Staff Scheduling by a Genetic Algorithm with a Two-Dimensional Chromosome Structure

John S. Dean

Box 24, 8700 N.W. River Park Dr., Parkville, MO 64152

Park University, Information and Computer Science Department

jdean4@kc.rr.com

913 422-8822

Abstract— Staff scheduling is the assignment of employees to time slots such that certain constraints are satisfied. In this project, I address the specific problem of scheduling nurses on daily shifts for the duration of a four-week schedule. The solution attempts to assign shifts with certain required constraints (specified number of nurses on a particular shift, must work every third weekend, etc.) satisfied and certain suggested constraints (avoid overtime hours, avoid three consecutive work days, etc.) satisfied to an acceptable degree.

In this study, I implement two genetic algorithm staff-scheduling solutions for scheduling nurses at a hospital. One solution uses a traditional bit-string chromosome structure to represent each schedule. The other solution uses a two-dimensional array chromosome structure to represent each schedule. Research literature indicates that prior genetic algorithm studies have used traditional bit-string structures for chromosomes and not two-dimensional arrays. Experimental results show that my two-dimensional array staff-scheduling implementation performs better than my bit-string staff-scheduling implementation.

Keywords— evolutionary computation, genetic algorithms, medical services, heuristics, staff scheduling

INTRODUCTION

Need for the study

The problem of automated staff scheduling has been extensively addressed over the past several decades (Ernst, Jiang, Krishnamoorthy, Owens, & Sier, 2004). In particular, since hospital nurse care is so important, and since nurse shift scheduling is so complicated, there has been a considerable research effort in automated staff scheduling for hospital nurses (Burke, Causmaecker, Berghe, & Landeghem, 2004).

Staff scheduling is complicated by the fact that the constraints are usually numerous and varied, and sometimes competing. Even without such complications, with a relatively small number of straightforward constraints, bare-bones staff scheduling is known to be an NP-complete problem (Fukunaga et al., 2002). Since NP-complete problems cannot be solved in a reasonable amount of time with brute-force methods, they are good candidates for heuristic solutions. Applying heuristic solutions to the staff-scheduling problem has been an active area of research. No solutions have emerged with clearly optimal results, and there is no consensus as to the best implementation strategy. Genetic algorithms (which are heuristic in nature) have yielded fairly good results when applied to scheduling problems such as the job-shop scheduling problem (Alves, Guimaraes, & Fernandes, 2006). Studies have shown reasonably good results when genetic algorithms are applied to the staff-scheduling problem. In this project, I address the need for improvement when applying a genetic algorithm to the staff-scheduling problem.

Although the trend of recent publications is towards very complex problems where finding optimal solutions is time-prohibitive, there is still some benefit in improving the convergence speed for problems where optimal or near-optimal solutions are possible. This study addresses that

need by attempting to improve the speed in which an optimal or near-optimal solution can be found for an actual staff-scheduling problem for nurses in a real hospital. The techniques used in this study should be useful for more complex problems, but that has not been verified.

Barriers and issues

The staff-scheduling problem has a fairly broad definition (the assignment of employees to time slots such that certain constraints are satisfied), and, as such, different studies focus on different types of shifts and different sets of constraints. Many studies have shifts with varying start times and varying lengths (Bard, Binici, & deSilva, 2003). Most staff-scheduling studies assign an employee to one shift per day, but some assign an employee to multiple short shifts within one day (Fukunaga et al., 2002). The constraints for staff-scheduling problems are even more varied than the shifts. Some schedulers (but not all) are required to have certain types of employees work simultaneously with other types of employees. For example, Aickelin, Burke, and Li (2007) modeled nurses of different skill levels where higher skill-level nurses can substitute for lower skill-level nurses Some schedulers (but not all) try to accommodate employee requests, such as maximizing or minimizing consecutive working days (Dowsland, 2000).

With such diversity of shift types and constraints, setting up a new study can be somewhat challenging. A researcher might choose to pick shift and constraint parameters from among the vast pool of shift and constraint parameters used in prior studies, but such choices would be arbitrary, at least to a certain degree. To avoid such arbitrariness, in this study I implement shifts and constraints used in an actual hospital – Shawnee Mission Medical Center in Overland Park, Kansas.

A particular challenge of the staff-scheduling problem is that one constraint might compete with another constraint to the point where a solution is impossible. For example, suppose one constraint attempts to balance the number of extended weekends (i.e., consecutive days Friday through Monday off) between all employees. And suppose another constraint attempts to accommodate individual employee preferences to avoid more than two consecutive working days. Such constraints compete because the first constraint encourages more consecutive working days and the second constraint encourages less consecutive working days.

Research questions investigated

The primary question to be answered is whether a genetic algorithm can be created for the staff-scheduling problem that is (1) able to generate successful schedules, and (2) reasonably generic. In past studies, genetic algorithm solutions to real-world, non-trivial staff-scheduling problems have provided good, but not optimal, schedules. Some studies have achieved optimality by adding repair operators to their genetic algorithm solutions. Such repair operators lead to solutions that are less generic – they cannot be translated easily to other staff-scheduling domains. In this project, I attempt to implement a successful solution without repair operators. By omitting the repair operators, the genetic algorithm part of the solution is kept reasonably separate from the domain-specific part of the solution, and that helps to make the solution more generic.

Another question to be answered is what type of chromosome structure is best for each schedule in a population of schedules. Research literature indicates that in prior studies that use genetic algorithms, schedules have typically used the traditional bit-string structure for chromosomes, and they have not used two-dimensional structures for chromosomes. Using a two-dimensional chromosome structure appears to be a natural fit for the staff-scheduling problem since employee schedules are often printed as two dimensional tables with the table's rows representing individual employees and the table's columns representing individual days. In this project, I use a two-dimensional structure for chromosomes, and I compare its effectiveness to that of a bit-string structure.

