ matrix multiplications followed
by looking at every entry in the resulting matrix. This would require a runtime of $O(0(n^2)*O(\text{Matrix Multiply}) + O(n^2)) = O(n^2*O(\text{Matrix Multiply}))$. Utilizing repeated squares would reduce the number of matrix multiplications. By using the pseudo-code in Figure 1.3 to compute $M^{n^2-2n+2}$ as opposed to that in Figure 1.2, the runtime can be reduced to $O(\log n * O(\text{Matrix Multiply}))$.

```
power ← M
for i ← 1 to (n-1)^2
    do power ← power * M
```

*Figure 1.2 Pseudo-code for computing $M^{n^2-2n+2}$ using $O(n^2)$ matrix multiplications.*

```
power ← M
i ← 1
while i < (n-1)^2 + 1
    do i ← 2 * i
        power ← power * power
```

*Figure 1.3 Pseudo-code for computing $M^{n^2-2n+2}$ using repeated squares.*

This is due to the fact that the repeated squares method doubles the power of $M$ on each execution of the while loop. Also if $M^k$ is strictly greater than zero then $M^{k+1}$ is also strictly greater than zero, therefore computing a higher power than necessary cannot affect determining whether or not the matrix is primitive.

The standard method for matrix multiplication—row times column—has a runtime of $O(n^3)$. Using subtraction in the performance of this multiplication enabled
Strassen to reduce the runtime to $O(n^{\log_2 7})$ in 1969 [Cu91]. Coppersmith and Winograd further reduced the time required to $O(n^{2.376})$ by means of arithmetic progressions [CoW87]. Currently, this is the fastest known method for performing matrix multiplication; this means that matrix multiply based methods for determining primitivity cannot be sped up anymore at this time.

1.4 Overview

The rest of this paper will focus on an algorithm for determining primitivity based on the theorem that a nonnegative, $n \times n$ matrix, $M$, is primitive if and only if $M$ is irreducible and the greatest common divisor of the lengths of cycles in the graph, $G(M)$, is one. An algorithm implementing this theorem will be outlined followed by a proof of its correctness and its required runtime. Subsequently, a C implementation will be used to obtain experimental results. Conclusions will then be drawn by comparing the theoretical and experimental results.

2 Algorithm

The algorithm discussed in this paper uses Theorem 1.1 to determine whether or not a given $n \times n$ nonnegative matrix, $M$, is primitive. This chapter outlines the algorithm developed followed by a theoretical proof of its correctness and runtime. Table 2.1 gives a brief description of the input required for the algorithm.
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>integer</td>
<td>dimension of $M$ is $n \times n$</td>
</tr>
<tr>
<td>$G(M)$</td>
<td>adjacency list</td>
<td>directed graph associated with $M$, (see Definition 1.1)</td>
</tr>
<tr>
<td>$G^B(M)$</td>
<td>adjacency list</td>
<td>backward directed graph associated with $M$, (see Definition 1.2)</td>
</tr>
</tbody>
</table>

Table 2.1 Algorithm Input

An outline of the following algorithm can be found in Figure 2.1. The algorithm first picks a starting vertex, $v$. This can be an arbitrary vertex but it makes more sense to pick the vertex with the highest out degree. Then a standard breadth-first search\(^1\) starting from $v$ is performed on the graph $G^B(M)$. This is done in order to fill in the array, $P$, which holds the depth at which each vertex is discovered in the search (i.e. $P[w]$ equals the number of edges that would have to be followed in order to reach the vertex $w$ from $v$ in the backward graph or to reach $v$ from $w$ in the forward graph). The numbers stored in $P$ can be calculated by using Equation 2.1.

\[
P[v] = 0 \quad P[i] = P[\text{predecessor}(i)] + 1
\]

*Equation 2.1* $P[i] =$ number of edges needed to be followed to reach vertex $i$ from starting vertex, $v$ in the backward graph (i.e. the number of edges needed to be followed to reach starting vertex, $v$, from vertex $i$ in the forward graph). During a breadth-first search, when scanning the adjacency list of an already discovered vertex $w$ and a new vertex $i$ is found then $\text{predecessor}(i) = w$ [CLR90].

\(^1\)Pseudo-code for a standard breadth-first search can be found in [CLR90].
So, during the breadth-first search on $G^B(M)$, set $P[v] = 0$; then, each time a new vertex is found in the search $P[new\ \text{vertex}] = P[\text{predecessor(new\ \text{vertex})}] + 1$. If a vertex was not found during this search, the graph is not strongly connected by Definition 1.4 and therefore, not primitive. Also, during this search, a check can be made to see if any vertices have edges to themselves. If so, and the graph is also strongly connected, then the matrix is primitive [Sch65]. It is easy to see that this is true. If there is a self-loop then the graph contains a cycle of length one so the gcd of all the cycle lengths will equal one. Next, a breadth-first search is performed on $G(M)$ starting at $v$. This is done to fill in $S$. $S$ is an array, of length $n$, of sets of integers. So, $S[i]$ equals the set of vertices found at depth $i$ (that is, all the vertices that can be reached from $v$ by following exactly $i$ edges) excluding those vertices that can be reached at a smaller depth. The sets in $S$ are represented by Equation 2.2.

\[
\begin{align*}
S[0] &= \{ v \} \\
S[i] &= \{ \text{successors}(S[i-1]) - \text{already reached vertices} \} \\
\text{already reached vertices} &= \bigcup_{k=0}^{i-1} S[k]
\end{align*}
\]

*Equation 2.2* $S[i]$ = the set of vertices that can be reached from starting vertex, $v$, by following exactly $i$ edges excluding those vertices that can be reached by following less than $i$ edges. 

