2001

Scheduling and weighted coloring

Sandy Weihong Zhang
San Jose State University

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

SCHEDULING AND WEIGHTED COLORING

A Thesis

Presented to

The Faculty of the Department of Mathematics

and Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by Sandy Weihong Zhang

June 2001

UMI Number: 1405524

UMI Microform 1405524

Bell & Howell Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

APPROVED FOR THE DEPARTMENT OF MATHEMATICS
AND COMPUTER SCIENCE

Brad Jackson

Dr. Brad Jackson

Dr. Sin-Min Lee

John Mitchem

Dr. John Mitchem

APPROVED FOR THE UNIVERSITY

Joseph J. Pascal

ABSTRACT

SCHEDULING AND WEIGHTED COLORING

by Sandy Weihong Zhang

Scheduling problems can be represented by graphs with weighted vertices. Making a valid schedule using the least amount of time equates to finding the weighted chromatic number of the corresponding graph. First, the weighted chromatic number of some basic graphs such as bipartite graphs, even cycles, and odd cycles are determined; then by looking at the standard theorems for the chromatic number, one can try to find which of those theorems generalize to the weighted chromatic number.

DEDICATED TO THE MEMORY OF

My Grandfather
Guan Quan Yang
1918-1998

For My Grandmother
Jin Rui Yang

ACKNOWLEDGMENT

Many thanks to my thesis advisor, Dr. Brad Jackson for all he has done. He has been very helpful especially when I was lost in my research. He knew the exact direction to point me in. He seemed to apply the right amount of pressure on me; so I could finish this thesis within a reasonable amount of time. I have studied two graph theory courses from Dr. Jackson. His good lectures attracted me to stay in the field of graph theory.

My sincere thanks to Dr. Sin-Min Lee and Dr. John Mitchem for their careful reading and helpful criticism on my thesis. It was due to Dr. Mitchem's encouragement that I signed up for a graph theory class. As a math graduate coordinator, Dr. Mitchem has also helped me with many of the administrative steps; and has given me many useful math-related suggestions and guidance. Through working with Dr. Lee, I have a better understanding of my thesis. He has been very helpful and dedicated time for me even though his office is always full of students eager for his help.

TABLE OF CONTENTS

1. Introduction to Scheduling and Weighted Coloring

1.1 Introduction

Graph theory is a relatively young field in Mathematics. The publication of Euler's solution to the Königsberg Bridge Problem in 1736 is considered by many to be the birth of graph theory. The first full-length book on the subject, written by Dénes König, was published in 1936. Although many mathematicians study graph theory because of its intrinsic beauty, it has applications in operation research, chemistry, sociology, genetics, etc.. The Four Color Problem, has played an essential role in the development of graph theory. It has sparked the interest of mathematicians and amateurs alike.

Some applications of graph coloring are storage problems, zero knowledge passwords, circuit testing, and scheduling. The following sections describe some of these applications briefly.

1.2 A Storage Problem (Bondy and Murty)

A company manufactures n chemicals. Certain pairs of chemical can't be stored together in the same compartment because an explosion would occur if they were brought into contact with each other. What is the minimum number of compartments needed for storing these n chemical? This problem can be modeled with a graph. Let $v_1, v_2, \ldots, v_n$ be the vertices which represent the n chemicals. Draw an edge between two vertices if and only if they represent two incompatible chemicals. Two vertices are adjacent with each other if there is an edge joining the two. A proper coloring of a graph is one such that no two adjacent vertices receive the same color. The least number of

colors needed to color this graph is equal to the least number of compartments needed to store these chemicals.

1.3 Zero-knowledge Passwords (McHugh)

The least number of colors needed to color a graph is called the chromatic number of a graph. In fact, finding the chromatic number of a graph is an NP-complete problem. This application to be talked about relies precisely on the difficulty of solving such problems even for graphs with chromatic number 3. For access to a restricted system, a user identification and a password are assigned to each valid user. Each user is assigned a randomly generated three-colorable graph (i.e., a graph that can be colored with 3 colors) as the user identification number. "To access the system, a user provides a graph-id, and proves ownership of the graph by three-coloring it." The construction of a random three-colorable graph is simple, (see page 226 McHugh) and by its construction, one can three-color the final graph. However it is computationally intractable to find a three-coloring of a given graph. A particular three-coloring serves as the password; but it is unlike the ordinary password. "This three-coloring password does not even need to be shared with the system being accessed." To illustrate this idea, say that a valid user is trying to convince an observer that the user actually has the right password without revealing the password - the particular three-coloring of the graph-id. For simplicity, the graph is put on a table with all its vertices covered; the coloring of the graph is not revealed. Let the observer pick a random pair of adjacent vertices; then the user uncovers those two vertices, showing their different colors. The user then secretly and randomly permutes the colors of the graph. The user repeats this process until the

observer is convinced that the user has a three-coloring of the graph. Probabilistically speaking, after some sufficiently large number of tests, the chance that the user could deceive the observer can be made arbitrarily small. Because of the permutations of the colors, the observer can't figure out the password even though the observer can be convinced that the valid user has the right password.

1.4 A Scheduling Problem

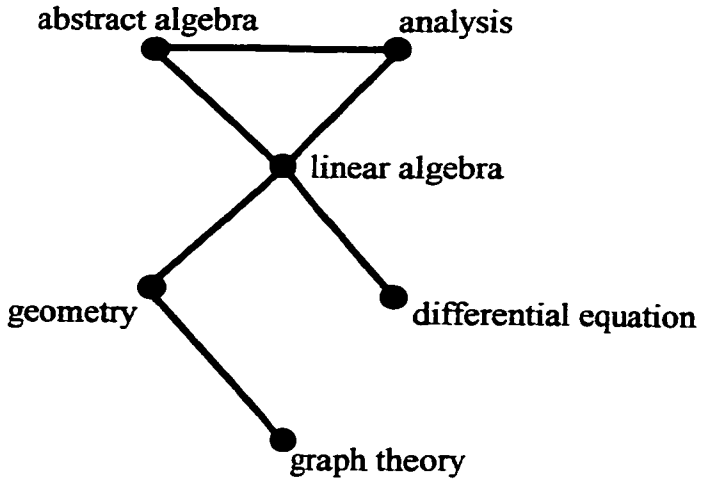
Suppose one is responsible for scheduling all the math classes for the coming semester. For simplicity, assume all the math classes are 1-hour long; and each class is scheduled to start at the beginning of the hour. To make a good schedule, one must consider the following two requirements. First, there are certain courses that can't be scheduled at the same time. Secondly, a schedule is set up with the fewest number of time periods required. This allows one to eliminate some unpopular times or bad times for class meetings such as lunch time or 2 pm when many students are the sleepiest. The question is: what is the minimum number of hours needed for such a schedule?

Model this scheduling problem using a graph. One will create a graph by representing each course with a vertex. Two vertices are joined by an edge if and only if the two courses represented by these two vertices can't be scheduled at the same time. Here is a small example. Suppose six courses will be offered in the next semester. They are graph theory, analysis, geometry, abstract algebra, linear algebra, and differential equations. The constraints are :

1. Graph theory and geometry can't be scheduled at the same time.

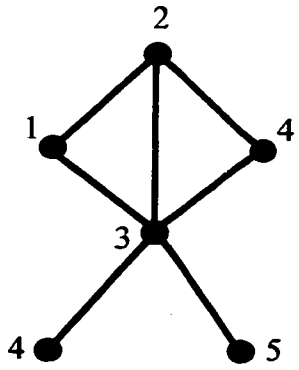
2. Any of the two courses from analysis, abstract algebra, and linear algebra can not be scheduled at the same time.
3. Linear algebra and differential equations can not be scheduled at the same time.
4. Geometry and linear algebra can not be scheduled at the same time.

The following graph G models this situation .



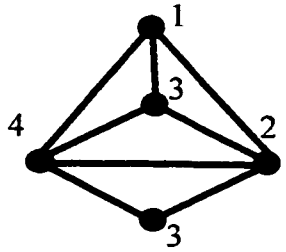
A **coloring** of a graph G is an assignment of colors to the vertices of G such that each vertex receives one color and the adjacent vertices are assigned different colors. An **n-coloring** of G is a coloring of G using n colors. The following two figures show a 5-coloring of G_1 and a 4-coloring of G_2 where the colors are denoted by numbers.

G_1:



(a 5-coloring of G_1)

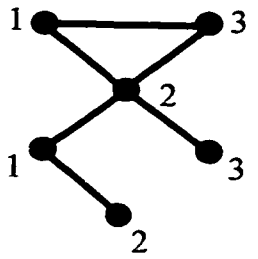
G_2:



(a 4-coloring of G_2)

The minimum number of colors needed to color a graph G is called the **chromatic number** of G. $\chi(G)$ is the standard notation for the chromatic number of G. Look at G_1 and G_2 again. G_1 can be colored also by 4 colors or even better by 3 colors. Hence, $\chi(G_1) = 3$ because 3 is actually the smallest number of colors required. For G_2, $\chi(G_2) = 4$ which is a simple verification for the reader.

Now back to the graph which models the scheduling problem. The minimum number of colors needed to color that graph is the minimum number of hours required for the schedule of classes. That is to say, the minimum number of hours required for the schedule of classes in this schedule problem is $\chi(G)$. One can verify that $\chi(G)$ in this model is 3. Here is a 3-coloring of G.

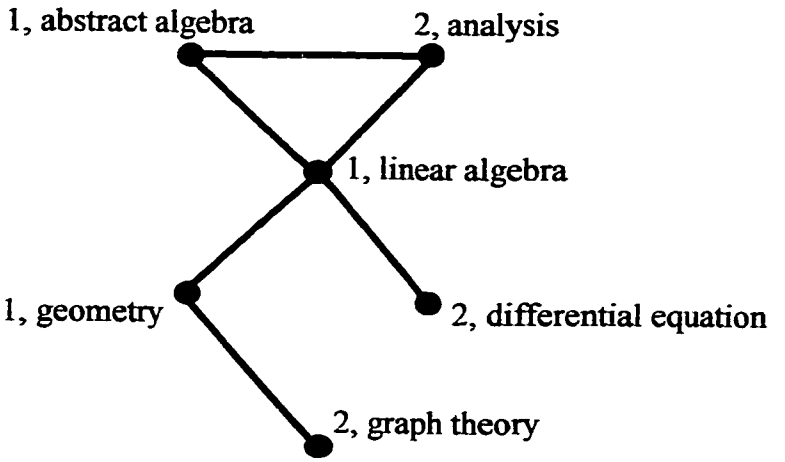


(a 3-coloring of G)

1.5 Scheduling and Weighted Coloring

Suppose in the previous scheduling problem, not all the math classes are of the same length. Some of them are 1-hour long while the rest are 2-hour long. To model this new situation, a weight is assigned to each vertex; and the weight of each vertex is the duration time of the class that the vertex represents. Suppose abstract algebra, linear

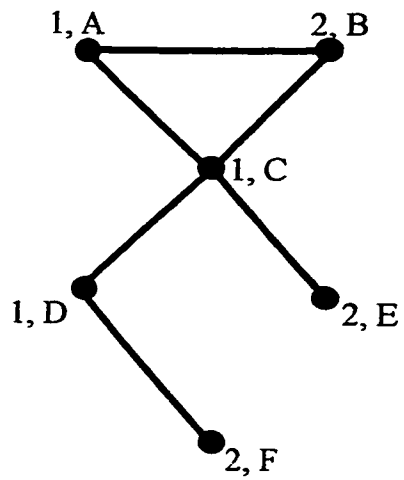
algebra, and geometry are 1-hour classes while analysis, graph theory, and differential equation are 2-hour classes. The new graph would be a weighted graph as follows.



(a weighted graph)

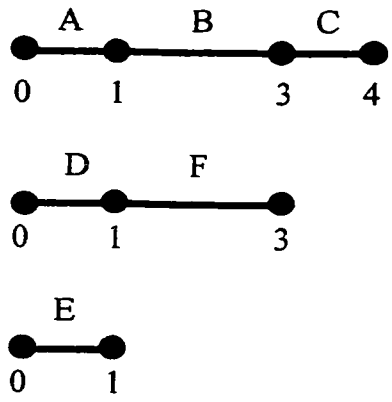
The minimum number of hours required for the schedule of a weighted graph G is called **the weighted chromatic number** of G. The weighted chromatic number of G is denoted by $\chi^*(G)$. Assign a time interval = "color" to each vertex, with the length of the interval = the "weight" of the corresponding vertex. It is common to label the vertices by capital letters. The weighted chromatic number of the model graph G will be discussed on the next page.

G:



(a weighted graph)

To make a schedule out of this weighted graph, one can proceed with the following drawing.



The above drawing says that event A (abstract algebra in this example) is scheduled from time 0 to time 1, event B from time 1 to time 3, event C from time 3 to time 4, event D form time 0 to 1, etc. Based on this schedule, 4 hours are used. In fact, 4 hours are the minimum of time required for this weighted graph G. So, $\chi^*(G) = 4$. Note that there are

other valid schedules which also use only 4 hours. So the schedule leading to the weighted chromatic number may not be unique.

In this thesis, the weighted chromatic number of a graph is studied. All the theorems and statements about the weighted chromatic number in this paper are original or at least no reference books of weighted chromatic number could be found or were used upon the completion of this paper. How were such theorems or statements derived? By looking at the theorems of the (regular) chromatic number, one can try to see if there were analogous theorems about the weighted chromatic number.

2. Definitions and Notations

It is assumed that the reader knows about the basic terms and notations in graph theory. Please refer to "Applied and Algorithmic Graph Theory" by Gary Chartrand and Ortrud R. Oellermann for basic definitions and notations in graph theory. Some selected definitions and notations are listed here. The usual definitions are mostly from Chartrand and Oellermann's book while other definitions are introduced for weighted graphs.

DN1. **V(G)** denotes the vertex set of a graph G; **E(G)** denotes the edge set of a graph G.

DN2. A **walk** in a graph G is an alternating sequence W: $v_0, e_1, v_1, e_2, v_2, \ldots, v_{n-1}, e_n, v_n$ ($n \geq 0$) of vertices and edges, beginning and ending with vertices, such that $e_i = v_{i-1} v_i$, for $i = 1, 2, \ldots, n$. We refer to W as a v_0 - v_n walk. Also W is said to have length n because W encounters n (not necessarily distinct) edges. Walks are mostly commonly expressed just by the vertices.

DN3. A **trail** is a walk in which no edge is repeated.

DN4. A **path** is a walk in which no vertex is repeated. The **length of a path** is the number of (not necessarily distinct) edges the path has.

DN5. The **weight of a path P** in a weighted graph G, is the sum of the weights of the vertices along P.

DN6. A **cycle** is a walk $v_0, v_1, \ldots, v_n$ in which $n \geq 3$, $v_0 = v_n$, and the n vertices $v_1, v_2, \ldots, v_n$ are distinct. A cycle of length n is referred to as an n-cycle. An **even cycle** is a cycle with an even length. An **odd cycle** is a cycle with an odd length.

DN7. A graph G is **bipartite** if V(G) can be partitioned into two (nonempty) subsets V_1 and V_2 such that every edge of G joins a vertex of V_1 and a vertex of V_2. The sets V_1 and V_2 are called the **partite sets** of G.

