U.S.N.A. --- Trident Scholar project report; no. 317 (2003)

EFFICIENT ACADEMIC SCHEDULING AT THE U.S. NAVAL ACADEMY

By

Midshipman David L. Zane, Class of 2003
United States Naval Academy
Annapolis, Maryland

_____
(signature)

Certification of Advisers Approval

Assistant Professor William Traves
Department of Mathematics

_____
(signature)

_____
(date)

Assistant Professor Christopher Brown
Department of Computer Science

_____
(signature)

_____
(date)

Acceptance for the Trident Scholar Committee

Professor Joyce E. Shade
Deputy Director of Research & Scholarship

_____
(signature)

_____
(date)

USNA-1531-2

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATE COVERED
	2 May 2003	

4. TITLE AND SUBTITLE
Efficient academic scheduling at the U.S. Naval Academy

5. FUNDING NUMBERS

6. AUTHOR(S)
Zane, David L. |q(David Lawrence), |d1981-

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)		8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSORING/MONITORING AGENCY REPORT NUMBER
US Naval Academy Annapolis, MD 21402		Trident Scholar project report no. 317 (2003)

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT	12b. DISTRIBUTION CODE
This document has been approved for public release; its distribution is UNLIMITED.	

13. ABSTRACT: This research project examined academic scheduling problems at the U.S. Naval Academy. The focus was on devising methods to construct good final exam schedules and improve existing course schedules by facilitation course changes. The final exam scheduling problem is an example of an NP-hard problem. These difficult problems do not admit efficient deterministic solutions. Several heuristic methods to treat these problems were considered. An approach using genetic algorithms showed particular promise. Genetic algorithms involve mating “parent” schedules to form favorable “offspring” schedules and then subjecting these new schedules to local mutation. A computer program implementing these ideas was created and tested. Section changes at the Naval Academy had been done on an ad-hoc basis, but this project determined that it could be streamlined and improved by using a centralized barter system. The barter technique accepts input listing desired section changes and identifies multi-student section changes to accommodate their desires. A prototype computer program that uses network flow algorithms to find such section changes was devised. In addition, a method incorporating integer programming techniques was examined and tested.

14. SUBJECT TERMS:	15. NUMBER OF PAGES
academic scheduling; final exams; U.S. Naval Academy; integer programming techniques	81
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT

ABSTRACT: "Efficient Academic Scheduling at the U.S. Naval Academy"

This Trident Scholar Independent Research Project examined academic scheduling problems at the U.S. Naval Academy. The focus was on devising methods to construct good final exam schedules and improve existing course schedules by facilitating course changes.

The final exam scheduling problem is an example of an NP-hard problem. These difficult problems do not admit efficient deterministic solutions. Several heuristic methods to treat these problems were considered. An approach using genetic algorithms showed particular promise. Genetic algorithms involve mating "parent" schedules to form favorable "offspring" schedules and then subjecting these new schedules to local mutation. A computer program implementing these ideas was created and tested.

Section changes at the Naval Academy had been done on an ad-hoc basis, but this project determined that it could be streamlined and improved by using a centralized barter system. The barter technique accepts input listing desired section changes and identifies multi-student section changes to accommodate their desires. A prototype computer program that uses network flow algorithms to find such section changes was devised. In addition, a method incorporating integer programming techniques was examined and tested.

Acknowledgements

I would like to thank the following people. My advisors, Assistant Professor William Traves and Assistant Professor Chris Brown – thank you for your guidance, expertise, and patience. In addition, I want to thank Professor Joyce Shade and the Trident Committee, for their relentless pursuit of perfection in our projects. Finally, I would like to thank my family. To my mother Nina, who is my inspiration – you are everything I could have asked for and then some. To my brother Michael – thank you for setting the standard so high and tough to reach. To my girlfriend Caroline – thank you for keeping me happy on those difficult days and tolerating me on those instances that I could not spend time with you.

Table of Contents:

The modern Navy faces numerous logistical problems. These resource allocation issues come in many forms, one being the problem of efficient scheduling. Since each problem incorporates a myriad of factors and constraints, they do not admit simple solutions. Similar types of scheduling problems arise at the United States Naval Academy. This Trident Scholar research project focuses on two of these problems: the scheduling of final examinations and an aspect of course scheduling known as the course bartering system.

Scheduling Final Examinations:

How do you form a good final examination schedule for each semester? The first step to solving this problem is to define the factors involved. Foremost, good schedules avoid scheduling conflicts – the situation where midshipmen are scheduled for two examinations at once. Another important condition is to have the exams that are the most difficult to grade given first, since all of the professors must submit final grades by a certain date. Other problems include midshipmen taking three exams in a row or scheduling two common technical courses consecutively, such as Plebe Chemistry and Calculus.

Once these factors have been determined and prioritized, a schedule can be formed. Note that there are trade-offs to be made. If fewer exam periods are used to accommodate grading concerns, then there are many conflicts. Conversely, if the time between exams is lengthened in hopes of producing more time to study and fewer conflicts, then some instructors will have little time for grading (and less midshipmen leave!).

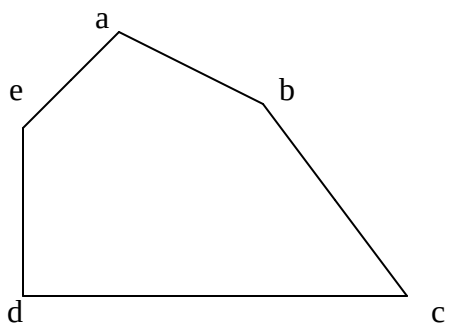
Course Scheduling:

In course scheduling at the Naval Academy, midshipmen first preregister – a process where students request the courses they want to take. Once midshipmen are approved for their course choices, they then register for particular sections within a course. Each section, however, has a maximum capacity, and midshipmen are not always placed in the section they desire. The goal of the course bartering system is to allow midshipmen to submit the changes they would like to their schedules. Then a centralized system inputs all of these requests and determines the course exchanges that are possible. The key constraint in determining whether to accommodate a course exchange is the Academic Registrar's request to keep the section size of a course the same. Thus, the program defines an optimal solution as accommodating the largest number of midshipmen without changing the section size of any course.

Graphs

Graphs are a primary tool for modeling the course bartering system. [BR] A graph is a pair G = (V, E) where V is the set of vertices, or points, and E is the set of edges, or line segments between two vertices. Directed edges are edges with direction; in other words, they have a start vertex and a terminal vertex.

Figure 1: A Graph

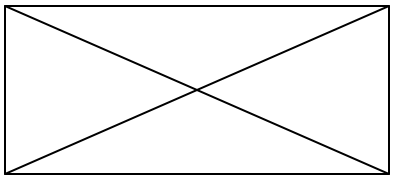


A typical graph with both directed edges and regular edges. Regular edges can be thought of as edges directed in both directions.

A graph was constructed to represent the information needed to produce a course bartering system. A vertex represented each course and section pair. A directed edge between two vertices indicates that a midshipmen desires to move from one course and section to another. For instance, the edge [b,c] in Figure 1 is a directed edge. A path from a starting vertex *1* to a terminal vertex *2* involves moving along any set of edges in the proper direction from *1* to *2* without repeating edges. In Figure 1, a path from e to c is given by the chain of vertices e, a, b, c. A cycle in the graph is a path where the starting vertex is also the terminal vertex. There are no directed cycles in Figure 1; however, if the edge [d,c] is replaced by the edge [c,d] (reversing orientation), then there is a cycle involving all of the vertices.

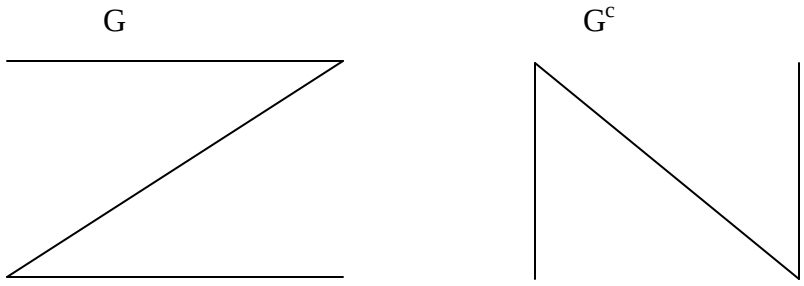
The vertex set of a graph G is the set of all vertices in G. The edge set is the set of all edges in G. If all of the possible edges of a given set of vertices are present, then the graph is called a complete graph.

Figure 2: A Complete Graph on Four Vertices



A subgraph G' of G consists of a vertex set V' $\subseteq$ V, where V is the vertex set of G. The edge set of G' consists of all of the edges between the vertices of V' that were in the edge set of G. A clique in G is a complete subgraph induced by a given subset of the vertex set. Finally, the complement G^c of G is the graph with vertex set V and all of the edges in a complete graph less the edges already present in G.

Figure 3: A Graph and Its Complement



Some of the scheduling problems fall into a class of problems classified as NP-hard. [CPPS] These are computationally difficult problems, for which no fast deterministic algorithms are known. It remains a difficult open problem to show that none of these problems admit solution by suitably fast methods, but the consensus among the computer science community is that these problems do not admit simple solutions. As a

result, attention has been focused on finding good heuristic approaches to scheduling problems. In fact, they are of such central importance that they have received a tremendous amount of attention. A search on MathSciNet located over 1800 research articles on NP-hard problems and heuristic approaches to them. [www.ams.org/mathscinet]

Course Bartering System:

As stated earlier, in the course bartering system, midshipmen who have already received their course schedule can submit requests for course changes. A computer program was developed that finds any possible switches. To that end, several approaches were examined: a deterministic greedy algorithm method, a cycle graph method, an algorithm incorporating the depth-first search to find cycles, a network flow algorithm, and an integer programming approach. Since the first two approaches were not implemented, they will not be discussed.

In the course bartering system, every midshipman was permitted to declare one switch as long as their move is to a period that is already open in their existing schedule. With this restriction in place, the following process on a graph was developed:

Steps of the Course Bartering System

1. For each course and period pair, (e.g. Electrical Engineering, 6^{th} period) assign a vertex.
2. For each midshipman requesting to move from one course and period to another, draw a directed edge from the vertex associated with the current course and period to the course the midshipmen desires to enter.
3. Remove vertices from the graph that cannot be changed. This would be a vertex where no midshipman desires to enter or leave that course and period. In this graph, any vertex without both incoming and outgoing edges may be eliminated. This is due to the constraint that the section size of any course must remain the same.
4. Apply an algorithm to look for cycles in the remaining graph. The search is for cycles instead of paths also because of the constraint that section size of any course must remain constant.

An important step in this process is the simplification step. After the simplification, there must be a cycle in the remaining graph and thus possible switches.

Claim: There exists a directed cycle in a graph where all vertices have both incoming and outgoing edges.

Assume that the graph G does not have a directed cycle and arrive at a contradiction. Start at any vertex v_1. Travel along any of its outgoing edges. Then travel to another vertex, which is called v_2. For the outgoing edge of v_2, travel to any other vertex in G, except that there is no directed edge (v_2, v_1) because there are no directed cycles. Continue the path to a vertex now called v_3. For the outgoing edge on v_3, travel to any other vertex in G, except v_1 or v_2, since (v_3, v_1) and (v_3, v_2) do not exist because there are no directed cycles. Thus, for a given vertex v_i, any of its outgoing edge can be used except ones leading to vertices used earlier. (Or the hypothesis is violated). The path eventually includes all of the vertices. Call the final vertex in the path in G v_n. The vertex v_n must have an outgoing edge as given by the conditions of the graph G. This outgoing edge must go to one of the previously used vertices, and thus there is a contradiction.

Q.E.D.

The goal now became to search for the cycles in the graph. In light of this, a C++ program that finds a directed cycle in a graph with a depth-first search, removes it, and searches again for another directed cycle, was constructed. It then terminates when the graph is reduced to the null set. (i.e. when all cycles have been removed) This program accepts a data file containing course-section pairs and midshipmen desires as input and yields a possible collection of cycles. (*See Appendices A and B for the C++ program and an example*)

The program, entitled the "Cycle-Elimination Program," is quite efficient. Utilizing the depth-first search allows the program to be run several times and get all of the possible cycle combinations on small graphs. However, the task then becomes difficult when the search transitions into finding the best collection of cycles.

Network Flows:

To deal with optimality concerns, a different algorithm that uses network flows to implement the course bartering system was examined. [BMMN, DIMACS, OMA] The

problem background remains the same: vertices are course-section pairs and directed edges between vertices indicate that a midshipman desires to move from one section to another. There is still the search for cycles using a depth-first method. The combination of cycles using the most edges possible corresponds to the optimal course change. However, the graph is thought of as a network, and the goal becomes to flow the most midshipmen from their current course-section pair to the one they desire. Each edge is assigned a flow, capacity, and cost. The flow is a number that represents the movement of midshipmen from one section to another. The capacity is the total number of midshipmen that wish to make the section change. Each midshipman move (traversing an edge) is assigned a cost of "-1." The number is negative because these switches are desired – switches correspond to a benefit. For example, two midshipmen might want to move from a sixth period Electrical Engineering section to a fourth period section. This leads to a directed edge in the graph with capacity 2. The objective is to find circulations of minimum cost. A circulation, C, is a collection of directed cycles. Note that in the collection, a directed cycle can appear more than once. In fact, for each edge in the circulation, the number of cycles where that edge appears is limited by the capacity of that edge. Each circulation is thought of as giving a movement of commodities; in the course bartering system, the commodities are midshipmen as they move from one section to another.

A vector notation is sometimes used to describe this flow of commodities. For each circulation C, a "flow" vector of non-negative integers is constructed. Each component in the vector is labeled by an edge in G. The flow x_{ij} corresponding to edge [i,j] is the number of times edge [i,j] appears in a cycle of C.

Thus the vector x satisfies:

1. For each vertex i the flow into the vertex is the same as the flow leaving the vertex:

$$\text{for each i} \qquad \sum_j x_{ij} = \sum_j x_{ji} \ .$$

2. For each edge the flow along the edge is not greater than the capacity of the edge.

The cost, cx, of a flow x is the dot product of the flow vector and the cost vector, $c \bullet x$.

Conversely, any vector of non-negative integers satisfying conditions 1 and 2 is the vector associated with a circulation. The circulation is constructed as follows. Start at any of the vertices (v_1) that has an outgoing edge with positive flow. Traverse this edge to another vertex, v_2. Then transverse an outgoing edge of v_2 to another vertex, v_3. Such an edge exists because of condition (1) above. Continuing in this manner, the path eventually arrives at a vertex already used. This corresponds to a directed cycle. At this point, the flow along each of the edges of this cycle must be examined. Flow around this directed cycle as much as possible. That is, flow an amount equal to the lowest flow of all of the edges in the cycle. Remove this flow, and the original circulation is reduced to a smaller circulation. Continue this process of finding directed cycles in the circulation and passing flow around them as much as possible. Ultimately, the flow vector reaches zero and the original graph is decomposed into a collection of directed cycles.

An algorithm that uses network flows to find the optimal cycle combination in the network G was used. [OA] First, the depth-first search finds a cycle, which induces a flow along its edges. (Alternatively, start with the zero flow – the function that assigns

flow zero to each edge.) Then form a residual graph G(x) from the flow x. This is a new graph, consisting of the same vertex set as G. Join vertex x to vertex y in the residual graph if either (1) the residual capacity, the capacity of $[x, y]$ in G minus the flow on the edge $[x, y]$ is positive (here we label this new edge with its residual capacity and assign it cost -1) or (2) the flow on edge $[y, x]$ in G is positive. In this second case, flow along the edge $[x, y]$ in the residual graph corresponds to reducing the flow along $[y, x]$ in the original network. In the second case, the new edge is assigned capacity equal to the flow along $[y, x]$ in G and assigned cost 1.