Successors($w$) = the set of vertices pointed to by the edges out of $w$. 
This search is also used to fill in the set $C$. $C$ is the set of cycle lengths found in the graph $G(M)$. The cycle lengths are found by using the values stored in $S$ and $P$ in accordance with Equation 2.3.

\[
\text{for } i \leftarrow 0 \text{ to } n-1 \\
\forall w \in \text{successors}(S[i]) \quad \text{put } (P[w] + i + 1) \text{ into } C
\]

*Equation 2.3* Calculating cycle lengths in graph.

As with the last search, if not all vertices are visited in this search, the graph is not strongly connected and therefore the matrix is not primitive. Otherwise, since both searches visited all the vertices this means that the graph representation of the matrix is strongly connected. Meaning that, it is now necessary to compute the greatest common divisor of $C$. If $\gcd(C) = 1$, then the graph—and therefore corresponding matrix—is primitive. If $\gcd(C) \neq 1$, then both the graph and matrix are not primitive.
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$</td>
<td>array of length $n$, of integers</td>
<td>$P[i] =$ number of edges required to reach vertex $i$ from starting vertex, $v$ in the backward graph, with $P[v] = 0$. (see Equation 2.1)</td>
</tr>
<tr>
<td>$S$</td>
<td>array of length $n$, of integer sets</td>
<td>$S[i] =$ all vertices that can be reached from $v$ by following exactly $i$ edges excluding those vertices found following less edges. (see Equation 2.2)</td>
</tr>
<tr>
<td>$C$</td>
<td>set of integers between 1 and $2n$</td>
<td>A set of integers representing the cycle lengths found in the graph. (see Equation 2.3)</td>
</tr>
<tr>
<td>$g$</td>
<td>positive integer</td>
<td>Greatest common divisor of $C$.</td>
</tr>
</tbody>
</table>

*Table 2.2 Algorithm Data Structures*

**PRIMITIVITY** ($G(M), G^B(M), n$)

1. Pick a starting vertex, $v$
2. $P =$ Backward-Breadth-First Search($G^B(M), v$) (see Equation 2.1)
3. $S =$ Forward-Breadth-First Search($G(M), v$) (see Equation 2.2)
4. if NOT every vertex visited in Backward-Breadth First Search not strongly connected return NOT PRIMITIVE
   
   if NOT every vertex visited in Forward-Breadth First Search not strongly connected return NOT PRIMITIVE
5. fill in $C$ with the "short" cycle lengths in $G(M)$ (see Equation 2.3)
6. $g = \text{gcd}(C)$
7. if $g = 1$
   
   return PRIMITIVE
   
   else
   
   return NOT PRIMITIVE

*Figure 2.1 Algorithm pseudo-code*
2.1 Example

Here is a small example to help understand the aforementioned algorithm. Is the matrix, \( M = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix} \), primitive? To determine the primitivity of \( M \) requires using \( M \)'s associated adjacency lists, which are shown in Figure 2.2.

\[ G(M) = \]

\[ G^B(M) = \]

*Figure 2.2 Adjacency Lists*

Now when the algorithm PRIMITIVITY is called on these graphs, it first picks an arbitrary starting vertex. As stated earlier, it makes sense to pick the vertex with the highest out degree—but it does not matter which one is chosen for correctness, only speed—for this example \( v = 1 \) was arbitrarily chosen. Next, a breadth-first search is performed on \( G^B(M) \) to fill in \( P \). Figure 2.3 shows the steps performed to fill in \( P \).
Figure 2.3  Example of Backward-Breadth-First search starting at vertex 1, filling in $P$. Dashed edges represent edges that have not been followed while full line edges represent edges that have been followed.

Next, $S$ is filled in by performing a breadth-first search on $G(M)$. Note, to speed up the algorithm, $C$ should be filled in while performing the forward breadth-first search. Figure 2.4 shows the execution of this step.
Figure 2.4 Example of Forward-Breadth-First search starting at vertex 1, filling in $S$ and $C$. Dashed lines represent edges that haven't been followed while full line edges represent edges that have been followed.
So, $C = \{2, 3\}$. At this point, the algorithm checks to see if the graph $G(M)$ is strongly connected. In this example both searches reached all the vertices so $G(M)$ is strongly connected. Therefore, the last step of determining the primitivity of $M$ is to calculate the greatest common divisor of $C = \gcd(\{2, 3\})$, which is obviously $1$. This means that the matrix, $M$, is primitive. Note that by matrix multiplication $M^4 \gg 0$ but $M^3$ is not. This means the $M$'s exponent of primitivity is $4$. It can also be noted that Wielandt's upper bound on the exponent of primitivity is $(n-1)^2 + 1 = (3-1)^2 + 1 = 5$ is greater than the actual exponent of primitivity, as it should be.