DN8. The **degree** of a vertex, **deg(v)**, is the number of vertices adjacent to v.

DN9. **A weighted graph** is a graph with a weight associated with each vertex. The weight of each vertex (task) of the graphs in this paper will be the number of hours needed to complete that task. The weight of vertex v is denoted by **w(v)**.

DN10. **The weighted degree** of a vertex in a weighted graph, $\mathbf{deg^*(v)}$, is the sum of the weight of v, and the weights of all the neighbors of v.

DN11. **The maximum degree** of a graph G, **Δ(G)**, is the degree of a vertex v such that $\deg(v) \geq \deg(u)$ for every vertex u of G.

DN12. **The weighted maximum degree** of a weighted graph G, $\mathbf{\Delta^*(G)}$, is the weighted degree of a vertex v such that $\deg^*(v) \geq \deg^*(u)$ for every vertex u of G.

DN13. **The minimum degree** of a graph G, **δ(G)**, is the degree of a vertex v such that $\deg(v) \leq \deg(u)$ for every vertex u of G.

DN14. **The weighted minimum degree** of a weighted graph G, $\mathbf{\delta^*(G)}$, is the weighted degree of a vertex v such that $\deg^*(v) \leq \deg^*(u)$ for every vertex u of G.

DN15. Let G be a graph with n vertices. The **adjacency matrix** $A = [a_{ij}]$ of G is the $n \times n$ matrix defined by

$$a_{ij} = \begin{cases} 1 \text{ if } v_i v_j \in E(G) \\ 0 \text{ otherwise} \end{cases}$$

Thus, A is a symmetric matrix in which every entry on the main diagonal is 0.

DN16. Let G be a weighted graph with n vertices. The **weighted adjacency matrix** A^* of G is the $n \times n$ matrix with all the non-diagonal entries identical to that of the usual adjacency matrix A. The i^{th} diagonal entry of A^* is the weight of vertex v_i.

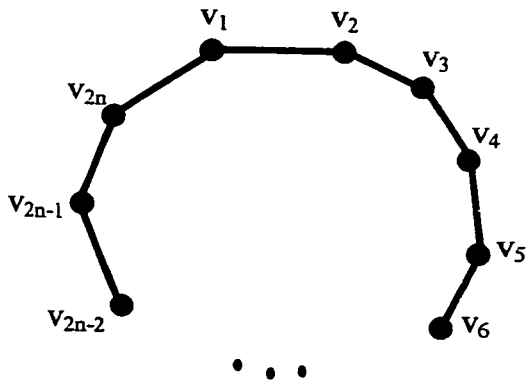
3. Weighted Chromatic Number of Some Basic Graphs

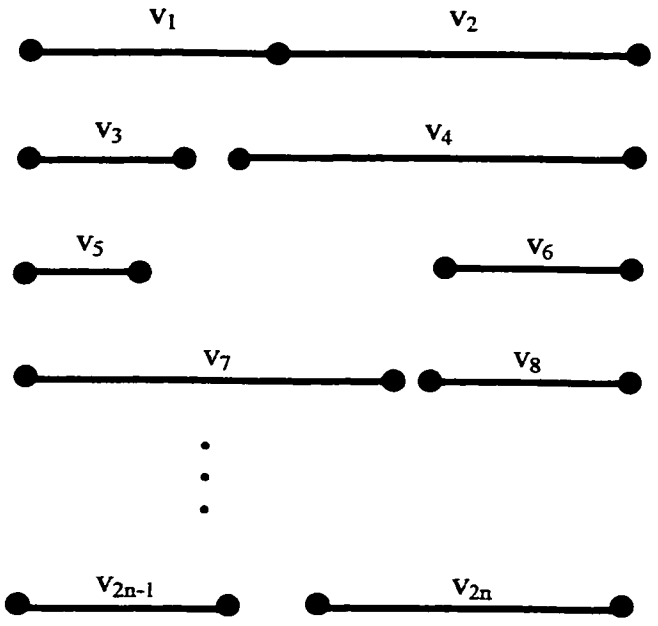
It is natural to find the weighted chromatic number of some basic graphs first because the ordinary chromatic numbers of those basic graphs are very simple to find. For even cycles, the minimum number of colors needed is 2; so $\chi(C_{2n}) = 2$. For odd cycles, the minimum number of colors needed is 3; so $\chi(C_{2n+1}) = 3$. For bipartite graphs, color one partite set by one color, and color the other partite set by another color. That implies $\chi(G) = 2$ when G is a bipartite graph. Note that even cycles are bipartite graphs.

Because the even cycles are easier to understand, it is logical to look at even cycles before making a generalization to bipartite graphs.

Theorem 3.1 For an even cycle C_{2n}, $\chi^*(C_{2n})$ = max sum of two consecutive weights.

Proof: Without loss of generality, let $w(v_1) + w(v_2)$ produce the max sum of two consecutive weights. Schedule $v_1, v_3, v_5, \ldots, v_{2n-1}$ to start at time zero; and schedule each of $v_2, v_4, v_6, \ldots, v_{2n}$ to end at time $w(v_1) + w(v_2)$.





This schedule puts v_2 immediately following v_1. Note that v_3 and v_4 do not overlap with each other because $w(v_3) + w(v_4) \leq w(v_1) + w(v_2)$. The same thing is true for v_5 and v_6, v_7 and v_8, ..., v_{2n-1} and v_{2n}. Also note v_3 and v_2 do not overlap with each other because of the left and right alignments of the tasks and $w(v_3) + w(v_2) \leq w(v_1) + w(v_2)$. The same thing is true for v_5 and v_4, v_7 and v_6, ..., v_{2n-1} and v_{2n-2}, v_1 and v_{2n}. This schedule implies that $\chi^*(C_{2n}) \leq w(v_1) + w(v_2)$.

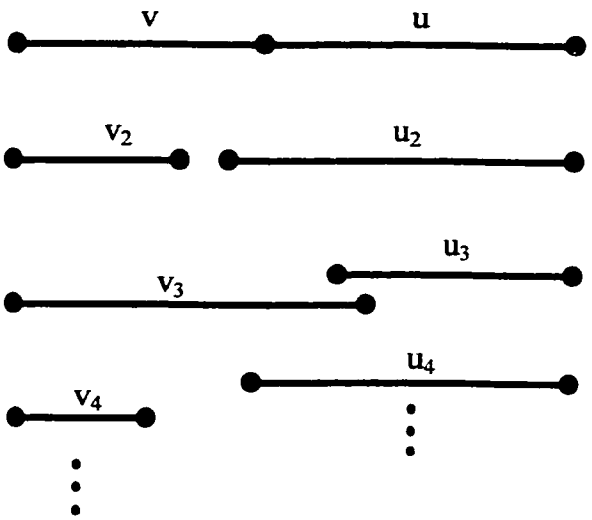
The fact that v_1 and v_2 cannot be scheduled in conflicting time implies that $\chi^*(C_{2n}) \geq w(v_1) + w(v_2)$. Therefore, $\chi^*(C_{2n}) = w(v_1) + w(v_2)$ = the max sum of two consecutive weights.

Q.E.D.

Next, a generalization of theorem 3.1 to weighted bipartite graphs will be presented.

Theorem 3.2 Let G be a bipartite graph without any isolated vertices with partite sets $\{v_1, v_2, \ldots, v_k\}$ and $\{u_1, u_2, \ldots, u_h\}$; then $\chi^*(G) = \max \{w(v_i) + w(u_j)\}$ over all v_i and u_j where $v_i u_j \in E(G)$.

Proof: Suppose $w(v) + w(u) = \max \{w(v_i) + w(u_j)\}$ over all v_i and u_j where $v_i u_j \in E(G)$ and $vu \in E(G)$. Clearly, $\chi^*(G) \geq w(v) + w(u)$ since $vu \in E(G)$. Consider the following schedule.



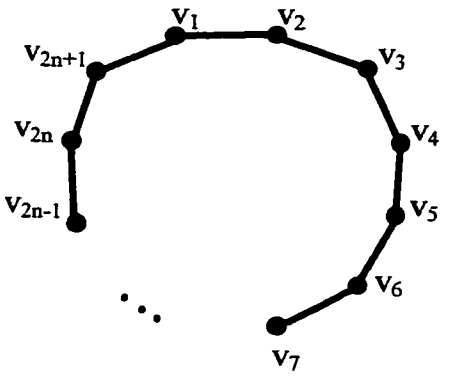
[WLOG, let $v = v_1$ and $u = u_1$]

All vertices in one partite set begin at time zero while all vertices in the other partite set end at the final time $w(v) + w(u)$. Suppose events v_i and u_j overlap; then $w(v_i) + w(u_j) > w(v) + w(u)$. This implies $v_i u_j \notin E(G)$. That is to say that it is okay for v_i and u_j to overlap. Therefore, this scheduling is valid. It follows that $\chi^*(G) \leq w(v) + w(u)$. Together with the earlier result, the equality is established. Q.E.D.

What about odd cycles, C_{2n+1}? The following result for odd cycles is interesting.

Theorem 3.3 For any odd cycle C_{2n+1}, $n \geq 1$, $\chi^*(C_{2n+1})$ = max{max sum of 2 consecutive weights, min sum of 3 consecutive weights}

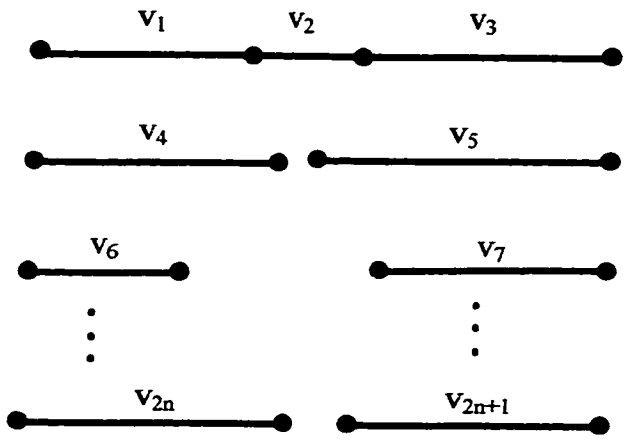
Note that min sum of 3 consecutive weights is the minimum weighted degree of the cycle, $\delta^*(C_{2n+1})$.



Proof: Let m = max{max sum of 2 consecutive weights, min sum of 3 consecutive weights} Without loss of generality, let min sum of 3 consecutive weights be $w(v_1)+w(v_2)+w(v_3)$.

(Case 1) m = min sum of 3 consecutive weight = $w(v_1)+w(v_2)+w(v_3)$.

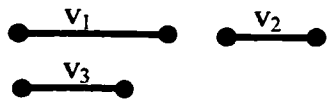
Then the scheduling on the next page is feasible.



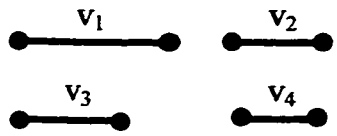
This above scheduling implies that $\chi^*(C_{2n+1}) \leq w(v_1) + w(v_2) + w(v_3)$.

It will be shown that there are always three consecutive vertices scheduled in three non-overlapping time intervals. v_1 and v_2 have to be scheduled in two non-overlapping time intervals. Without loss of generality, assume v_2 starts after v_1.

If v_3 is scheduled in a time interval that does not overlap the time intervals for v_1 and v_2, then that's it; otherwise v_3 must be scheduled in a time interval that overlaps the time interval for v_1.

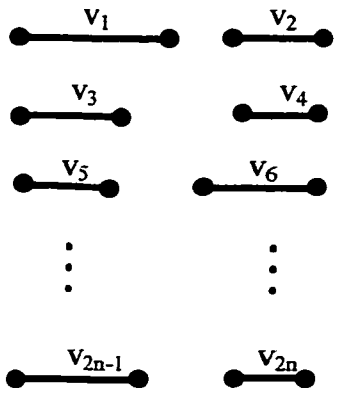


If v_4 is scheduled in a time interval that does not overlap the time intervals for v_2 and v_3, then that's it; otherwise v_4 must be scheduled in a time interval that overlaps the time interval for v_2.



If v_5 is scheduled in a time interval that does not overlap the time intervals for v_3 and v_4, then that's it; otherwise ...

In the worst case scenario of this proof, it comes down to the following scheduling.

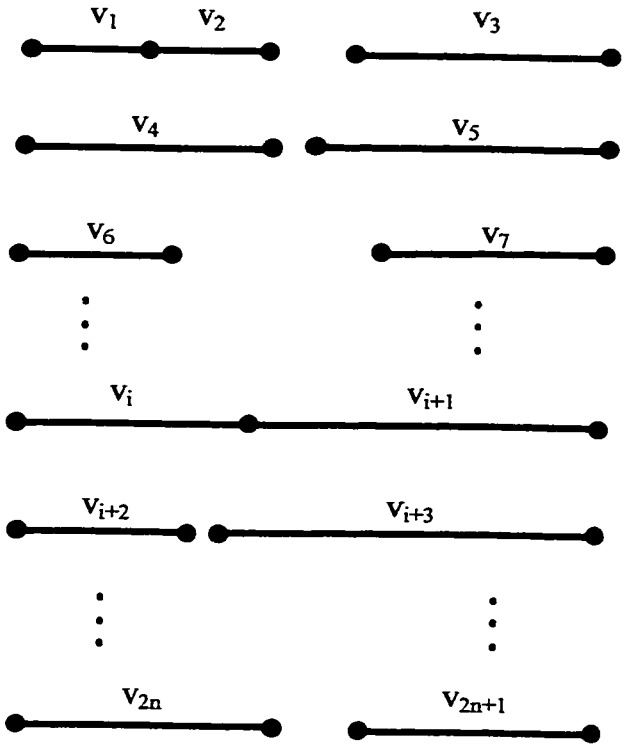


Now when can v_{2n+1} be scheduled? The only time v_{2n+1} can be scheduled is in a time interval that does not overlap the time intervals of v_1 and v_{2n}. Therefore, there are three consecutive vertices (v_{2n}, v_{2n+1}, v_1) scheduled in three time intervals which do not overlap each other. Because there are always three consecutive vertices scheduled in three non-overlapping time intervals, and the minimum of three consecutive weights equals $w(v_1)+w(v_2)+w(v_3)$, one can conclude that $\chi^*(C_{2n+1}) \geq w(v_1)+w(v_2)+w(v_3)$. Therefore, $\chi^*(C_{2n+1}) = w(v_1)+w(v_2)+w(v_3) = m$.

(Case 2) m = max sum of 2 consecutive weights = $w(v_i) + w(v_{i+1})$ where $i = 1$ to $2n+1$; and where v_{i+1} is v_1 if v_i is v_{2n+1}.

Recall that min sum of 3 consecutive weights is $w(v_1) + w(v_2) + w(v_3)$. This case says that $w(v_i) + w(v_{i+1}) \geq w(v_1) + w(v_2) + w(v_3)$. First, schedule v_1 to start at time 0; v_3 to end at time $w(v_i) + w(v_{i+1})$; v_2 immediately following v_1. Next, schedule all vertices with even subscripts, v_4, v_6 , v_6 , ..., and v_{2n} to start at time 0. Finally, schedule all vertices with odd subscripts, v_5, v_7 , v_9 , ..., and v_{2n+1} to end at time $w(v_i) + w(v_{i+1})$.

One such schedule may look like the following.