Once the residual graph G(x) is made, then search for a cycle in G(x) with negative total cost. Choose that cycle and overlay it on the original graph: for each edge $[x, y]$ in the cycle that also appears in G, increase flow along $[x, y]$ by 1; for each edge $[x, y]$ in the cycle that does not appear in G, the reverse edge $[y, x]$ must appear in G (this follows from the definition of the residual graph) and thus, decrease the flow by 1 along edge $[y, x]$ in G. Then form a new residual graph and use the depth-first search to find another negative cost cycle. Repeat this process until there are no negative cost cycles in the residual graph. When this is the case, a circulation in G that minimizes the total cost has been found. This corresponds to finding the section changes that accommodate the greatest number of midshipmen while keeping section sizes constant.

The key mathematical concept in this algorithm is as follows: a feasible flow x is an optimal solution of a minimum cost flow problem if and only if the residual network G(x) contains no negative cost cycles. Below are a construction and a proof. [OMA]

Construction: For a given flow x, every flow x in the network G corresponds to a flow x' in the residual network G(x).*

Define the flow vector x' as the unique vector of non-negative integers satisfying:

$x'_{ij} - x'_{ji} = x^*_{ij} - x_{ij}$ and $x'_{ij}\, x'_{ji} = 0$

The second equality implies that x'_{ij} and x'_{ji} cannot both be positive.

If $x_{ij} < x^*_{ij}$, set $x'_{ji} = 0$ and $x'_{ij} = x^*_{ij} - x_{ij}$. If x^*_{ij} is not greater than the capacity u_{ij} of the edge [i,j], then $x'_{ij} \le u_{ij} - x_{ij} = r_{ij}$, the residual capacity on the edge [i,j]. Also $x'_{ji} = 0$ so the flow x' satisfies the capacity constraints in G(x).

If $x_{ij} = x^*_{ij}$, set $x'_{ji} = -(x^*_{ij} - x_{ij})$ and $x'_{ij} = 0$. As before, the flow x' satisfies the capacity constraints in G(x).

Thus, if x* is a feasible flow in G, its corresponding flow x' is a feasible flow in G(x).

Proof that a feasible flow x is an optimal solution of a minimum cost flow problem if and only if the residual network G(x) contains no negative cost cycles:

Part 1:
Suppose that x is a feasible flow and that the residual graph G(x) contains a net negative cost cycle. Positive flow can be augmented along the negative cycle to get a new circulation of smaller cost. Therefore, if x is an optimal flow, then G(x) does not contain any net negative cost cycles.

Part 2:
Suppose x is a feasible flow on G and that G(x) contains no net negative cost cycles. Let x* be an optimal flow in G and $x \neq x^*$. The flow x* corresponds to a flow x' in the residual network G(x). Let c' denote the costs assigned to each edge in G(x), and let c denote the costs in G. For every directed edge [i, j] in G, $c'_{ij} = c_{ij}$ and $c'_{ji} = -c_{ij}$. For the flow x^*_{ij} on edge [i, j] in the network G the cost of the associated flow x' on the edges [i, j] and [j, i] in G(x) is $c'_{ij}\, x'_{ij} + c'_{ji}\, x'_{ji} = c'_{ij}\,(x'_{ij} - x'_{ji}) = c_{ij}\, x^*_{ij} - c_{ij}\, x_{ij}$.
Thus, $c'x' = cx^* - cx$.

The flow x' in G(x) corresponds to the difference vector x* - x. The circulation corresponding to x' can be decomposed into a finite collection of cycles in G(x). But the costs of all cycles in G(x') are nonnegative. Therefore, $cx' = cx^* - cx \ge 0$, or $cx^* \ge cx$. Also, since x* is an optimal flow, $cx^* \le cx$. Thus $cx^* = cx'$, and x' is also an optimal flow.

Q.E.D.

In short, the following is the network flow algorithm applied to the course bartering system.

Network Flows:

1. For each course and period pair, assign a vertex. (i.e. Electrical Engineering, 6th period)
2. For each midshipman requesting to move from one course and period to another, draw a directed edge from the vertex associated with the current course and period to the vertex the midshipmen desires to enter.
3. Simplification process – elimination of any vertex that does not have both incoming and outgoing edges.
4. Assign to each edge an original flow of 0 and a cost of -1.
5. Form the residual graph.
6. Use Karp's algorithm to determine if there are net negative cycles in the residual graph. If not terminate the algorithm.
7. Search for a net negative cycle in the residual graph.
8. Overlay it on the original graph with the adjusted flow and cost values.
9. Repeat steps 5-8 until there are no net negative cycles in the residual graph.

The determination on whether a net negative cycle is in the residual graph can be found in polynomial time using Karp's algorithm. Below is the basis behind the algorithm and its conclusion. [KA]

Karp's Algorithm: Terminology and Conclusion:

Let G = (V, E) be a graph with n vertices.
Each edge in E has weight f(e).
Given any sequence of edges $s = e_1, e_2, \ldots, e_p$:

Let the weight of s, $w(s) = \sum_{i=1}^{p} f(e_i)$.

Let the mean weight of s, m(s) = w(s)/p

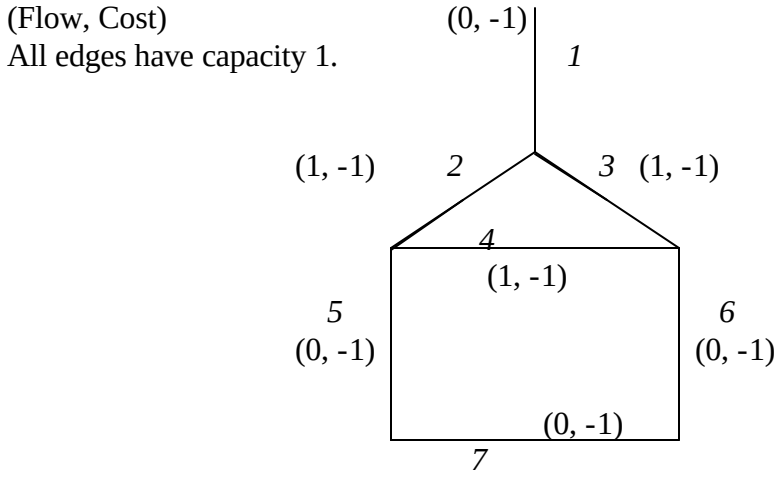
Let s be an arbitrarily chosen vertex in G. For every vertex v and every nonnegative integer k, define $F_k(v)$ as the minimum weight edge progression of length k from s to v.
Let ?* denote the minimum cycle mean = $\min_c m(C)$ where C ranges over all directed cycles in G. Then ?* can also be computed as:

$$l^* = \min_{v \in V} \max_{0<k<n-1} \left[\frac{F_n(v) - F_k(v)}{n-k} \right].$$

A negative cycle exists if and only if ?* < 0.

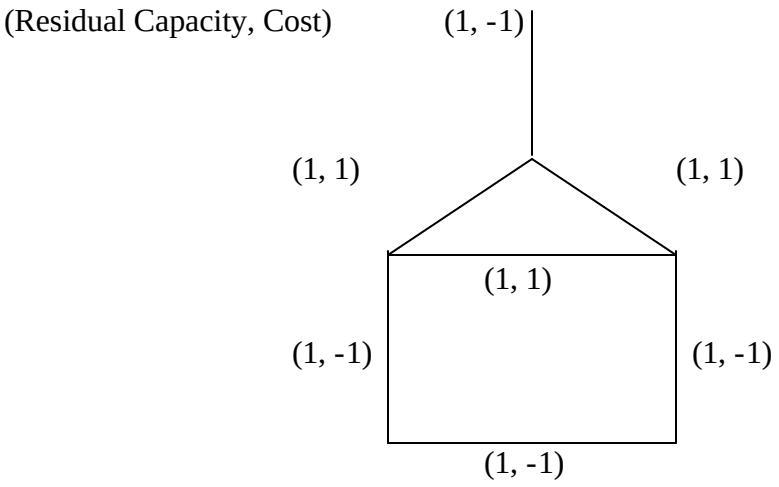
To give an example of the network flow algorithm in action, consider the “Hanging House” network. (See Figure 4) By observation, it can be seen that the triangle and the square are possible cycles, but since they share an edge of capacity 1, both cannot be chosen. If the “cycle-elimination” program were run several times, it would find both cycles. The square would then be chosen because it leads to switches that accommodate the most number of midshipmen. However, in more complicated graphs, where there are potentially a large number of directed cycles, it is not guaranteed that the current program would find the best collection of directed cycles (i.e. the collection that accommodates the most midshipmen). Thus, it is assumed in this example that the depth-first search found the less beneficial cycle (the triangle). Then the network flow algorithm is applied. The following graph designates that the triangle has been chosen as a flow, as represented by the "1" values in the flow component in the ordered pair.

Figure 4: Hanging House Network Graph



Following the algorithm, the next step is to produce the residual graph.

Figure 5: Hanging House Network Residual Graph



There is a net negative cycle in the residual graph that traces around the perimeter of the house. Choose it because it has a net negative cost of -1. Then overlay[1] it on the original graph and only the square cycle remains. Create a new residual graph from the new flow. Since this residual graph has no net negative cycles, the optimal network flow has been found.

Integer Programming

Another promising approach to the course bartering system involved the use of linear and integer programming. [S, V] Simply put, in a linear programming problem, the goal is to maximize a certain objective function subject to known constraints. The only extra stipulation on an integer programming problem is that some of the variables are constrained to be integers.

This integer programming approach was examined to see if it produced the optimal solution to the hanging house network. The problem was modeled as follows: A

[1] That is, add the vector (0,1,1,1,0,0,0) corresponding to the triangle cycle to the vector corresponding to the perimeter (0,-1,-1,0,1,1,1).

matrix, **M**, where the rows represented midshipmen desires and the columns were course-section pairs, was developed. In its initial phases, all values in the matrix could either be a -1, 0, or 1. A -1 signified a person wanting to leave a certain course and section and a 1 represented a student wanting to change to that course and section. The rest of a row would be filled with zeros. The goal is to maximize the amount of changes that could be made subject to the constraint that each course and section size must not increase. The request vector **w** is used to determine whether a request can be accommodated.

In other words,

$$w_i = \begin{cases} 0, & \text{if student i's switch is not made} \\ 1, & \text{if student i's switch is made} \end{cases}$$

Thus, the values of this vector are either one or zero, the former if a change can be made and the latter if not. The amount of requests accommodated is constrained by the components of the resultant vector, **r**, which is a course's section size. This must be less than zero. Thus, in

$$\mathbf{w} \bullet \mathbf{M} = \mathbf{r}$$

the goal is to maximize the sum of the entries in **w** subject to $\mathbf{r} \leq 0$. Like in the network flow algorithm, where it is desired to minimize costs, maximizing **w** also yields a solution that accommodates the largest number of midshipmen.

The matrix, **M**, for the hanging house network looked as follows:

Sections of Courses

Midshipmen Requests

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

This IP problem was submitted to the *NEOS Server*, a group of processors that solve large-scale linear programming problems expediently. It could be given to a linear programming solver because this example shares the properties of a transportation problem, which always has an integral optimal solution. It is an interesting problem to see whether the optimal solution is always integral when all scheduling requests are made.

The *NEOS Server* uses a heuristic known as the simplex method. This method is an iterative process that starts with a feasible solution that satisfies the given linear programming problem and then looks for a better solution that produces a larger objective function value. This process is continued until the objection function value cannot be increased. In the case of this particular IP problem, the server did indeed find the optimal solution of four section changes. (*See Appendix C*)

Final Examination Scheduling:

The problem of scheduling final examinations has been approached using a variety of methods. [C, CLC, LD, MRP] However, each problem is unique since the constraints are different. Thus, an algorithm that produces good schedules at large civilian schools might not yield the same results at the Naval Academy. The Naval Academy initially scheduled final exams by hand, using a collection of magnets. When computing facilities became available, there was a transition to the Stilwell One-Pass algorithm [St] designed by MIDN Mahlon Stilwell. In this algorithm, the courses are ordered and input into a software program. The computer then produces a schedule with a small number of conflicts. For example, in the spring of 1996, 261 courses were entered and a schedule with ninety midshipmen conflicts was produced. The Stilwell algorithm was well suited for minimizing conflicts and its code could be adjusted. However, it responded poorly to many other constraints that were placed on the schedule.

In 1996, Professor Mark Meyerson, Chair of the United States Naval Academy's Mathematics Department, researched alternative ways to produce a final examination schedule. [M] He recommended that the Naval Academy transition to the Strathmann Schedule Expert. [SSE] The Strathmann Schedule Expert produced a schedule for the spring semester of 104 conflicts. Although this exceeded the conflict number of the Stilwell algorithm, the program was simpler to use, more easily adaptable, and also responded better to many constraints placed on the schedule. Further, its schedule consisted of fewer days where midshipmen had two examinations. The downside of the Strathmann Schedule Expert is that it is proprietary and thus there is no access to the code. In other words, though Strathmann Associates works closely with the Registrar's

Office, the Naval Academy schedulers have no idea how their algorithm works. Other methods, such as the descent method, simulated annealing, and partial search methods, were also identified in Professor Meyerson's research. The descent method is an exhaustive search that begins at a schedule in the space of schedules and then moves to another schedule if it has a lower midshipmen conflict number. It continues in this manner until the search leads to a local minimum, known as a valley. The terminology of searching a "space of schedules" and "valleys" will be referred to.

Genetic Algorithms:

Another approach to constructing good final exam schedules involves using genetic algorithms. [K] Applying genetic algorithms to the final examination problem is a two-step process. First, mate two "parent" schedules to form "child" schedules. Then subject the "child" schedule to local mutation by randomly selecting and moving courses from slot to slot. If the "child" schedule is better than its parents' schedules, keep the child; otherwise, discard it and perform the process again. The example below illustrates the mating process.

Figure 6: Genetic Algorithm Example

Parent 1

MONDAY	TUESDAY	WEDNESDAY
Chemistry	Leadership	Calculus
Physics	Naval Law	Electrical Engineering
Navigation	International Relations	Thermodynamics
English	Political Science Methods	Boats

+

Parent 2

MONDAY	TUESDAY	WEDNESDAY
Boats	Political Science Methods	Physics
Calculus	Thermodynamics	Navigation
Electrical Engineering	International Relations	Naval Law
English	Leadership	Chemistry

=

Child

MONDAY	TUESDAY	WEDNESDAY
Chemistry	Leadership	Electrical Engineering
Physics	Naval Law	Political Science Methods
Navigation	Boats	Thermodynamics
English	Calculus	International Relations

In the example, the highlighted portion of parent 1 is carried over into the child. The rest of the slots in the child schedule are then taken from parent 2 in the order that they appear. In a normal genetic algorithm, the child schedule would then be subject to the local mutation step.

The benefit of the genetic algorithm is shown in the example as well. Parent 1 is unfavorable because it schedules two required sophomore-year Political Science courses, Political Science Methods and International Relations, in the same exam slot. Parent 2 is also unfavorable because it schedules two common sophomore-level courses, Physics and Navigation, in the same exam slot. Their child, however, avoids both of these trouble areas, and thus its conflict number is probably less than both of its parents.