2.2 Proof of Correctness

A nonnegative $n \times n$ matrix, $M$, is primitive if and only if the corresponding graph, $G(M)$, is strongly connected and $\gcd(\text{cycle lengths in } G(M)) = 1$ by Theorem 1.1. The following section will prove the correctness of the algorithm, based on this theorem and outlined in Figure 2.1. This will be done by providing individual proofs for specific portions of the algorithm.

First, it will be proven that to determine if a graph is strongly connected, it is sufficient to perform a forward and backward breadth-first search and check if both searches reached every vertex in the graph. A depth-first search version of this method has been previously outlined by Karp and Tarjan [KaT80].
Theorem 2.1: If every vertex of a graph, G, is visited in both a forward and backward breadth-first search, starting from the same vertex, then G is strongly connected.

Proof: By definition, a graph is strongly connected if every two vertices are reachable from each other.

(1) Also by definition, a forward breadth-first search systematically explores the edges of G to "discover" every vertex that is reachable from the starting vertex [CLR90]. Therefore, if all vertices are "discovered" in a forward breadth-first search and i is the starting vertex then

\[ v_i \rightarrow \ldots \rightarrow v_k \quad \forall k \in V \]

(2) In contrast, a backward breadth-first search systematically follows edges in G backwards to "discover" every vertex that can reach the starting vertex. Therefore, if all vertices are "discovered" in a backward breadth-first search and i is the starting vertex then

\[ v_i \leftarrow \ldots \leftarrow v_k \quad \forall k \in V \]

Therefore, if every vertex of a graph, G, is visited in both a forward and backward breadth-first search, starting from the same vertex, i, then G is strongly connected, because to get from any vertex, w, to any other vertex, z, can be done as follows:
Next, it needs to be shown that it is not necessary to compute all the cycle lengths in the graph in order to find their greatest common divisor. It would be favorable to only compute the lengths of the simple cycles in the graph, but this problem seems to be computationally hard (the Hamiltonian Circuit problem is a special case of determining if a graph has a simple cycle of a specified length). This is why the outlined algorithm computes the lengths of "short cycles" which, unlike simple cycles, may contain repeated vertices. Below it is shown that the greatest common divisor of the set of cycle lengths computed by the aforementioned algorithm is equivalent to the greatest common divisor of all the cycle lengths present in the graph.

**Definition 2.1:** If $C$ is a set of natural numbers, then the ± closure of $C$ is the smallest set $C^\pm$ which contains $C$ and is closed under addition and subtraction, if $a \in C^\pm$ and $b \in C^\pm$, then $a + b \in C^\pm$ and $|a - b| \in C^\pm$ [CFR02].

**Theorem 2.2:** Let $g$ be the greatest common divisor of a set, $C$, of natural numbers, then $g$ is also the greatest common divisor of the set $C^\pm$. Also, $C^\pm = g\mathbb{N}$, that is, $C^\pm$ consists of all multiples of $g$. 
Proof: This can be shown by assigning all the elements in $C^c$ a type where the type number indicates the number of operations (i.e. + or -) necessary to create the element from the elements in $C$. Any element in $C$ is assigned the type $T_0$. Any other element, $x$, in $C^c$ has type $T_K$ if $x = a + b$ or $x = |a - b|$ and $\max(\text{type}(a), \text{type}(b)) = T_{K-1}$. For notational purposes, let $\{T_K\}$ represent the set of all elements that have type $T_K$. Now it is easy to show by induction that if $g = \gcd(C)$ then $g = \gcd(C^c)$.

Inductive Hypothesis: If $g = \gcd(\{T_0\})$ then $g = \gcd(\bigcup_{i=0}^{K} \{T_i\})$. (i.e. If $g = \gcd(C)$ then $g = \gcd(C^c)$).

Base Case: $K = 1$.

By definition of the types, all the elements of type, $T_1$, must equal either $a + b$ or $|a - b|$ where $a, b \in \{T_0\}$. Since we are given that $g = \gcd(\{T_0\})$, by the following basic gcd properties, $g = \gcd(\{T_0\} \cup \{T_1\})$.

1. $\gcd(a, b) = \gcd(a, b+ma) \forall m \in \mathbb{Z}$ (just let $m = 1$)
2. $\gcd(a, b) = \gcd(b, |a - b|)$

Inductive Step: Assume that if $g = \gcd(\{T_0\})$ then $g = \gcd(\bigcup_{i=0}^{K} \{T_i\})$. Show that if $g = \gcd(\{T_0\})$ then $g = \gcd(\bigcup_{i=0}^{K+1} \{T_i\})$. 


\[ \gcd(\bigcup_{i=0}^{K+1} \{T_i\}) = \gcd( \gcd(\bigcup_{i=0}^{K} \{T_i\}), \{T_{K+1}\}) \]

\[ = \gcd( g, \{T_{K+1}\} ) \quad \text{by inductive hypothesis} \]

\[ = g \]