Such scheduling tells $\chi^*(C_{2n+1}) \leq w(v_i) + w(v_{i+1}) = m$. The weighted chromatic number must of course be greater than the sum of the weights of two adjacent vertices; so it is obvious that $\chi^*(C_{2n+1}) \geq w(v_i) + w(v_{i+1}) = m$. Therefore, $\chi^*(C_{2n+1}) = m$ in case 2.

Q.E.D.

4. Upper Bounds for the Weighted Chromatic Number

In this chapter, some upper bounds for the weighted chromatic number are found by looking at the corresponding upper bounds for the ordinary chromatic number. The following is a known result relating the chromatic number and the length of a longest path.

Theorem 4.1 For every graph G, $\chi(G) \leq 1 + \ell(G)$, where $\ell(G)$ denotes the length of a longest path in G.

Proof: Let $\chi(G) = k$. Then divide the vertices of G into k independent classes (i.e. color classes), say V_1', V_2', V_3', ... , V_k' with color i assigned to the vertices of V_i'. Order the vertices of G starting with the vertices of V_2', then the vertices of V_3', ..., and finally the vertices of V_k'. Use the greedy algorithm to assign new colors to the vertices of G. It proceeds in the following manner. Let u be a vertex in V_2'. If u is not adjacent with any vertex in V_1', then reassign u with color 1 instead. If u is adjacent with some vertex in V_1', then u keeps its color, namely color 2. Note that some vertex in V_2' is assigned color 2; otherwise $\chi(G) = k - 1$. Let v be a vertex in V_3'. If v is not adjacent to any vertex with color 1, then reassign v with color 1 instead. If v is adjacent to some vertex with color 1, but not adjacent to any vertex with color 2, then reassign v with color 2. If v is adjacent to some vertex with color 1, and also adjacent to some vertex with color 2, then v keeps its color 3. Similarly, some vertex in V_3' is assigned color 3; otherwise $\chi(G) = k - 1$. In this process, one always tries to assign the lowest color to every vertex. Keep doing this process, the final step yields color classes V_1, V_2, V_3, ... , V_k.

Let u_k be a vertex in $V_k \neq \varnothing$, then u_k must be adjacent with a vertex u_{k-1} in V_{k-1} due to the process. Similarly, u_{k-1} in V_{k-1} must be adjacent with a vertex u_{k-2} in V_{k-2}. And u_{k-2} must be adjacent with a vertex u_{k-3} in V_{k-3}; ... This gives a path $u_1 - u_2 - u_3 - \ldots - u_{k-3} - u_{k-2} - u_{k-1} - u_k$ of length k-1. Hence, $\ell(G) \geq k - 1$. It follows $k \leq 1 + \ell(G)$; i.e., $\chi(G) \leq 1 + \ell(G)$.

Q.E.D.

Theorem 4.2 Let G be any weighted graph. If $\chi(G) = k$, then $\chi^*(G) \leq \max\{\text{weight of P}\}$ where path P has length k-1 or less.

Proof: Since $\chi(G) = k$, there are k independent sets V_1', V_2', V_3', ..., V_k' of V(G) where V_i' is the color class in which vertices of G are assigned color i. Color the vertices of G with colors {1,2,3, ..., k} by the greedy algorithm in the sense that the lowest available color is assigned to each vertex (for details, see the proof of the previous theorem.) This yields k new independent sets (color classes) V_1, V_2, V_3, ..., V_k such that V_i contains all vertices with color i for i = 1 to k. For i = 2 to k, each vertex in V_i must then be adjacent to some vertex in each of V_1, V_2, V_3, ..., and V_{i-1}.

Next apply the weighted greedy algorithm to the scheduling in the sense that each task is scheduled as early as possible. Schedule all the vertices (tasks) in V_1 to start at time zero. For each vertex u in V_2, consider all the vertices of V_1 that are adjacent with u. Among those vertices, suppose $w(v_1) + w(u)$ is the maximum. Schedule u to start at time $w(v_1)$. Let $P_m(u)$ denote this maximum (heaviest) path $v_1 - u$. For convenience, let

$P_m(z)$ equal the path of length zero, z for each vertex z of V_1. For each vertex u in V_3, consider all the vertices in V_1 and V_2 that are adjacent with u. Among those vertices, suppose $w(P_m(v_2)) + w(u)$ is the maximum. Note that v_2 could be a vertex either from V_1 or V_2. Schedule u right after v_2. Let $P_m(u)$ denote this new path formed by attaching u to $P_m(v_2)$.

Apply the same process to each vertex in V_4, then in V_5, V_6, ..., V_k. That is, for each vertex u in V_i where i = 4 to k, consider all vertices in V_1, V_2, ..., V_{i-1} that are adjacent with u. Among those vertices, suppose $w(P_m(v)) + w(u)$ is the maximum. Schedule u right after v. Let $P_m(u)$ denote this new path formed by attaching u to $P_m(v)$.

When this process is done, there is a schedule with total time equal to $\max\{w(P_m(v_i))\}$ where v_i are over all the vertices in G. This implies that $\chi^*(G) \leq$ max{weight of P} where path P has length k-1 or less.

Q.E.D.

A immediate result from the above theorem:

Corollary $\chi^*(G) \leq \max\{\text{weight of P}\}$ where P is a path of any length.

The following corollary of this theorem is weaker than Theorem 3.2.

Corollary Let G be a bipartite graph (not necessarily connected),

then $\chi^*(G) \leq \max\{\text{weight of P}\}$ where P has length 0 or 1.

Here is another classic result for the chromatic number.

Theorem 4.3 Let G be a connected graph with maximum degree $\Delta(G)$.

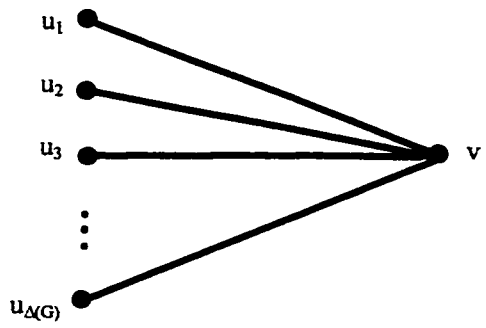
Then $\chi(G) \leq 1+ \Delta(G)$.

Proof: (by induction)

Initial cases: When the order of G equals 1, then $\Delta(G) = 0$ and $\chi(G) = 1$.

When the order of G equals 2, then $\Delta(G) = 1$ and $\chi(G) = 2$.

Assume $\chi(G) \leq 1+ \Delta(G)$ holds for any graph G with order n-1.



The goal is to show that $\chi(G) \leq 1+ \Delta(G)$ for any graph G with order n. Suppose G has order n, and let $\deg(v) = \Delta(G)$. By the induction hypothesis, $\chi(G-v) \leq 1+ \Delta(G-v)$. In addition, $\Delta(G-v) \leq \Delta(G)$. It follows that $\chi(G-v) \leq 1+ \Delta(G)$. Let the neighbors of v in G be $u_1, u_2, \ldots, u_{\Delta(G)}$. Then $u_1, u_2, \ldots, u_{\Delta(G)}$ receive at most $\Delta(G)$ colors in the coloring of G-v. Add vertex v back to G-v, forming G. Color v with one of the $1+ \Delta(G)$ colors not used to color a neighbor of v. Therefore, $\chi(G) \leq 1+ \Delta(G)$.

Q.E.D.

An analogous result to weighted colorings is stated in the following theorem.

Theorem 4.4 For any connected weighted graph G, $\chi^*(G) \leq \Delta^*(G)$ and for every vertex u of G, u can be scheduled to end no later than $\deg^*(u)$.

Proof: It is obviously true for graphs of orders 1 or 2. Assume the statement is true for graphs of order n-1. Let the order of G be n. Let v be a vertex of G such that $\deg^*(v) = \Delta^*(G)$; and let its neighbors be $v_1, v_2, \ldots, v_k$. Consider the new graph G-v of order n-1. By the induction hypothesis, $\chi^*(G\text{-}v) \leq \Delta^*(G\text{-}v) \leq \Delta^*(G)$; and furthermore, in the scheduling of G-v, every v_i (for i = 1 to k) ends no later than $\deg_{G\text{-}v}^*(v_i)$.

But $\deg_{G\text{-}v}^*(v_i) = \deg_G^*(v_i) - w(v)$

$\leq \deg_G^*(v) - w(v)$

$= \Delta^*(G) - w(v)$.

That is, $\chi^*(G\text{-}v) \leq \Delta^*(G)$; and in the scheduling of G-v, every v_i (for i = 1 to k) ends no later than $\Delta^*(G) - w(v)$. Now put the vertex v back in the graph; and schedule v from time $\Delta^*(G) - w(v)$ to time $\Delta^*(G)$. Note that v ends no later than $\Delta^*(G)$ which equals $\deg_G^*(v)$. This shows that $\chi^*(G) \leq \Delta^*(G)$ for $|G| = n$, and that every vertex u of G can be scheduled to end no later than $\deg^*(u)$. By induction, the theorem is proved.

Q.E.D.

5. A Conjecture about the Weighted Chromatic Number

5.1 Introduction

Based on the following theorem, a similar result about the weighted chromatic number can be conjectured.

Theorem 5.1 For any graph G, $\chi(G) \leq 1+ \max \delta(G')$, where the maximum is taken over all induced subgraphs G′ of G.

Proof: The result is immediate for empty graphs. Let G be an arbitrary n-chromatic graph, $n \geq 2$. Let H be any smallest induced subgraph such that $\chi(H) = n$. The graph H therefore has the property that $\chi(H-v) = n-1$ for all its vertices v. It follows that $\deg(v) \geq n-1$. So $\delta(H) \geq n-1$; and hence $n-1 \leq \delta(H) \leq \max \delta(H') \leq \max \delta(G')$, the first maximum taken over all induced subgraphs H′ of H and the second over all induced subgraphs G′ of G. This implies $\chi(G) = n \leq 1+ \max \delta(G')$. Q.E.D.

Corollary 5.2 For any graph G, the chromatic number is at most one greater than the maximum degree; that is, $\chi(G) \leq 1+ \Delta(G)$.

Note that this corollary is just Theorem 4.3. So Theorem 4.3 could have been proven in this way. Here is the conjecture based on the previous theorem.

Conjecture 5.3 $\chi^*(G) \leq \max\{\delta^*(H)\}$ where H is taken over all induced subgraphs of G.

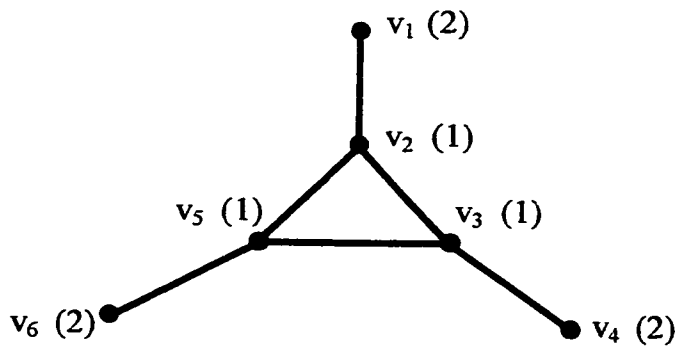
It was called a conjecture for a period of time because it could not be shown as a true statement. While trying to prove the conjecture, difficulties arose. After failing to provide a proof, it was suspected that it might be false. However, it was also hard to

disprove it. It seemed that the easiest step to take was to write a computer program which could find a counter-example if a small enough counter-example existed. The program was written in C++; but it mainly used the ideas from C. The time spent on the program paid off because it found four counter-examples after looking at 6624 weighted graphs. Please refer to section 5.4 (Input Data) to find out how those 6624 weighted graphs were chosen.

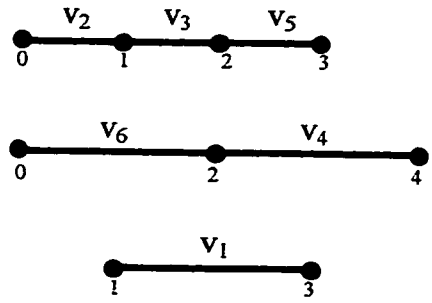
5.2 The Four Counter-examples

These four counter-examples may look simple and in fact, some of their $\chi^*(G)$ and $\max\{\delta^*(H)\}$ can be easily verified by hand; but it is a difficult task to come up with counter-examples without the help of the computer program. The source codes of the program can be found on pages 39 to 49.

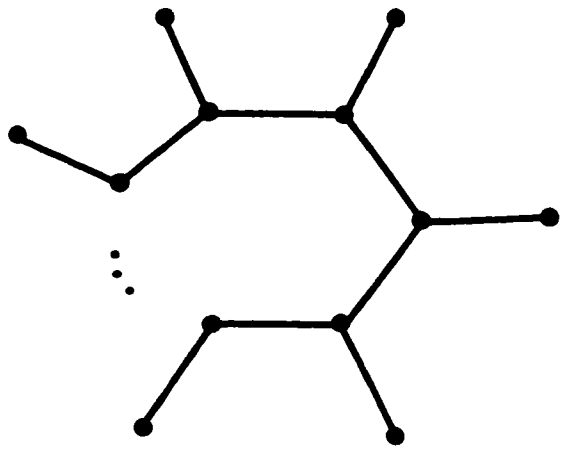
Counter-example 1: It was found when the adjacency matrix of graph card #94 was input into the program. The following weighted graph G is a counter-example with $\chi^*(G) = 4$, and $\max\{\delta^*(H)\} = 3$. The weights of the vertices are displayed in parentheses.



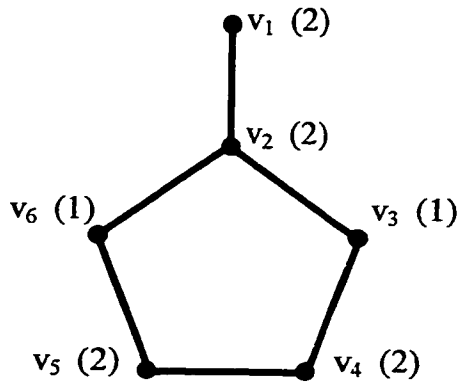
The following is a schedule using 4 hours.



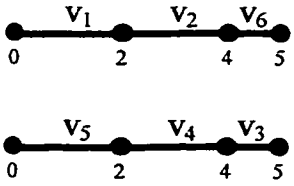
Counter-example 1 gives rise to a family of counter-examples. This family of counter-examples are represented by the following graph. In the middle, there is an odd cycle with weight 1 on each of its vertices; and each vertex outside the odd cycle has weight 2. It is simple to verify by hand that $\chi^*(G) = 4$, and $\max\{\delta^*(H)\} = 3$.



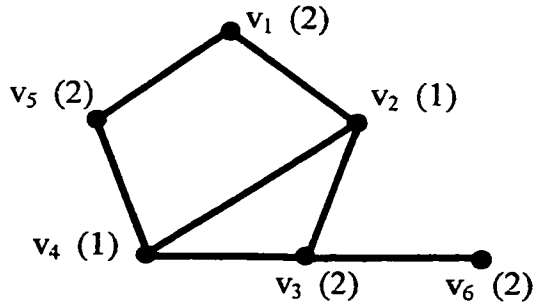
Counter-example 2: This was found when the adjacency matrix of graph card # 104 was entered into the program. The following graph G has $\chi^*(G) = 5$, and $\max\{\delta^*(H)\} = 4$.