Review of the Literature

Prior research specific to the topic

Over the years, researchers have used various techniques to implement solutions for the staff-scheduling problem. Deterministic techniques have been used successfully to find complete

solutions. For example, integer programming (IP) has been used to schedule postal workers (Bard et al., 2003), and constraint logic programming (CLP) has been used to schedule nurses (Winstanley, 2002). But, unfortunately, such deterministic solutions require a great deal of computing resources (in particular, a large search space), and their ability to handle many constraints is somewhat limited.

Several researchers have used tabu search to solve the staff-scheduling problem (Burke, De Causmaecker, Petrovic, & Berghe, 2006; Burke, De Causmaecker, & Vanden Berghe, 1999; Burke, Kendall, & Soubeiga, 2003; Dowsland, 2000). Tabu search is a heuristic approach that attempts to escape local optimality by avoiding search space points that have already been visited. Although tabu search solutions have been successful, a criticism of the tabu search approach is that it requires a solution that is very specific to a particular domain. It is not general purpose enough (Aickelin & Dowsland, 2004).

A number of researchers have used genetic algorithms to solve the staff-scheduling problem. Gomez, de la Fuente, Puente, and Parreno (2006) successfully implement a genetic algorithm solution, but in so doing, they point out how difficult it is to satisfy the myriad of constraints in a typical staff-scheduling problem. To overcome some of this difficulty, they use "hard constraints," which must be satisfied, and "soft constraints," which should be satisfied if possible. Even with this improvement, Gomez et al. lament how staff-scheduling genetic algorithm solutions unfortunately and necessarily become tied to a particular domain.

Aickelin and Dowsland (2000) successfully implement a staff-scheduling genetic algorithm solution, but they admit that their solution suffers from being too dependent on problem-specific knowledge. In a more recent effort, Aickelin and Dowsland (2004) present an alternative solution – an "indirect genetic algorithm" where the problem-specific knowledge is moved out of the genetic algorithm code. They do that by splitting the solution into two parts. The first part, called the GA decoder, uses the problem-specific knowledge to create input for the second part. The second part is a relatively generic genetic algorithm that uses the GA decoder's output to find an optimal staff scheduling solution. Although this approach does lead to a more generic genetic algorithm implementation, it generates optimal schedules in just more than half of all the problems addressed.

Several studies (Cai & Li, 2000; Gröbner & Wilke, 2001; Tanomaru, 1995) have addressed the difficult task of finding a complete solution by adding repair operators to their genetic algorithms. The repair operators use problem-specific knowledge to modify individuals so that they satisfy the required problem constraints. The repair operators are applied to each individual after selection, crossover, and mutation, and before calculating the fitness function. In applying repair operators, Gröbner and Wilke admit the risk of favoring local maxima too much. To avoid such bias, his algorithm applies the repair operators to a lesser degree initially and to a greater degree later on as convergence takes place. Genetic algorithms with repair operators have been shown to work well, but repair operators have been criticized because they cause the genetic algorithm solution to rely even more heavily on problem-specific knowledge (Aickelin & Dowsland, 2004).

A recent study by Aickelin et al. (2007) uses an estimation of distribution algorithm (EDA) with a local search, and they call their approach a memetic EDA. An EDA is similar to a genetic algorithm in that it produces generations of chromosomes that tend to converge to a better solution, but EDAs choose subsequent chromosomes by sampling from a probabilistic model for the different chromosome possibilities. To improve chromosome content during each genetic algorithm generation, Aickelin et al.'s solution embeds an ant-miner local search within the EDA algorithm. With the ant-miner algorithm, greater amounts of pheromone are assigned to successful trails and future schedule searches are consequently encouraged to repeat prior successes. When empirically compared to a genetic algorithm solution without a probabilistic model and without a local search (Aickelin & White, 2004), Aickelin et al.'s 2007 solution performed about the same when local search was not used and performed much better when local search was used.

Significance of this study

In, Gröbner and Wilke's study, their initial staff scheduling genetic algorithm "yielded relatively good rosters, but still an unacceptably high number of hard constraints [were] violated" (2001, p. 468). In particular, Gröbner and Wilke note that their solution wasted a significant amount of time by assigning an employee to more than one shift on the same day. As mentioned above, the researhers' "fix" was to use repair operators, and repair operators have been criticized since they lead to genetic algorithm solutions that are too problem-specific.

As mentioned above, Aickelin and Dowsland (2004) implement a more generic genetic algorithm solution by using a two-part approach where the problem-specific knowledge is moved out of the genetic algorithm part and into the GA decoder part. Unfortunately, their program generates optimal schedules in only just over half of their test cases.

The goal of this project is to implement an improved genetic algorithm for solving the staff-scheduling problem. In the project's solution, I refrain from using repair operators (which lead to genetic algorithm solutions that are too problem-specific). Not using repair operators helps to make my solution more generic, but it is not completely generic. As with most other staff-scheduling studies, my solution embeds domain-specific information in the constraints.

In this project's solution, I introduce the use of two-dimensional arrays with a genetic algorithm. Two-dimensional arrays have been used with other techniques to solve the staff-scheduling problem, but the literature suggests a lack of genetic algorithm studies that use two-dimensional arrays. My study indicates that using a two-dimensional array with a genetic algorithm can improve the ability to find optimal and near-optimal solutions for a real-world staff-scheduling problem.

Problem Domain

Schedules, units, employee types, and shift types

This project schedules nurses on daily shifts for the duration of a four-week schedule. The study's shifts and constraints are those used at the Shawnee Mission Medical Center. The hospital generates four-week schedules for each of its units. The hospital has one unit for each type of patient. For example, there is an oncology unit and an emergency room unit.

In testing the scheduler, I use the unit with the largest staffing need – the oncology unit. I use the unit with the largest staffing need because getting the scheduler to work for that population group should indicate that the scheduler will work for the other, smaller population groups.

The hospital schedules different types of employees for each unit. Each unit schedules resident nurses (RNs), clinical associates (CAs), information associates (IAs), and charge nurses (CNs). In testing the scheduler, I use the type of employee with the largest staffing need – resident nurses – because getting the scheduler to work for that population group should indicate that the scheduler will work for the other, smaller population groups. As mentioned earlier, some studies allow higher skill-level nurses to substitute for lower skill-level nurses. That constraint can add significantly to the complexity of a problem. My study does not allow for higher-level nurse substitution because the Shawnee Mission Medical Center does not allow for that.