Since \( g \) divides all elements of type \( T_{K+1} \),

**Proof:** Since \( g \) divides all elements in \( \bigcup_{i=0}^{K} \{T_i\} \), by inductive hypothesis, \( g \) also divides all elements of type \( T_{K+1} \), because given two elements, \( a \in \{T_K\} \) and \( b \in \bigcup_{i=0}^{K} \{T_i\} \), this means that all elements of type \( T_{K+1} \), equal \( a+b \) or \( |a-b| \) by definition of the types, and since \( g \) divides \( a \) and \( g \) divides \( b \) by the two aforementioned gcd properties, \( g \) divides \( a+b \) and \( g \) divides \( |a-b| \).

and the largest number that can divide \( g \) is \( g \); this means that

\[ \gcd( g, \{T_{K+1}\} ) = g. \]

This means that \( g = \gcd(C^\pm) \). All that is left to be proved is the second part of the statement (\( C^\pm = gN \)). This can be done by showing that \( C^\pm = \{0g, 1g, 2g, 3g, \ldots\} \). Zero is obviously in \( C^\pm \) by closure under subtraction. For any \( a \in C^\pm \), by closure under subtraction \( |a-a| = 0 \) must also be in \( C^\pm \). The greatest common divisor of \( C^\pm \), \( g \), is in \( C^\pm \) by the following algorithm for computing the gcd and the fact that \( C^\pm \) is closed under subtraction.
This means that the greatest common divisor of $C^\pm$ is either a number that was already in $C^\pm$ or a number created by subtraction, and since $C^\pm$ is closed under subtraction, $g$ is in $C^\pm$. Now all the multiples of $g$ are in $C^\pm$ by closure under addition since multiplication can be performed by addition (e.g. $3g = g + g + g$). Finally, all elements in $C^\pm$ are multiples of $g$, since $g$ is the greatest common divisor of $C^\pm$.

So now, if it can be shown that all cycle lengths are in $C^\pm$, constructing $C^\pm$ will not be necessary, but rather it will be sufficient to calculate the gcd of $C$, which has been proven equivalent to that of $C^\pm$.

**Lemma 2.1:** If $C$ is the set of numbers found by the above algorithm, then the length of every cycle in $G$ is in $C^\pm$.

**Proof:**

$P[v_i] =$ the depth at which $v_i$ is first encountered in the backward breadth-first search

$Q[v_i] =$ the depth at which $v_i$ is first encountered in the forward breadth-first search
Consider a cycle $v_1 \rightarrow v_2 \rightarrow \ldots v_L \rightarrow v_L$ of length $L$. Since there is an edge from $v_i \rightarrow v_{i+1}$, when $v_i$ is found, $v_{i+1}$ is encountered in the next step. So, both

\[ P[v_i] + Q[v_i] \quad \text{and} \quad P[v_{i+1}] + Q[v_i] + 1 \]

are put into $C$. So, since $C^\times$ is closed under addition and subtraction, the following is in $C^\times$:

\[
\sum_{i=1}^{L} (P[v_{i+1}] + Q[v_i] + 1) - \sum_{i=1}^{L} (P[v_i] + Q[v_i])
\]

\[
= \sum_{i=1}^{L} P[v_{i+1}] + \sum_{i=1}^{L} Q[v_i] + \sum_{i=1}^{L} 1 - \sum_{i=1}^{L} P[v_i] - \sum_{i=1}^{L} Q[v_i]
\]

\[
= \sum_{i=1}^{L} P[v_{i+1}] + \sum_{i=1}^{L} 1 - \sum_{i=1}^{L} P[v_i]
\]

\[
= \sum_{i=1}^{L} P[v_{i+1}] + L - \sum_{i=1}^{L} P[v_i]
\]

\[
= \sum_{i=2}^{L} P[v_i] + P[v_{L+1}] + L - \sum_{i=1}^{L} P[v_i]
\]

(Here the subscripts are taken mod $L$ so $v_{L+1}$ is simply $v_1$)

\[
= \sum_{i=2}^{L} P[v_i] + P[v_1] + L - \sum_{i=1}^{L} P[v_i]
\]

\[
= \sum_{i=1}^{L} P[v_i] + L - \sum_{i=1}^{L} P[v_i]
\]

\[
= L
\]
Thus $L \in C^\pm$. (Notice that there is no assumption that the vertices are distinct. So this argument holds for both simple and non-simple cycles.)

**Theorem 2.3:** If the graph $G(M)$ is strongly connected and $C$ is the set of numbers found by the primitivity algorithm, then $\gcd(C)$ is the greatest common divisor of the cycle lengths present in the graph. More precisely, if $G(M)$ is strongly connected, then it is primitive if and only if $\gcd(C) = 1$.

**Proof:** By Lemma 2.1, $C^\pm$ contains all the cycle lengths in the graph. Also, by Theorem 2.2, $\gcd(C) = \gcd(C^\pm)$. Therefore, $\gcd(C)$ is the greatest common divisor of the cycle lengths in the graph. So, by Definitions 1.3 and 1.4, a graph is primitive if and only if it is strongly connected (i.e. irreducible) and the gcd of its cycle lengths is 1.