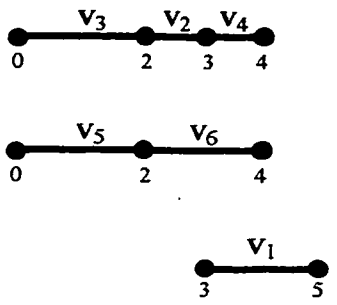
Here is a schedule that takes 5 hours.



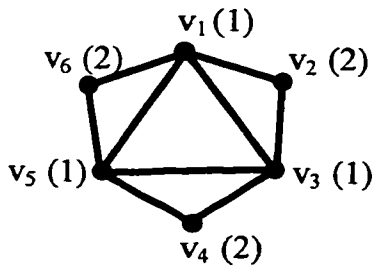
Counter-example 3: This counter-example was found when the adjacency matrix of graph card #125 was entered into the program. This graph G has $\chi^*(G) = 5$, and $\max\{\delta^*(H)\} = 4$.



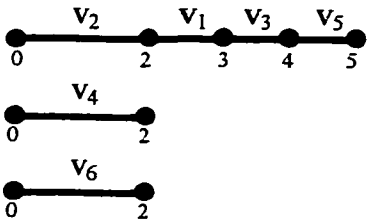
The following schedule takes 5 hours.



Counter-example 4: This counter-example was found when the adjacency matrix of graph card #163 was entered. This graph G has $\chi^*(G) = 5$, and $\max\{\delta^*(H)\} = 4$.



The following schedule takes 5 hours.



5.3 A Summary of the Computer Program Used to Find Counter-examples

The source codes of the computer program used to find counter-examples are included in appendix 7.1; and one output is included in appendix 7.2. Several techniques

learned in combinatorics were used in the program. Those techniques include how to tell the computer to find the subsets of a given set, and how to find all the permutations of a given set.

A summary of the computer program follows. The program looks at the adjacency matrix of a graph. It assigns a weight of 1 or 2 to each vertex of the graph; and it generates a new matrix - an adjacency matrix that has the weights of each vertex on its diagonal. Based on this new matrix, the program finds its weighted chromatic number $\chi^*(G)$, and the maximum of the minimum weighted degree $\max\{\delta^*(H)$ over all induced subgraphs H. If $\chi^*(G) > \max\{\delta^*(H)\}$ where H is taken over all induced subgraphs of G, G is a counter-example to the conjecture.

To find $\max\{\delta^*(H)\}$, the program uses the binary string to generate an induced subgraph H of G. The program computes the weighted degree of all the vertices in each induced subgraph H to find the minimum weighted degree $\delta^*(H)$ for each H. After it has looked at every induced subgraph H, the $\max\{\delta^*(H)\}$ can be found. The process is very labor intensive; so the program definitely helps.

To find $\chi^*(G)$, the program applies the weighted greedy algorithm to each ordering of the vertices of G, to find out the number of hours needed. Out of all the orderings of the vertices of G, the minimum number of hours needed is the weighted chromatic number. To get all the orderings of the vertices, the program needs to generate all the permutations of the vertices. The program uses an algorithm stated on page 126

in "Applied Combinatorics" by Jackson and Thoro, which enumerates all permutations of the vertex set of G.

The **weighted greedy algorithm for scheduling** works as follows. First schedule vertex(task) v_1 to start at time 0. Schedule all the vertices in the increasing order of their subscripts. To schedule each v_i, looked at all the neighbors of v_i which have been scheduled; that is, look at only the neighbors of v_i with the subscript less than i. Out of all those neighbors being looked at, pick out the neighbor say v_j which ends at the latest time. Schedule v_i to start at the time which v_j ends. That is, schedule each v_i at its earliest possible time according to that ordering of the vertices.

Claim: There is an ordering of vertices such that when the weighted greedy algorithm is applied. It yields the weighted chromatic number.

Proof: Let G be a weighted graph with n vertices. Suppose $\chi^*(G) = k$. Then there is a scheduling *S* of G that uses exactly k hours. Order all the vertices of G in a non-decreasing order based on their starting times. Call them $v_1, v_2, v_3, \ldots, v_n$ according to that order. Let $s(v_i)$ denote the starting time of v_i in schedule *S*, then $0 = s(v_1) \leq s(v_2) \leq s(v_3) \leq \ldots \leq s(v_n)$. Now apply the weighted greedy algorithm to G with that ordering of vertices. Strong induction is used to show that for each i from 1 to n, v_i is scheduled (by the weighted greedy algorithm) to start at the same time or at an earlier time as in schedule *S*. Let $s'(v_i)$ denote the starting time of v_i in the new schedule produced by the weighted greedy algorithm. The weighted greedy algorithm schedules v_1 to start at time zero; that is, $s'(v_1) = 0$. Then $s'(v_1) \leq s(v_1)$. Suppose $s'(v_1) \leq s(v_1)$, $s'(v_2) \leq s(v_2)$, $s'(v_3) \leq$

$s(v_3)$, ..., and $s'(v_r) \leq s(v_r)$ for some integer r such that $1 \leq r < n$. It is needed to show that $s'(v_{r+1}) \leq s(v_{r+1})$. Let $e'(v_i)$ stand for the ending time of v_i in the new schedule produced by the greedy algorithm, and $e(v_i)$ for the ending time of v_i in schedule *S*. Note that $s'(v_{r+1}) = \max \{e'(v_j)\}$ where v_j is taken over all the vertices that have been scheduled (i.e. $j < r+1$) and adjacent to v_{r+1}. Now suppose that $\max \{e'(v_j)\} = e'(v_t)$ where t must be less than r+1 and v_t is adjacent to v_{r+1}. By the induction hypothesis, one can see $s'(v_t) \leq s(v_t)$; This implies $e'(v_t) \leq e(v_t)$. But $e(v_t) \leq \max \{e(v_j)\}$ where v_j is taken over all the vertices that have been scheduled (i.e. $j < r+1$) and adjacent to v_{r+1}. Therefore, $\max \{e'(v_j)\} \leq \max \{e(v_j)\}$ where v_j is taken over all the vertices that have been scheduled (i.e. $j < r+1$) and adjacent to v_{r+1}. If $j < r+1$, and v_j is adjacent to v_{r+1}, then $e(v_j) \leq s(v_{r+1})$; hence, $\max \{e(v_j)\} \leq s(v_{r+1})$. It follows that $s'(v_{r+1}))\} \leq s(v_{r+1})$. It was shown for each i from 1 to n, v_i is scheduled to start at the same time or at an earlier time as in schedule *S*. Because of this, the total time used according to the weighted greedy algorithm $\leq k$. But it cannot be less than k; otherwise $\chi^*(G) < k$. Therefore, the total time used according to the weighted greedy algorithm must be k.

Q.E.D.

5.4 The Input Data

For complete graphs, bipartite graphs, and odd cycles, the conjecture is true. Therefore, it was not needed to consider those graphs when looking for counter-examples

to the conjecture. For a complete graph G, it is easy to see that $\chi^*(G) = \Delta^*(G) = \delta^*(G) = \max\{\delta^*(H)\}$ where H is taken over all induced subgraphs of G..

Theorem 5.4 For a bipartite graph G, $\chi^*(G) \leq \max\{\delta^*(H)\}$ where H is taken over all induced subgraphs of G.

Proof: Suppose G is bipartite. By theorem 3.2, $\chi^*(G) = \max\{\delta^*(H_1)\}$ where H_1 is over all induced subgraphs of order two. It follows that $\chi^*(G) \leq \max\{\delta^*(H)\}$ where H is over all induced subgraphs of G.

Q.E.D

Theorem 5.5 For an odd cycle G, $\chi^*(G) \leq \max\{\delta^*(H)\}$ where H is taken over all induced subgraphs of G.

Proof: Suppose G is an odd cycle. Theorem 3.3 implies either $\chi^*(G) = \max\{\delta^*(H_1)\}$ where H_1 is over all induced subgraphs of order 2; or $\chi^*(G) = \delta^*(G)$. Therefore, $\chi^*(G) \leq \max\{\delta^*(H)\}$ where H is over all induced subgraphs of G.

Q.E.D.

On pages from 19 to 24 of "Graphs - An Introductory Approach" by Robin J. Wilson and John J. Watkins, there is a list of all the graph cards with six or fewer vertices. The graph cards contain the picture of the graphs, but not their adjacency matrices. In order for the computer to understand the graphs, their adjacency matrices except for those of the complete graphs, the bipartite graphs, and the odd cycles were needed. Each adjacency matrix was braced on a single line so the computer program can read it. Here is the list of all adjacency matrices being fed into the computer program. To help one understand

the input data format, the line for graph card #4 stands for the following adjacency matrix.

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

n = 4

graph card #4: {{0,1,0,0},{1,0,1,1},{0,1,0,1},{0,1,1,0}}
graph card #17 {{0,1,0,1},{1,0,1,1},{0,1,0,1},{1,1,1,0}}

n = 5

graph card #34 {{0,0,1,0,0},{0,0,1,0,0},{1,1,0,1,1},{0,0,1,0,1},{0,0,1,1,0}}
graph card #35 {{0,1,0,0,1},{1,0,1,0,1},{0,1,0,0,0},{0,0,0,0,1},{1,1,0,1,0}}
graph card #36 {{0,1,0,0,0},{1,0,1,0,0},{0,1,0,1,1},{0,0,1,0,1},{0,0,1,1,0}}
graph card #40 {{0,1,0,0,0},{1,0,1,1,1},{0,1,0,1,0},{0,1,1,0,1},{0,1,0,1,0}}

graph card #41 {{0,1,0,0,0},{1,0,1,0,1},{0,1,0,1,1},{0,0,1,0,1},{0,1,1,1,0}}
graph card #42 {{0,0,0,1,1},{0,0,1,1,0},{0,1,0,1,0},{1,1,1,0,1},{1,0,0,1,0}}
graph card #43 {{0,1,0,0,1},{1,0,1,0,1},{0,1,0,1,0},{0,0,1,0,1},{1,1,0,1,0}}
graph card #45 {{0,1,0,0,0},{1,0,1,1,1},{0,1,0,1,1},{0,1,1,0,1},{0,1,1,1,0}}

graph card #46 {{0,1,1,1,1},{1,0,1,0,0},{1,1,0,1,1},{1,0,1,0,0},{1,0,1,0,0}}
graph card #47 {{0,1,0,0,1},{1,0,1,1,1},{0,1,0,1,0},{0,1,1,0,1},{1,1,0,1,0}}
graph card #48 {{0,1,0,0,1},{1,0,1,1,0},{0,1,0,1,1},{0,1,1,0,1},{1,0,1,1,0}}
graph card #49 {{0,1,1,0,1},{1,0,1,0,1},{1,1,0,1,1},{0,0,1,0,1},{1,1,1,1,0}}

graph card #50 {{0,1,1,0,1},{1,0,1,1,0},{1,1,0,1,1},{0,1,1,0,1},{1,0,1,1,0}}
graph card #51 {{0,1,1,1,1},{1,0,1,1,0},{1,1,0,1,1},{1,1,1,0,1},{1,0,1,1,0}}

n = 6

#92 {{0,0,0,0,0,1},{0,0,0,0,0,1},{0,0,0,0,0,1},{0,0,0,0,1,1},{0,0,0,1,0,1},{1,1,1,1,1,0}}
#93 {{0,0,1,0,0,0},{0,0,1,0,0,0},{1,1,0,0,1,1},{0,0,0,0,1,0},{0,0,1,1,0,1},{0,0,1,0,1,0}}
#94 {{0,1,0,0,0,0},{1,0,1,0,1,0},{0,1,0,1,1,0},{0,0,1,0,0,0},{0,1,1,0,0,1},{0,0,0,0,1,0}}

#96 {{0,1,1,0,0,0},{1,0,1,0,0,0},{1,1,0,1,1,0},{0,0,1,0,0,0},{0,0,1,0,0,1},{0,0,0,0,1,0}}
#99 {{0,1,0,0,0,0},{1,0,1,1,0,0},{0,1,0,1,0,0},{0,1,1,0,1,0},{0,0,0,1,0,1},{0,0,0,0,1,0}}
#101 {{0,1,0,0,0,0},{1,0,1,1,0,0},{0,1,0,0,0,0},{0,1,0,0,1,1},{0,0,0,1,0,1},{0,0,0,1,1,0}}
#102 {{0,1,1,0,0,0},{1,0,1,0,0,0},{1,1,0,1,0,0},{0,0,1,0,1,0},{0,0,0,1,0,1},{0,0,0,0,1,0}}

#104 {{0,1,0,0,0,0},{1,0,1,0,0,1},{0,1,0,1,0,0},{0,0,1,0,1,0},{0,0,0,1,0,1},{0,1,0,0,1,0}}
#111 {{0,1,1,1,0,0},{1,0,1,0,0,0},{1,1,0,1,1,1},{1,0,1,0,0,0},{0,0,1,0,0,0},{0,0,1,0,0,0}}
#112 {{0,1,0,0,0,0},{1,0,1,1,0,1},{0,1,0,1,0,0},{0,1,1,0,1,1},{0,0,0,1,0,0},{0,1,0,1,0,0}}
#113 {{0,1,0,0,0,0},{1,0,1,0,1,0},{0,1,0,1,1,0},{0,0,1,0,1,0},{0,1,1,1,0,1},{0,0,0,0,1,0}}

#114 {{0,1,0,1,0,0},{1,0,1,1,0,0},{0,1,0,1,1,1},{1,1,1,0,0,0},{0,0,1,0,0,0},{0,0,1,0,0,0}}
#116 {{0,1,0,0,0,0},{1,0,1,0,0,1},{0,1,0,1,0,1},{0,0,1,0,1,1},{0,0,0,1,0,0},{0,1,1,1,0,0}}
#117 {{0,1,1,0,0,0},{1,0,1,0,0,0},{1,1,0,1,1,1},{0,0,1,0,1,0},{0,0,1,1,0,0},{0,0,1,0,0,0}}
#118 {{0,1,1,0,0,0},{1,0,1,0,0,0},{1,1,0,1,1,0},{0,0,1,0,1,0},{0,0,1,1,0,1},{0,0,0,0,1,0}}

#119 {{0,1,0,1,0,0},{1,0,1,1,0,0},{0,1,0,1,0,0},{1,1,1,0,1,0},{0,0,0,1,0,1},{0,0,0,0,1,0}}
#120 {{0,1,1,0,1,0},{1,0,0,1,0,0},{1,0,0,1,0,0},{0,1,1,0,1,1},{1,0,0,1,0,0},{0,0,0,1,0,0}}
#121 {{0,1,0,1,1,0},{1,0,1,0,0,0},{0,1,0,1,0,0},{1,0,1,0,1,1},{1,0,0,1,0,0},{0,0,0,1,0,0}}
#122 {{0,1,1,1,0,0},{1,0,1,0,0,0},{1,1,0,1,0,0},{1,0,1,0,1,0},{0,0,0,1,0,1},{0,0,0,0,1,0}}

