Highly Constrained Sports Scheduling With Genetic Algorithms

Andrew Brian Goldberg

April 14, 2003

Submitted to the
Department of Mathematics and Computer Science
of Amherst College
in partial fulfillment of the requirements
for the degree of
Bachelor of Arts with Distinction

Faculty Advisor: Professor Cathy C. McGeoch

Abstract

The quality of a sports schedule has a direct impact on a sports league's fairness of competition; the satisfaction of players, coaches, and fans; and revenue brought in through ticket sales. Because many leagues, including the New England Small College Athletic Conference (NESCAC), currently create their schedules though a lengthy and inefficient manual process, often producing sub-optimal results, an automated sports scheduler is of great practical importance. Since the search space of possible schedules grows exponentially as more teams are added to a league, devising a schedule that meets all of the league administrators' criteria regarding when and where games may be played is an NP-Hard optimization problem. Therefore, heuristic methods must be employed to find highly desirable schedules in a short period of time. This thesis presents a genetic algorithm that finds such schedules for the NESCAC men's soccer season by mimicking the processes of natural selection and evolution. As the only algorithmic components tied to this specific problem are the decoding mechanism and evaluation function, this algorithm could be adapted to suit other sports leagues and non-sports scheduling tasks. This work successfully demonstrates that a genetic algorithm is an appropriate heuristic method for solving real-world optimization tasks for which direct methods often fail.

Acknowledgments

I would first like to thank Professor Cathy McGeoch for spending countless Friday afternoons listening to me ramble on about sports scheduling. I am extremely grateful for her expert guidance, constructive criticism, and voracious editing.

This work would not have been possible without the collaboration of Andrea Savage, the NESCAC Administrative Director. Many lengthy phone calls and e-mails provided me with invaluable data on which to base my work.

I would also like to thank the rest of the Computer Science faculty, whose advising and instruction over the past four years have produced a firm foundation on which to build my independent research.

A thank you is definitely in store for my fellow Computer Science thesis writers who provided hours of thought-provoking discussion and companionship while living in the Sun Lab.

I must also thank my roommates and friends whose steady support and personal involvement in NESCAC sports have provided inspiration throughout this project.

Most of all, though, I would like to thank my family for offering constant encouragement, always taking a vested interest in my education, and pushing me to strive for excellence throughout my life.

Contents

Chapter 1

Introduction

Scheduling is one of many heavily explored and researched practical problems in combinatorial optimization. Scheduling consists of developing a timetable of events within a specified time-frame and given a finite set of resources such that no events conflict with one another or violate any *constraints*. Scheduling problems arise in many fields, such as scheduling jobs in a factory, scheduling processes in a computer system, scheduling school final examinations, or, the focus of this thesis, scheduling sporting competitions between teams in a league.

In most cases, scheduling problems are NP-hard: there is no known polynomial time algorithm capable of finding a solution that combines the given resources in the optimal way without violating constraints. The search space consisting of all combinations of events, times, and locations can be far too great to search exhaustively in a reasonable amount of time. In many real scheduling applications, this work is done by hand, with human schedulers utilizing heuristics acquired over years of experience to find what would be considered a near optimal or favorable solution. This manual process takes a substantial amount of time, and the final result is usually far from the optimal schedule. For example, the New England Small College Athletic Conference (NESCAC) currently develops its sports schedules through a lengthy process of rearranging index cards [16].

To eliminate the inefficient practice of manual scheduling, researchers have tried to automate the task of scheduling using computers, since a computer should be able to apply a set of heuristics and produce schedules in significantly less time than a human. As in other situations involving heuristics, the task of encoding this human knowledge into a computerized form can be rather

daunting. Nevertheless, various techniques have been applied in attempts to automate difficult scheduling tasks. Many come from mathematics and operations research, including linear programming, simulated annealing, tabu search, and genetic algorithms. Each of these techniques has been shown to work fairly successfully in some situations.

This thesis explores the use of genetic algorithms in sports scheduling by building one to schedule the NESCAC men's soccer season. The goal is to find an algorithm which runs relatively quickly and finds a set of solutions that we can conclude are highly desirable. For a real-world optimization task, it is nearly impossible to calculate the absolute optimum solution, but we can use league administrators' input to determine what is "near optimal."