The hospital schedules just two types of shifts – a 13-hour day shift from 6:45 am to 7:45 pm and a 12½ hour night shift from 6:45 pm to 7:15 am. When an employee agrees to work at Shawnee Mission Medical Center, he/she signs a contract that specifies day shift or night shift. Unless the employee renegotiates his/her contract, he/she works that type of shift (and not the other type of shift) for the duration of his/her employment at the hospital.

Constraints

The hospital creates schedules that are subject to certain constraints. The constraints/rules are divided into two types – hard rules and soft rules. The hard rules are considered to be more important, and, as such, the scheduler tries harder to satisfy them.

Hard rules:

1. Create 4-week schedules that start on Sunday.
2. For each day of the entire schedule, assign a constant number of working employees.
3. For each employee, assign at least 3 working days per week (= 39 hours).
4. For each employee, maximum number of consecutive working days = 6.
5. Each employee works every third weekend.

Soft rules:

1. Try to avoid more than 3 working days per employee per week because that leads to overtime pay.
2. Try hard to avoid more than 4 working days per employee per week because that leads to

additional overtime pay.

3. Try to balance the number of Monday and Friday off days that coincide with off weekends.
4. If an individual wants to avoid working 3 days in a row, try to accommodate.
5. If an individual wants to maximize the number of grouped working days, then try to maximize the number of grouped working days.

Number of employees

The hospital determines the number of employees needed for a particular unit by first estimating the number of patients for the upcoming four-week schedule. The estimated number of patients is then plugged into a look-up table where the number of employees needed is found. Figure 1 is the look-up table for the number of employees needed in the oncology unit. It handles 1-26 possible patients, where 26 is the number of beds in the oncology unit and therefore the maximum number of patients. The table's top row indicates that if 26 patients per day are expected, then each day 6 daytime RNs , 4 daytime CAs, and 1 daytime IA should be scheduled for work. The table does not show the number of CNs because there is always one CN per unit. The table's fractional values are always rounded up, so in the nighttime CA column for the 22-patient row, the 2.4 value indicates that 3 CAs are needed.

In testing the scheduler, I use the maximum staffing need of 6 resident nurses for 26 patients because getting the scheduler to work for that maximum-size population group should indicate that the scheduler will work for the other, smaller population groups.

Oncology Staffing Guidelines 2004

DAY	RN	CA	IA	NIGHT	RN	CA	IA
CENSUS				CENSUS			
26	6	4	1	26	4	3	0.3
25	6	4	1	25	4	3	0.3
24	6	3	1	24	4	3	0.3
23	6	3	1	23	4	3	0.3
22	6	3	1	22	4	2.4	0.3
21	5	3	1	21	4	2.4	0.3
20	5	3	1	20	3	2	0.3
19	5	3	1	19	3	2	0.3
18	4	3	1	18	3	2	0.3
17	4	3	1	17	3	2	0.3
16	4	2	1	16	3	2	0.3
15	4	2	1	15	3	1.4	0.3
14	3	2	1	14	3	1.4	0.3
13	3	2	1	13	2	1.7	0.3
12	3	2	1	12	2	1.4	0
11	3	1.4	0.7	11	2	1.4	0
10	3	1	0.7	10	2	1	0
9	2	1.4	0.3	9	2	1	0
8	2	1	0.3	8	2	0.8	0
7	2	1	0.3	7	2	0	0
6	2	0.5	0.0	6	2	0	0
5	2	0	0.0	5	2	0	0
4	2	0	0.0	4	2	0	0
3	2	0	0.0	3	2	0	0
2	2	0	0.0	2	2	0	0
1	2	0	0.0	1	2	0	0

Figure 1 Number of employees needed in the oncology unit.

To determine the maximum number of employees in a particular unit's employee rotation, this formula can be used:

$$\begin{aligned} &\textit{employees working per day} = \\ &\quad \textit{maximum employees in rotation} \times \textit{probability of employee working on one day} \end{aligned} \tag{1}$$

to obtain this formula:

$$\textit{maximum employees in rotation} = \frac{\textit{employees working per day}}{\textit{probability of employee working on one day}} \tag{2}$$

If *employees working per day* equals 6 (the maximum staffing need for oncology RNs is 6), and *probability of employee working on one day* equals 0.4286 (the goal is for an employee to work 3 days in a 7-day week and 3/7 = 0.4286), then:

$$maximum\ employees\ in\ rotation = \frac{6}{0.4286} \quad (3)$$

Thus, while developing the scheduler program, I used a baseline test set of 14 for the size of the employee rotation pool and 6 for the number of employees working per day. An employee rotation pool of size 14 is comparable to employee rotation pools used in other staff-scheduling studies (Aickelin & Dowsland, 2000; Fukunaga et al., 2002; Gomez et al., 2006; Gröbner & Wilke, 2001; Tanomaru, 1995). Upon completion of the scheduler program, I conducted tests with smaller employee rotation pools and also larger employee rotation pools.

Methodology

Genetic algorithm overview

In this study, I apply a genetic algorithm to the staff-scheduling problem. Genetic algorithms have been studied since the 1960's. For an overview and historical background, see (Fogel, 2005). This study's genetic algorithm solution implements each schedule as a chromosome/individual in a population of schedules. Each schedule is evaluated with a fitness function (also called an objective function). Schedules with greater fitness function values are allowed to "mate" with other schedules via crossover. Mutation provides for diversity in the population. The crossover and mutation operations generate new populations of schedules. New generations are created until a schedule is formed that is deemed acceptable. Generally speaking, a schedule is deemed acceptable if its fitness function value is high enough.