#123 {{0,1,0,0,1,0},{1,0,1,0,1,0},{0,1,0,1,0,0},{0,0,1,0,1,1},{1,1,0,1,0,0},{0,0,0,1,0,0}}
#124 {{0,1,0,1,0,0},{1,0,1,0,1,0},{0,1,0,1,0,0},{1,0,1,0,1,0},{0,1,0,1,0,1},{0,0,0,0,1,0}}
#125 {{0,1,0,0,1,0},{1,0,1,1,0,0},{0,1,0,1,0,1},{0,1,1,0,1,0},{1,0,0,1,0,0},{0,0,1,0,0,0}}
#126 {{0,1,0,1,0,0},{1,0,1,0,0,0},{0,1,0,1,0,0},{1,0,1,0,1,1},{0,0,0,1,0,1},{0,0,0,1,1,0}}

#127 {{0,1,1,0,0,0},{1,0,1,0,0,0},{1,1,0,1,0,0},{0,0,1,0,1,1},{0,0,0,1,0,1},{0,0,0,1,1,0}}
#129 {{0,1,0,0,0,1},{1,0,1,1,0,0},{0,1,0,1,0,0},{0,1,1,0,1,0},{0,0,0,1,0,1},{1,0,0,0,1,0}}
#130 {{0,1,0,1,1,0},{1,0,1,0,0,0},{0,1,0,1,0,1},{1,0,1,0,0,0},{1,0,0,0,0,1},{0,0,1,0,1,0}}
#133 {{0,1,1,1,0,0},{1,0,1,1,0,0},{1,1,0,1,0,0},{1,1,1,0,1,1},{0,0,0,1,0,0},{0,0,0,1,0,0}}

#134 {{0,1,0,0,0,0},{1,0,1,1,1,0},{0,1,0,1,1,0},{0,1,1,0,1,0},{0,1,1,1,0,1},{0,0,0,0,1,0}}
#135 {{0,1,1,1,1,0},{1,0,0,0,1,0},{1,0,0,0,1,0},{1,0,0,0,1,0},{1,1,1,1,0,1},{0,0,0,0,1,0}}
#136 {{0,1,1,0,1,0},{1,0,1,0,0,0},{1,1,0,1,1,1},{0,0,1,0,1,0},{1,0,1,1,0,0},{0,0,1,0,0,0}}
#137 {{0,1,1,1,1,0},{1,0,1,0,0,0},{1,1,0,1,0,1},{1,0,1,0,1,0},{1,0,0,1,0,0},{0,0,1,0,0,0}}

#138 {{0,1,0,1,0,0},{1,0,1,1,1,0},{0,1,0,1,0,0},{1,1,1,0,1,0},{0,1,0,1,0,1},{0,0,0,0,1,0}}
#139 {{0,1,1,1,0,0},{1,0,1,1,0,0},{1,1,0,1,0,0},{1,1,1,0,1,0},{0,0,0,1,0,1},{0,0,0,0,1,0}}
#140 {{0,1,0,0,1,0},{1,0,1,1,1,0},{0,1,0,1,0,1},{0,1,1,0,1,0},{1,1,0,1,0,0},{0,0,1,0,0,0}}
#141 {{0,1,0,1,1,0},{1,0,1,1,0,0},{0,1,0,1,1,0},{1,1,1,0,0,1},{1,0,1,0,0,0},{0,0,0,1,0,0}}

#142 {{0,1,1,1,0,0},{1,0,0,0,1,0},{1,0,0,1,1,0},{1,0,1,0,1,0},{0,1,1,1,0,1},{0,0,0,0,1,0}}
#143 {{0,1,1,1,0,0},{1,0,1,0,1,0},{1,1,0,1,0,0},{1,0,1,0,1,0},{0,1,0,1,0,1},{0,0,0,0,1,0}}
#144 {{0,1,0,1,0,0},{1,0,1,1,0,0},{0,1,0,1,0,0},{1,1,1,0,1,1},{0,0,0,1,0,1},{0,0,0,1,1,0}}
#145 {{0,1,1,1,0,1},{1,0,1,0,0,0},{1,1,0,1,1,0},{1,0,1,0,0,0},{0,0,1,0,0,1},{1,0,0,0,1,0}}

#147 {{0,1,1,0,1,1},{1,0,1,0,0,0},{1,1,0,1,0,0},{0,0,1,0,1,0},{1,0,0,1,0,1},{1,0,0,0,1,0}}
#148 {{0,1,0,1,0,1},{1,0,1,0,0,0},{0,1,0,1,1,1},{1,0,1,0,0,0},{0,0,1,0,0,1},{1,0,1,0,1,0}}
#149 {{0,1,0,1,1,1},{1,0,1,0,0,0},{0,1,0,1,0,0},{1,0,1,0,1,0},{1,0,0,1,0,1},{1,0,0,0,1,0}}
#150 {{0,1,1,1,0,0},{1,0,1,0,0,0},{1,1,0,1,0,0},{1,0,1,0,1,1},{0,0,0,1,0,1},{0,0,0,1,1,0}}

#151 {{0,1,0,0,1,1},{1,0,1,1,0,0},{0,1,0,1,0,0},{0,1,1,0,1,0},{1,0,0,1,0,1},{1,0,0,0,1,0}}
#153 {{0,1,0,1,0,1},{1,0,1,1,0,0},{0,1,0,1,1,0},{1,1,1,0,0,0},{0,0,1,0,0,1},{1,0,0,0,1,0}}
#154 {{0,1,0,0,1,0},{1,0,1,0,0,1},{0,1,0,1,0,0},{0,0,1,0,1,1},{1,0,0,1,0,1},{0,1,0,1,1,0}}
#156 {{0,1,1,1,1,0},{1,0,1,0,1,0},{1,1,0,1,1,1},{1,0,1,0,0,0},{1,1,1,0,0,0},{0,0,1,0,0,0}}

#157 {{0,1,0,0,0,0},{1,0,1,0,1,1},{0,1,0,1,1,1},{0,0,1,0,1,0},{0,1,1,1,0,1},{0,1,1,0,1,0}}
#158 {{0,1,0,1,1,0},{1,0,1,0,1,0},{0,1,0,1,1,0},{1,0,1,0,1,0},{1,1,1,1,0,1},{0,0,0,0,1,0}}
#159 {{0,1,1,1,0,0},{1,0,1,0,1,0},{1,1,0,1,1,0},{1,0,1,0,1,0},{0,1,1,1,0,1},{0,0,0,0,1,0}}
#160 {{0,1,1,1,0,0},{1,0,1,1,1,0},{1,1,0,1,1,0},{1,1,1,0,0,0},{0,1,1,0,0,1},{0,0,0,0,1,0}}

#161 {{0,1,1,1,1,1},{1,0,0,0,0,1},{1,0,0,0,0,1},{1,0,0,0,0,1},{1,0,0,0,0,1},{1,1,1,1,1,0}}
#162 {{0,1,1,1,0,1},{1,0,1,0,0,0},{1,1,0,1,1,1},{1,0,1,0,0,0},{0,0,1,0,0,1},{1,0,1,0,1,0}}
#163 {{0,1,1,0,1,1},{1,0,1,0,0,0},{1,1,0,1,1,0},{0,0,1,0,1,0},{1,0,1,1,0,1},{1,0,0,0,1,0}}
#164 {{0,1,1,1,0,0},{1,0,1,1,0,0},{1,1,0,1,1,1},{1,1,1,0,0,0},{0,0,1,0,0,1},{0,0,1,0,1,0}}

#165 {{0,1,1,1,1,1},{1,0,1,0,0,0},{1,1,0,1,0,0},{1,0,1,0,1,0},{1,0,0,1,0,1},{1,0,0,0,1,0}}
#166 {{0,1,0,0,1,1},{1,0,1,1,1,0},{0,1,0,1,0,0},{0,1,1,0,1,0},{1,1,0,1,0,1},{1,0,0,0,1,0}}
#167 {{0,1,0,1,0,1},{1,0,1,1,1,0},{0,1,0,1,0,0},{1,1,1,0,1,0},{0,1,0,1,0,1},{1,0,0,0,1,0}}
#168 {{0,1,0,1,1,1},{1,0,1,0,0,0},{0,1,0,1,1,1},{1,0,1,0,0,0},{1,0,1,0,0,1},{1,0,1,0,1,0}}

#169 {{0,1,0,0,1,1},{1,0,1,0,1,1},{0,1,0,1,0,0},{0,0,1,0,1,0},{1,1,0,1,0,1},{1,1,0,0,1,0}}
#170 {{0,1,0,0,1,0},{1,0,1,0,1,1},{0,1,0,1,0,1},{0,0,1,0,1,0},{1,1,0,1,0,1},{0,1,1,0,1,0}}
#171 {{0,1,0,0,1,1},{1,0,1,0,0,0},{0,1,0,1,0,1},{0,0,1,0,1,1},{1,0,0,1,0,1},{1,0,1,1,1,0}}
#172 {{0,1,0,0,1,1},{1,0,1,1,0,1},{0,1,0,1,0,0},{0,1,1,0,1,0},{1,0,0,1,0,1},{1,1,0,0,1,0}}

#173 {{0,1,0,1,1,1},{1,0,1,0,0,0},{0,1,0,1,0,1},{1,0,1,0,1,0},{1,0,0,1,0,1},{1,0,1,0,1,0}}
#174 {{0,1,0,1,1,0},{1,0,1,0,0,1},{0,1,0,1,0,1},{1,0,1,0,1,0},{1,0,0,1,0,1},{0,1,1,0,1,0}}
#177 {{0,1,1,1,1,0},{1,0,1,1,0,0},{1,1,0,1,1,1},{1,1,1,0,1,0},{1,0,1,1,0,0},{0,0,1,0,0,0}}
#178 {{0,1,1,1,1,0},{1,0,1,1,0,1},{1,1,0,1,1,0},{1,1,1,0,1,0},{1,0,1,1,0,0},{0,1,0,0,0,0}}

#179 {{0,1,1,1,0,0},{1,0,1,1,1,1},{1,1,0,1,1,1},{1,1,1,0,0,0},{0,1,1,0,0,0},{0,1,1,0,0,0}}
#180 {{0,1,0,0,1,0},{1,0,1,1,1,1},{0,1,0,1,0,0},{0,1,1,0,1,1},{1,1,0,1,0,1},{0,1,0,1,1,0}}
#181 {{0,1,0,1,1,1},{1,0,1,1,1,0},{0,1,0,1,0,0},{1,1,1,0,1,0},{1,1,0,1,0,1},{1,0,0,0,1,0}}
#182 {{0,1,0,0,0,1},{1,0,1,1,1,1},{0,1,0,1,0,1},{0,1,1,0,1,0},{0,1,0,1,0,1},{1,1,1,0,1,0}}

#183 {{0,1,1,1,1,0},{1,0,1,1,1,1},{1,1,0,1,0,0},{1,1,1,0,0,0},{1,1,0,0,0,1},{0,1,0,0,1,0}}
#184 {{0,1,0,0,1,0},{1,0,1,0,1,1},{0,1,0,1,0,1},{0,0,1,0,1,1},{1,1,0,1,0,1},{0,1,1,1,1,0}}
#185 {{0,1,0,1,1,1},{1,0,0,0,1,0},{0,1,0,1,1,1},{1,0,1,0,1,0},{1,0,1,1,0,1},{1,0,1,0,1,0}}
#186 {{0,1,0,0,1,1},{1,0,1,0,1,1},{0,1,0,1,0,1},{0,0,1,0,1,0},{1,1,0,1,0,1},{1,1,1,0,1,0}}

#187 {{0,1,0,0,1,1},{1,0,1,0,0,1},{0,1,0,1,0,1},{0,0,1,0,1,1},{1,0,0,1,0,1},{1,1,1,1,1,0}}
#188 {{0,1,1,0,0,1},{1,0,1,0,0,1},{1,1,0,1,1,0},{0,0,1,0,1,1},{0,0,1,1,0,1},{1,1,0,1,1,0}}
#189 {{0,1,1,0,1,1},{1,0,1,1,0,1},{1,1,0,1,0,0},{0,1,1,0,1,0},{1,0,0,1,0,1},{1,1,0,0,1,0}}
#190 {{0,1,0,1,0,1},{1,0,1,0,1,1},{0,1,0,1,0,1},{1,0,1,0,1,0},{0,1,0,1,0,1},{1,1,1,0,1,0}}

#191 {{0,1,1,1,1,0},{1,0,1,1,1,1},{1,1,0,1,1,0},{1,1,1,0,1,0},{1,1,1,1,0,0},{0,1,0,0,0,0}}
#192 {{0,1,1,1,0,0},{1,0,1,1,1,1},{1,1,0,1,1,1},{1,1,1,0,1,0},{0,1,1,1,0,0},{0,1,1,0,0,0}}
#193 {{0,1,1,1,1,0},{1,0,1,1,0,1},{1,1,0,1,1,1},{1,1,1,0,1,0},{1,0,1,1,0,0},{0,1,1,0,0,0}}
#194 {{0,1,1,1,1,0},{1,0,1,1,1,0},{1,1,0,0,1,1},{1,1,0,0,1,1},{1,1,1,1,0,0},{0,0,1,1,0,0}}

#195 {{0,1,1,0,0,1},{1,0,1,0,0,1},{1,1,0,1,1,1},{0,0,1,0,1,1},{0,0,1,1,0,1},{1,1,1,1,1,0}}
#196 {{0,1,1,1,1,1},{1,0,1,0,1,0},{1,1,0,1,0,1},{1,0,1,0,1,0},{1,1,0,1,0,1},{1,0,1,0,1,0}}
#197 {{0,1,0,1,1,1},{1,0,1,1,0,0},{0,1,0,1,0,1},{1,1,1,0,1,1},{1,0,0,1,0,1},{1,0,1,1,1,0}}
#198 {{0,1,0,1,1,1},{1,0,1,1,1,0},{0,1,0,1,0,1},{1,1,1,0,1,0},{1,1,0,1,0,1},{1,0,1,0,1,0}}

#199 {{0,1,1,1,1,0},{1,0,1,0,0,1},{1,1,0,1,0,1},{1,0,1,0,1,1},{1,0,0,1,0,1},{0,1,1,1,1,0}}
#200 {{0,1,1,1,1,0},{1,0,1,1,1,0},{1,1,0,1,1,1},{1,1,1,0,1,1},{1,1,1,1,0,0},{0,0,1,1,0,0}}
#201 {{0,1,0,1,0,1},{1,0,1,1,1,1},{0,1,0,1,0,1},{1,1,1,0,1,1},{0,1,0,1,0,1},{1,1,1,1,1,0}}
#202 {{0,1,1,1,1,1},{1,0,1,1,1,0},{1,1,0,1,0,0},{1,1,1,0,1,1},{1,1,0,1,0,1},{1,0,0,1,1,0}}

#203 {{0,1,1,1,1,1},{1,0,1,0,1,1},{1,1,0,1,0,1},{1,0,1,0,1,0},{1,1,0,1,0,1},{1,1,1,0,1,0}}

#204 {{0,1,1,1,1,0},{1,0,1,0,1,1},{1,1,0,1,0,1},{1,0,1,0,1,1},{1,1,0,1,0,1},{0,1,1,1,1,0}}
#205 {{0,1,1,1,1,1},{1,0,1,0,1,1},{1,1,0,1,1,1},{1,0,1,0,1,0},{1,1,1,1,0,1},{1,1,1,0,1,0}}
#206 {{0,1,1,0,1,1},{1,0,1,1,0,1},{1,1,0,1,1,1},{0,1,1,0,1,1},{1,0,1,1,0,1},{1,1,1,1,1,0}}

#207 {{0,1,1,1,1,1},{1,0,1,1,1,1},{1,1,0,1,1,0},{1,1,1,0,1,1},{1,1,1,1,0,1},{1,1,0,1,1,0}}

For each of these graphs, the computer program assigned either the weight 1 or 2 to each vertex. For instance, one can feed the adjacency matrix of graph card #207 with 6 vertices into the program. The program automatically assigns a weight of 1 or 2 to each vertex; and hence the program generates 64 (i.e. 2^6) weighted graphs to look at. For a graph with 5 vertices, the program will automatically generates 32 weighted graphs to examine.