1.1 What Is Sports Scheduling?

For four teams labeled A, B, C, and D, a possible timetable might look something like this:

Round 1	Round 2	Round 3
A vs. B, C vs. D	A vs. C, B vs. D	A vs. D, B vs. C

Clearly, for four teams, there are only six possible games between the teams. Three rounds are necessary to ensure that no team is expected to be in more than one place at the same time.

1.1.1 Feasible, Infeasible, and Desirable Schedules

The simple schedule devised above is a *feasible* schedule: it can be played out in the real world. The only restrictions here are that each team plays each other once and that no games overlap. Therefore, this is a viable, practicable schedule. In general, a feasible solution to a scheduling problem is one that does not violate any of the requirements of the problem. Given S, the search space of all schedules, let the set of feasible schedules be denoted $F \subseteq S$.

While creating feasible schedules is fundamental in the design of any sports scheduling algorithm, most real sports scheduling situations involve more complex constraints imposed by the sporting organization's administrators or based on factors such as arena availability. As is the case for general scheduling problems, constraints in sports scheduling can be divided into *hard constraints* and *soft constraints*.

Hard constraints are those limitations which must be upheld absolutely in order for a schedule to be feasible or at all acceptable to the league. They are either satisfied or violated; there is no middle ground. For example, the basic criteria that all teams play each other exactly once and that no teams' games overlap are considered hard constraints.

Soft constraints, on the other hand, are those restrictions for which violations make a schedule less *desirable* but not less feasible. Unlike binary hard constraints, they can be partially satisfied. For example, one possible soft constraint imposed by administrators is that all teams should travel the same distance or amount of time throughout the course of the season. This is merely a preference, for it is near impossible to balance travel time or distance exactly. A soft constraint is therefore generally assessed on a rating scale. In an optimal solution, or the "best" possible schedule, all of the league's hard constraints and soft constraints would be completely satisfied. Since this is fairly unrealistic, a certain threshold will need to be set for what determines a desirable enough schedule for consideration by league administrators. Let the set of desirable schedules be $D \subseteq F$. The final goal of an automated scheduler is to produce a schedule $d \in D$ such that d is the best or optimum of all D.

1.2 The NESCAC Problem

The NESCAC men's soccer league is composed of 10 teams: Amherst, Bates, Bowdoin, Colby, Connecticut College, Middlebury, Trinity, Tufts, Wesleyan, and Williams. The complexities of scheduling this league were outlined by Andrea Savage, NESCAC Administrative Director, as follows [16]:

1.2.1 Hard Constraints

Constraints that must be satisfied for a schedule to be accepted by the league:

1. Ten teams must play each other in the space of eight weeks.
    - Each team plays eight weekend games and one mid-week game.
2. Teams must play each other exactly once.

3. Teams cannot play more than one game at the same time.
4. Out of the nine games played by each team, either five must be home and four away, or vice versa.
5. No team can ever play more than two consecutive away games.
6. Some teams must be scheduled at home on certain dates.
    - Homecomings and Family Weekends require home sporting events.
7. Within any group of three rival teams, a *triplet*, each team should play one of its rivals home and the other away.
    - Rival triplets: Amherst, Williams, and Wesleyan; Bates, Bowdoin, and Colby.
    - Example: If Amherst plays Williams at home, it must play Wesleyan away.

1.2.2 Soft Constraints

Constraints that make a schedule more desirable:

8. No team should spend a disproportionately large amount of time traveling to away games throughout the season. For example:
    - No team should play all of its most distant opponents away in one season.
9. Mid-week games must either involve fewer than two hours of travel or be scheduled for the same weekend that the away team plays at an opponent within three hours of the mid-week opponent. For example:
    - Middlebury cannot travel to Amherst on a weekday, but Middlebury may play at Amherst and at Williams in the same weekend.
10. Important rival match-ups, such as Amherst versus Williams, should occur during a preferred part of the season.

1.3 What Approaches Have Been Used?

There are many possible ways to tackle the task of developing a sports schedule that meets as many of the criteria laid out by the league administrators as possible. Some approaches focus directly on the constraints and try to incrementally construct a desirable solution with a deterministic algorithm. A direct approach is often difficult or near impossible to design, however, given the complexity of both the hard and soft constraints. Therefore, some lines of attack rely on creating a random feasible schedule and then evaluating it in terms of soft constraint satisfaction to determine the level of desirability. The schedule can be accepted, modified, or discarded based on this evaluation.