Fitness functions

The genetic algorithm's fitness functions are derived from the hospital's constraints. As dictated by the hospital and as exemplified by Gomez et al. (2006), I organize the constraints into two categories – hard rules and soft rules. In evaluating a schedule, the genetic algorithm uses separate fitness functions, one for each of the two categories. I use two techniques to ensure that the hard-rules fitness function is given more prominence than the soft-rules fitness function: (1) I include hard-rules satisfaction in the program's termination condition. Specifically, the program continues to generate new schedule populations until all of the hard rules are satisfied. At termination, soft rules might be fully satisfied, but it is not a requirement. (2) I weight hard-rules fitness-function constraints more than soft-rules fitness-function constraints. Adding up all the fitness-function constraint values shows that the maximum hard-rules fitness-function value is roughly 10 times greater than the maximum soft-rules fitness-function value.

For each schedule, the genetic algorithm calculates "goodness" values for both the hard-rules fitness function and the soft-rules fitness function by adding up weighted values for each of the hard and soft constraints. For details, see below:

Hard-Rules Fitness Function	
Correct number of employees per day	For each day of the schedule: If correct number of employees per day, add 20 to goodness. If number of employees per day is within 1 of correct value, add 8 to goodness. If number of employees per day is within 2 of correct value, add 4 to goodness.
Minimum 3 days per week	For each week for each employee: If working 3 days per week, add 10 to goodness. If working 2 days per week, add 4 to goodness. If working 1 day per week, add 2 to goodness.
Maximum 6 consecutive days	For each employee: If maximum consecutive working days ≤ 6, add 20 to goodness. If maximum consecutive working days = 7, add 8 to goodness. If maximum consecutive working days = 8, add 4 to goodness.
Work every 3rd weekend	For each employee and for each weekend the employee is supposed to work: Add 20 to goodness for working both Saturday and Sunday or for working the last Saturday. Add 8 to goodness for working one of Saturday or Sunday.

Soft-Rules Fitness Function	
Minimize	For each week for each employee: If working ≤ 4 days per week, add 1 to

overtime	goodness. If working 5 days per week, add .25 to goodness.
Extended weekend days off	Calculates the standard deviation between all employees for each employee's number of off Mondays and off Fridays that coincide with off weekends. If standard deviation < .5, add 4 × numOfEmps to goodness. If .5 ≤ standard deviation < 1, add 1 × numOfEmps to goodness.
Requested maximum 2 consecutive days	For each employee that requests a maximum of 2 consecutive days: If there is an employee with a max of 2 consecutive days, add 4 to goodness. If there is an employee with 3 consecutive days, but not 4, add 1 to goodness.
Requested maximize grouped days	For each employee that requests maximization of grouped days: If there is an employee with ≤ 2 solitary days, add 4 to goodness. If there is an employee with 3 solitary days, add 2 to goodness.

Constraints that depend on prior schedules

Certain hospital constraints require that values be saved from prior schedules. For example, in applying the maximum-6-consecutive-working-days rule, the avoid-3-consecutive-working-days rule, and the maximize-consecutive-working-days rule to an employee's schedule, the scheduler must know how many days the employee worked at the end of the prior schedule. In a development environment, there is no actual prior schedule, so reasonable values should be generated. This study's program creates a `priorConsecutiveDays` array, where each element stores the number of consecutive days (0, 1, 2, or 3) worked by an employee prior to the start of the current schedule. The elements are assigned values such that there are a probabilistically reasonable number of 0's, a probabilistically reasonable number of 1's, etc.

In applying the work-every-3-weekends rule to an employee's schedule, the scheduler must know how many weekends ago the employee was assigned to work at the end of the prior schedule. Prior to the generation of schedules, the program creates a `numOfWeekendsAgo` array, where each element stores the number of weekends ago the employee worked prior to the start of the current schedule (0, 1, or 2, where 2 indicates 2 weekends ago). The elements are assigned values cyclically – 0, 1, 2, 0, 1, 2, 0, and so on.

Two-dimensional chromosome structure

Although most genetic algorithms use bit strings to represent chromosome data, studies (from domains other than the staff scheduling domain) show that alternative representations often work better, and that, in general, chromosomes should resemble the data that they represent (Mitchell, 1996). With that in mind, I use a two-dimensional array for the genetic algorithm's chromosome structure. The array's columns represent individual days and the array's rows represent individual employees. Each array element implements a gene, and each gene holds a Boolean value where true indicates that an employee is scheduled to work on a particular day and false indicates that an employee is not working on a particular day.

When performing a crossover operation between two schedules, the genetic algorithm randomly selects a crossover line that signals the starting position for swapped information. See Figure 2. Such a crossover approach is not as radical as it first appears. It is equivalent to stringing a chromosome array's rows together into one row and then having multiple crossover points – one crossover point for each employee in the employee rotation. But it is different from traditional bit-string operations in that a relatively large number of crossover points are used (one for each employee) and the positions of the crossover points are evenly spaced (for a given crossover operation, the crossover points are positioned between the same pair of days).

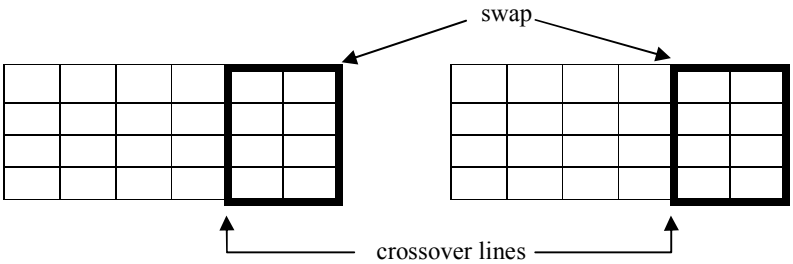

Figure 2 Crossover between two 2-dimensional array chromosomes

Using a two-dimensional array with a vertical crossover line has the benefit of automatically enforcing one of the hard-rule constraints – fixed number of employees working on a particular day. The difficulty of satisfying that constraint was the primary reason that Gröbner and Wilke (2001) introduced repair operators, so having it automatically satisfied is particularly beneficial. The reason for the automatic constraint satisfaction is that (1) the scheduler creates an initial population with the constraint already satisfied and (2) the vertical crossover line means that each day's group of employees remains intact.