A preliminary computer program that takes a possible schedule as input and continually applies the local mutation step was developed. In other words, it just randomly changes the slot assignment for one course and sees if the conflict number has decreased. If it has, it keeps the change; otherwise, it backtracks and tries again. In the program, 10,000 local changes were tried.

When the program was applied to the fall semester final exam data, it produced a conflict number of 24, as opposed to the conflict number of 76 obtained by the Academic Registrar. Even though the Strathmann program used by the registrar takes into account more factors than the conflict number, genetic algorithms showed tremendous promise.

A complete genetic algorithm was then developed to be applied to the spring semester exam data. The first part of the process involves generating an initial population of schedules. From the exam data, the conflict graph was formed. This graph depends on a parameter, ccut. The graph is made of vertices that represent courses that have final examinations. Two vertices are joined by an edge if there are at least ccut midshipmen taking both final examinations. Then the courses are ordered by the degree of their vertex.[2] Since the final exam schedule consists of fifteen exam slots, the fifteen

[2] The degree of a vertex is the number of edges incident to that vertex.

vertices of highest degree are chosen and placed in different exam slots. The rest of the courses are filled in randomly to complete the schedule.

After this preprocessing step, the initial population is subject to local mutation. To reiterate, the local mutation step makes random moves of a course to a different exam slot to see if the midshipman conflict number decreases. If it does, the move is made; otherwise, it is discarded and the process is repeated. Once the initial number of local mutations is made, the mating process is ready to begin.

In this genetic algorithm, the following method was devised to simulate the mating process of two schedules. Parent 1's first exam slot is carried to the child. Parent 2 then fills in the child's next exam slot with their second slot. If a course in their second slot is already in the child schedule (from Parent 1), it is just omitted. In other words, there is no chance of having the same course in the child scheduled more than once. Parent 1 then fills in the child's third exam slot, Parent 2 the fourth, and so forth. Once all of the child's slots are filled, there may be courses that were not scheduled in the child schedule. To ensure every course is present in the child, there is one more run through the exam slots. In this run, Parent 2 contributes its first exam slot to exam slot one of the child, and Parent 1 follows suit for the child's second exam slot. Following this process all the way through the exam schedule a second time ensures that every course is scheduled. An example of the intermingle mating process is provided below.

Figure 7: Intermingle Mating Process

The population undergoes the mating process for a given number of generations. After each generation, the child schedules are subject to another local mutation. The theory behind the local mutation process is that often in the search for an optimal schedule in terms of low conflict number, a local minimum can be encountered. In other words, when searching the space of schedules, the lowest conflict number in a small region might be found, vice the smallest number overall. In these cases, it is often favorable to make "jumps," where the search can move to another region. Although this

might lead to a higher value conflict number initially, the search in this new region may eventually descend to a lower valley. To ensure that there is not a transition to higher valleys, a given number of "best" schedules are kept. In other words, after every generation a given number of favorable schedules, or those with the lowest conflict number, are passed to the next generation for more mating. In this way, the most favorable schedules from previous generations are always kept and hopefully these will improve further with more generations. The complete genetic algorithm was as follows:

Genetic Algorithm

1. Form initial population of schedules. Use the conflict graph to separate the "most conflicted" courses and then randomly fill in the remaining courses in a given exam slot.
2. Subject the initial population to local mutations, or random moves of courses between exam slots.
3. Mate for a given number of generations. Mating two parents involves alternatively filling in the consecutive exam slots with the corresponding courses from the parents.
    a. Subject the children schedules to local mutations after each generation.
    b. Produce more child schedules by mating random pairs in the current population. Discard the current population but keep a given number of best schedules.
4. Subject the population to one more local mutation.
5. Follow the process for incorporating courses with long grading concerns. (Explained below)

When the genetic algorithm program was first run on the Spring 2003 data, the results were extremely good. The best schedule had nine midshipman conflicts and eight course conflicts. The final exam schedule produced for this semester with the Strathmann Schedule Expert had well over 150 midshipmen conflicts. These numbers were a little misleading, however, considering that there were many factors, such as long grading considerations and exams that must be taken together, that were taken into account in the Strathmann program. Nonetheless, when the Strathmann program was run without any of these factors included, its best result was still over thirty conflicts. This

genetic algorithm program had produced a raw schedule with one third of the midshipmen conflicts.

There were still many concerns with the initial genetic algorithm program. In particular, there were three extra factors that needed to be incorporated into the genetic algorithm code. The first factor was setting a limit on the amount of midshipmen scheduled to take an exam in a given exam slot. The Academic Registrar currently uses two thousand midshipmen as a capacity for each slot. In the Spring 2003 schedule with only nine midshipmen conflicts, there were exam slots with over 2800 and 3700 midshipmen scheduled to take an exam. Thus, the constraint that only 2400 midshipmen can take an exam in one exam slot was added. If this value were exceeded, the overflow would be added to the conflict number, or total cost.

The next constraint incorporated into the genetic algorithm program was the requests by various academic departments to have certain final exams given at the same time. These "groupings" were added into the original exam data file by making any courses fitting this criterion into one larger course. With the new "groupings" consideration and maximum exam slot capacity, the genetic algorithm program was tested again. This time, the best results were twelve midshipman conflicts and twelve course conflicts. Furthermore, none of the exam slots exceeded midshipman capacity. The number of midshipmen in each exam slot ranged from 636 to 2215, and the average was 1390.5.

The final constraint added to the genetic algorithm program deals with proximity concerns. Every semester, each academic department can submit requests for courses they feel require a long time for grading. This list of courses was made into a data file

and a method to accommodate the largest amount of courses possible was devised. The exam slots in the final schedule produced by the genetic algorithm program are distinct, and thus they can be arranged in any order. Therefore every course with a long grading concern was labeled with a value one. All of the other courses had value zero. These values were summed for each exam slot. Then the slots were placed in descending order so that the slots with more long grading courses went first.

Analysis:

There are seven variables in the genetic algorithm program. They include: the threshold in the conflict graph ("ccut"), the initial size of the population, the number of local mutations performed on the initial population, the number of generations, the number of children, the number of these children that were good schedules and kept for more matings, and the number of local mutations performed on the children after each generation. The initial experimentation began with establishing a baseline of variable values (that were picked following promising initial results) and then isolating a given variable to see its effect on the number of midshipmen conflicts, course conflicts, and the time it took to produce the final schedule. This effect was determined by changing a designated variable's baseline value over successive trials while fixing all of the other variables. For each part of this first set of experiments, five trials were performed for each value of a given variable. The following are charts showing the results of these isolation experiments. The midshipmen conflict numbers are averages of the five trials.