While the focus of this thesis is on the latter type of approach, namely a generate-and-test model, it is important to note two other approaches that have had success in certain situations: reducing the problem to one of graph edge coloring; and linear programming. Scheduling can be achieved by thinking of teams as nodes in a graph and games as edges between team nodes. Edges can then be assigned a color corresponding to a specific round of the season. Additional constraints can be added by introducing new nodes and by adding edges between these new nodes and the team nodes [8]. Linear programming has also proven to be effective in scheduling a college sports tournament similar to the NESCAC men's soccer competition. Using this approach, Nemhauser and Trick successfully found highly desirable men's and women's basketball schedules that are now used by the Atlantic Coast Conference [13].

The second type of approach to sports scheduling treats the problem as one of *combinatorial optimization*. Combinatorial optimization can be described as follows: Given a set X of solutions $s \in X$ and an objective function $f(s)$, find the solution $s^* \in X$ for which $f(s^*)$ is minimized (or maximized, depending on the direction of optimization). In most real-world combinatorial optimization problems, X is either far too large for an exhaustive search, or it is difficult to explicitly define the set of all solutions. For example, the current problem has slightly fewer than $\frac{N(N-1)}{2}!$ solutions in its search space. Therefore, any algorithm attempting to perform an exhaustive search will not scale well.

Because the search spaces of these problems grow exponentially, research has focused on using heuristic, rather than exact, methods to attempt to find near optimal solutions. Three ideas from Operations Research and Artificial Intelligence used with great success, as evidenced by the fact that their computer-generated schedules have been adopted by national sports leagues, are *tabu search*, *simulated annealing*, and *genetic algorithms*. Each of these involves encoding human schedulers' heuristics, or "tricks of the trade," to navigate the search space in an intelligent way, hopefully considering those solutions that are more likely to be close to the optimal solution. Tabu search and simulated annealing are both *iterative search procedures* or *local search* algorithms, whereby solutions are examined one at a time: the next solution is generated by changing the current solution in some small way. By applying certain rules at each step, this process guides the search toward a solution which best satisfies some criteria, i.e. a solution with a sufficiently low (or high) objective function value. For more information on how tabu search and simulated annealing operate and how they can be applied to scheduling, see [1, 8, 18].

1.3.1 Genetic Algorithms

Genetic algorithms use ideas of natural evolution, specifically Darwin's notion of the "survival of the fittest," to solve optimization problems. Beginning with an initial, random *population* of feasible solutions called *chromosomes*, a genetic algorithm generates progressively more desirable, or "fitter," solutions over a series of *generations* by mimicking the processes occurring on DNA in real chromosomes. A solution's fitness is determined by applying an *evaluation function*, which incorporates a human scheduler's knowledge in distinguishing better and worse solutions. A genetic algorithm carries out evolution by selecting parent chromosomes, probabilistically based on their fitness values, and applying *crossover* and *mutation* operators to produce offspring chromosomes. A crossover operator takes parts of two presumably good solutions and merges them into one new, hopefully even better, solution. A mutation operator applies a random transformation to a current solution in order to introduce new variations into the population. This helps prevent the search from getting trapped in local minima. These operators must be designed with care to ensure that the characteristics of the best solutions are preserved across generations and that enough variability

is introduced to allow the search to cover a substantial portion of the search space. Some genetic algorithms, including the one developed here, always promote a certain percentage of strong, or *elite*, chromosomes unchanged and always discard (i.e., never select for crossover or mutation) a certain percentage of weak, or *culled*, chromosomes. If implemented correctly, genetic algorithms produce better and better solutions by following the general heuristic of keeping what is good and discarding what is bad. Although genetic algorithms seem "blind" because they do not use a knowledge-driven deterministic procedure, they have been shown to work quite well in many optimization tasks [1, 10].

A Simple Genetic Algorithm

1. **Initialization:** Generate a random population of n chromosomes.
2. **Evaluation:** Evaluate the fitness of each chromosome in the population, and order chromosomes by their fitness.
3. **Elitism:** Automatically add the top e chromosomes of the current population to a new population. Let c be the number of culled, disregarded chromosomes such that $c + e \leq n$.
4. **Reproduction:** Complete the new population by repeating the following steps $n - e$ times:
   (a) **Operator Selection:** Choose an operator to apply.
   (b) **Parent Selection:** Select either one or two parents, depending on the chosen operator, from the top $n - c$ chromosomes based on their fitness, using one of several probabilistic selection procedures.
   (c) **Apply Operator:** Apply the chosen operator to the selected parent (or parents).
      i. **Crossover:** Combine random subparts of the two parents to form a new offspring.
      ii. **Mutation:** Randomly modify the selected parent to produce a new offspring.
   (d) **Acceptance:** Place this new offspring into the new population.
5. **Replace:** Use this new population in place of the previous population.