When performing a mutation operation on a schedule, the genetic algorithm randomly selects a column; that is, it selects one day from the 28-day schedule. It then replaces that column with a randomly selected column from a pool of columns. The column pool consists of all the combinations of the specified number of employees working on a particular day. As with the two-dimensional array crossover scheme, this two-dimensional array mutation scheme has the benefit of automatically enforcing the fixed-number-of-employees-working-on-a-particular-day constraint.

Creating the initial population

One possible drawback with the two-dimensional array approach with multiple crossover points along a vertical line is that it will cause a particular day's employees to be grouped. For example, if a schedule initially has nurses 1, 3, 8, and 12 working on a particular day, then that same group of nurses will persist as long as its enclosing schedule(s) survives. If this problem is not addressed, such sustained grouping might lead to less than the entire search space being available.

To ensure that the entire search space is available, all combinations must be provided initially of employees working on a particular day. As mentioned above, while developing the scheduler program, I used a baseline test set of 14 for the size of the employee rotation pool and 6 for the number of employees working per day. With 14 employees in the employee pool and 6 employees working each day, the number of combinations of working employees is 3003:

$$14 \; choose \; 6 = \frac{14!}{6! \times (14-6)!} = 3003 \tag{4}$$

Thus, for the baseline test set of 14 for the size of the employee rotation pool and 6 for the number of employees working per day, 3003 schedules are needed in the population to ensure that the entire search space is available.

Bit-string chromosome structure

For comparison purposes, I implement and test an alternative staff-scheduling solution that uses a traditional bit-string chromosome structure. It still uses a two-dimensional array for storing a schedule's values, and the initial population is created the same as when a two-dimensional chromosome structure is used. But crossover and mutation operators work differently than when a two-dimensional chromosome structure is used. The crossover and mutation operators look at the chromosome as if all the employees' schedules were strung together on one long bit string. The bit-string-based solution performs crossover at randomly selected points in the bit string. It performs mutation at randomly selected days for two randomly selected employees.

Unbiased tournament selection

There are several different components to decide on when implementing a genetic algorithm. One such component is the manner in which parents are chosen, where parents are the chromosomes that generate the next generation via crossover. For choosing parents, I use an

unbiased tournament selection scheme (Sokolov & Whitley, 2005). The unbiased tournament selection scheme has two primary benefits:

- Speed. In particular, it is faster than ranking schemes because sorting is not required.
- Good at preserving diversity, which is important for the staff-scheduling state space where local maxima are far apart. In particular, it is much better than the roulette scheme in terms of preserving diversity.

The traditional tournament selection process relies on randomly sampling groups of individuals and selecting the best individual (the one with the best fitness function) from each group. A problem with that scheme is that due to the randomness, individuals are not sampled uniformly, regardless of their fitness function value. Thus, diversity suffers.

Unbiased tournament selection attempts to remove the sampling omissions by creating two-individual groups where one individual comes from one permutation of all of the individuals and the other individual comes from another permutation of all of the individuals. That way, all individuals are in both groups.

This same two-permutation scheme is used when choosing crossover partners. Using all the parent chromosomes, two-individual groups are formed where one individual comes from one permutation of all of the parent chromosomes and the other individual comes from another permutation of all of the parent chromosomes.

Balancing diversity and convergence

With genetic algorithms in general and the staff-scheduling problem in particular, preserving population diversity is essential. But there needs to be a balance between achieving diversity and achieving convergence. In addition to using unbiased tournament selection to foster diversity, this project's program uses a high crossover probability (100 percent) and a high mutation probability (33 percent) to foster diversity. I determined those probabilities by running a series of tests with different values for crossover and mutation probabilities. To foster convergence, the program uses an elitism strategy. With elitism, the schedule with the best fitness-function value is saved and, after crossover and mutation, it overlays the schedule with the worst fitness-function value (Maulik & Bandyopadhyay, 2000).

Fine tuning the crossover and mutation rates

I implement both the two-dimensional array chromosome approach and the bit-string chromosome approach within the same scheduler program. The two approaches differ only in regard to their crossover and mutation operators. Specifically, the approaches differ in terms of how the operators impact their parent schedules and how often the operators are called. The above "chromosome structure" subsections explain how the operators impact their parent schedules. I determined how often the operators are called by running a series of tests with different values for crossover and mutation rates. I settled on crossover and mutation rates that yielded the best results in terms of optimality of the resulting schedule and faster convergence.

For the bit-string chromosome approach, the best results occurred when (1) the number of crossover points in each of two parent chromosomes was equal to the number of employees in the employee pool, and (2) the number of mutated bits (for schedules that were to be mutated) was also equal to the number of employees in the employee pool. Consequently, I used those crossover and mutation rates for all subsequent comparison tests with the two-dimensional array chromosome approach.

Graduated mutation rates for the two-dimensional array chromosome approach

For the two-dimensional array chromosome approach, I found that the best results occurred when there was just one crossover line, as shown in Figure 2. For mutation, I found that the best results occurred when adjustable mutation rates were used. I found that it was best to use a default of just one mutated column for each schedule that is to be mutated, and to gradually increase the number of mutated columns if a local maximum was reached that was too far from optimality. By increasing the number of mutated columns, the program was better able to break free from the local maximum and explore alternative schedules in the search space.

The stabilityCount variable helps with checking for a local maximum by keeping track of the number of consecutive generations where the same combined (hard rules plus soft rules) fitness-function value is achieved in the population's best schedule. Each time the stability count increases to a multiple of 10, the number of mutated columns increases by 1. The maximum number of

mutated columns is the number of days in the schedule. Recall that only a certain percentage of schedules are mutated (33%), so even if the number of mutated columns reaches the number of days in the schedule, it does not mean that all prior learning is destroyed.

If a new best schedule is found, then the stabilityCount variable gets reset to zero, and the number of mutated columns gets reset to 1. Resetting the number of mutated columns allows the genetic algorithm to return to its default behavior where the current local maximum can be more easily improved.