Figure 8

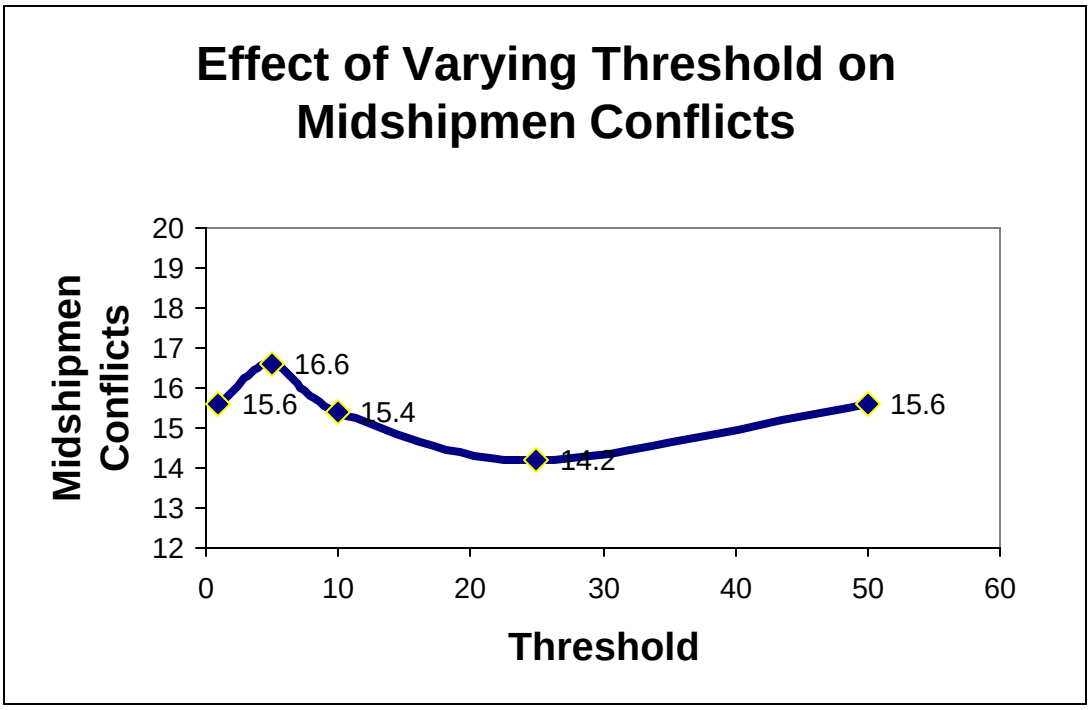


Figure 9

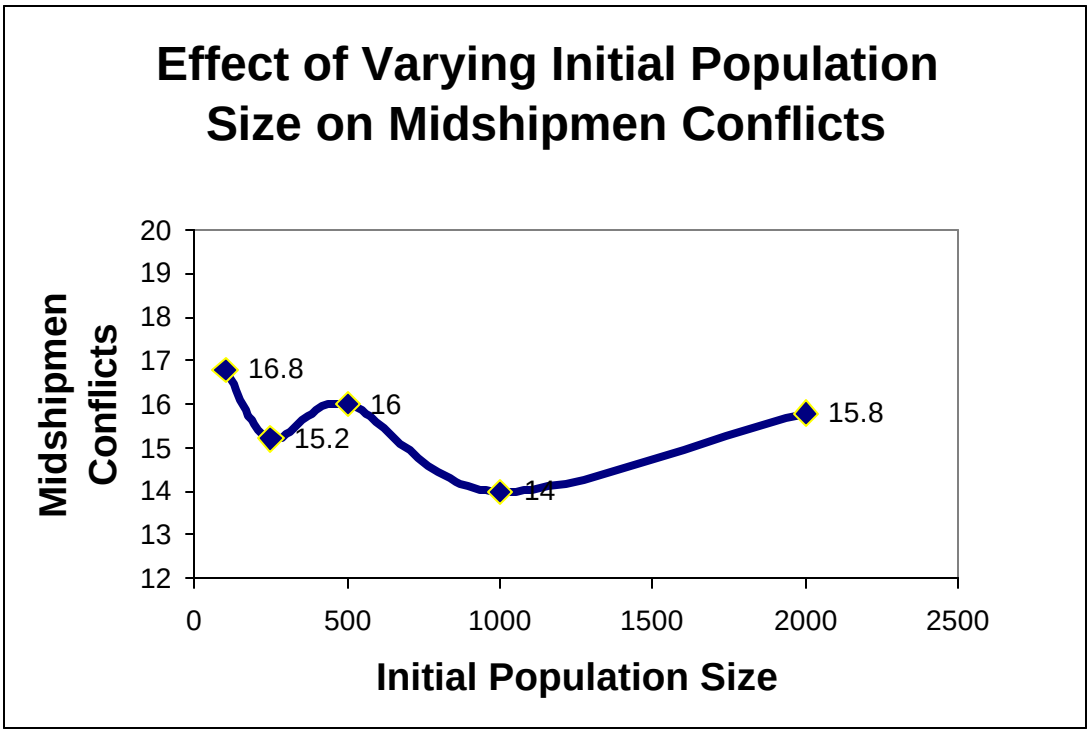


Figure 10

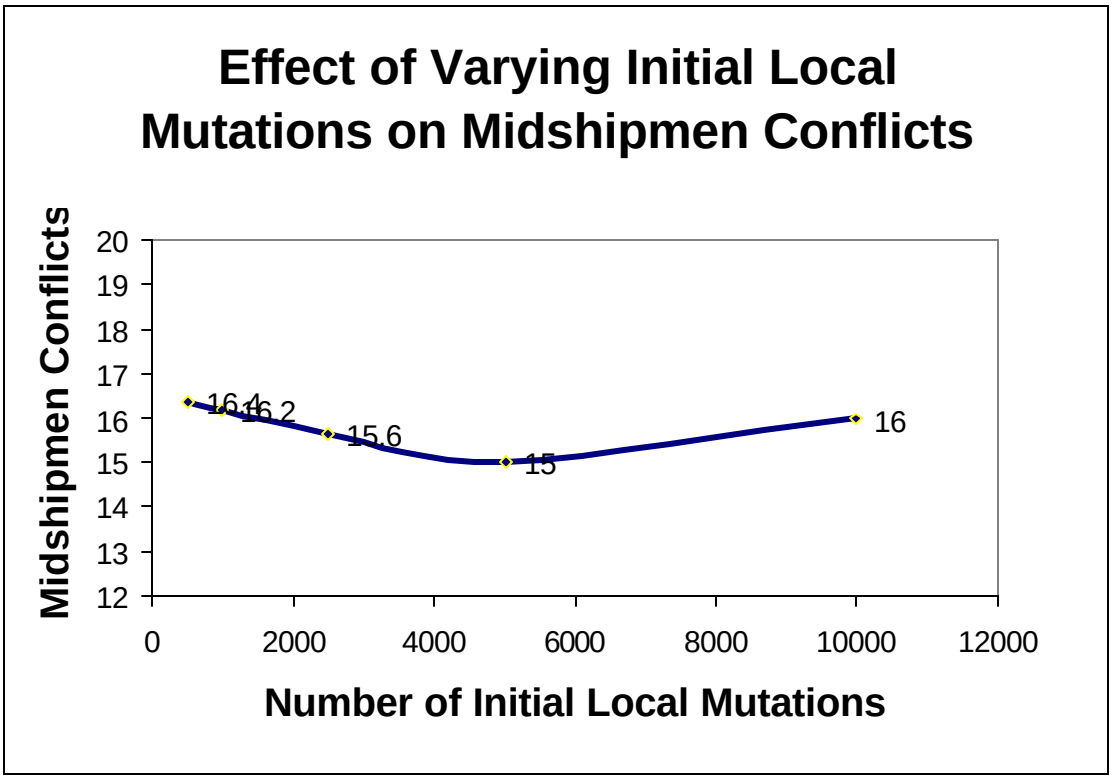


Figure 11

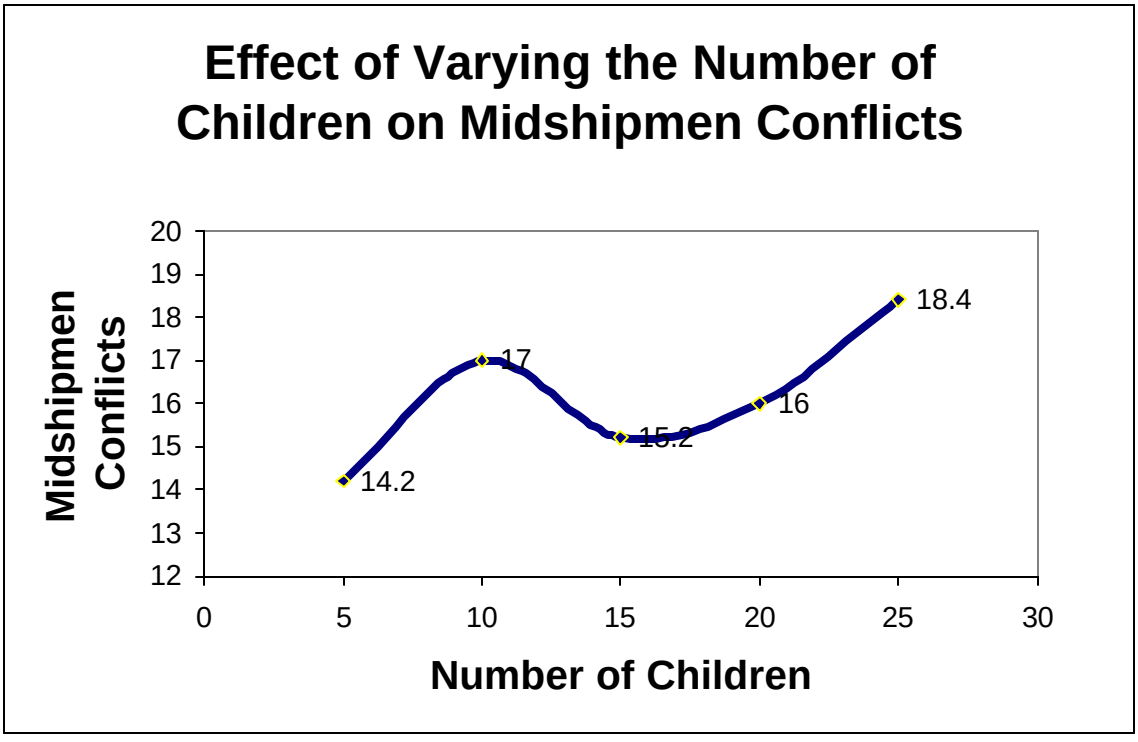


Figure 12

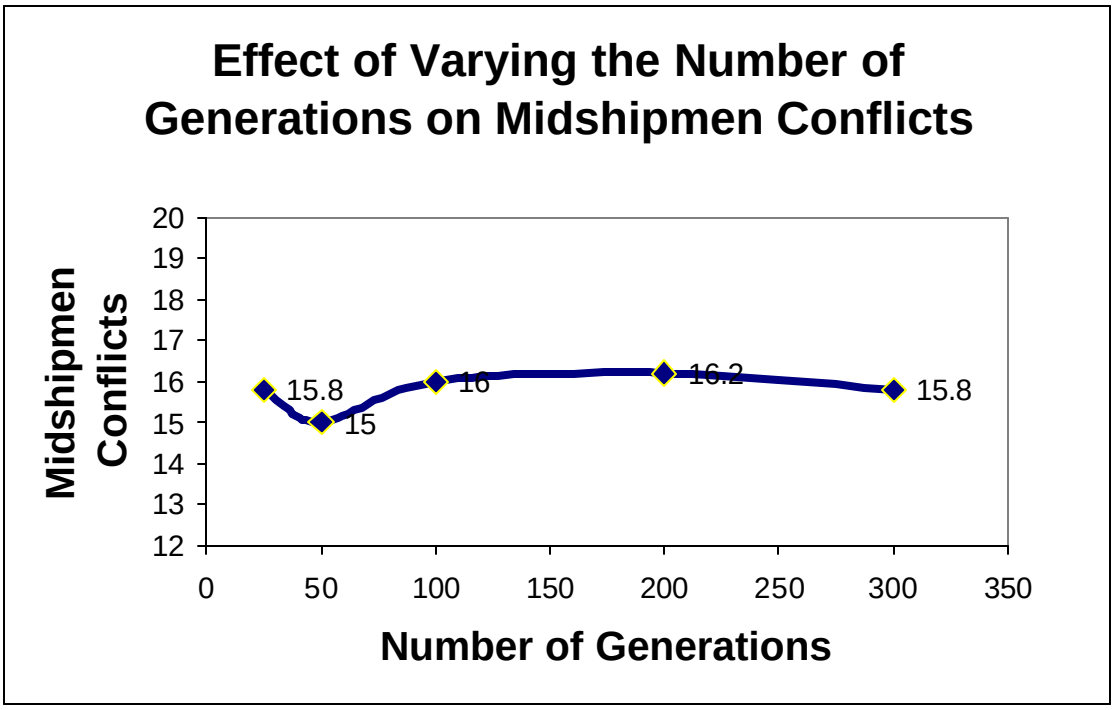


Figure 13

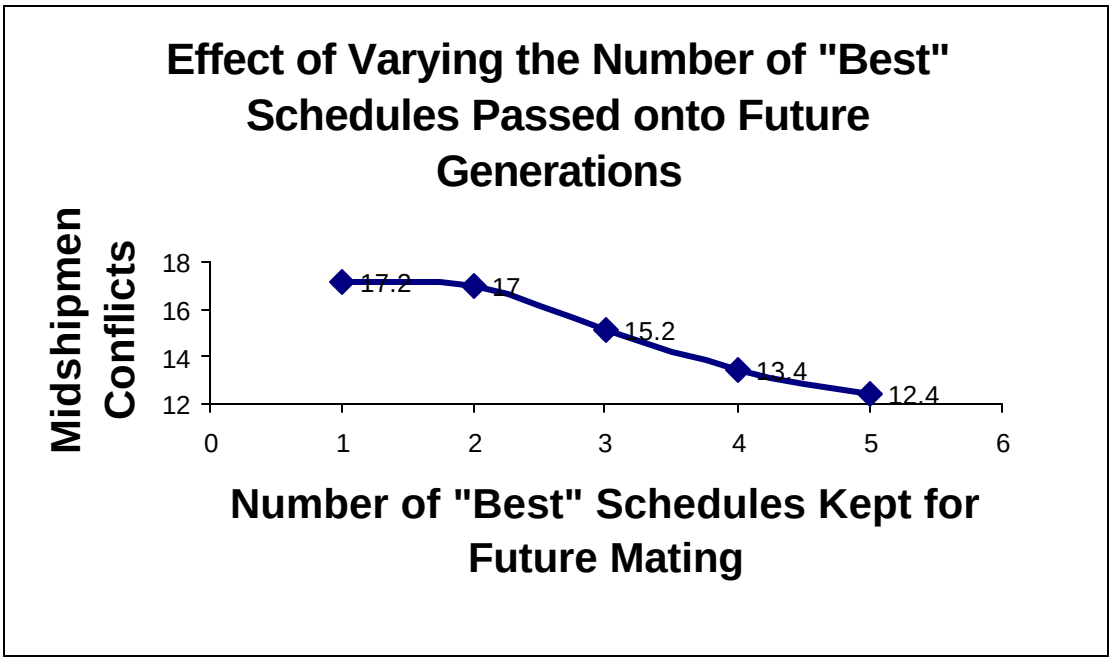
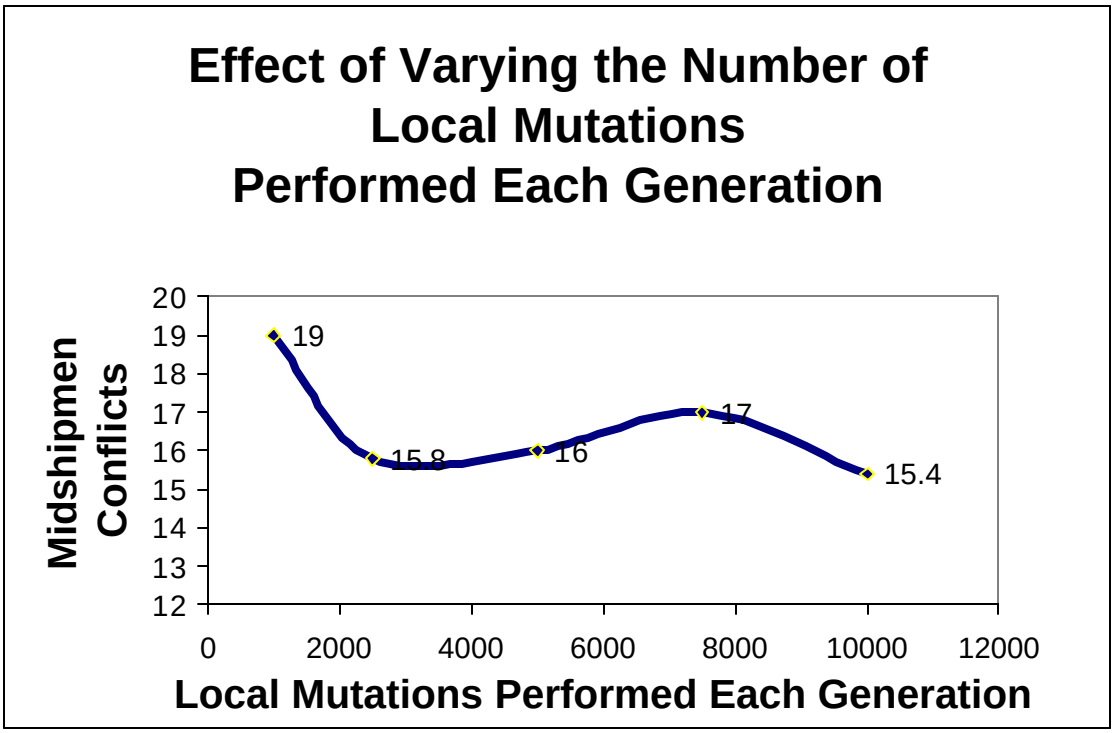


Figure 14



In terms of pure conflict number, most of the variables did not follow a consistent pattern. This displays the randomness of the genetic algorithm, which is indeed desired in such a scheduling problem heuristic. Two variables, however, did show some correlation in their experiments. When the "ccut" variable was increased, there was a gradual decline in the number of course conflicts and essentially no change in the time. The number of "best" schedules after each generation also showed a consistent response in the results. When this value was increased from one to five, there was a significant decrease in the number of midshipmen conflicts and no effect on the time. In other words, the lowest group of midshipmen conflict numbers resulted when five out of the ten schedules produced from each generation were "best" schedules from previous generations. Throughout all of this first set of experiments, which consisted of over 175

trials, the best result was ten midshipmen conflicts, ten course conflicts, and a time of 471 seconds, or roughly eight minutes. The whole experiment took nearly 24 hours on one processor. However, the genetic algorithm program can be easily run on several processors in parallel. Thus, the running time would be less on the sixteen processors of the Beowulf cluster, which is a parallel processing network located in Chauvenet Hall.

The next set of experiments sought to find out whether the mating process was more significant than the local mutation step, or vice versa. An age component was added to the genetic algorithm program. When the experiment outputs the best schedule from each generation, it would now also indicate the age of the schedule. The age of the schedule is the number of local mutations that have been performed to the schedule. This set of experiments had fewer generations than the normal baseline and less mutations after each generation. Another addition to the genetic algorithm program for this experiment was a part to the mutation step that would equate the number of mutations performed on the newly produced children schedules to the kept "best" schedules. The number of "best" schedules kept from previous generations was then varied. The following graphs display the results of this set of experiments.

Figure 15

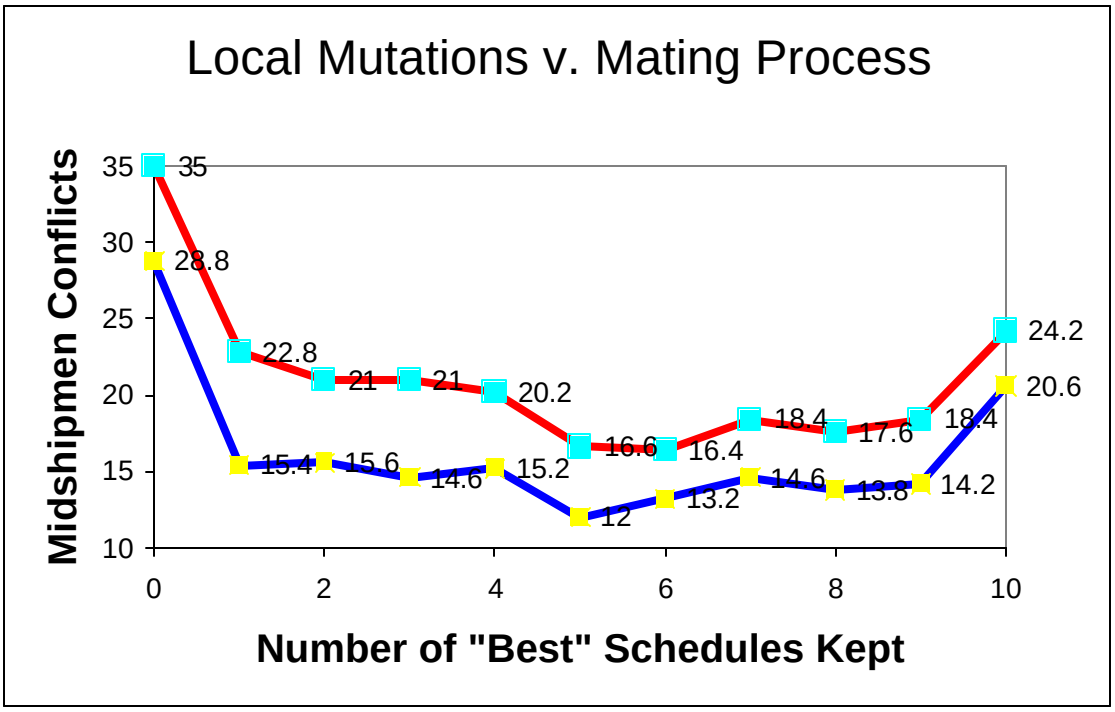
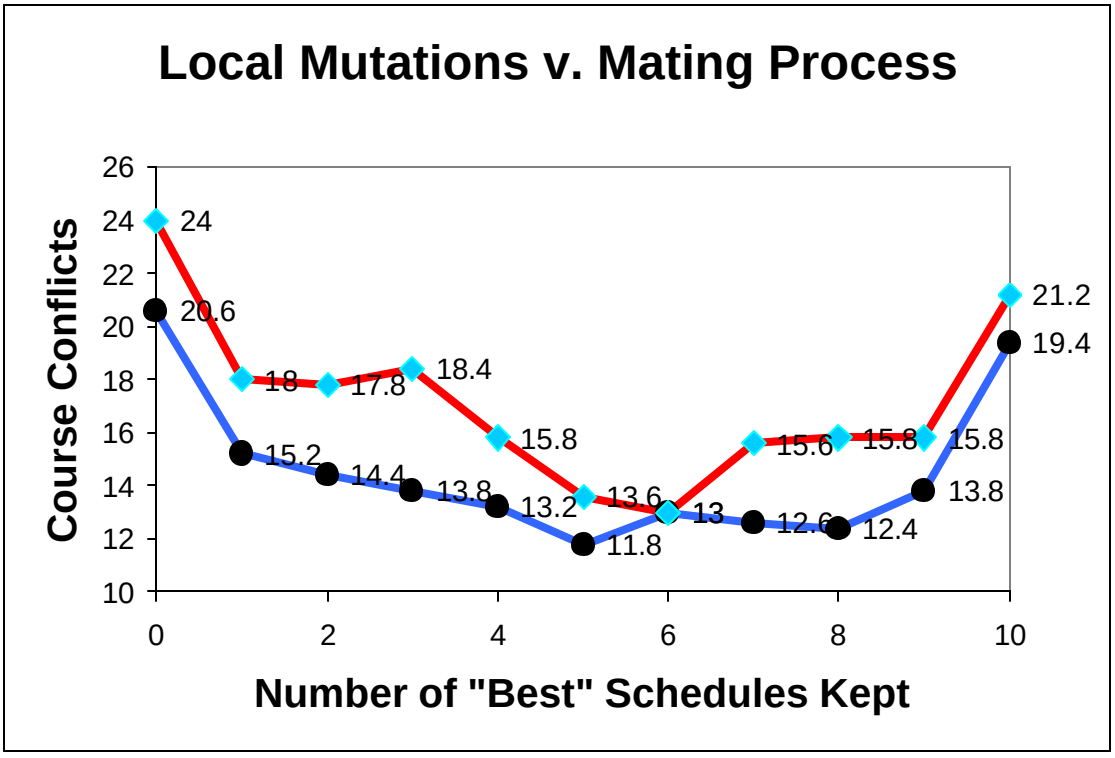


Figure 16



This final set of experiments showed that both the mating process and the local mutation steps are important. As the amount of "best" schedules was increased from zero to five, there was a significant decrease in the amount of midshipmen conflicts. In fact, the best Spring semester schedule of ten midshipmen conflicts, nine course conflicts, and only 556 seconds (just under 10 minutes) was produced when five out of the ten schedules were "best" schedules. When this number was further increased, the schedules increased in conflict number. This correlates to the fact that when there was less of a significant mating process, the schedules became less favorable. When this experiment was run on the Fall semester exam data, there were similar results. This showed that the number of schedules kept from previous generations should be around half of the number of total children used in the mating process.