### 2.3 Runtime Analysis

Now that this algorithm has been proven correct, the next step is to look at its runtime and compare it with the previously known algorithms. The following is a breakdown of the runtime. The step numbers correspond to those in Figure 2.1 of the code. The variable $n$ will refer to the number of vertices in the graph, and the variable $e$ will refer to the number of edges, which is upper-bounded by $n^2$. 
Step 1 is just picking a starting vertex. This can be done in constant time if just an arbitrary vertex is picked or $O(n)$ time if the vertex with the largest number of outgoing edges is picked. Step 2 performs a standard breadth-first search on the backwards representation of the graph and therefore has a runtime of $O(n + e)$ [CLR90]. Step 3 performs the forward breadth-first search, which is used to fill in the sets $S$ and $C$. This can be done with the following pseudo code.

\[
\begin{array}{|l|l|}
\hline
\text{FwdBFS}(G(M), \text{startVertex}) & \text{Runtime} \\
\hline
\text{for } i \leftarrow 0 \text{ to max short cycle length } (2n) & O(n) \\
& \text{do } \text{tempC}[i] \leftarrow 0 \\
\hline
\text{for each vertex } u \in \text{VertexSet( } G(M) ) & O(n) \\
& \text{do } \text{color}[u] \leftarrow \text{WHITE} \\
\hline
S[0] \leftarrow \{ \text{startVertex} \} & O(1) \\
\text{color[startVertex]} \leftarrow \text{BLACK} \\
currentDepth \leftarrow 0 & \\
\hline
\text{while } S[\text{currentDepth}] \neq \{ \} & O(e) \\
& \text{do for each vertex } v \in S[\text{currentDepth}] \\
& \text{do for each vertex } w \in \text{AdjList}[v] \\
& \text{do if } \text{color}[w] = \text{WHITE} \\
& \text{then } w \in S[\text{currentDepth} + 1] \\
& \text{color}[w] \leftarrow \text{BLACK} \\
& \text{tempC}[P[w] + \text{currentDepth} + 1] \leftarrow 1 \\
& \text{currentDepth} \leftarrow \text{currentDepth} + 1 \\
\hline
j \leftarrow 0 & O(n) \\
\text{for } i \leftarrow 1 \text{ to max short cycle length } (2n) & \\
& \text{if } \text{tempC}[i] = 1 \\
& \text{then } C[i] \leftarrow i \\
& j \leftarrow j + 1 \\
\hline
\text{return } S \text{ and } C & O(1) \\
\end{array}
\]

*Figure 2.5* Forward breadth-first search pseudo-code with runtimes.

This code fills in $C$ as it runs and therefore also completes Step 5 in the process. A breakdown of the runtime of FwdBFS goes as follows. The time for all the initialization performed before the while statement is $O(n)$. The time for the while
loop is a little more complicated. In the worst case, the code is going to put every vertex into one of the sets in $S$ exactly once, because if a vertex has already been in one of $S$'s sets then it would have been marked BLACK and vertices are only added to $S$ if they are marked WHITE. This means in the worst case every vertex will have all its edges followed once and each time one of the edges is followed, constant time operations are performed. This provides a runtime of $O(e)$ for the while loop. Now the values need to be pulled out of tempC—ignoring zero since it will have no effect—and put into set $C$, which requires $O(n)$ time. So the total runtime for this section is $O(n + e)$.

Step 4 checks to see if the graph is strongly connected. Since both a backward and forward breadth-first search has already been performed, all that is required is to check that every vertex was visited in both searches. Initializing $P$ to infinity before the backward breadth-first search can be used to do this. If after the search, any of the values are still infinity, the corresponding vertices were not reached and therefore the graph is not strongly connected. This has a runtime of $O(n)$. Now the color array from the forward breadth-first search needs to be checked; if any of the vertices are still WHITE they were not reached and therefore the graph is not strongly connected. Again this has a runtime of $O(n)$.

Step 6 computes the gcd of $C$. (Due to the function for adding numbers to $C$, the numbers in $C$ range from 1 to $2n-1$ meaning that the length of $C$ is at most $2n-1$).
Therefore, computing the \( \text{gcd} \) of \( C \) can be done in \( O(n) \) time, as can be seen by the following outlined algorithm.

To find the \( \text{gcd} \) of \( k \) unique numbers which are all between 0 and \( 2n \):

1. **Obvious**: Given \( x_1, \ldots, x_k \), where \( x_1 < x_2 < \ldots < x_k \), compute \( \text{gcd}(x_{k-1}, x_k) \) by Euclid's algorithm in time \( O(\log n) \). Then compute \( \text{gcd}(x_{k-2}, \text{gcd}(x_{k-1}, x_k)) \). So, using \( k \) \( \text{gcd} \) calculations each taking \( O(\log n) \) time suffices for computing the \( \text{gcd} \). This method has an overall runtime of \( O(k \log n) \) which is equivalent to \( O(n \log n) \) since \( k \) is upper bounded by \( 2n \).

2. **Faster**: If \( k \) is "small" use the above method. If \( k \) is "big" then the \( \text{gcd} \) is 1 because if \( k > n \), then there is an \( i \) so that \( 1 + x_i = x_{i+1} \). This is because out of a range of \( 2n \) consecutive numbers, \( n \) of them are odd and \( n \) of them are even. So, if \( k > n \) then there must be both odd and even numbers—meaning \( \text{gcd} \) equals 1. In fact, if \( k > \frac{2n}{\log n} \), then there is an \( i \) so that \( |x_{i+1} - x_i| \leq \log n \). This is because to make the smallest difference between two consecutive numbers as big as possible the numbers should be evenly distributed (i.e. all consecutive numbers separated by \( \log n \)). Let this smallest consecutive difference be \( d \), and compute \( x_i \mod d \) for each \( x_i \). All of these numbers will be at most \( \log n \), and their \( \text{gcd} \).
can be computed in $O(\log^2 n)$. The pre-processing phase takes $O(n)$. So, overall the "big" case takes $O(n)$.