I tested using a graduated mutation rate for the bit-string chromosome approach, but it was not helpful, so I abandoned it. It was not helpful because the bit-string chromosome approach already exhibits a great deal of random behavior, and, as such, it is highly resistant to getting stuck in local maxima.

Time complexity analysis

As mentioned earlier, the staff-scheduling problem is known to be NP-complete. As such, except for problems with a small number of staff and a small number of days, it is impractical to generate every possible schedule and test each schedule for constraint satisfaction.

If a bit-string chromosome structure were used, each schedule's working-versus-off-day value would be independent of the other working-versus-off-day values, so there would be $2^{(emps \times days)}$ possible schedules, where *emps* is the total number of employees and *days* is the number of days in each schedule. If an exhaustive-search algorithm were used, then a worst-case scenario would check all of those schedules, and that would represent a time complexity of $O(2^{(emps \times days)})$. Using such an approach would be impractical. For our standard 14-employees, 28-days test set, there would be $2^{(14 \times 28)} = 1.01 \times 10^{118}$ schedules to check.

If a two-dimensional chromosome structure with a vertical crossover point were used, each schedule's column would represent a schedule for a particular day. The number of different possible schedules for each day would be *emps* choose *crewSize*, where *emps* is the total number of employees and *crewSize* is the number of employees that work together on one day. For example, with 14 employees and 6 employees working on a particular day, the number of different possible schedules for each day is 3003 because 14 choose 6 equals 3003. Taking into account all of the days in a schedule, there are (*emps* choose *crewSize*)^{days} schedules, where *days* is the number of days in each schedule. If an exhaustive-search algorithm were used, then a worst-case scenario would check all of those schedules, and that would represent a time complexity of O((*emps* choose *crewSize*)^{days}).

Although using a two-dimensional chromosome structure reduces the time complexity significantly, using an exhaustive schedule generation strategy is still intractable. To combat this problem, I use a heuristic solution, in the form of a genetic algorithm. The study's genetic algorithm program limits the search for an acceptable schedule by favoring the generation of schedules that improve in quality as the program proceeds. This reduced search space leads to a significant improvement in search times for acceptable schedules. But since the genetic algorithm is a heuristic device, and subject to randomness, worst-case analysis would indicate that finding an acceptable-quality schedule might take an infinite amount of time. To address this worst-case scenario, the scheduler program contains a termination condition (in addition to the all-hard-rules-satisfied termination condition mentioned in the above "Termination conditions" section) that checks for a user-specified maximum number of schedule attempts. If the maximum number of schedule attempts is reached without finding an acceptable schedule, then the program terminates. At that point, the user would presumably run the scheduler program again.

Space complexity analysis

Using a two-dimensional chromosome structure, my scheduler program's space complexity is the number of chromosomes in the population times the number of elements in each chromosome. The number of chromosomes in the population is *emps* choose *crewSize*. The number of elements in each chromosome is *emps* times *days*. Thus, the space complexity is O((*emps* choose *crewSize*) $\times$ *emps* $\times$ *days*).

As mentioned earlier, I use a standard test set of 14 for the size of the employee rotation pool and 6 for the number of employees working per day, which requires a population size of 3003 schedules. Each schedule is a two-dimensional array – 14 rows for 14 employees and 28 columns for 28 days. With 3003 two-dimensional arrays, the storage requirements for the entire population are fairly high. I can successfully run the scheduler program for that case and also for the next two

larger cases – 16 in the employee pool and 7 working each day, 18 in the employee pool and 8 working each day. But for the next larger case of 21 in the employee pool and 9 working each day, the number of schedules in the population is 293,930, and the program generates an out-of-memory error during the creation of the initial population. The simplest solution would be to increase the maximum heap size used by Java, but the problem would still exist for larger cases. A more robust, but slower, solution would be to store populations as files and load parts of the population into memory on an as needed basis. I chose not to explore that type of implementation for this study because the hospital's needs do not call for it. But exploring such an implementation could be a useful project for future work.

Results

Example output

See Figure 3's example output. It shows the result of running the scheduler program with the standard test input of 14 in the employee pool and 6 employees working each day. Note that all of the hard rules are satisfied with 1300 hard-rule fitness-function points out of a maximum of 1300. Note that soft rules are satisfied with 150 soft-rule fitness-function points out of a maximum of 152. The header numbers indicate the day of the week for each of the four weeks in the schedule. The numbers under the headers indicate the days that the employees are working. For example, employee 1 is scheduled to work on Sunday, Tuesday, and Wednesday during the first week and

```
Enter number of employees: 14
Enter number of employees working per day: 6
Percent of schedules in population that are mutated (e.g., 15 for 15%): 33
Enter number of employees who want a maximum of 2 consecutive days: 5
Enter number of employees who want to maximize grouped days: 5
Enter maximum number of generations: 5000
Enter number of schedules to be printed: 1

Number of generations = 721, stability count = 0.

Best possible goodness values:
Hard rules = 1860, Soft rules = 152, Combined = 2012

The following schedule's goodness values:
Hard rules = 1860, Soft rules = 152, Combined = 2012
All hard rules satisfied.

Emp = employee number, pcd = prior consecutive days, nwa = number of weekends
ago (worked).

emp pcd nwa    1 2 3 4 5 6 7   1 2 3 4 5 6 7   1 2 3 4 5 6 7   1 2 3 4 5 6 7
--- --- ---    - - - - - - -   - - - - - - -   - - - - - - -   - - - - - - -
 1   1   0   | 1   2 3       |       1   2 3 |       1 2   3 | 1 2       3
 2   0   1   |     1 2 3     | 1     2     3 | 1   2 3       | 1 2   3
 3   0   2   |   1     2   3 | 1   2 3       |   1 2       3 |   1   2     3
 4   1   0   | 1 2         3 |     1   2 3   |   1 2       3 | 1   2 3
 5   0   1   |     1     2 3 |     1   2   3 | 1     2   3   |     1   2 3
 6   0   2   |   1   2     3 | 1       2 3   |   1     2 3   |     1   2   3
 7   1   0   | 1       2 3   |   1 2     3   |     1 2     3 | 1     2 3
 8   0   1   |   1   2   3   |   1   2     3 | 1   2     3   |       1 2   3
 9   0   2   | 1       2   3 | 1 2   3       |   1   2 3     |   1       2 3
10   2   0   | 1   2     3   |     1 2   3   | 1         2 3 | 1 2 3
11   0   1   |       1 2 3   |   1     2   3 | 1 2     3     |   1     2   3
12   0   2   |   1 2       3 | 1       2 3   |       1 2 3   |       1   2 3
13   3   0   | 1   2     3   |   1 2   3     |         1 2 3 | 1   2     3
14   0   1   |   1   2 3     | 1 2         3 | 1 2 3         |     1   2 3
```