The local mutation step probably accounts for most of the genetic algorithm's efficiency. However, this step only serves to get the midshipman conflict number to the lowest point in the local region. Conversely, the mating process has the potential to make a drastic enough change in the children schedules to move to another region and seek its lowest point. By keeping a certain number of "best" schedules from previous generations, the algorithm does not suffer when the search moves to another region with a higher valley. Therefore, the local mutation step allows the search to reach the best possible schedule in the local area, and keep it on record in case it is the best schedule in all of the areas. The mating process is equally important because it allows the search to span more of the space of schedules and potentially find a region that descends to a lower midshipmen conflict number. Overall, both contribute to the genetic algorithm.

An important practical question is how to best take advantage of increased running time (for example, if the Registrar wants to run the program overnight). Several possibilities are: increase the initial population size, increase the number of schedules in each generation, increase the number of generations and increase the number of local mutations in each generation. However, increasing many of these variables just adds time, but does not decrease the midshipmen conflict number. For instance, increasing the number of generations does not produce better schedules, as Figure 12 shows. If more time is allocated, the number of local mutations in each generation should be increased. This assessment is supported by Figure 14, where the lowest midshipmen conflict number is obtained when the number of local mutations is greatest.

Recommendations:

The course bartering system is a reasonable addition to the course scheduling process. The "Cycle Elimination" program is extremely efficient in finding the cycles that would correspond to possible course-section switches between midshipmen. Furthermore, network flow techniques provide an algorithm to determine the optimal accommodation of midshipmen. Finally, the integer programming approach also provides an efficient way of finding the optimal accommodation. This approach allows for midshipmen to submit more than one change to their schedule. In addition, the model of the course bartering system as an integer programming problem is quite similar to other well-known problems that could be examined. It is recommended that the course bartering system be added to the scheduling process and that either the network flow algorithm or integer programming approach be used.

In terms of final examination scheduling, the genetic algorithm program is extremely efficient in finding good schedules. Its raw best schedule produced one third less midshipmen conflicts than the Strathmann Schedule Expert that is used today. Many of the additional constraints were added into the program, such as the grouping of course exams where desired, early exams for those courses requiring a long grading period, and a maximum capacity of midshipmen for each exam slot. Even with these additions, a Spring semester schedule with ten midshipmen conflicts and nine course conflicts was produced in roughly ten minutes. The baseline values for the fixed variables in the second set of experiments should be kept the same. The number of "best" schedules passed onto future generations should be half of the total children, or in this case, five. Note that the Strathmann Schedule Expert is still very good at producing exam statistics

and incorporating other constraints. Ultimately, it is recommended that the genetic algorithm program be used to find an initial schedule that can be used as input into the Strathmann program. Then the Strathmann software could be used to add any further improvements to the schedule if necessary.

Bibliography

[BEW] Burke, E.K.; Elliman, D.G. and Weare, R. *A University Timetabling System Based on Graph Colouring and Constraint Manipulation*. Journal of Research on Computing in Education **27**, no. 1 (1994), 1—18.

[BMMN] Ball, M.O.; Magnanti, T.L.; Monma, C.L.; Nemhauser, G.L. Network Models. Elsevier, Amsterdam, 1995.

[BR] Brualdi, R.A. Introductory Combinatorics, Third edition. Prentice Hall, Upper Saddle River, NJ, 1999.

[C] Carter, Michael W. *A Survey of Practical Applications of Examination Timetabling Algorithms.* Operations Research no. 2 (1986), 193 – 202.

[CLC] Carter, Michael W.; Laporte, Gilbert and Chinneck, John W. *A General Examination Scheduling System*. Interfaces **24 (**1994), 109—120.

[CR1] Chandru, Vijay; Rao, M. R. *Integer programming*. Algorithms and theory of computation handbook, 32-1--32-45, CRC, Boca Raton, FL, 1999.

[CR2] Chandru, Vijay; Rao, M. R. *Linear programming*. Algorithms and theory of computation handbook, 31-1--31-37, CRC, Boca Raton, FL, 1999.

[CCPS] Cook, W.J; Cunningham, W.H.; Pulleyblank, W.R.; Schrijver, A. Combinatorial Optimization. John Wiley & Sons, New York, 1998.

[DIMACS] Johnson, D.S.; McGeoch, C.C., editors. Network Flows and Matching. American Mathematical Society, Providence, RI, 1993.

[H] Hu, T.C. Combinatorial Algorithms. Addison-Wesley, Reading, MA, 1982.

[K] Kreher, Donald L.; Stinson, Douglas R. Combinatorial Algorithms: Generation, Enumeration, and Search. CRC, Boca Raton, FL, 1999.

[KA] Karp, R.M. *A Characterization of the Minimum Cycle Mean in a Digraph.* Discrete Mathematics **23** (1978), 309—311.

[KV] Korte, B.; Vygen, J. Combinatorial Optimization. Second Edition. Springer, Berlin, 2002.

[LD] Laporte, G.; Desroches, S. *Examination Timetabling by Computer*. Computer Operations Research, **11** (1984), 351 – 360.

[M] Meyerson, Mark D. *On the Mathematics of Exam Scheduling at USNA*. Unpublished report (1996).

[MRP] Mooney, Edward L.; Radin, Ronald L. and Parmenter, W. J. *Large Scale Classroom Scheduling*. IIE Transactions, **28:5** (1996).

[OA] Orlin, J.B.; Ahuja, R.K. *Minimum Cost Flows*. In: Handbook of Discrete and Combinatorial Mathematics. Kenneth H. Rosen, editor. CRC, Boca Raton, FL, 2000. 673 – 682.

[OMA] Orlin, J.B.; Magnanti, T.L.; Ahuja, R.K. Network Flows: Theory, Algorithms, and Applications. Prentice Hall, Englewood Cliffs, NJ, 1993.

[PMX] Pardalos, Panos M.; Mavridou, Thelma and Xue, Jue. *The Graph Coloring Problem: A Bibliographic Survey*. Pages 331—395 In: Handbook of Combinatorial Optimization, Vol. 2. Edited by: D-Z. Du and P.M. Pardalos (1998), Kluwer Academic Press.

[S] Strayer, J.K.; Linear Programming and Its Applications. Springer-Verlag, New York, 1989.

[SSE] *The Strathmann Schedule Expert*. Software program. Strathmann and Associates.

[St] Stilwell, Mahlon F. *A Computer-Assisted Examination Schedule*. Unpublished report.

[V] Vanderbei, Robert J. Linear Programming: Foundations and Extensions. Kluwer Academic, Boston, 1996.

Appendix A: Cycle Elimination Program

Purpose: This C++ program uses an iterative process to find and remove directed cycles in a given graph.

Input: A data file consisting of the course bartering system graph. In this graph, vertices represent course section pairs. Directed edges join two vertices when a midshipmen requests to move from one section to another.

Description: The program first performs a simplification process whereby any vertex without both incoming and outgoing edges is eliminated. It then uses a depth-first search to find a directed cycle in the graph. Once a cycle is found, it is removed and recorded. The process is repeated until the graph consists of the null set.

Output: All of the alpha codes and corresponding switches that were made on a given graph.

```
/************************************
 ** This version takes the first
 ** cycle it finds and removes it!
 ************************************/
#include <iostream>
#include <vector>
#include <string>
#include <stdlib.h>
using namespace std;

//Define new type edge
class edge
{
public:
  bool used;
  int weight;
  int destination;
  vector <string> A;
};

//Define new type vertex
class vertex
{
public:
  bool live;
  vector <edge> N;
  string course;
  string period;
  char color;
};

int getindex(vector<vertex>&,string,string);
void addedge(vector<vertex>&,int,int,string);
void print(vector<vertex> &);
bool are_out_edges(vector<vertex>&,int);
bool are_in_edges(vector<vertex>&,int);
bool simplify(vector<vertex>&);
void deletevertex(vector<vertex>&,int);
vector<int> findcycle(vector<vertex>&);
void deleteedge(vector<vertex>&,int,int);
bool nonempty(vector<vertex>&);
void deletepath(vector<vertex>&, vector<int> &);
int index_in_Nbrs(vector<vertex>&, int, int);
void printcycle(vector<vertex>&,vector<int>&);

int main()
{
  // Read in number of vertices, and rest on first line
  int n;
  string temp;
  cin >> n >> temp >> temp;

  // Read vertices of course  period
  vector<vertex> V;
  vertex w;
  for(int i = 0; i < n; i++)
  {
```

```
    cin >> w.course >> w.period;
    w.live = true;
    V.push_back(w);
  }

  // Read in number of edges, and rest on line
  int e;
  cin >> e >> temp >> temp;

  // Read and store edges!
  for(int i = 0; i < e; i++)
  {
    // Read through student info, which we'll ignore for now!
    cin >> temp;

    // Read in edge info
    string alpha, course0, period0, course1, period1;
    cin >> alpha >> course0 >> period0 >> course1 >> period1;

    // Add edge to graph
    int s = getindex(V,course0,period0);
    int d = getindex(V,course1,period1);
    addedge(V,s,d,alpha);
   }

  // Print graph
  print(V);
  cout << endl << endl;

  bool f = true;
  while(f)
    f = simplify(V);
  cout << "After simplify: " << endl;
  print(V);
  cout << endl;

  while(nonempty(V))
  {
    vector<int> Cycle = findcycle(V);
    printcycle(V,Cycle);
    deletepath(V,Cycle);

    cout << "After path delete: " << endl; print(V);

    bool f = true;
    while(f)
      f = simplify(V);
    cout << "After simplify: " << endl;
    print(V);
    cout << endl;
   }
  print(V);
 return 0;
}

bool nonempty(vector<vertex>& V)
{
```

```
  for(int a = 0; a < V.size(); a++)
    if(V[a].live == true)
      return true;
  return false;
}

// Gets the index in V of the vertex (course,period)
int getindex(vector<vertex>& V,string course, string period)
{
  int index = 0;
  while(index < V.size())
  {
    // Is V[index] the one we want?
    if (V[index].course == course && V[index].period == period)
      return index;
    index++;
  }

  // If we ever get here, things are messed up!  We didn't find the
vertex!
}

// Add edge assuming that such an edge is not already in the graph!
void addedge(vector<vertex>& V,int s,int d, string alpha)
{
  int k = index_in_Nbrs(V,s,d);
  if (k > -1)
  {
    // Edge already exists
    V[s].N[k].weight++;
    V[s].N[k].A.push_back(alpha);
  }
  else
  {
    // Create the edge object to add
    edge newedge;
    newedge.used = false;
    newedge.weight = 1;
    newedge.destination = d;
    newedge.A.push_back(alpha);

    // Add edge to vector V[s].N
    V[s].N.push_back(newedge);
  }
}

//Print the graph with given vertices and edges!
void print(vector<vertex> &G)
{
  // loop through each vertex & print edges out of that vertex
  for(int u = 0; u < G.size(); u++)
  {
    if (G[u].live)
    {
      for(int k = 0;  k < G[u].N.size(); k++)
      {
        int v = G[u].N[k].destination;
```

```
        cout << "Edge: (" << G[u].course << "," << G[u].period
             << ") to (" << G[v].course << "," << G[v].period << ")"
             << " of weight " << G[u].N[k].weight << endl;
      }
    }
  }
}


bool are_out_edges(vector<vertex>& G, int u)
{
  if (G[u].N.size() == 0)
    return false;
  else
    return true;
}

bool are_in_edges(vector<vertex>& G, int u)
{
  for(int a  = 0; a < G.size(); a++)
  {
    if (G[a].live)
    {
      for(int k = 0;  k < G[a].N.size(); k++)
      {
        if (G[a].N[k].destination == u)
          return true;
      }
    }
  }
  return false;
}

// Simplification of graph:
// returns true if graph was changed and false otherwise
bool simplify(vector<vertex>& G)
{
  bool found = false;
  for(int u = 0; u < G.size(); u++)
  {
    if(G[u].live &&(are_out_edges(G,u) == false || are_in_edges(G,u) ==
false))
    {
      //code that deletes vertex u!
      deletevertex(G,u);
      found = true;
    }
  }
  return found;
}

//Function that will delete a vertex!
void deletevertex(vector<vertex>& G, int u)
{
  // Set vertex u to dead
  G[u].live = false;
```

```
  // Delete any edges into u
  for(int a  = 0; a < G.size(); a++)
  {
    if (G[a].live)
    {
      for(int k = 0;  k < G[a].N.size(); k++)
      {
        if (G[a].N[k].destination == u)
        {
          int n = G[a].N.size() - 1;
          edge t = G[a].N[n];
          G[a].N[n] = G[a].N[k];
          G[a].N[k] = t;
          G[a].N.pop_back();
        }
      }
    }
  }
}

bool dfvisit(vector<vertex>& G, int v, vector<int>& Path)
{
  G[v].color = 'g';
  Path.push_back(v);

  for(int i = 0; i < G[v].N.size(); i++)
  {
    int w = G[v].N[i].destination;
    if (G[w].live)
    {
      bool cyc = false;
      if (G[w].color == 'w')
        cyc = dfvisit(G,w,Path);
      else if (G[w].color == 'g')
      {
        Path.push_back(w);
        cyc = true;
      }

      if (cyc) return true;
    }
  }

  G[v].color = 'b';
  Path.pop_back();
  return false;
}

vector<int> findcycle(vector<vertex>& G)
{
  //Initialize the color of all vertices to white
  for(int i = 0; i < G.size(); i++)
    G[i].color = 'w';

  vector<int> Path;

  for(int v = 0; v < G.size() && Path.size() == 0; v++)
```

```
  {
    if (G[v].color == 'w' && G[v].live)
      dfvisit(G,v,Path);
  }

  return Path;
}

void printcycle(vector<vertex> &G, vector<int> &Path)
{
  if (Path.size() > 0)
  {
    // Find first entry of Path that matches the *last* index
    int i = 0;
    while(Path[i] != Path[Path.size()-1])
      i++;

    // Print from Path[i] onwards
    while(i < Path.size()-1)
    {
      // Print switch from Path[i] to Path[i+1]
      cout << G[Path[i]].course << G[Path[i]].period << " " <<  "to" <<
" "
           << G[Path[i+1]].course << G[Path[i+1]].period << ' ';

      //Find edge from Path[i] to Path[i+1]
      int k = index_in_Nbrs(G,Path[i],Path[i+1]);

      //Print out the last alpha associated with this edge
      cout << G[Path[i]].N[k].A[ G[Path[i]].N[k].A.size()-1 ] << endl;

      i++;
    }
    cout << endl;
  }

}

void deleteedge(vector<vertex>& G, int s, int d)
{
  int a = index_in_Nbrs(G,s,d);

  if (G[s].N[a].weight == 1)
  {
    // swap G[s].N[a] with the last element of G[s].N
    int last = G[s].N.size() - 1;
    G[s].N[a] = G[s].N[last];
    // delete the edge
    G[s].N.pop_back();
  }
  else
  {
    G[s].N[a].weight--;
    G[s].N[a].A.pop_back();
  }
  return;
}
```

```
void deletepath(vector<vertex>& G, vector<int> &Path)
{
  int n = 0;
  while(Path[n] != Path[Path.size() - 1])
    n++;
  while( n < Path.size() - 1)
  {
    int s = Path[n];
    int d = Path[n+1];
    deleteedge(G,s,d);
    n++;
  }
}

int index_in_Nbrs(vector<vertex>& V, int s, int d)
{
  for(int i = 0; i < V[s].N.size(); i++)
    if(V[s].N[i].destination == d)
      return i;
  return -1;
}
```

```
/Sample data set to be tested

/6 course-period pairs

SM101 1st
SM101 3rd
SM101 6th
HH202 1st
HH202 3rd
HH202 6th


10 Mid preferences

Jones 046754 SM101 3rd SM101 6th
Brown 037876 SM101 1st SM101 3rd
Smith 049888 SM101 6th SM101 1st
Thomas 031234 SM101 1st HH202 6th
Poindexter 041234 HH202 1st HH202 6th
Kant 056789 HH202 6th HH202 3rd
Cohen 034567 HH202 6th SM101 6th
Colt 057891 SM101 3rd SM101 1st
Holmes 034657 HH202 6th SM101 1st
Freud 051234 SM101 1st HH202 3rd
*/
```