6. When is $\chi^*(G) = \Delta^*(G)$?

Theorem 4.3 says that $\chi(G) \leq 1 + \Delta(G)$ for any connected graph G. Brook's Theorem can be found in most graph theory books. Brook's Theorem says that $\chi(G) \leq \Delta(G)$ if and only if G is neither a complete graph nor an odd cycle. Theorem 4.3 and Brook's Theorem together imply that **$\chi(G) = 1 + \Delta(G)$ if and only if G is either a complete graph or an odd cycle.**

On the other hand, Theorem 4.4 says that $\chi^*(G) \leq \Delta^*(G)$ for any connected weighted graph G. When is that upper bound reached? That's, when is $\chi^*(G) = \Delta^*(G)$? **Is it as simple as $\chi^*(G) = \Delta^*(G)$ if and only if G is either a complete graph or an odd cycle?** The answer is no. The following two results were found so far.

1) If G is a complete weighted graph, then $\chi^*(G) = \Delta^*(G)$.
2) If G is a regular weighted odd cycle, then $\chi^*(G) = \Delta^*(G)$ where regular weighted odd cycle means that every vertex of the odd cycle has the same weighted degree.

The first result is easily seen. The second result follows immediately from Theorem 3.3. It is suspected that there are other graphs other than complete graphs and regular weighted odd cycles such that their weighted chromatic number is equal to their weighted maximum degree. A computer program is written to search for other G such that $\chi^*(G) = \Delta^*(G)$. This program is just a modification of the program written earlier in section 5.3, and this new program is shorter because $\Delta^*(G)$ is much easier to find than $\max\{\delta^*(H)\}$. Please refer to Appendix 7.4 for the source codes of the program. Most of the 6-vertex graphs with weights 1, 2 or 3 have been fed into the program; but none of

them satisfied $\chi^*(G) = \Delta^*(G)$. $\chi^*(G)$ and $\Delta^*(G)$ are close to each other for all permutations of the weights 1, 2 or 3 if G is any one of the graphs in graph card #186, 191, 194, 198, 200, 205, and 207. Perhaps one can try to build a graph of order 7 (7 vertices) or higher order based on the characteristics of those graphs; and see whether it has the property that $\chi^*(G) = \Delta^*(G)$.

7. APPENDIX

7.1 The Computer Program Used to Find Counter-examples

```
// By Sandy Zhang
// program FO_sht_n6MaxMin&Wght_Chrmtc
// Output is sent to a file.

// In this program, a lot of details are commented out to make run time short.
// That is, the program prints out only the important messages, and it skips
// printing most of the intermediate results. Those comments can be brought
// back to the program anytime if it is desirable to do so in the future.

// This program looks at an input graph of six vertices, and assigns either
// weight 1 or 2 for each vertex. So it will look at all the 64 weighted
// graphs associcated with the original graph. For each of those 64 weighted
// graphs, the program calculates the weighted chromatic number and the max of
// the min weighted degree of H where H is taken over all induced subgraphs of
// G (the term MaxMin will be used throughout the program.) If the weighted
// chromatic number is greater than the MaxMin, a counter-example is found.


#include <fstream.h>
#include <math.h>

ofstream fout("card94MaxMin&WghtChrmtc.txt");  // name of the output file

const n = 6;   // 6 vertices

bool makeComparision (int [][n]);
int findMaxMin (int [][n]);
int findWeightedChromatic(int gMatrix[][n]);
int applyGreedy(int [][n]);

void generateNewMatrix(int [][n], int [][n], int [n]);
void initialArray(int [n]);
void printStartTime(int [n]);
void printMatrix(int [][n]);
void initialize(int [n]);
void printArr(int [n]);

int factorial(int);
int findBigJ(int [n]);
int findBigK(int, int [n]);
void sortJPlusOneOn(int, int [n]);
int positionOf(int,int Arr[n]);
int minDeg(int[][n]);
void SubGraph(int [],int[][n]);
```

```
int main()
{
    int incidentMatrix[n][n] = {{0,1,0,0,0,0},{1,0,1,0,1,0},{0,1,0,1,1,0},{0,0,1,0,0,0},
                                {0,1,1,0,0,1},{0,0,0,0,1,0}};
    // That's the input for the incident matrix of graph card #163.

    int numOfTotalGraphs = 0;  // numbers of weighted graphs we have looked
                               // through for the input graph
    int numOfCounterEx = 0;    // number of counter-examples found
    bool counterExampleFound = false;
    for (int indexA = 1; indexA <= 2; indexA++)  // assign weights 1 or 2 to
                                                 // each vertex until all
    {                                            // permutations are exhausted.
        incidentMatrix[0][0] = indexA;
        for (int indexB = 1; indexB <= 2; indexB++)
        {
            incidentMatrix[1][1] = indexB;
            for (int indexC = 1; indexC <= 2; indexC++)
            {
                incidentMatrix[2][2] = indexC;
                for (int indexD = 1; indexD <= 2; indexD++)
                {
                    incidentMatrix[3][3] = indexD;
                    for (int indexE = 1; indexE <= 2; indexE++)
                    {
                        incidentMatrix[4][4] = indexE;
                        for (int indexF = 1; indexF <= 2; indexF++)
                        {
                            incidentMatrix[5][5] = indexF;
                            counterExampleFound = makeComparision(incidentMatrix);
                            // to see if the weighted graph is a
                            // counter-example by making a comparison of the
                            // weighted chromatic number with the MaxMin.

                            if (counterExampleFound)  // if a counter-example
                                                      // is found, update the
                                numOfCounterEx++;  // numOfCounterEx counter by 1.

                            numOfTotalGraphs++; // keeping track of how many
                                                //weighted graphs being looked at
                        }
                    }
                }
            }
        }
    }

    fout<<endl<<"!!!!!!!!!!!! CONCLUSION !!!!!!!!!!!!"<<endl;
    fout<<"Number of total graphs we have looked through is "
        <<numOfTotalGraphs<<"."<<endl;
    fout<<"Number of counter-examples found from these graphs is "
```

```
            <<numOfCounterEx<<"."<<endl;
        return 0;
}


bool makeComparision (int graphMatrix[][n])
// This function takes a weighted graph.  It calls two other functions
// - findMaxMin and findWeightedChromatic.  It then compares the two results.
// Based on the comparison, it returns the "true" boolean value if a
// counter-example is found; otherwise it returns the "false" boolean value.
{
    bool counterExFound = false;
    int maxOfMinWghtDeg = findMaxMin(graphMatrix);
    int weightedChromatic = findWeightedChromatic(graphMatrix);

    if (weightedChromatic > maxOfMinWghtDeg)
    {
        counterExFound = true;
        fout<<endl<<"***** Summary:  Yes, here is a counter-example!  *****"
            <<endl;
        fout<<"This following matrix has the Weighted Chromatic Number greater than"
            <<endl
            <<"the Maximum of the Minimum Weighted Degree over all the induced subgraphs."
            <<endl;
        printMatrix(graphMatrix);
    }
    else
        fout<<endl<<"***** Summary:  not a counter-example *****"<<endl;

    fout<<"The Maximum of the Minimum Weighted Degree over all the induced subgraphs is "
        <<maxOfMinWghtDeg<<"."<<endl;
    fout<<"The Weighted Chromatic Number is "<<weightedChromatic<<"."<<endl;
    fout<<"##########################################################################"
        <<endl<<endl;
    return counterExFound;
}


int findMaxMin (int gMatrix[][n])
// This function returns the MaxMin of an input weighted graph.
{
    int MaxMin;
    int hMatrix[n][n];  // the adjacency matrix of the induced subgraph H
    bool done;
    int Bstr[n];   // Bstr stands for Binary String; this is needed to let
                   // the program know which induced subgraph to look at next.

    for (int i = 0; i < n; i++)
        Bstr[i] = 0;  // initialize the Bstr string to be all zeroes
```

```
//    fout<<endl;
//    fout<<"Start to look for the Maximum of the Minimum Weighted Degree over "
//       <<"all induced subgraphs..."<<endl<<endl;
//    fout<<"The matrix being consideration is:"<<endl<<endl;
//    printMatrix(gMatrix);
//    fout<<endl;
      MaxMin = minDeg(gMatrix);    // call the function minDeg to find out the minimum
                                   // degree of the weighted graph; and assign this
                                   // value to MaxMin for now.
//    fout<<"The minimum weighted degree of the graph associated with this matrix is "
//        <<MaxMin<<"."<<endl<<endl;


      for (int s = 0; s < (pow(2,n) - 2); s++)
      // This for loop counts the binary string from all zeroes
      // to all ones by the usual oder.
      {
          int index = n-1;  // always start at the right most digit
          done = false;
          while (!(done))
          {
              if (Bstr[index] == 0)  // if the digit is a zero, change it to an "1";
                                     // then we are done counting to the next binary number.
              {
                  Bstr[index] = 1;
                  done = true;
              }
              else  // if the digit is an "1", change it to a "0"; then decrease
                    // the index by one. That's, the same while loop will be
                    // performed again for the digit just to the left of the
                    // current digit.
              {
                  Bstr[index] = 0;
                  index--;
              }
          }
//        printArr(Bstr);
          for (i = 0; i < n; i++)
              for (int j = 0; j < n; j++)
                  hMatrix[i][j] = gMatrix[i][j]; // make the adjacency matrix of
                  // the induced subgraph to be the same as the adjacency matrix
                  // of the graph for now; so that we just need a slight
                  // modification (call the function SubGraph) to get the
                  // hMatrix right.

          SubGraph(Bstr,hMatrix); // call SubGraph to come up with the correct hMatrix
//        fout<<"The matrix of a subgraph is:"<<endl<<endl;
//        printMatrix(hMatrix);
          int minD = minDeg(hMatrix);
//        fout<<"minDeg(hMatrix) is "<<minD<<endl;
          if (MaxMin < minD)
```

```
            MaxMin = minD;

//          fout<<endl;
    }

//  fout<<"*** MaxMin is "<<MaxMin<<". ***"<<endl<<endl;
//  fout<<"=========================================================================="<<endl<<endl;

    return MaxMin;
}


int findWeightedChromatic(int gMatrix[][n])
// This function returns the weighted chromatic number of an input weighted graph.
// To calculate the weighted chromatic number of a graph G, we rename the vertices.
// That is, we re-order the vertices of G; call them v1, v2, v3, ... again.  Apply
// the weighted greedy algorithm to each ordering of all the vertices, to find out
// the number of hours needed for each ordering.  The minimum number of hours taken
// over all those orderings is the weighted chromatic number.
{

    int numArr[n];  // a permutation of vertices in term of their subscripts
    int count = 1;
    int BIGj, BIGk, temp;
    int wghtChromatic;
    int newMatrix[n][n];  // a new ajacency weighted matrix for a new permuation
                          // of vertices
    int totalTime;

    initialize(numArr);  // initialize numArr to be the permutation 1, 2, 3, ... n.

//  fout<<"Start to look for the Weighted Chromatic Number..."<<endl;
//  printArr(numArr);
//  fout<<"The matrix being consideration is:"<<endl<<endl;
//  printMatrix(gMatrix);
    totalTime = applyGreedy(gMatrix);  // apply the weighted greedy algorithm to
                                       // the graph with its original ordering of
                                       // vertices to find out the number of
                                       // hours needed for this ordering.
//  fout<<"The number of colors needed by the greedy algorithm"<<endl
//      <<"according to this arrangement of vertices is "<<totalTime<<"."<<endl;
    wghtChromatic = totalTime;

    // The following do loop generates all the permutations of the vertices.
    // For each permutation, a new incident weighted matrix has to be created;
    // then the weighted greedy algorithm is applied by calling the function
    // applyGreedy.
    do
    {
        // The following six lines, and the line "count++" are based on the
        // algorithm stated on page 126 of "Applied Combinatorics" by Jackson
```

```
            // and Thoro. This algorithm generates all permuations of {1,2,3, ..., n}.
            // Here it is:
            // 0. Input n.
            // 1. Let i = 1 and p1 = 1,2,3,...n.
            // 2. Output p1 and stop if i = n! .
            // 3. If p1 = X1,X2,X3,..., Xn, find the largest j so that Xj < Xj+1.
            //    (Thus Xj is the last digit that is smaller than some digit after it,
            //    and the digits after Xj are arranged in decreasing order.)
            // 4. Find the largest k so that Xk > Xj.
            // 5. Increase i by 1. The new permutation pi is obtained by
            //    interchanging Xk and Xj and arranging the digits after the jth
            //    digit in increasing order.
            //    (Thus pi = X1,X2,...,Xj-1,Xk,Xn,Xn-1,...,Xk+1,Xj,Xk-1,...,Xj+1.)
            //    Return to Step 2.


            BIGj = findBigJ(numArr);
            BIGk = findBigK(BIGj,numArr);
            temp = numArr[BIGj];
            numArr[BIGj] = numArr[BIGk];
            numArr[BIGk] = temp;
            sortJPlusOneOn(BIGj,numArr);
//          printArr(numArr);
            generateNewMatrix(gMatrix,newMatrix,numArr);
//          printMatrix(newMatrix);
            totalTime = applyGreedy(newMatrix);
            if (wghtChromatic > totalTime)
                wghtChromatic = totalTime;
//          fout<<"The number of colors needed by the greedy algorithm"<<endl
//              <<"according to this arrangement of vertices is "<<totalTime<<"."<<endl;
            count++;
        } while (count < factorial(n));

//      fout<<endl<<"*** The Weighted Chromatic number is "<<wghtChromatic<<". ***"<<endl;
        return wghtChromatic;
}


int minDeg(int a[][n])
// This function returns the minimum degree of an induced subgraph.

{
//      fout<<endl;
        int min;
        int count = 0;
        for (int i = 0; i < n; i++)
        {
            if (a[i][i] != -1)  // If vi is not one of the deleted vertices, then
                                // proceed to calculate its weighted degree.
            {
                count++;
```

```
                int deg = a[i][i];      // assign the weight of vi to the weighted
                                        // degree of vi temporarily. The next for loop
                                        // will update the weighted degree of vi based
                                        // on its neighbors (its adjacent vertices).

                for (int j = 0; j < n; j++)
                {
                    if (i != j)
                    {
                        if (a[i][j] == 1)
                            deg = deg + a[j][j];
                    }
                }
//              fout<<"Weighted Degree of V"<<i+1<<" = ";
//              fout<<deg<<endl;
                if (count == 1)     // If that's the very first weighted degree being
                                    // calculated, we have to treat it as the min so far.
                    min = deg;
                else
                    if (deg < min)
                    min = deg;
            }
        }
        return min;
}


void SubGraph(int B[],int hMatr[][n])
// Which induced subgraph to look at next depends on what binary string
// we have next. hMatr stands for the incident weighted matrix of the
// induced subgraph H and B stands for the binary string.  If there is
// a "1" on the Kth position of the B string, delete vertex vk from the
// weighted graph.  To indicate that a vertex is deleted, we just assign
//  a "-1" on the entire Kth column and the entire Kth row in the hMatr.
{
    for (int k = 0; k < n; k++)
    {
        if (B[k] == 1)
        {
            for (int j = 0; j < n; j++)
                hMatr[k][j] = -1;
            for (int i = 0; i < n; i++)
                hMatr[i][k] = -1;
        }
    }
}

void initialize(int Arr[n])
// intitialize the array Arr to be 1, 2, 3,..., n.
{
    for (int i = 0; i < n; i++)
```

```
            Arr[i] = i + 1;
}

void printArr(int Arr[n])
// print out the entire array
{
     fout<<endl<<endl;
     for (int i = 0; i < n; i++)
          fout<<Arr[i]<<" ";

     fout<<endl<<endl;
}

int factorial(int num)
// calculate the factorial of a number num
{

     if (num <= 1)
          return 1;
     else
          return num*factorial(num-1);
}

int findBigJ(int Arr[n])
// Please read the algorithm documented on the function findWeightedChromatic.
{
     int indexJ = -1;
     for (int i = 0; i < n-1; i++)
     {
          if (Arr[i] < Arr[i+1])
               indexJ = i;
     }
     return indexJ;
}

int findBigK(int indexJ, int Arr[n])
// Please read the algorithm documented on the function findWeightedChromatic.
{
     int indexK;
     for (int i = indexJ+1; i < n; i++)
     {
          if (Arr[i] > Arr[indexJ])
               indexK = i;
     }
     return indexK;
}

void sortJPlusOneOn(int ind_j, int Arr[n])
// Please read the algorithm documented on the function findWeightedChromatic.
// Sort the (j+1)th to (n-1)th elements of the array in increasing (actually
// non-decreasing) order.  It uses "bubble sort."  On each pass, successive pairs
```

```
// of elements are compared. If a pair is in non-decreasing order, we leave the
// values as they are. If a pair is in decreasing order, their values are swapped
// in the array.
{
    int hold;

    for (int pass = ind_j+1; pass < n-1; pass++)
    {
        for (int index = ind_j+1; index < n-1; index++)
            if (Arr[index] > Arr[index+1])
            {
                hold = Arr[index];
                Arr[index] = Arr[index+1];
                Arr[index+1] = hold;
            }
    }
}


void generateNewMatrix(int origMatrix[][n], int permMatrix[][n], int permArr[n])
// generate a new weighted ajacency matrix according to the new permuation of
// the vertices
{
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
            permMatrix[i][j] = origMatrix[positionOf(i+1,permArr)][positionOf(j+1,permArr)];
    }
}

int positionOf(int x,int Arr[n])
// find the position of x in an array Arr
{
    for (int i = 0; i < n; i++)
    {
        if (Arr[i] == x)
            return i;
    }
}


int applyGreedy(int matrix[][n])
// Apply the weighted greedy algorithm to a weighted graph with a particular
// ordering represented by the weighted adjacency matrix. This function will
// return the number of hours needed to make a schedule according to the
// weighted greedy algorithm. The weighted greedy algorithm for scheduling
// works as follows. First schedule vertex(task) v1 to start at time 0.
// We schedule all the vertices in the increasing order of their subscripts.
// To schedule each vi, we looked at all the neighbors of vi which have been
// scheduled; that's we look at only the neighbors of vi with the subscript
// less than vi. Out of all those neighbors we look at, we pick out the
```

```
// neighbor say vj which ends at the lastest time.  Schedule vi to start
// at the time which vj ends.

{
    int tempLastestEndTime;
    int startTimeArr[n];     // to record the starting time for each vertex (task).
    int adjToArr[n];         // to record the subscripts of the neighbors of each vi
    int totalTimeUsed = matrix[0][0];
    int indCount;       // a counter
    int arrInd;     // to keep track of how many neighbors vi has
                    // The number of neighbors = arrInd + 1.  Also it serves as
                    // the array index for the array adjToArr.
    int neighborInd;
    int possLastestEndTime;
    int temp;

    initialArray(startTimeArr); // initialize the starting time of each
                                // vertex (task) to be time 0.
    for (int indi = 1; indi < n; indi++)
    // In C++, the index 0 in an array positions the 1st element of the array;
    // so index 1 refers to the 2nd element of the array.
    // Beginning at indi = 1 means that we begin to find the starting time
    // of vertex v2.  Remember the starting time of v1 is always zero;
    // so there is no need to find the starting time of v1.
    {
        initialArray(adjToArr);  // initialize adjToArr to be all zeroes
        arrInd = 0;

        for (int indj = 0; indj < n; indj++)
        {
            if (matrix[indi][indj] == 1)
            {
                adjToArr[arrInd] = indj;    // for vi, record all the subscipts of
                                            // the vertices that are adjacent to vi.
                arrInd++;
            }
        }

        indCount = 0;
        tempLastestEndTime = 0;
        while (indCount < arrInd)
        // This while loop finds out the lastest ending time taken over
        // all the neighbors of vi.
        {
            neighborInd = adjToArr[indCount];
            if (neighborInd < indi)
            {
                possLastestEndTime = startTimeArr[neighborInd] + matrix[neighborInd][neighborInd];
                if (possLastestEndTime > tempLastestEndTime)
                    tempLastestEndTime = possLastestEndTime;
            }
```

```
            indCount++;
        }

        startTimeArr[indi] = tempLastestEndTime;  // Assign the starting time of vi
                                                  // to be the latest ending time
                                                  // taken over all the neighbors of vi.
        temp = startTimeArr[indi] + matrix[indi][indi];
        if (temp > totalTimeUsed)
            totalTimeUsed = temp;  // We keep track of the total time used so far
                                   // after we schedule each vi.
    }

//  printStartTime(startTimeArr);
    return totalTimeUsed;
}


void initialArray(int Arr[n])
// initialize the entire array to be all zeroes
{
    for (int i = 0; i < n; i++)
        Arr[i] = 0;
}

void printStartTime(int array[n])
// print out the starting time for each vertex (task)
{
    fout<<endl;
    for (int i = 0; i < n; i++)
        fout<<"The start time assigned to V"<<i+1<<" is "<<array[i]<<"."<<endl;
}

void printMatrix(int a[][n])
// print out the content of a matrix
{
    for (int i = 0; i < n; i++ )
    {
        for (int j = 0; j < n; j++)
            fout<< a[i][j] << ' ';

        fout<< endl;
    }
}
```