Figure 3 Example output from the scheduler program

Monday, Tuesday, and Saturday during the second week.

The maximize-grouped-working-days soft constraint might need some explanation in regard to

Figure 3. As shown by the input (italicized), five employees have a preference for maximizing the number of grouped work days. Employees 1, 2, 6, and 8 have schedules that satisfy that constraint because they have no more than two working days that are isolated. All their other working days are adjacent to at least one other working day. Employee 11 has a schedule with three working days that are isolated. Having just three such working days is good, but less than ideal, and that accounts for the two point difference between the best possible soft-constraint fitness-function value and the actual soft-constraint fitness-function value.

Summarized results

See Figure 4's summarized results of running the scheduler program with a two-dimensional array chromosome approach. Of the 24 tests conducted on employee pool sizes of 2 through 14, all except two generate a schedule where all of the hard rules are satisfied. For the 8 tests conducted on employee pool sizes of 16 and 18 (which are larger than needed by the hospital), all except one generate a schedule where all of the hard rules are satisfied. Figure 4 shows that the soft rules are completely satisfied 62.5% of the time when the program is run with employee pool sizes of 2 through 18. When the soft rules are not completely satisfied, they are still satisfied to a great extent. Specifically, when the program is run with employee pool sizes of 2 through 18 and hard

Staff scheduling with two-dimensional array chromosom structure

Each percent and average value is based on four program executions.
Standard inputs used by all program executions: mutation = 33%, maximum generations = 5000

Input				Output			
Number of employees in employee pool	Number of employees working on same day	Number of employees preferring maximum 2 consecutive days	Number of employees preferring to maximize grouped days	Percent of program executions where all hard rules satisfied	Percent of program executions where all soft rules satisfied	Average % of fitness function points achieved versus best possible fitness function points	*Average number of generations to achieve a successful solution
2	1	1	0	100	50	98.6	138
4	2	1	1	100	100	100	224
6	3	2	2	100	0	99.6	147
9	4	3	3	75	25	99.5	253
11	5	4	4	100	100	100	168
14	6	5	5	75	50	99.9	1606
**16	7	6	6	100	75	99.9	780
**18	8	6	6	75	100	99.8	254
totals:				90.6	62.5	99.7	446.3

* Average number of generations takes into account only program executions that achieve a successful solution, where a successful solution is a schedule with all hard rules satisfied and stability count = 500.
** Employee pools of size 16 and 18 generated for test purposes. They are not needed by the hospital.

Figure 4 Staff scheduling results with two-dimensional chromosome structure and uniform constraint satisfaction

rules are completely satisfied and soft rules are not completely satisfied, soft-rule fitness-function points are 93% of the best possible soft-rule fitness-function points.

See Figure 5's summarized results of running the scheduler program with a bit-string chromosome approach. With all of the test sets viewed collectively, the average percent of fitness function points achieved versus the best possible fitness function points was 99.1%, which was slightly less than the value of 99.7% found with the two-dimensional chromosome structure test sets. The greater disparity between the two approaches was found for the speed of convergence. With all of the test sets viewed collectively, the average number of generations to achieve a successful solution was 1058.4 for the bit-string chromosome approach and 446.3 for the two-dimensional array chromosome approach. The two-dimensional array chromosome approach tended to converge to slightly better solutions than the solutions generated by the bit-string chromosome solution, and the convergence took less than half the time.

Staff scheduling with bit-string chromosom structure

Each percent and average value is based on four program executions.
Standard inputs used by all program executions: mutation = 33%, maximum generations = 5000

Input				Output			
Number of employees in employee pool	Number of employees working on same day	Number of employees preferring maximum 2 consecutive days	Number of employees preferring to maximize grouped days	Percent of program executions where all hard rules satisfied	Percent of program executions where all soft rules satisfied	Average % of fitness function points achieved versus best possible fitness function points	*Average number of generations to achieve a successful solution
2	1	1	0	0	0	97.6	n/a
4	2	1	1	75	50	98.2	512
6	3	2	2	100	75	99.6	614
9	4	3	3	75	50	98.5	1010
11	5	4	4	100	75	99.2	812
14	6	5	5	50	100	99.8	669
**16	7	6	6	100	100	100	1626
**18	8	6	6	100	100	99.8	2166
totals:				75.0	68.8	99.1	1058.4

* Average number of generations takes into account only program executions that achieve a successful solution, where a successful solution is a schedule with all hard rules satisfied and stability count = 500.
** Employee pools of size 16 and 18 generated for test purposes. They are not needed by the hospital.

Figure 5 Staff scheduling results with two-dimensional chromosome structure and uniform constraint satisfaction

Conclusion

In this study, I successfully implemented a genetic algorithm solution for a nurse staff-scheduling problem at an actual hospital. My solution uses a two-dimensional array chromosome structure for each nurse schedule. Research literature indicates that my solution is novel in that prior genetic algorithm studies have used traditional bit-string structures for chromosomes and not two-dimensional arrays.