Appendix B: Sample Run of the Cycle Elimination Program

Data File:

6 course-period pairs

SM101 1st
SM101 3rd
SM101 6th
HH202 1st
HH202 3rd
HH202 6th

10 Mid preferences

Jones 046754	SM101 3rd SM101 6th
Smith 049888	SM101 6th SM101 3rd
Thomas 031234	SM101 3rd SM101 6th
Poindexter 041234	SM101 6th HH202 6th
Kant 056789	HH202 6th HH202 3rd
Cohen 034567	HH202 3rd SM101 3rd
Colt 057891	SM101 3rd SM101 1st
Holmes 034657	HH202 6th SM101 1st
Freud 051234	SM101 1st HH202 3rd
Jos 046754	SM101 3rd SM101 6^{th}

Cycle Elimination Program on Data File

Edge: (SM101,1st) to (HH202,3rd) of weight 1
Edge: (SM101,3rd) to (SM101,6th) of weight 3
Edge: (SM101,3rd) to (SM101,1st) of weight 1
Edge: (SM101,6th) to (SM101,3rd) of weight 1
Edge: (SM101,6th) to (HH202,6th) of weight 1
Edge: (HH202,3rd) to (SM101,3rd) of weight 1
Edge: (HH202,6th) to (HH202,3rd) of weight 1
Edge: (HH202,6th) to (SM101,1st) of weight 1

After simplify:
Edge: (SM101,1st) to (HH202,3rd) of weight 1
Edge: (SM101,3rd) to (SM101,6th) of weight 3
Edge: (SM101,3rd) to (SM101,1st) of weight 1
Edge: (SM101,6th) to (SM101,3rd) of weight 1
Edge: (SM101,6th) to (HH202,6th) of weight 1
Edge: (HH202,3rd) to (SM101,3rd) of weight 1

Edge: (HH202,6th) to (HH202,3rd) of weight 1
Edge: (HH202,6th) to (SM101,1st) of weight 1

SM1013rd to SM1016th 046754
SM1016th to SM1013rd 049888

After path delete:
Edge: (SM101,1st) to (HH202,3rd) of weight 1
Edge: (SM101,3rd) to (SM101,6th) of weight 2
Edge: (SM101,3rd) to (SM101,1st) of weight 1
Edge: (SM101,6th) to (HH202,6th) of weight 1
Edge: (HH202,3rd) to (SM101,3rd) of weight 1
Edge: (HH202,6th) to (HH202,3rd) of weight 1
Edge: (HH202,6th) to (SM101,1st) of weight 1
After simplify:
Edge: (SM101,1st) to (HH202,3rd) of weight 1
Edge: (SM101,3rd) to (SM101,6th) of weight 2
Edge: (SM101,3rd) to (SM101,1st) of weight 1
Edge: (SM101,6th) to (HH202,6th) of weight 1
Edge: (HH202,3rd) to (SM101,3rd) of weight 1
Edge: (HH202,6th) to (HH202,3rd) of weight 1
Edge: (HH202,6th) to (SM101,1st) of weight 1
HH2023rd to SM1013rd 034567
SM1013rd to SM1016th 031234
SM1016th to HH2026th 041234
HH2026th to HH2023rd 056789

After path delete:
Edge: (SM101,1st) to (HH202,3rd) of weight 1
Edge: (SM101,3rd) to (SM101,6th) of weight 1
Edge: (SM101,3rd) to (SM101,1st) of weight 1
Edge: (HH202,6th) to (SM101,1st) of weight 1
After simplify:

Empty graph

Appendix C

The following code was used when the course bartering system's "Hanging House Network" was modeled as a linear / integer programming problem and submitted to NEOS Servers. NEOS Servers can be found at www-neos.mcs.anl.gov.

```
Set Students;
Set Sections;

param ADDDROP{Students, Sections) default 0;

var w{Students} >= 0, <=1; # binary;
var r{Sections} <= 0;

maximize Total_Change: sum {i in Students} w[i]
subject to Change_Section{j in Sections}: sum {i in students} w[i] *AddDrop[i,j] = r[j];
```

```
set Students := Cindy David Matt Jennifer Mike Will James;
set Sections := SM112SI SM112S2 SM112S3 SM112S4 SM112S5 SM112S6;

param AddDrop :=
Cindy SM112S1 -1
Cindy SM112S2 1
David SM112S2 -1
David SM112S6 1
Matt SM112S3 -1
Matt SM112S6 1
Jennifer SM112S3 1
Jennifer SM112S4 -1
Mike SM112S2 1
Mike SM112S5 -1
Will SM112S4 1
Will SM112S5 -1
James SM112S5 1
James SM112S6 -1;
```

```
solve;
display w, r;
```


```
************************************************************

                    NEOS Server Version 4.0

    Job#    : 231584
    Solver : XPRESS (AMPL input)
    Start   : 03/04/2003 10:57:30
    End     : 03/04/2003 10:57:38
    Host    : pergamon.iems.northwestern.edu

    Kestrel Interface: check out our new client for sending jobs
    to the NEOS Server from your AMPL or GAMS modeling session
    and receiving results back into the session for further
    computation.  See http://www-neos.mcs.anl.gov/neos/kestrel.html

    Feedback Requested!!
    Our funding agency has asked for a report on our work, and
    feedback from our users would be greatly appreciated.  The
    information of interest is the type of work you are doing
    on NEOS, whether for business, education, or other purposes,
    and the solvers used.
    Please send your comments to dolan@mcs.anl.gov, more@mcs.anl.gov.

    Disclaimer:

    This information is provided without any express or
    implied warranty. In particular, there is no warranty
    of any kind concerning the fitness of this
    information for any particular purpose.

************************************************************


%% XPRESS OUTPUT %%%%
The license for this AMPL processor will expire in 11.3 days.

13 variables, all linear
6 constraints, all linear; 20 nonzeros
1 linear objective; 7 nonzeros.

XPRESS-MP Workstation Integer Barrier Optimiser Release 12.13
(c) Copyright Dash Associates 1984-2000
XPRESS-MP 12.13: maxtim=3600
XPRESS-MP 12.13: Optimal solution found; objective 4
 5 Simplex iterations
:          w   r    :=
Cindy      0   .
David      0   .
James      1   .
Jennifer   1   .
Matt       1   .
Mike       0   .
SM112S1    .   0
SM112S2    .   0
SM112S3    .   0
SM112S4    .   0
SM112S5    .   0
SM112S6    .   0
Will       1   .
;
```

Appendix D: Schedule Formation Program

Purpose: This program provides an efficient way of recording and maintaining all of the statistics of a final examination schedule. These include: number of midshipmen conflicts, number of course conflicts and the number of midshipmen that exceed the 2400-person capacity of each examination time slot.

Input: A base schedule

Description: This program essentially introduces the statistics by which a final examination schedule will be measured.

Output: N/A

```
/************************************************
 ** Definition of class Schedule and related functions.
 **
 ** A Schedule is simply a BaseSchedule augmented with
 ** facilities for efficiently keeping track of the
 ** total number of midshipmen and course conflicts.
 ** The key new functions (i.e. not in BaseSchedule)
 ** are:
 **
 ** int numconflicts();         // # mid conflicts
 ** int num_course_conflicts(); // # course conflicts
 **
 ** A "mid conflict" is a tuple ({c1,c2},alpha) such
 ** that midshipman alpha is enrolled in courses c1
 ** and c2, and c1 and c2 are scheduled in the same
 ** slot.  "numconflicts()" returns the number of
 ** distinct mid conflicts.  A course conflict is
 ** an unordered pair (c1,c2) such that course c1 and course c2
 ** are scheduled in the same slot.  The function
 ** "num_course_conflict" returns the number of
 ** distinct pairs of this kind.
 ************************************************/
#include "BaseSchedule.h"

extern int MaxPerSlot;
inline void setMaxPerSlot(int M) { MaxPerSlot = M; }

class Schedule : public BaseSchedule
{
 public:
  //-- NEW FUNCTIONS ---------------------------------//
  int numconflicts() { return MidConflictCount; }
  int num_course_conflicts() { return CourseConflictCount; }

  //-- REIMPLEMENTATIONS OF OLD FUNCTIONS ----------//
  virtual void resize(int numSlots, int numCourses)
  {
    MidConflictCount = 0;
    CourseConflictCount = 0;
    OverFlowPenalty = 0;
    BaseSchedule::resize(numSlots,numCourses);
    StusInSlot.resize(numSlots);
    for(int i = 0; i < numSlots; i++)
      StusInSlot[i] = 0;
  }
  virtual int add(SlotIdx s, CourseIdx c, CourseList &L)
  {
    CourseConflictCount += course_conflicts(s,c,L);
    int k = BaseSchedule::add(s,c,L);
    MidConflictCount += k;
    int contrib = max(0,StusInSlot[s] - MaxPerSlot);
    StusInSlot[s] += L.Course[c].size();
    int delta = max(0,StusInSlot[s] - MaxPerSlot) - contrib;
    OverFlowPenalty += delta;
    return k;
  }
  virtual int remove(SlotIdx s, CourseIdx c, CourseList &L)
```

```
  {
    int k = BaseSchedule::remove(s,c,L);
    MidConflictCount -= k;
    CourseConflictCount -= course_conflicts(s,c,L);
    int contrib = max(0,StusInSlot[s] - MaxPerSlot);
    StusInSlot[s] -= L.Course[c].size();
    int delta = max(0,StusInSlot[s] - MaxPerSlot) - contrib;
    OverFlowPenalty += delta;
    return k;
  }
  virtual int overflowpenalty() { return OverFlowPenalty; }
  virtual int cost() { return OverFlowPenalty + MidConflictCount; }
  virtual int cost2add(SlotIdx s, CourseIdx c, CourseList &L)
  {
    int conflictincrease = conflicts(s,c,L), overflowincrease;
    if (StusInSlot[s] - MaxPerSlot >= 0) overflowincrease =
L.Course[c].size();
    else overflowincrease = max(0,StusInSlot[s] +
int(L.Course[c].size()) - MaxPerSlot);
    return conflictincrease + overflowincrease;
  }
  int stusInSlot(int k) { return StusInSlot[k]; }

 private:
  //-- DATA ----------------------------------------//
  int MidConflictCount;
  int CourseConflictCount;
  int OverFlowPenalty; // # of stus over MaxPerSlot summed over all
slots
  vector<int> StusInSlot;
};

// This class simply provides a simple comparison of conflict objects.
class CompSchedByMidConflicts
{
 public:
  bool operator()(Schedule *a, Schedule *b)
  {
    return a->numconflicts() < b->numconflicts();
  }
};

// This class simply provides a simple comparison of cost.
class CompSchedByCost
{
 public:
  bool operator()(Schedule *a, Schedule *b)
  {
    return a->cost() < b->cost();
  }
};
```

Appendix E: Genetic Algorithm Program

Purpose: This C++ program produces good final examination schedules.

Input: Two data files: First, a data file that contains all of the courses with final examinations and the list of midshipmen taking each exam. The second data file incorporates values for the threshold variable, initial population size, number of initial local mutations, number of children produced from each mating, number of generations, number of local mutations performed after each generation and number of "best" schedules passed onto future generations.

Description: The program first forms an initial population of schedules using a preprocessing (deterministic) step and then a random assignment of courses to exam slots. The program then follows a process of performing local mutations to the exam schedule and mating schedules. Local mutations are single moves of a course exam from one slot to another while the mating, or combining, of two schedules follows a special intermingling method.