1.4 Why Use Genetic Algorithms over Other Heuristics?

According to Falkenauer, while optimization tasks vary greatly, one general rule employed by heuristic methods is the idea that one should "Stick to what you already know is good [4]." These algorithms explore the search space based on motivation from the best solution discovered so far, so as to avoid a purely random search. Tabu search and simulated annealing both iteratively examine solutions in the vicinity of the current one until one that meets certain stopping criteria is encountered. Both of these heuristic methods have mechanisms to ensure that the search does not get trapped in local optima. Again, for more information on these algorithms, see [1, 8, 18].

Genetic algorithms follow a similar rule for sticking to what is known to be good when searching for improved solutions. However, this approach diverges from the previous methods, according to Falkenauer, because it rests on the premise that a good solution contains several good subparts [4]. Therefore, a genetic algorithm builds on what is already known to be good by taking parts of two solutions and merging them to create a new solution. Through the use of well-designed crossover operators, a genetic algorithm can combine the good parts within individual solutions in the genetic algorithm's population to produce, in theory, even better solutions containing more good subparts.

1.4.1 Schema Theorem

While the genetic algorithm approach sounds promising, there is one hurdle to get over. How do we know which parts out of a whole solution make it a good solution? Luckily, it does not matter. The evaluation function of a genetic algorithm only works on whole solutions, so genetic algorithms judge a part of a solution by the quality of the various solutions that contain it [4]. The Schema Theorem, formulated by Holland, states that well-performing *schema*, parts of good solutions, will be sampled at an exponentially increasing rate over the generations run by the genetic algorithm [9]. Since higher quality solutions are selected for reproduction at a greater rate than poor solutions, individual parts of these good solutions will be included in an exponentially increasing number of individuals in the population. Therefore, after some relatively small number of generations, the population will be saturated with successful solutions. While Holland's work was done under a

set of rather simplified, idealistic conditions that do not exist in most real-world problems, the Schema Theorem holds the key to the success of the genetic algorithm approach. The idea of recombination, or crossover, to merge parts of solutions in the hope of generating promising new solutions, differentiates genetic algorithms from the other heuristic approaches discussed above [4].

We see that genetic algorithms have an advantage over other heuristic search methods. Rather than exploring the neighborhoods of good solutions, genetic algorithms assume that solutions are strong due to favorable parts. The search for better solutions rests in combining randomly selected parts of good solutions, and, as the Schema Theorem proves, increasingly better whole solutions will be produced. At the same time, though, better solutions may exist in completely different sections of the search space, so it is important to always introduce some random mutations to current solutions. The parallel nature of the genetic algorithm paradigm allows for multiple solutions to be examined at the same time. Thus solutions found by crossover operators can be compared to solutions found by mutation operators. Genetic algorithms provide the robustness of local search methods while taking advantage of the added benefit of crossover or recombination [4].

This thesis demonstrates that a genetic algorithm is a suitable method for solving a real-world optimization problem like highly constrained sports scheduling. When a direct, constructive approach cannot be developed, this heuristic approach is capable of guiding a search toward a set of very good solutions in a relatively short period of time. Illustrating this point, a genetic algorithm is developed here that can evolve highly desirable schedules for the NESCAC men's soccer season.

The remaining chapters are organized in the following manner:

- Chapter two reviews the foundations of genetic algorithms.
- Chapter three investigates the considerations involved in designing a genetic algorithm for a real-world optimization task. It carefully describes the current algorithm, consisting of customized components best suited for the NESCAC problem: a permutation-based, indirect encoding of schedules; a schedule-building routine that decodes chromosomes into schedules that meet most hard constraints; and an evaluation function that assesses the strength of

schedules based on their satisfaction of the remaining hard and soft constraints. Other critical parts of a genetic algorithm, such as the selection procedure, genetic operators, and parameter values, are also discussed in the context of these choices.

- Chapter four presents the results obtained with the current algorithm, demonstrating its success in finding schedules that satisfy all of the NESCAC administrators' requirements remarkably well. The best schedule ever found, which satisfies Constraint 8 to a high degree and all other constraints fully, is analyzed in depth.