In the small case $k \leq \frac{2n}{\log n}$; this means that $O(k \log n)$ will equal $O(n)$. So, the time required to compute gcd is $O(n)$. Since Step 7 requires constant time, the overall runtime of this algorithm for determining primitivity is $O(n + e)$.

3 Experiments

The above sections have proved that the primitivity of a matrix can be determined in $O(n + e)$ time. This will be further enforced by the following experiments. A C implementation of the algorithm has been used for this analysis [see Appendix for code].

Here is an explanation of the experiment that was performed. First, the number of vertices and edges are set. Then, for those numbers a primitive graph is randomly created. This is done by:

- Creating a super-loop – That is, $v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_{\text{last}} \rightarrow v_0$. This is done to force the graph to be strongly connected.
- Randomly pick a starting vertex, $i$, and an ending vertex, $j$. If there is not an edge from $i$ to $j$, create one. Repeat this until there are $e$ edges in the graph.
Note that this just creates a random strongly connected graph; it does not guarantee primitivity. Primitivity can be forced by creating a self-loop. For this experiment, this was not done. Instead, every graph was checked to see if it was primitive—it worked out that all the random graphs created were primitive.

Then the backwards version of the graph is created. The test for primitivity function is now called 500 times in a row. This process of creating the graph and then checking whether or not it’s primitive is repeated 100 times. So for all the graphs from this point on all the points represent the average of 100 different times from different graphs to perform the specified task 500 consecutive times. This means that to find the average time to perform the task once, simply divide the points by 500. The following four graphs are used in analyzing the runtime of determining primitivity when the graphs used are primitive. The first graph shows that the runtime for determining primitivity does in fact follow the order of $O(e)$. The three graphs following it are a breakdown of the contributing factors to the overall runtime. From them it can be seen that the two breath-first searches are where all the time is being used. The gcd seems to run in constant time, as do all the remaining computations.
Figure 3.1 Breakdown of runtime for determining primitivity of random primitive graphs
For the same vertex and edges numbers specified, the process is repeated exactly except that the random graphs created are not primitive. The non-primitive graphs are created by

- Create a super-loop.
- Randomly pick a starting vertex, $i$, and an ending vertex, $j$. If there is not an edge from $i$ to $j$, and $|i-j|$ modulo 2 equals one, create an edge from $i$ to $j$. Repeat this until there are $e$ edges in the graph. Note, since the this method is checking that only cycles that are divisible by 2 are created, the number of vertices in the graph must be even so that the super loop is also divisible by 2.

For the following graphs each point is determined by averaging the 100 different runs that had the same vertex and edge numbers. To help illustrate what parts of the code are having the most effect on the overall runtime, runtimes of specific parts of the code were also recorded—again the time to run 500 times. These graphs follow the same runtime breakdown as that for the primitive graphs. The gcd again seems to actually run in constant time.
Figure 3.2 Breakdown of runtime for determining primitivity of random non-primitive graphs
The following graph shows the comparison of the runtime for primitive and non-primitive graphs. Showing non-primitivity seems to be slower than showing primitivity by a constant factor.

Figure 3.3 Graph comparing runtime of determining primitivity of primitive and non-primitive matrices.

The reason for this can be seen in the next graph. Though performing the gcd seems to take a constant amount of time for both primitive and non-primitive graphs—a very minute constant—the non-primitive graph's constant is a little bigger.
Figure 3.4 Graph comparing runtime of computing gcd of primitive and non-primitive matrices.

Therefore, according to the experiment performed, the runtime for determining primitivity is $O(e)$.

4 Conclusions

4.1 Results

Several methods for determining primitivity are widely known. This paper focused on an algorithm for determining primitivity based on the theorem that a nonnegative, $n \times n$ matrix, $M$, is primitive if and only if $M$ is irreducible and the
greatest common divisor of the lengths of cycles in the graph, $G(M)$, is one. The algorithm depended on using the speed given by breadth-first search. An algorithm implementing this theorem was outlined followed by a proof of its correctness. It was proven to run in $O(n + e)$ time. An empirical study of a C implementation of this algorithm led to the conclusion that calculating the gcd of a set in this case does not require its theoretical time of $O(n)$ but rather ran in constant time. Either way the overall runtime is expected to be linear in terms of $e$.