Output: The best schedule after each generation and best overall schedule. The program also produces the schedule statistics for these schedules. These statistics include the number of midshipmen conflicts, the number of course conflicts and the running time of the program to form the schedule.

```
#include "Schedule.h"
#include <ctime>
#include <cmath>
#include <algorithm>

int MaxPerSlot;

/*********************************************************
 ** Function Prototypes
 *********************************************************/
// sets S to a random schedule of the courses in L into N exam slots
void rand_sched(Schedule &S, CourseList &L, int N);

// sets S to a schedule that's random, except that the Top Conflicts
// are all in different slots
void not_quite_rand_sched(Schedule &S, CourseList &L, int N,
vector<CourseIdx> &TopConfs);

// Tries to improve schedule "S" by reassigning course "course".
// "course" is assigned to slot resulting in fewest conflicts.  If
// there are multiple slots realizing this fewest number of conflicts,
// one of them is chosen at random.

void GreedyImproveMidConflicts(Schedule &S, CourseList &L, CourseIdx
course);
void GreedyImproveCost(Schedule &S, CourseList &L, CourseIdx course);

Schedule* mate(Schedule &A, Schedule &B, CourseList &L);


int main(int argc, char **argv)
{
  int TStart = time(0);
  srand(time(0));

  /*** READ IN COURSE DATA ***************************/
  if (argc < 4) {
    cerr << "Insufficient arguments: <configfile> <inputfile>
<outputfile>" << endl;
    return 1; }
  ifstream ConfIN(argv[1]);
  if (!ConfIN) {
    cerr << "File " << argv[1] << " could not be opened!" << endl;
    return 2; }
  ifstream IN(argv[2]);
  if (!IN) {
    cerr << "File " << argv[2] << " could not be opened!" << endl;
    return 3; }
  ofstream OUT(argv[3]);
  if (!OUT) {
    cerr << "File " << argv[3] << " could not be opened!" << endl;
    return 4; }

  /*** INITIALIZE ************************************/
  setMaxPerSlot(2400);
  int N = 15;    // number of exam time slots
```

```
  int ccut = 50; // we start with looking at # of conflicts involving
ccut or more students
  int initsize = 1000, initimprove = 5000;
  int restsize = 10, restimprove = 10000;
  int generations = 200; // number of generations
  int hold = 2;        // the top "hold" schedules get held over each
generation

  // Read parameter values config file
  string name;
  if (!(ConfIN >> name && name == "ccut" && ConfIN >> ccut)) {
    cerr << "Error reading ccut from config file" << endl; return 5; }
  if (!(ConfIN >> name && name == "initsize" && ConfIN >> initsize)) {
    cerr << "Error reading initsize from config file" << endl; return
5; }
  if (!(ConfIN >> name && name == "initimprove" && ConfIN >>
initimprove)) {
    cerr << "Error reading initimprove from config file" << endl;
return 5; }
  if (!(ConfIN >> name && name == "restsize" && ConfIN >> restsize)) {
    cerr << "Error reading restsize from config file" << endl; return
5; }
  if (!(ConfIN >> name && name == "restimprove" && ConfIN >>
restimprove)) {
    cerr << "Error reading restimprove from config file" << endl;
return 5; }
  if (!(ConfIN >> name && name == "generations" && ConfIN >>
generations)) {
    cerr << "Error reading generations from config file" << endl;
return 5; }
  if (!(ConfIN >> name && name == "hold" && ConfIN >> hold)) {
    cerr << "Error reading hold from config file" << endl; return 5; }

  CourseList L; // create course data structure and read in course info
  L.read(IN);

  /*** Find the "top N" courses ***/
  vector<CourseIdx> TopConfs;

  // Find degrees
  vector<CourseIdx> TCI;
  for(CourseIdx i = 0; i < L.courses(); i++)
  {
    int d = 0;
    for(CourseIdx j = 0; j < L.courses(); j++)
      if (i != j && L.conflicts(i,j) >= ccut)
        d++;
    TCI.push_back(d*5000 + i);
  }

  // Push course indices with N highest degrees on TopConfs
  sort(TCI.begin(),TCI.end());
  for(int k = 1; k < N-1; k++)
    TopConfs.push_back(TCI[TCI.size() - k] % 5000);

  /*** GENERATE AN INITIAL POPULATION OF RANDOM SCHEDULES **/
  vector<Schedule*> Pi;
```

```
  for(int i = 0; i < initsize; i++) {
    Pi.push_back(new Schedule());
    not_quite_rand_sched(*(Pi.back()),L,N,TopConfs);
  }

  /*** locally improve 1st population ******************/
  for(int j = 0; j < Pi.size(); j++) {
    for(int k = 0; k < initimprove; k++)
      GreedyImproveCost(*(Pi[j]),L,L.randcourse());
  }

  /*** Take the "restsize" best of the initial population ***/
  sort(Pi.begin(),Pi.end(),CompSchedByCost());
  vector<Schedule*> P(restsize), HallOfFame;
  for(int i = 0; i < restsize; i++)
    P[i] = Pi[i];
  for(int i = restsize; i < initsize; i++)
    delete Pi[i];
  Pi.clear();

  /*** DO THE GENETIC ALGORITHM THING!   ***/
  for(int i = 0; i < generations; i++)
  {
    sort(P.begin(),P.end(),CompSchedByCost());
    vector<Schedule*> Old = P;
    P.clear();

    cout << "Generation " << i << " best is " << Old[0]->cost() << ' ';
    cout << "born = " << Old[0]->born << " age = " << Old[0]->age << '
';
    int sum = 0;
    for(int j = 0; j < Old.size(); j++)
      sum += Old[j]->cost();
    cout << "Average is " << double(sum)/Old.size() << endl;

    for(int j = 0; j < hold; j++)
      P.push_back(new Schedule(*Old[j]));

    /*** mate x pairs ************************************/
    int M = Old.size();
    for(int j = 0; j < M - hold; j++)
    {
      int i1 = int(sqrt(double(rand()%(M*M)))), i2;
      do { i2 = int(sqrt(double(rand()%(M*M)))); } while (i1 == i2);
      P.push_back(mate(*Old[i1],*Old[i2],L));
      P[P.size()-1]->born = i + 1;
    }

    /*** locally improve new population ******************/
    for(int j = 0; j < P.size(); j++) {
      for(int k = 0; k < (j < hold ? restimprove : (i+1)*restimprove);
k++)
      //      for(int k = 0; k < restimprove; k++)
        GreedyImproveCost(*(P[j]),L,L.randcourse());
    }

    /*** Save best of the Old and kill the rest ***********/
```

```
    HallOfFame.push_back(new Schedule(*Old[0]));
    for(int j = 0; j < Old.size(); j++)
      delete Old[j];
  }


  HallOfFame = P; // HACK!  Look at ending population and ignore
HallOfFame


  /*** One last local mutate on the hall of fame ********/
  for(int j = 0; j < HallOfFame.size(); j++) {
    for(int k = 0; k < 2*restimprove; k++)
      GreedyImproveCost(*(HallOfFame[j]),L,L.randcourse());
  }


  /*** Print number of conflicts in sorted order ***********/
  cout << endl << endl << "Final Phase:" << endl;
  sort(HallOfFame.begin(),HallOfFame.end(),CompSchedByCost());
  for(int i = 0; i < HallOfFame.size(); i++)
    cout << "cost = " << HallOfFame[i]->cost()
         << " born = " << HallOfFame[i]->born
         << " age = " << HallOfFame[i]->age << endl;

  /*** Write best schedule to output file ******************/
  cout << endl << endl << "Best schedule had "
       << HallOfFame[0]->cost() << " cost and "
       << HallOfFame[0]->numconflicts() << " midshipman conflicts and "
       << HallOfFame[0]->num_course_conflicts() << " course conflicts"
       << endl;
  HallOfFame[0]->print(L,OUT);
  cout << "That took " << (time(0) - TStart) << " seconds" << endl;

  return 0;
}


/********** FUNCTIONS INVOLVING BaseSchedule **********/
void rand_sched(Schedule &S, CourseList &L, int N)
{
  S.resize(N,L.courses());
  for(int i = 0; i < L.courses(); i++)
    S.add(rand()%N,i,L);
}

void not_quite_rand_sched(Schedule &S, CourseList &L, int N,
vector<CourseIdx> &TopConfs)
{
  S.resize(N,L.courses());

  //Place list of N courses with top conflicts in different exam slots
  for(int j = 0; j < TopConfs.size(); j++)
    S.add(j,TopConfs[j],L);

  //Randomly fill in the rest
    for(int course_i = 0; course_i < L.courses(); course_i++)
```

```
    {
      // Search for course_i in the cells of TopConfs
      int flag = 1;
      for(int vect_i = 0; vect_i < TopConfs.size(); vect_i++)
        if(course_i == TopConfs[vect_i])
          flag = 0;
      if(flag == 1)
        S.add(rand()%N,course_i,L);
    }
}

void GreedyImproveMidConflicts(Schedule &S, CourseList &L, CourseIdx
course)
{
  S.age++;

  // Remove chosen course
  int oldslot = S.slot(course);
  int delta = S.remove(oldslot,course,L);

  // Collect all optimal choices for assigning "course"
  int fewestconfs = delta;
  vector<SlotIdx> choice(L.courses());
  int i = 0;
  for(SlotIdx newslot = 0; newslot < S.numslots(); newslot++)
  {
    int c = S.conflicts(newslot,course,L);
    if (c < fewestconfs)
    {
      fewestconfs = c;
      choice[0] = newslot;
      i = 1;
    }
    else if (c == fewestconfs)
      choice[i++] = newslot;
  }

  // Randomly choose one of the optimal assignments and make it!
  S.add( choice[rand() % i],course,L);
}

void GreedyImproveCost(Schedule &S, CourseList &L, CourseIdx course)
{
  S.age++;

  // Remove chosen course
  int oldslot = S.slot(course);
  S.remove(oldslot,course,L);
  int delta = S.cost2add(oldslot,course,L);

  // Collect all optimal choices for assigning "course"
  int fewestconfs = delta;
  vector<SlotIdx> choice(L.courses());
  int i = 0;
  for(SlotIdx newslot = 0; newslot < S.numslots(); newslot++)
  {
    int c = S.cost2add(newslot,course,L);
```

```
    if (c < fewestconfs)
    {
      fewestconfs = c;
      choice[0] = newslot;
      i = 1;
    }
    else if (c == fewestconfs)
      choice[i++] = newslot;
  }

  // Randomly choose one of the optimal assignments and make it!
  S.add( choice[rand() % i],course,L);
}

Schedule* mate(Schedule &A, Schedule &B, CourseList &L)
{
  // Create new schedule
  Schedule *p = new Schedule;
  Schedule &Baby = *p;
  Baby.resize(A.numslots(),L.courses());

  // Round one of copying!
  for(SlotIdx i = 0; i < A.numslots(); i++)
  {
    // 2a. Interleave exam slots from Schedule A and B
    if(i % 2 == 0)
    { //Copy slot from A
      for(set<CourseIdx>::iterator q = A[i].begin(); q != A[i].end();
++q)
        if (Baby.slot4Course[*q] < 0)
          Baby.add(i,*q,L);
    }
    else
    { //Copy slot from B
      for(set<CourseIdx>::iterator q = B[i].begin(); q != B[i].end();
++q)
        if (Baby.slot4Course[*q] < 0)
          Baby.add(i,*q,L);
    }
  }

  // Round two! Go through slots in opposite manner & fill in left-out
courses
  for(SlotIdx i = 0; i < A.numslots(); i++)
  {
    // 2a. Interleave exam slots from Schedule A and B
    if(i % 2 == 1)
    { //Copy slot from A
      for(set<CourseIdx>::iterator q = A[i].begin(); q != A[i].end();
++q)
        if (Baby.slot4Course[*q] < 0)
          Baby.add(i,*q,L);
    }
    else
    { //Copy slot from B
      for(set<CourseIdx>::iterator q = B[i].begin(); q != B[i].end();
++q)
```

```
        if (Baby.slot4Course[*q] < 0)
          Baby.add(i,*q,L);
    }
  }

  return p;
}
```

Appendix F: Verification of Schedule Program

Purpose: This C++ program establishes the validity of any final examination schedule produced and incorporates the proximity step.

Input: A final examination schedule and a data file with all of the courses requiring a long period to grade.

Description: This program first verifies that all of the final exams are present in the final examination schedule produced. It then incorporates a proximity step that orders the exam slots by the number of "long grading" courses that each exam slot contains.

Output: A schedule in proximity order. In addition, it yields values for the number of "long grading" courses in each exam slot and the number of students taking an exam in each exam slot.

```
/*****************************************************
 ** verifyshed <coursefile> <schedfile>
 **
 ** Program reads data from coursefile and reads a
 ** proposed schedule from schedfile and reports the
 ** conflict numbers, as well as verifying that all
 ** courses are scheduled.  It also implements the
 ** proximity step.
 *****************************************************/
#include "Schedule.h"
#include <ctime>
#include <cmath>
#include <algorithm>

int MaxPerSlot;

Schedule* reorder(string fname, Schedule &A, CourseList &L);

int main(int argc, char **argv)
{

  /*** INITIALIZE ************************************/
  int N = 15; // number of exam time slots

  /*** READ IN COURSE DATA ***************************/
  if (argc < 3) {
    cerr << "Insufficient arguments:  <coursefile> <schedfile>
[<orderfile> <reorderedfile>]" << endl;
    return 1; }
  ifstream C_IN(argv[1]);
  if (!C_IN) {
    cerr << "File " << argv[1] << " could not be opened!" << endl;
    return 2; }
  ifstream S_IN(argv[2]);
  if (!S_IN) {
    cerr << "File " << argv[2] << " could not be opened!" << endl;
    return 3; }

  string orderfile;
  if (argc >= 4)
    orderfile = string(argv[3]);

  ofstream ReOUT;
  bool reout;
  if (reout = (argc == 5))
  {
    ReOUT.open(argv[4]);
    if (!ReOUT) {
      cerr << "File " << argv[4] << " could not be opened!" << endl;
      return 5; }
  }

  CourseList L;
  L.read(C_IN);

  /*** READ IN SCHEDULE ******************************/
  Schedule S;
```

```
  S.resize(N,L.courses());
  SlotIdx slot = -1;
  string s;
  while(S_IN >> s)
  {
    if (s == "Slot")
    {
      S_IN >> s;
      slot++;
    }
    else
    {
      map<string,CourseIdx>::iterator p = L.NameIndex.find(s);
      CourseIdx course = p->second;
      S.add(slot,course,L);
    }
  }

  /*** REPORT ****************************************/
  bool cflag = true;
  for(CourseIdx course = 0; course < L.courses(); course++)
  {
    if (S.slot4Course[course] < 0 || S.slot4Course[course] >
S.numslots())
    {
      cflag = false;
      cout << "Course with index " << course << " is not scheduled!" <<
endl;
    }
  }
  if (cflag)
    cout << "All courses are scheduled!" << endl;
  cout << "Schedule has " << S.numconflicts() << " Mid conflicts!" <<
endl;
  cout << "Schedule has " << S.num_course_conflicts() << " course
conflicts!" << endl;

  cout << "Students scheduled in slot: " << endl;
  for(int i = 0; i < N; i++)
    cout << S.stusInSlot(i) << ' ';
  cout << endl;


  // Order info
  Schedule *p = 0;
  if (orderfile != "")
    p = reorder(orderfile,S,L);
  if (reout)
  {
    p->print(L,ReOUT);
    cout << "Reordered schedule in file " << argv[4] << endl;
  }

  return 0;
}
```

```
Schedule* reorder(string fname, Schedule &A, CourseList &L)
{
  // Create set of all course indices that are long grading
  ifstream IN(fname.c_str());
  set<CourseIdx> EGC;
  string s;
  while(IN >> s)
    EGC.insert(L.NameIndex[s]);

  // Fill V with (numearlygradecourses,slotindex)
  vector< pair<int,int> > V(A.numslots());
  for(int i = 0; i < A.numslots(); i++)
  {
    vector<int> I;

set_intersection(EGC.begin(),EGC.end(),A[i].begin(),A[i].end(),back_ins
erter(I));
    V[i] = pair<int,int>(I.size(),i);
  }

  // Print before sort
  cout << "Early grade courses in slots originally: " << endl;
  for(int i = 0; i < V.size(); i++)
    cout << V[i].first << ' ';
  cout << endl;

  sort(V.begin(),V.end());

  // Print after  sort
  cout << "Early grade courses in slots reordered: " << endl;
  for(int i = V.size() - 1; i >= 0; i--)
    cout << V[i].first << ' ';
  cout << endl;

  // Create schedule by reordering A in decreasing number of early
grading courses
  Schedule *p = new Schedule();
  p->resize(A.numslots(),L.courses());
  for(int i = V.size()-1, j = 0; i >= 0; i--, j++)
  {
    int si = V[i].second;
    for(set<CourseIdx>::iterator itr = A[si].begin(); itr !=
A[si].end(); ++itr)
      p->add(j,*itr,L);

  }
  return p;
}
```

Data Sets

Experiment 1

In this part of Experiment 1, threshold was varied and all other variable values were fixed.