Staff scheduling studies, when based on a real-world scheduling problem, tend to have different domains. Specifically, different scheduling organizations have different types of shifts and different types of constraints. With such diversity, it is difficult to compare results across different studies with certainty. However, in this study, it is straightforward to compare my newly proposed genetic algorithm approach (which uses a two-dimensional chromosome array structure) to a traditional genetic algorithm approach (which uses a bit-string chromosome structure) because I implement both approaches and apply them to the same domain.

This study's results show an improvement in staff scheduling when a two-dimensional chromosome structure is used for the schedules, as opposed to a bit-string chromosome structure. Both approaches generate a comparable high percentage of optimal schedules, but the two-dimensional array chromosome approach converges much more rapidly to an optimal solution. The faster convergence is due to the crossover and mutation operators used by the two-dimensional array chromosome approach. Those operators preserve the correct-number-of-employees-per-day constraint by using crossover points and mutation points that are in the same vertical line within a given two-dimensional array chromosome structure.

An interesting features of the two-dimensional array chromosome approach is its graduated mutation rate. I found that it was best to use a default of one mutated column and to gradually increase the number of mutated columns if a local maximum was reached that was too far from optimality. Such an approach helps the program break free from local maxima and explore alternative schedules in the search space.

It is not recommended that two-dimensional chromosomes be applied to all genetic algorithm problems because they can lead to populations with limited diversity. But for problems where solutions are in the form of tables, two-dimensional chromosomes might do well. In particular, they should do well when the order of column data relates to a constraint.

In the future, I would like to use the two-dimensional chromosome structure with genetic algorithms and apply it to other domains – other staff-scheduling domains, and also to other NP-complete problems. In particular, I would like to use a two-dimensional chromosome structure for

staff-scheduling problems with more complex domains. Two common features that add to a staff-scheduling problem's complexity are: (1) handling multiple types of employees where more highly skilled employees are allowed to substitute for less highly skilled employees, and (2) handling multiple types of shifts where employees are allowed to work on more than one type of shift.

References

Aickelin, U., Burke, E., & Li, J. (2007). An estimation of distribution algorithm with intelligent local search for rule-based nurse rostering. *Journal of the Operational Research Society, 58*(12), 1574-1585.

Aickelin, U., & Dowsland, K. (2004). An indirect genetic algorithm for a nurse-scheduling problem. *Computers & Operations Research, 31*(5), 761-778.

Aickelin, U., & Dowsland, K. A. (2000). Exploiting problem structure in a genetic algorithm approach to a nurse rostering problem. *Journal of Scheduling, 3*(3), 139-153.

Aickelin, U., & White, P. (2004). Building better nurse scheduling algorithms. *Annals of Operations Research, 128*(1-4), 159-177.

Alves, F. S. R., Guimaraes, K. F., & Fernandes, M. A. (2006). Modeling workflow systems with genetic planner and scheduler. *18th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'06)*, *0*, 381-388.

Bard, J. F., Binici, C., & deSilva, A. H. (2003). Staff scheduling at the united states postal service. *Computers and Operations Research, 30*(5), 745-771.

Burke, E., Causmaecker, P. D., Berghe, G. V., & Landeghem, H. V. (2004). The state of the art of nurse rostering. *Journal of Scheduling, 7*(6), 441-499.

Burke, E., De Causmaecker, E. P., Petrovic, S., & Berghe, G. V. (2006). Metaheuristics for handling time interval coverage constraints in nurse scheduling. *Applied Artificial Intelligence, 20*(3).

Burke, E., De Causmaecker, P., & Vanden Berghe, G. (1999, November 1998). A hybrid tabu search algorithm for the nurse rostering problem. *Simulated Evolution and Learning: Second Asia-Pacific Conference on Simulated Evolution and Learning*, Canberra, Australia, 187-194.

Burke, E., Kendall, G., & Soubeiga, E. (2003). A tabu-search hyperheuristic for timetabling and rostering. *Journal of Heuristics, 9*(6), 451-470.

Cai, X., & Li, K. N. (2000). A genetic algorithm for scheduling staff of mixed skills under multi-criteria. *European Journal of Operational Research, 125*(2), 359-369.

Dowsland, K. A., Thompson, J.M. (2000). Solving a nurse scheduling problem with knapsacks, networks and tabu search. *The Journal of the Operational Research Society, 51*(7).

Ernst, A. T., Jiang, H., Krishnamoorthy, M., Owens, B., & Sier, D. (2004). An annotated bibliography of personnel scheduling and rostering. *Annals of Operations Research, 127*(1-4), 21-144.

Fogel, D. B. (2005). *Evolutionary computation: Toward a new philosophy of machine intelligence* (3 ed.): Wiley-IEEE Press.

Fukunaga, A., Hamilton, E., Fama, J., Andre, D., Matan, O., & Nourbakhsh, I. (2002). Staff scheduling for inbound call and customer contact centers. *AI Magazine, 23*(4), 30-40.

Gomez, A., de la Fuente, D., Puente, J., & Parreno, J. (2006). A case-study about shift work management at a hospital emergency department with genetic algorithms. *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, Seattle, Washington, USA.

Gröbner, M., & Wilke, P. (2001). Optimizing employee schedules by a hybrid genetic algorithm. In E. J. W. Boers (Ed.), *Applications of evolutionary computing : Evoworkshops 2001* (pp. 463–472): Springer-Verlag.

Maulik, U., & Bandyopadhyay, S. (2000). Genetic algorithm-based clustering technique. *Pattern Recognition, 33*.

Mitchell, M. (1996). *An introduction to genetic algorithms*: MIT Press.

Sokolov, A., & Whitley, D. (2005). Unbiased tournament selection. *Proceedings of the 2005 conference on Genetic and evolutionary computation*, Washington DC, USA.

Tanomaru, J. (1995). Staff scheduling by a genetic algorithm with heuristic operators. *IEEE International Conference on Evolutionary Computation*, Perth, WA, Australia, *1*, 456-462.

Winstanley, G. (2002). A hybrid ai approach to staff scheduling. *Proceedings of ES-02, volume XIX of Research and Development in Intelligent Systems*, 367–380.