4.2 Future Work

Future work to be done in this area could include further speeding up the process of determining whether or not a matrix is primitive. This could include speeding up the original method by speeding up matrix multiplication. More likely though, any speed up will require working with the algorithm outlined in this paper or developing a new one. Although, since every edge has to be looked at, $\Omega(e)$ is a lower bound for the problem, so the increase in speed could only be by a constant factor. It is possible that calculating the gcd may be sped up due to the fact that determining primitivity does not require the actual gcd of the set, rather it only needs to find if the gcd is one or not one.
References


**Appendix: C Implementation of Algorithm**

```c
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <string.h>

#define n 1000
#define e 60000 //min = n, max = n*n
               //for non primitive min = n, max = (n*n)/2

#define lenC 2*n //found from Dr Cull's paper
#define WHITE -1
#define BLACK -2
#define INFINITY -3
```

void makePrimitiveGraphs();
void makeNonPrimitiveGraphs();
void makeBackwardGraph();
int notAlreadyEdge(long int from, long int to);

long int Primitivity();
int BkwdBFS(long int startVertex);
void FwdBFS(long int startVertex);
long int gcdSet(long int * A, long int len);
long int gcd(long int a, long int b);
long int sorter[2*n]; //help with gcd

long int P[n];       //depth each vertex found at in a bkwdBFS
long int setC[lenC]; //actual cycle numbers
long int C[lenC];    //set of cycle lengths, represented by
                     //index with a one
long int S[n][n];   //S[i][j] for all j >=0 and < lenS[i] =
                     //vertex found at depth i from start with
                     //fwd BFS

long int lenS[n+1];
long int color[n];    //used to see if a vertex has been visited
                      //or not