Varying ccut	1trial	midconf	courseconf	Time(s)	
initsize	1000	1	14	13	471
initimprove	5000	2	17	16	471
restsize	10	3	15	15	471
restimprove	10000	4	15	15	472
hold	2	5	17	16	471
generations	200				

Varying ccut	5trial	midconf	courseconf	time	
initsize	1000	1	16	15	470
initimprove	5000	2	14	14	470
restsize	10	3	18	15	471
restimprove	10000	4	19	17	470
hold	2	5	16	15	470
generations	200				

Varying ccut	10trial	midconf	courseconf	time	
initsize	1000	1	16	15	472
initimprove	5000	2	11	11	471
restsize	10	3	16	10	472
restimprove	10000	4	18	16	470
hold	2	5	16	15	472
generations	200				

Varying ccut	25trial	midconf	courseconf	time	
initsize	1000	1	14	11	470
initimprove	5000	2	11	11	471
restsize	10	3	14	14	471
restimprove	10000	4	14	14	472
hold	2	5	18	14	471
generations	200				

Varying ccut	50trial	midconf	courseconf	time	
initsize	1000	1	13	10	471
initimprove	5000	2	14	10	471
restsize	10	3	17	16	470
generations	10000	4	17	13	472
hold	2	5	17	12	471

In this part of Experiment 1, the initial population size was varied and all other variable values were fixed.

ccut	50	trial	midconf	courseconf	time	
varying initsize	100	1	19	19	385	
initimprove	5000	2	14	14	385	
restsize	10	3	20	16	386	
restimprove	10000	4	15	12	386	
hold	2	5	16	16	387	
generations	200					

ccut	50	trial	midconf	courseconf	time	
varying initsize	250	1	14	11	402	
initimprove	5000	2	14	14	401	
restsize	10	3	13	11	402	
restimprove	10000	4	16	16	401	
hold	2	5	19	16	402	
generations	200					

ccut	50	trial	midconf	courseconf	time	
varying initsize	500	1	18	18	424	
initimprove	5000	2	15	14	424	
restsize	10	3	16	13	425	
restimprove	10000	4	17	16	424	
hold	2	5	14	14	424	
generations	200					

ccut	50	trial	midconf	courseconf	time	
varying initsize	1000	1	16	11	471	
initimprove	5000	2	12	11	471	
restsize	10	3	13	12	472	
restimprove	10000	4	15	12	472	
hold	2	5	14	14	471	
generations	200					

ccut	50	trial	midconf	courseconf	time	
varying initsize	2000	1	12	11	564	
initimprove	5000	2	14	10	563	
restsize	10	3	18	17	564	
restimprove	10000	4	18	18	564	
hold	2	5	17	13	565	
generations	200					

In this part of Experiment 1, the amount of initial mutations was varied and all other variable values were fixed.

ccut	50	trial	midconf	courseconf	time
initsize	1000	1	18	16	387
varying initimprove	500	2	18	15	388
restsize	10	3	14	11	387
restimprove	10000	4	15	12	388
hold	2	5	17	16	387
generations	200				

ccut	50	trial	midconf	courseconf	time
initsize	1000	1	18	8	397
varying initimprove	1000	2	12	9	396
restsize	10	3	18	16	397
restimprove	10000	4	20	19	396
hold	2	5	13	12	397
generations	200				

ccut	50	trial	midconf	courseconf	time
initsize	1000	1	13	13	425
varying initimprove	2500	2	20	16	425
restsize	10	3	11	11	425
restimprove	10000	4	18	17	426
hold	2	5	16	10	425
generations	200				

ccut	50	trial	midconf	courseconf	time
initsize	1000	1	18	16	470
varying initimprove	5000	2	15	14	470
restsize	10	3	13	13	471
restimprove	10000	4	14	11	470
hold	2	5	15	14	472
generations	200				

ccut	50	trial	midconf	courseconf	time
initsize	1000	1	18	16	564
varying initimprove	10000	2	17	16	563
restsize	10	3	16	15	564
restimprove	10000	4	15	15	563
hold	2	5	14	14	563
generations	200				

In this part of Experiment 1, the amount of children allowed was varied and all other variable values were fixed.

ccut	50	trial	midconf	courseconf	time
initsize	1000	1	15	15	282
initimprove	5000	2	15	14	283
varying restsize	5	3	12	12	282
restimprove	10000	4	15	12	282
hold	2	5	14	14	283
generations	200				

ccut	50	trial	midconf	courseconf	time
initsize	1000	1	17	16	472
initimprove	5000	2	16	13	471
varying restsize	10	3	16	16	471
restimprove	10000	4	18	14	472
hold	2	5	18	16	471
generations	200				

ccut	50	trial	midconf	courseconf	time
initsize	1000	1	14	13	660
initimprove	5000	2	17	15	659
varying restsize	15	3	14	10	661
restimprove	10000	4	14	11	661
hold	2	5	17	14	660
generations	200				

ccut	50	trial	midconf	courseconf	time
initsize	1000	1	15	11	849
initimprove	5000	2	16	14	851
varying restsize	20	3	15	15	850
restimprove	10000	4	17	14	849
hold	2	5	17	13	848
generations	200				

ccut	50	trial	midconf	courseconf	time
initsize	1000	1	20	18	1037
initimprove	5000	2	18	17	1037
varying restsize	25	3	22	19	1037
restimprove	10000	4	15	14	1037
hold	2	5	17	16	1038
generations	200				

In this part of Experiment 1, the amount of generations was varied and all other variable values were fixed.

ccut	50	trial	midconf	courseconf	time
initsize	1000	1	15	14	143
initimprove	5000	2	16	16	144
restsize	10	3	14	13	143
varying generation	25	4	16	16	144
hold	2	5	18	17	143
restimprove	10000				

ccut	50	trial	midconf	courseconf	time
initsize	1000	1	18	17	190
initimprove	5000	2	14	14	190
restsize	10	3	15	15	190
varying generation	50	4	13	13	190
hold	2	5	15	12	190
restimprove	10000				

ccut	50	trial	midconf	courseconf	time
initsize	1000	1	15	14	284
initimprove	5000	2	16	14	283
restsize	10	3	18	14	284
varying generation	100	4	16	15	284
hold	2	5	15	15	284
restimprove	10000				

ccut	50	trial	midconf	courseconf	time
initsize	1000	1	12	9	471
initimprove	5000	2	19	18	472
restsize	10	3	16	15	471
varying generation	200	4	17	12	471
hold	2	5	17	14	470
restimprove	10000				

ccut	50	trial	midconf	courseconf	time
initsize	1000	1	13	13	659
initimprove	5000	2	16	13	659
restsize	10	3	17	16	658
varying generation	300	4	14	14	659
hold	2	5	19	16	659
restimprove	10000				

In this part of Experiment 1, the amount of "best" schedules kept was varied & all other variables were fixed.

ccut	50	trial	midconf	courseconf	time
initsize	1000	1	21	21	471
initimprove	5000	2	19	14	471
restsize	10	3	17	16	472
generations	200	4	15	14	470
varying hold	1	5	14	10	471
restimprove	10000				

ccut	50	trial	midconf	courseconf	time
initsize	1000	1	19	18	472
initimprove	5000	2	22	16	471
restsize	10	3	16	12	471
generations	200	4	13	13	471
varying hold	2	5	15	13	471
restimprove	10000				

ccut	50	trial	midconf	courseconf	time
initsize	1000	1	17	16	472
initimprove	5000	2	17	13	472
restsize	10	3	16	14	471
generations	200	4	**10**	**10**	**471**
varying hold	3	5	16	16	470
restimprove	10000				

ccut	50	trial	midconf	courseconf	time
initsize	1000	1	15	15	471
initimprove	5000	2	13	12	471
restsize	10	3	11	11	472
generations	200	4	14	14	471
varying hold	4	5	14	13	471
restimprove	10000				

ccut	50	trial	midconf	courseconf	time
initsize	1000	1	15	10	470
initimprove	5000	2	11	11	471
restsize	10	3	13	12	470
generations	200	4	11	11	470
varying hold	5	5	12	11	471
restimprove	10000				

In this part of Experiment 1, the amount of mutations following each mating was varied

and all other variable values were fixed.

ccut	50	trial	midconf	courseconf	time
initsize	1000	1	15	15	132
initimprove	5000	2	19	18	131
restsize	10	3	15	14	131
generations	200	4	21	21	132
hold	2	5	25	22	131
varying restimprove	1000				

ccut	50	trial	midconf	courseconf	time
initsize	1000	1	15	15	188
initimprove	5000	2	16	15	188
restsize	10	3	22	17	188
generations	200	4	13	12	188
hold	2	5	13	12	188
varying restimprove	2500				

ccut	50	trial	midconf	courseconf	time
initsize	1000	1	13	13	282
initimprove	5000	2	13	12	283
restsize	10	3	17	16	283
generations	200	4	19	16	282
hold	2	5	18	18	283
varying restimprove	5000				

ccut	50	trial	midconf	courseconf	time
initsize	1000	1	18	18	377
initimprove	5000	2	14	11	377
restsize	10	3	19	15	377
generations	200	4	14	12	377
hold	2	5	20	20	376
varying restimprove	7500				

ccut	50	trial	midconf	courseconf	time
initsize	1000	1	12	12	471
initimprove	5000	2	17	16	472
restsize	10	3	15	13	471
generations	200	4	15	11	471
hold	2	5	18	17	473
varying restimprove	10000				

Average Results - Experiment 1

	MidConf	CourseConf	Time	
CCUT				
1	15.6	15	471.2	**Baseline**
5	16.6	15.2	470.2	
10	15.4	13.4	471.4	CCUT 50
25	**14.2**	**12.8**	**471**	initsize 1000
50	15.6	12.2	471	initimprove 5000
				restsize 10
initsize				generations 200
100	16.8	15.4	385.8	hold 2
250	15.2	13.6	401.6	restimprove 10000
500	16	15	424.2	
1000	**14**	**12**	**471.4**	
2000	15.8	13.8	564	
initimprove				
500	16.4	14	387.4	
1000	**16.2**	**12.8**	**396.6**	
2500	15.6	13.4	425.2	
5000	15	13.6	470.6	
10000	16	15.2	563.4	
restsize				
5	**14.2**	**13.4**	**282.4**	
10	*17*	*15*	*471.4*	
15	15.2	12.6	660.2	
20	16	13.4	849.4	
25	18.4	16.8	1037.2	
generations				
25	15.8	15.2	143.4	
50	**15**	**14.8**	**190**	
100	16	14.4	283.8	
200	16.2	13.6	471	
300	15.8	14.4	658.8	
hold				
1	17.2	15	471	
2	17	14.4	471.2	
3	15.2	13.8	471.2	
4	13.4	13	471.2	
5	**12.4**	**11**	**470.4**	

restimprove

1000	19	18	131.4
2500	**15.8**	**14.2**	**188**
5000	16	15	282.6
7500	17	15.2	376.8
10000	15.4	13.8	471.6

Experiment 2

Experiment 2 - Spring Semester

Local Mutations v. Mating Process

Varying the Number of "Best" Schedules Kept with Lower Baseline Values for Other Variables

	MidConf	CourseConf	Time			
Varying Hold - T1					CCUT	50
	0	23	20	957	initsize	1000
	1	15	15	861	initimprove	5000
	2	13	13	773	restsize	10
	3	12	12	695	generations	100
	4	13	13	612	restimprove	1000
	5	**10**	**9**	**528**		
	6	12	12	466		
	7	15	15	377		
	8	13	10	288		
	9	14	14	198		
	10	22	19	108		
Varying Hold - T2						
	0	32	19	949		
	1	17	16	863		
	2	17	15	778		
	3	14	13	694		
	4	17	16	610		
	5	13	13	527		
	6	11	11	460		
	7	14	10	358		
	8	14	14	272		
	9	17	15	188		
	10	19	18	103		
Varying Hold - T3						
	0	24	15	949		
	1	16	16	863		
	2	17	16	778		
	3	15	13	694		
	4	16	10	611		
	5	13	13	525		
	6	14	14	441		
	7	14	10	357		
	8	12	12	272		
	9	15	15	188		
	10	21	20	103		

Varying Hold - T4

0	38	30	944
1	14	14	862
2	14	14	780
3	16	15	694
4	13	12	608
5	12	12	526
6	13	13	441
7	15	14	357
8	15	12	273
9	12	12	187
10	19	18	104

Varying Hold - T5

0	27	19	947
1	15	15	860
2	17	14	781
3	16	16	694
4	17	15	610
5	12	12	526
6	16	15	451
7	15	14	357
8	15	14	272
9	13	13	188
10	22	22	103

Averages

0	28.8	20.6	949.2
1	15.4	15.2	861.8
2	15.6	14.4	778
3	14.6	13.8	694.2
4	15.2	13.2	610.2
5	**12**	**11.8**	**526.4**
6	**13.2**	**13**	**451.8**
7	14.6	12.6	361.2
8	13.8	12.4	275.4
9	14.2	13.8	189.8
10	20.6	19.4	104.2

AllStar1 *Best Values from Experiment 1*

1	22	16	43	
2	17	17	44	
3	22	21	43	
4	25	21	43	
5	22	18	43	

CCUT	25
initsize	1000
initimprove	1000
restsize	10
generations	50
hold	5
restimprove	2500

AllStar2 *Keeping the Number of "Best" Schedules = 5*

1	14	13	472	
2	11	11	471	
3	12	12	472	
4	12	12	471	

CCUT	50
initsize	1000
initimprove	5000
restsize	10
generations	200
hold	5
restimprove	10000

Experiment 2 - Fall Exam Data

Local Mutations v. Mating Process

Varying the Number of "Best" Schedules Kept with Lower Baseline Values for Other Variables

	MidConflicts	**CourseConflicts**	**Time**			
Varying Hold - T1					CCUT	50
0	33	28	934	initsize	1000
1	26	20	852	initimprove	5000
2	23	22	769	restsize	10
3	23	19	686	generations	100
4	21	18	603	restimprove	1000
5	18	17	520		
6	16	14	437		
7	19	14	354		
8	18	14	270		
9	16	15	185		
10	27	22	102		
Varying Hold - T2						
0	45	21	935			
1	21	18	852			
2	20	16	768			
3	21	20	686			
4	18	15	603			
5	18	13	521			
6	14	11	438			
7	20	16	352			
8	19	16	269			
9	20	13	185			
10	23	23	101			
Varying Hold - T3						
0	36	21	935			
1	21	20	852			
2	18	14	769			
3	20	19	684			
4	23	18	602			
5	15	13	520			
6	18	13	438			
7	19	16	353			
8	18	18	269			
9	23	19	185			
10	26	20	102			

Varying Hold - T4

0	29	24	934
1	23	14	851
2	23	18	767
3	22	19	687
4	19	13	602
5	16	12	522
6	16	14	438
7	15	14	357
8	16	14	270
9	13	13	188
10	22	22	103

Varying Hold - T5

0	32	26	937
1	23	18	852
2	21	19	767
3	19	15	687
4	20	15	603
5	16	13	520
6	18	13	436
7	19	18	353
8	17	17	268
9	20	19	185
10	23	19	101

Averages

0	35	24	935
1	22.8	18	851.8
2	21	17.8	768
3	21	18.4	686
4	20.2	15.8	602.6
5	**16.6**	**13.6**	**520.6**
6	**16.4**	**13**	**437.4**
7	18.4	15.6	353.8
8	17.6	15.8	269.2
9	18.4	15.8	185.6
10	24.2	21.2	101.8