7.2 An Output

Most of the output to these 112 graph cards entered as an input are quite boring and similar. For completion, one output is included here. The following is the output for graph card #94 which is the first counter-example found.

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 3.
The Weighted Chromatic Number is 3.
##

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 3.
The Weighted Chromatic Number is 3.
##

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 4.
The Weighted Chromatic Number is 4.
##

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 4.
The Weighted Chromatic Number is 4.
##

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 3.
The Weighted Chromatic Number is 3.
##

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 3.
The Weighted Chromatic Number is 3.
##

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 4.
The Weighted Chromatic Number is 4.
##

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 4.
The Weighted Chromatic Number is 4.
##

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 4.
The Weighted Chromatic Number is 4.
##

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 4.
The Weighted Chromatic Number is 4.
##

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 5.
The Weighted Chromatic Number is 5.
##

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 5.
The Weighted Chromatic Number is 5.
##

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 4.
The Weighted Chromatic Number is 4.
##

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 4.
The Weighted Chromatic Number is 4.
##

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 5.
The Weighted Chromatic Number is 5.
##

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 5.

The Weighted Chromatic Number is 5.
##

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 4.
The Weighted Chromatic Number is 4.
##

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 4.
The Weighted Chromatic Number is 4.
##

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 5.
The Weighted Chromatic Number is 5.
##

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 5.
The Weighted Chromatic Number is 5.
##

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 4.
The Weighted Chromatic Number is 4.
##

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 4.
The Weighted Chromatic Number is 4.
##

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 5.
The Weighted Chromatic Number is 5.
##

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 5.
The Weighted Chromatic Number is 5.
##

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 5.
The Weighted Chromatic Number is 5.
##

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 5.
The Weighted Chromatic Number is 5.
##

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 6.
The Weighted Chromatic Number is 6.
##

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 6.
The Weighted Chromatic Number is 6.
##

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 5.
The Weighted Chromatic Number is 5.
##

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 5.
The Weighted Chromatic Number is 5.
##

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 6.
The Weighted Chromatic Number is 6.
##

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 6.
The Weighted Chromatic Number is 6.
##

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 3.

The Weighted Chromatic Number is 3.
##

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 3.
The Weighted Chromatic Number is 3.
##

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 4.
The Weighted Chromatic Number is 4.
##

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 4.
The Weighted Chromatic Number is 4.
##

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 3.
The Weighted Chromatic Number is 3.
##

***** Summary: Yes, here is a counter-example! *****
This following matrix has the Weighted Chromatic Number greater than
the Maximum of the Minimum Weighted Degree over all the induced subgraphs.
2 1 0 0 0 0
1 1 1 0 1 0
0 1 1 1 1 0
0 0 1 2 0 0
0 1 1 0 1 1
0 0 0 0 1 2
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 3.
The Weighted Chromatic Number is 4.
##

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 4.
The Weighted Chromatic Number is 4.
##

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 4.
The Weighted Chromatic Number is 4.

```
##################################################################

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 4.
The Weighted Chromatic Number is 4.
##################################################################

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 4.
The Weighted Chromatic Number is 4.
##################################################################

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 5.
The Weighted Chromatic Number is 5.
##################################################################

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 5.
The Weighted Chromatic Number is 5.
##################################################################

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 4.
The Weighted Chromatic Number is 4.
##################################################################

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 4.
The Weighted Chromatic Number is 4.
##################################################################

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 5.
The Weighted Chromatic Number is 5.
##################################################################

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 5.
The Weighted Chromatic Number is 5.
##################################################################
```

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 4.
The Weighted Chromatic Number is 4.
##

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 4.
The Weighted Chromatic Number is 4.
##

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 5.
The Weighted Chromatic Number is 5.
##

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 5.
The Weighted Chromatic Number is 5.
##

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 4.
The Weighted Chromatic Number is 4.
##

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 4.
The Weighted Chromatic Number is 4.
##

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 5.
The Weighted Chromatic Number is 5.
##

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 5.
The Weighted Chromatic Number is 5.

##

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 5.
The Weighted Chromatic Number is 5.
##

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 6.
The Weighted Chromatic Number is 6.
##

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 6.
The Weighted Chromatic Number is 6.
##

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 5.
The Weighted Chromatic Number is 5.
##

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 5.
The Weighted Chromatic Number is 5.
##

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 6.
The Weighted Chromatic Number is 6.
##

***** Summary: not a counter-example *****
The Maximum of the Minimum Weighted Degree over all the induced subgraphs is 6.
The Weighted Chromatic Number is 6.
##

!!!!!!!!!!! CONCLUSION !!!!!!!!!!!
Number of total graphs we have looked through is 64.
Number of counter-examples found from these graphs is 1.

7.3 A Computer Program to Find the (Regular) Chromatic Number

Before writing the program in section 7.1 to find counter-examples to the conjecture stated in chapter 5, a program was written to find the chromatic number. The part of the program in 7.1 which finds the weighted chromatic number is very similar to the one written here except that the code for the greedy algorithm is different. Surprisingly, it was a bit trickier to write the code of the greedy algorithm for the chromatic number than for the weighted chromatic number. However, neither of them are difficult to write. All the comments are left out because they are very similar to that in 7.1.

The greedy algorithm for finding the chromatic number works as follows. First assign color 1 to vertex v_1, then color all the vertices in the increasing order of their subscripts. To color each v_i, use the lowest color that is not equal to any of the colors used by the neighbors of v_i.

To find $\chi(G)$, the program applies the greedy algorithm to each ordering of the vertices of G. Out of all the orderings, the minimum number of colors needed is the chromatic number of G.

Claim: There is an ordering of vertices such that when the greedy algorithm is applied. It yields the chromatic number.

Proof: Suppose $\chi(G) = k$. Let V_1', V_2', V_3', ..., V_k' be the color classes of G with color i assigned to the vertices in V_i'. Proceed as follows. Let u be a vertex in V_2'. If u is not adjacent with any vertex in V_1', then reassign u with color 1 instead. If u is adjacent with

some vertex in V_1', then u keeps its color, namely color 2. Note that some vertex in V_2' is assigned color 2; otherwise $\chi(G) = k - 1$. Let v be a vertex in V_3'. If v is not adjacent to any vertex with color 1, then reassign v with color 1 instead. If v is adjacent to some vertex with color 1, but not adjacent to any vertex with color 2, then reassign v with color 2. If v is adjacent to some vertex with color 1, and also adjacent to some vertex with color 2, then v keeps its color 3. Similarly, some vertex in V_3' is assigned color 3; otherwise $\chi(G) = k - 1$. In this process, assign the lowest color to every vertex. Keep doing this process, the final step yields color classes $V_1, V_2, V_3, \ldots, V_k$. Now re-order all the vertices in G according to these new color classes, the vertices in V_1 first, then the vertices in V_2, and so on. Apply the greedy algorithm to this ordering of vertices, the number of colors needed equals the chromatic number of G.

Q.E.D.