long int Forward[n][n]; //forward adjacency list
long int Backward[n][n]; //backward adjacency list
long int lenF[n];       //number edges from vertex (index) in
                        //forward list
```
long int lenB[n];  //number edges from vertex (index) in //backward list

clock_t startBFS, startGCD, stopBFS, stopGCD;
clock_t sumBFS, sumGCD;
double timeBFS, timeGCD, avetime, aveBFS, aveGCD, aveLeft;

void main()
{
    FILE *fp;
    long int k;
    int j, i;
    clock_t start, stop;
    double timeR;

    srand((unsigned) time(NULL));

    fp = fopen("output.txt", "w");
    fprintf(fp,"n = %d\ne = %d\n", n, e);

    avetime = 0; aveBFS = 0; aveGCD = 0; aveLeft = 0;

    fprintf(fp, "time,BFS, GCD, timeLeft\n");
    for(j=0; j<100; j++)
    {
        makePrimitiveGraphs();

        sumBFS = 0; sumGCD = 0;
        start = clock();
        for(i=0; i<500; i++)
            k = Primitivity();
        stop = clock();

        timeR = (((double) (stop-start))/CLOCKS_PER_SEC);
        timeBFS = (((double) (sumBFS))/CLOCKS_PER_SEC);
        timeGCD = (((double) (sumGCD))/CLOCKS_PER_SEC);

        avetime += timeR;
        aveBFS += timeBFS;
        aveGCD += timeGCD;
        aveLeft += (timeR - timeBFS - timeGCD);

        fprintf(fp, "%.1f,%.1f,%.1f,%.1f,", timeR, timeBFS, timeGCD, timeR - timeBFS - timeGCD);

        switch(k)
        {
        case 1:
            fprintf(fp,"Primitive: gcd = 1\n");
            break;
        }
case 0:
    fprintf(fp,"Not Primitive: not strongly connected\n");
    break;
default:
    fprintf(fp,"Not Primitive: gcd = %d\n",k);
    break;
}
printf("primitive graph %d done.\n",j);

fprintf(fp, "\n%d,%f,%d,%f,%d,%f,%d,%f\n", e,
    avetime/100.0, e, aveBFS/100.0, e,
    aveGCD/100.0, e, aveLeft/100.0);

fprintf(fp, "\n\n\n\n")

avetime = 0; aveBFS = 0; aveGCD = 0; aveLeft = 0;
fprintf(fp, "time,BFS,GCD,timeLeft\n");
for(j=0; j<100; j++)
{
    makeNonPrimitiveGraphs();

    sumGCD = 0; sumBFS = 0;
    start = clock();
    for(i=0; i<500; i++)
        k = Prirnitivity();
    stop = clock();

    timeR = (((double) (stop-start) )/CLOCKS_PER_SEC);
    timeBFS = (((double) (sumBFS) ) /CLOCKS_PER_SEC);
    timeGCD = (((double) (sumGCD) ) /CLOCKS_PER_SEC);

    avetime += timeR;
    aveBFS += timeBFS;
    aveGCD += timeGCD;
    aveLeft += (timeR - timeBFS - timeGCD);

    fprintf(fp, "%f,%f,%f,%f," , timeR, timeBFS, timeGCD,
    timeR - timeBFS - timeGCD);

switch(k)
{
    case 1:
        fprintf(fp,"Primitive: gcd = 1\n");
        break;
    case 0:
        fprintf(fp,"Not Primitive: not strongly connected\n");
        break;
default:
    fprintf(fp,"Not Primitive: gcd = %d\n",k);
    break;
}
printf("non primitive graph %d done.\n",j);
}
fprintf(fp, "\n%d,%f,%d,%f,%d,%f,%d,%f\n", e, avetime/100.0, e, aveBFS/100.0, e, aveGCD/100.0, e, aveLeft/100.0);
fclose(fp);

//number of vertices must be even for this to work
void makeNonPrimitiveGraphs()
{
    long int edgeCount = 0;
    long int from, to, i;

    //force superloop, initialize len's
    for(i=0; i<n; i++)
    {
        Forward[i][0] = (i+1) % n;
        lenF[i] = 1;
        lenB[i] = 0;
        edgeCount++;
    }

    while(edgeCount < e)
    {
        from = rand() % n;
        to = rand() % n;

        if(abs(to-from)%2==1 && notAlreadyEdge(from, to))
        {
            Forward[from][ lenF[from++] ] = to;
            edgeCount++;
        }
    }

    makeBackwardsGraph();
}

void makePrimitiveGraphs()
{
    long int to, from, i, edgeCount = 0;

    //force superloop, initialize len's
    for(i=0; i<n; i++)
    {
Forward[i][0] = (i+1) % n;
lenF[i] = 1;
lenB[i] = 0;
edgeCount++;
}

while(edgeCount < e) {
    from = rand() % n;
to = rand() % n;
    if(notAlreadyEdge(from, to)) {
        Forward[from][ lenF[from]++ ] = to;
        edgeCount++;
    }
}

makeBackwardsGraph();

void makeBackwardsGraph() {
    long int vertex, i, j;
    for(i=0; i<n; i++) {
        for(j=0; j<lenF[i]; j++) {
            vertex = Forward[i][j];
            Backward[ vertex ][ lenB[vertex]++ ] = i;
        }
    }
}

// works for forward graph
int notAlreadyEdge(long int from, long int to) {
    long int i;
    for(i=0; i<lenF[from]; i++)
        if(Forward[from][i] == to)
            return 0;
    return 1;
}

long int Primitivity() {
    long int i,j, g;
    int self_loop;
    long int startVertex = 0;
}
// added feature, start vertex is one with highest degree
long int maxDegree = -1, associatedVertex = -1;
for(i=0; i<n; i++)
{
    if(maxDegree == -1)
        {maxDegree=lenF[i]; associatedVertex = i;}
    else if(maxDegree < lenF[i])
        {maxDegree=lenF[i]; associatedVertex = i;}
}
startVertex = associatedVertex;

startBFS = clock();
selself_loop = BkwdBFS(startVertex);
FwdBFS(startVertex);
stopBFS = clock();
sumBFS += stopBFS - startBFS;

// check strongly connected
for(i=0; i<n; i++)
{
    if(P[i] < 0 || color[i] == WHITE)
        return 0;  // not strongly connected
}

// check early break gcd = 1 by self loop
if(self_loop == 1)
    return 1;

for(i=1, j=0; i<lenC; i++)
    if(C[i]==1)
        setC[j++] = i;

// calculate and time gcd
startGCD = clock();
g = gcdSet(setC, j);
stopGCD = clock();
sumGCD += stopGCD - startGCD;
return g;

long int gcd(long int a, long int b)
{
    return (b==0) ? a : gcd(b, a%b);
}

// input must have smallest non zero entry in first position,
// all numbers must be positive and < 2*n
long int gcdSet(long int * A, long int len)
{
    long int i, j;
long int min = A[0];

if(len == 0) return 0;
if(len == 1) return A[0];
if(len == 2) return gcd(A[0], A[1]);

for(i=0; i<min; i++)
    sorter[i] = 0;

for(i=0; i<len; i++)
    sorter[ A[i]%min ] = 1;

for(i=1, j=0; i<min; i++)
    if(sorter[i] == 1)
        A[j++] = i;

if(j==0) return min;
return gcdSet(A, j);

void FwdBFS(long int startVertex)
{
    long int currentDepth, i, j;
    long int currentVertex, currentEdge;

    //initialize variables----------------------------------------
    for(i=0; i<lenC; i++)
        C[i] = 0;
    for(i=0; i<n; i++)
    {
        color[i] = WHITE;
        lenS[i] = 0;
    }
    lenS[n] = 0;
    S[0][lenS[0]++] = startVertex;
    color[startVertex] = BLACK;
    currentDepth = 0;
    //----------------------------------------------------------

    //while no vertices left that haven't been processed
while(lenS[currentDepth] > 0)
{
    //for all the vertices in the current path length
    for(i=0; i<lenS[currentDepth]; i++)
    {
        currentVertex = S[currentDepth][i];
        //add all edges from graph to the next path
        //length that are new
        for(j=0; j<lenF[currentVertex]; j++)
        {
            currentEdge=Forward[currentVertex][j];
int BkwdBFS(long int startVertex)
{
    long int queue[n];
    long int Qtail = 0;
    long int u, v, i;

    int self_loop = 0;

    //initialize variables-----------------------------
    for(i=0; i<n; i++)
    {
        color[i] = WHITE;
        P[i] = INFINITY;
    }

    color[startVertex] = BLACK;
    //process starting vertex
    P[startVertex] = 0;
    queue[Qtail++] = startVertex;
    //place in queue
    //-----------------------------------------------------
    //while queue is not empty
    while(Qtail != 0)
    {
        u = queue[0]; //grab head of queue
        for(i=0; i<lenB[u]; i++)
        {
            v = Backward[u][i];
            //check for self loop
            if(v==u)
                self_loop = 1;
            //if edge from v to u hasn't been processed
            if(color[v] == WHITE)
            {
...
color[v] = BLACK;
P[v] = P[u] + 1;
queue[Qtail++] = v; // put in queue

} // dequeue
queue[0] = queue[--Qtail];

return self_loop;
}