Here is the program.

```
// By Sandy Zhang
// This program finds the (regular) chromatic number of an input graph G.

// program FO_reg_chromatic
// output to a file.
#include <fstream.h>

const n = 6;
void initialArray (int [n]);
void printMatrix(int [][n]);
void printColoring(int [n]);
void initialize(int [n]);
void printArr(int [n]);
int factorial(int);
int findBigJ(int [n]);
int findBigK(int, int [n]);
void sortJPlusOneOn(int, int [n]);
```

```
void generateNewMatrix(int [][n], int [][n], int [n]);
int applyGreedy(int [][n]);
int positionOf(int,int Arr[n]);

ofstream fout("OutP_reg_chromatic.txt");

int main()
{

    int gMatrix[n][n]  ={{0,1,0,1,0,0},{1,0,0,0,1,1},{0,0,0,0,1,1},{1,0,0,0,1,1},
                         {0,1,1,1,0,0},{0,1,1,1,0,0}};
    int numArr[n];
    int count = 1;
    int BIGj, BIGk, temp;
    int chromatic;
    int newMatrix[n][n];
    int numColors;

    initialize(numArr);  // initialize numArr to be the permutation 1, 2, 3, ... n.
    printArr(numArr);
    fout<<"The matrix entered is:"<<endl<<endl;
    printMatrix(gMatrix);
    numColors = applyGreedy(gMatrix);
    fout<<"The number of colors needed by the greedy algorithm"<<endl
        <<"according to this arrangement of vertices is "<<numColors<<"."<<endl;
    chromatic = numColors;

    do
    {
        BIGj = findBigJ(numArr);
        BIGk = findBigK(BIGj,numArr);
        temp = numArr[BIGj];
        numArr[BIGj] = numArr[BIGk];
        numArr[BIGk] = temp;
        sortJPlusOneOn(BIGj,numArr);
        printArr(numArr);
        generateNewMatrix(gMatrix,newMatrix,numArr);
        printMatrix(newMatrix);
        numColors = applyGreedy(newMatrix);
        if (chromatic > numColors)
            chromatic = numColors;
        fout<<"The number of colors needed by the greedy algorithm"<<endl
            <<"according to this arrangement of vertices is "<<numColors<<"."<<endl;
        count++;
    } while (count < factorial(n));

    fout<<endl<<"The chromatic number is "<<chromatic<<"."<<endl;
    return 0;
}
```

```
void initialArray(int Arr[n])
{
     for (int i = 0; i < n; i++)
          Arr[i] = 0;
}


void printMatrix(int a[][n])
{
     for (int i = 0; i < n; i++ )
     {
          for (int j = 0; j < n; j++)
               fout<< a[i][j] << ' ';

          fout<< endl;
     }
}


void printColoring(int array[n])
{
     fout<<endl;
     for (int i = 0; i < n; i++)
          fout<<"The color assigned to V"<<i+1<<" is color "<<array[i]<<"."<<endl;
}


void initialize(int Arr[n])
{
     for (int i = 0; i < n; i++)
          Arr[i] = i + 1;
}

void printArr(int Arr[n])
{
     fout<<endl<<endl;
     for (int i = 0; i < n; i++)
          fout<<Arr[i]<<" ";

     fout<<endl<<endl;
}

int factorial(int num)
{
     if (num <= 1)
          return 1;
     else
          return num*factorial(num-1);
}
```

```
int findBigJ(int Arr[n])
{
     int indexJ = -1;
     for (int i = 0; i < n-1; i++)
     {
          if (Arr[i] < Arr[i+1])
               indexJ = i;
     }
     return indexJ;
}

int findBigK(int indexJ, int Arr[n])
{
     int indexK;
     for (int i = indexJ+1; i < n; i++)
     {
          if (Arr[i] > Arr[indexJ])
               indexK = i;
     }
     return indexK;
}

void sortJPlusOneOn(int ind_j, int Arr[n])
// sort the (j+1)th to (n-1)th elements of the Array in increasing order
{
     int hold;

     for (int pass = ind_j+1; pass < n-1; pass++)
     {
          for (int index = ind_j+1; index < n-1; index++)
               if (Arr[index] > Arr[index+1])
               {
                    hold = Arr[index];
                    Arr[index] = Arr[index+1];
                    Arr[index+1] = hold;
               }
     }
}


void generateNewMatrix(int origMatrix[][n], int permMatrix[][n], int permArr[n])
{
     for (int i = 0; i < n; i++)
     {
          for (int j = 0; j < n; j++)
          {
               if (i == j)
                    permMatrix[i][j] = 0;
               else
                    permMatrix[i][j] = origMatrix[positionOf(i+1,permArr)][positionOf(j+1,permArr)];
          }
```

```
        }
    }

    int applyGreedy(int matrix[][n])
    {
        int tempColor;
        int colorArr[n];
        int adjToArr[n];
        int numColorsUsed = 0;
        int indCount;
        int arrInd;

        initialArray(colorArr);
        for (int indi = 0; indi < n; indi++)
        {

            initialArray(adjToArr);
            arrInd = 0;

            for (int indj = 0; indj < n; indj++)
            {
                if (matrix[indi][indj] == 1)
                {
                    adjToArr[arrInd] = indj;
                    arrInd++;
                }
            }

            indCount = 0;
            tempColor = 1;
            while (indCount < arrInd)
            {
                if (tempColor == colorArr[adjToArr[indCount]])
                {
                    tempColor++;
                    indCount = 0;
                }
                else
                    indCount++;
            }

            colorArr[indi] = tempColor;
            if (colorArr[indi] > numColorsUsed)
                numColorsUsed = colorArr[indi];
        }

        printColoring(colorArr);
        return numColorsUsed;
    }
```

```
int positionOf(int x,int Arr[n])
{
    for (int i = 0; i < n; i++)
    {
        if (Arr[i] == x)
            return i;
    }
}
```

7.4 The Computer Program Trying to Find G Such that $\chi^*(G) = \Delta^*(G)$

```
// By Sandy Zhang
// This program is a modification of the program written in section 7.1.
// This program tells if an input graph G (with weights 1, 2, or 3 assigned to its
// vertices) has its weighted chromatic number equal to its weighted maximum degree.

// program FO_n6w3WghtChrom&MaxWghtDeg
// output to a file

#include <fstream.h>
#include <math.h>

ofstream fout("card207w3WghtChrom&MaxWghtDeg.txt");

const n = 6;

bool makeComparision (int [][n]);
int findWeightedChromatic(int gMatrix[][n]);
int applyGreedy(int [][n]);
void generateNewMatrix(int [][n], int [][n], int [n]);
void initialArray(int [n]);
void printStartTime(int [n]);
void printMatrix(int [][n]);
void initialize(int [n]);
void printArr(int [n]);
int factorial(int);
int findBigJ(int [n]);
int findBigK(int, int [n]);
void sortJPlusOneOn(int, int [n]);
int positionOf(int,int Arr[n]);
int maxDeg(int[][n]);


int main()
{
    int incidentMatrix[n][n] = {{0,1,1,1,1,1},{1,0,1,1,1,1},{1,1,0,1,1,0},{1,1,1,0,1,1},
                                {1,1,1,1,0,1},{1,1,0,1,1,0}};
    int numOfTotalGraphs = 0;
    int numOfDesiredEx = 0;
    bool DesiredExampleFound = false;
    for (int indexA = 1; indexA <= 3; indexA++)
    {
        incidentMatrix[0][0] = indexA;
        for (int indexB = 1; indexB <= 3; indexB++)
        {
            incidentMatrix[1][1] = indexB;
            for (int indexC = 1; indexC <= 3; indexC++)
            {
                incidentMatrix[2][2] = indexC;
                for (int indexD = 1; indexD <= 3; indexD++)
```

```
                    {
                        incidentMatrix[3][3] = indexD;
                        for (int indexE = 1; indexE <= 3; indexE++)
                        {
                            incidentMatrix[4][4] = indexE;
                            for (int indexF = 1; indexF <= 3; indexF++)
                            {
                                incidentMatrix[5][5] = indexF;
                                DesiredExampleFound = makeComparision(incidentMatrix);
                                if (DesiredExampleFound)
                                    numOfDesiredEx++;

                                numOfTotalGraphs++;
                            }
                        }
                    }
                }
            }
        }

        fout<<endl<<"!!!!!!!!!!!!  CONCLUSION  !!!!!!!!!!!!"<<endl;
        fout<<"Number of total graphs we have looked through is "<<numOfTotalGraphs<<"."<<endl;
        fout<<"Number of desired examples found from these graphs is "<<numOfDesiredEx<<"."<<endl;
        return 0;
}


bool makeComparision (int graphMatrix[][n])
{
    bool DesiredExFound = false;
    int maxWghtDeg = maxDeg(graphMatrix);
    int weightedChromatic = findWeightedChromatic(graphMatrix);

    if (weightedChromatic == maxWghtDeg)
    {
        DesiredExFound = true;
        fout<<endl<<"***** Summary:  Yes, here is a desired example!  *****"<<endl;
        fout<<"This following matrix has the Weighted Chromatic Number equal to"
            <<"the Maximum Weighted Degree. "<<endl;
        printMatrix(graphMatrix);
    }
    else
        fout<<endl<<"***** Summary:  not a desired example *****"<<endl;

    fout<<"The Maximum Weighted Degree is "
            <<maxWghtDeg<<"."<<endl;
    fout<<"The Weighted Chromatic Number is "<<weightedChromatic<<"."<<endl;
    fout<<"#########################################################################"
        <<endl<<endl;
    return DesiredExFound;
}
```

```
int findWeightedChromatic(int gMatrix[][n])
{
    int numArr[n];
    int count = 1;
    int BIGj, BIGk, temp;
    int wghtChromatic;
    int newMatrix[n][n];
    int totalTime;

    initialize(numArr);  // initialize numArr to be the permutation 1, 2, 3, ... n.

//  fout<<"Start to look for the Weighted Chromatic Number..."<<endl;
//  printArr(numArr);
//  fout<<"The matrix being consideration is:"<<endl<<endl;
//  printMatrix(gMatrix);
    totalTime = applyGreedy(gMatrix);
//  fout<<"The number of colors needed by the greedy algorithm"<<endl
//      <<"according to this arrangement of vertices is "<<totalTime<<"."<<endl;
    wghtChromatic = totalTime;

    do
    {
        BIGj = findBigJ(numArr);
        BIGk = findBigK(BIGj,numArr);
        temp = numArr[BIGj];
        numArr[BIGj] = numArr[BIGk];
        numArr[BIGk] = temp;
        sortJPlusOneOn(BIGj,numArr);
//      printArr(numArr);
        generateNewMatrix(gMatrix,newMatrix,numArr);
//      printMatrix(newMatrix);
        totalTime = applyGreedy(newMatrix);
        if (wghtChromatic > totalTime)
            wghtChromatic = totalTime;
//      fout<<"The number of colors needed by the greedy algorithm"<<endl
//          <<"according to this arrangement of vertices is "<<totalTime<<"."<<endl;
        count++;
    } while (count < factorial(n));

//  fout<<endl<<"*** The Weighted Chromatic number is "<<wghtChromatic<<". ***"<<endl;
    return wghtChromatic;
}

int maxDeg(int a[][n])
{
    // The 3 lines commented out are for the display of each vertex's
    // weighted degree if we ever want to know them.

    // cout<<endl;
    int max;
    for (int i = 0; i < n; i++)
```

```
        {
            int deg = a[i][i];
            //cout<<"Weighted Degree of V"<<i+1<<" = ";
            for (int j = 0; j < n; j++)
            {
                if (i != j)
                {
                    if (a[i][j] == 1)
                        deg = deg + a[j][j];
                }
            }
            //cout<<deg<<endl;
            if (i==0)
                max = deg;
            else
                if (deg > max)
                max = deg;
        }
        return max;
}

void initialize(int Arr[n])
{
    for (int i = 0; i < n; i++)
        Arr[i] = i + 1;
}

void printArr(int Arr[n])
{
    fout<<endl<<endl;
    for (int i = 0; i < n; i++)
        fout<<Arr[i]<<" ";

    fout<<endl<<endl;
}

int factorial(int num)
{
    if (num <= 1)
        return 1;
    else
        return num*factorial(num-1);
}

int findBigJ(int Arr[n])
{
    int indexJ = -1;
    for (int i = 0; i < n-1; i++)
    {
        if (Arr[i] < Arr[i+1])
            indexJ = i;
```

```
        }
        return indexJ;
    }

    int findBigK(int indexJ, int Arr[n])
    {
        int indexK;
        for (int i = indexJ+1; i < n; i++)
        {
            if (Arr[i] > Arr[indexJ])
                indexK = i;
        }
        return indexK;
    }

    void sortJPlusOneOn(int ind_j, int Arr[n])
    // sort the (j+1)th to (n-1)th elements of the Array in increasing order
    {
        int hold;

        for (int pass = ind_j+1; pass < n-1; pass++)
        {
            for (int index = ind_j+1; index < n-1; index++)
                if (Arr[index] > Arr[index+1])
                {
                    hold = Arr[index];
                    Arr[index] = Arr[index+1];
                    Arr[index+1] = hold;
                }
        }
    }

    void generateNewMatrix(int origMatrix[][n], int permMatrix[][n], int permArr[n])
    {
        for (int i = 0; i < n; i++)
        {
            for (int j = 0; j < n; j++)
                permMatrix[i][j] = origMatrix[positionOf(i+1,permArr)][positionOf(j+1,permArr)];
        }
    }


    int positionOf(int x,int Arr[n])
    {
        for (int i = 0; i < n; i++)
        {
            if (Arr[i] == x)
                return i;
        }
    }
```

```
int applyGreedy(int matrix[][n])
{
    int tempLastestEndTime;
    int startTimeArr[n];
    int adjToArr[n];
    int totalTimeUsed = matrix[0][0];
    int indCount;
    int arrInd;
    int neighborInd;
    int possLastestEndTime;
    int temp;

    initialArray(startTimeArr);
    for (int indi = 1; indi < n; indi++)
    // Begin at V2 which has index i = 1 because the start time of V1 is always zero;
    // no need to find the start time of V1.
    {
        initialArray(adjToArr);
        arrInd = 0;

        for (int indj = 0; indj < n; indj++)
        {
            if (matrix[indi][indj] == 1)
            {
                adjToArr[arrInd] = indj;
                arrInd++;
            }
        }

        indCount = 0;
        tempLastestEndTime = 0;
        while (indCount < arrInd)
        {
            neighborInd = adjToArr[indCount];
            if (neighborInd < indi)
            {
                possLastestEndTime = startTimeArr[neighborInd] + matrix[neighborInd][neighborInd];
                if (possLastestEndTime > tempLastestEndTime)
                    tempLastestEndTime = possLastestEndTime;
            }
            indCount++;
        }

        startTimeArr[indi] = tempLastestEndTime;
        temp = startTimeArr[indi] + matrix[indi][indi];
        if (temp > totalTimeUsed)
            totalTimeUsed = temp;
    }
//  printStartTime(startTimeArr);
    return totalTimeUsed;
}
```

```
void initialArray(int Arr[n])
{
     for (int i = 0; i < n; i++)
          Arr[i] = 0;
}

void printStartTime(int array[n])
{
     fout<<endl;
     for (int i = 0; i < n; i++)
          fout<<"The start time assigned to V"<<i+1<<" is "<<array[i]<<"."<<endl;
}

void printMatrix(int a[][n])
{
     for (int i = 0; i < n; i++ )
     {
          for (int j = 0; j < n; j++)
               fout<< a[i][j] << ' ';

          fout<< endl;
     }
}
```

References

1. Biggs, Norman, E. Keith Lloyd and Robin J. Wilson, *Graph Theory 1736-1936,* Clarendon Press, Oxford, 1976.

2. Bollobas, Bela, *Modern Graph Theory,* Springer, New York, 1998.

3. Bondy, J.A. and U.S.R. Murty, *Graph Theory with Applications,* Elsevier Science Publishing Co., New York, 1985.

4. Chartrand, Gray, *Graphs as Mathematical Models,* Prindle, Weber & Schmidt, Boston, Massachusetts, 1978.

5. Chartrand, G. and L. Lesniak, *Graphs & Digraphs,* Chapman & Hall, New York, 1996.

6. Chartrand, Gray and Ortrud R. Oellermann, *Applied and Algorithmic Graph Theory,* McGraw-Hill 1993.

7. Harary, Frank, *Graph Theory,* Addison-Wesley Publishing Company, Menlo Park, California, 1972.

8. Harary, Frank and John S. Maybee, *Graphs and Applications Proceedings of the First Colorado Symposium on Graph Theory,* John Wiley & Sons, New York, 1985.

9. Jackson, Brad and Dmitri Thoro, *Applied Combinatorics with Problem Solving,* Addison-Wesley Publishing Company, Menlo Park, California, 1990

10. McHugh, James A., *Algorithmic Graph Theory,* Prentice-Hall, New Jersey, 1990.

11. Watkins, John J. and Robin J. Wilson, *Graphs - An Introductory Approach,* John Wiley & Sons, New York, 1989.