- Chapter five introduces ideas for improving the effectiveness of the current algorithm and adapting it for other sports leagues and other NP-Hard scheduling tasks.

Chapter 2

Foundations of Genetic Algorithms

Now that the problem of scheduling has been outlined, the specific task of scheduling the NESCAC men's soccer season has been defined, and several approaches to scheduling have been presented, let us explore genetic algorithms in greater depth.

2.1 That's One Powerful Metaphor

By using natural evolution and genetic recombination as a metaphor for evolving solutions in a "survival of the fittest" manner, genetic algorithms are a prime candidate for solving highly constrained real-world optimization problems. Davis explains why the inventor of genetic algorithms, John Holland, first believed that they would provide useful mechanisms for computer algorithms in 1975 [3]. Evolutionary biologists have marveled at the process of natural selection since Darwin first presented the idea and subsequently when it was accepted by the scientific community. It is amazing that the complex level of life now present could have evolved in such a short period of time compared to the age of the physical universe [3].

2.1.1 Evolution by Computer

Holland recognized that evolutionary mechanisms are potentially valuable ways to solve difficult problems in a computer [9]. If nature could evolve multifaceted, intelligent beings, he reasoned, then the same processes, appropriately implemented, could evolve solutions to computational problems. Thus, Holland began experimenting with strings of binary digits analogous to chromosomes.

Holland's simulated evolution manipulated the bits in these strings based only on an evaluation of each chromosome. His algorithms were completely blind to all other knowledge of the actual problems they were solving. Like natural evolution, no memory across generations existed, and only a chromosome's evaluation could increase its chances of selection. Chromosomes with bad evaluations, therefore, rarely contributed genetic material to any new chromosomes. The first set of problems he explored were function optimizations, so the bit strings represented input values, and a chromosome was evaluated simply by the output of the function to be minimized or maximized. Employing only simple encoding and reproduction schemes, and lacking knowledge of how one typically solves such problems, Holland's algorithms functioned remarkably well. His Schema Theorem explains their success in generating progressively stronger populations of solutions [9]. Furthermore, after over twenty years of advancement, Holland's creations, now known as genetic algorithms, can be used to produce better solutions to intricate problems like scheduling and circuit design than other methods can [3].

2.1.2 Problem-Specific Components

In order to carry out simulated evolution on a population of solutions, there are two critical components that must link the otherwise abstract genetic algorithm to the specific problem at hand: an encoding scheme and an evaluation function [3].

Chromosome Representation

Genetic algorithm researchers have devised many methods to encode problem solutions into easily manipulatable chromosomes. Selecting an adequate yet not too cumbersome encoding is inescapably tied to the specific problem. Likewise, a corresponding method of decoding chromosomes is critical to the success of a genetic algorithm. Holland primarily used bit strings to represent solutions, but this technique is not appropriate for some more complex problems. The genetic algorithm for scheduling the NESCAC men's soccer schedule uses an encoding scheme based on permutations of integers, while other scheduling problems demand more complex representations of solutions [1]. Both encoding and decoding are discussed in Chapter 3.

Evaluation Function

The evaluation function is the fundamental connection between the genetic algorithm and the problem. It is a function that takes a chromosome as input and returns a measure of the quality of that chromosome, based on knowledge of the problem and an assessment of constraint satisfaction. The return value is typically a single number or, for a multi-objective function, a set of numbers, each evaluating the chromosome on a separate criterion. Just as real organisms survive and reproduce based on their fitness in the outside world, genetic algorithms' chromosomes are selected for reproduction based on the value or values produced by the evaluation function. Deciding how to incorporate domain knowledge and problem constraints into a suitable evaluation function is the subject of Section 3.2.

Chapter 3

Designing a Genetic Algorithm for a Real-World Task

The genetic algorithm framework discussed thus far is rather generic. For any real-world optimization task, each step in the process of designing a genetic algorithm takes time and serious consideration to create an efficient, high performance algorithm suitable to the problem at hand.

The design of a genetic algorithm can be broken down into five major steps. They are:

1. Design an encoding scheme, or representation, and an appropriate decoder.
2. Build an evaluation function that incorporates domain-specific knowledge.
3. Choose a method for selecting parent chromosomes for reproduction.
4. Construct operators appropriate for the encoding.
5. Fine-tune parameters to improve performance.

This chapter presents the main issues involved in each step and discusses how each component is designed in the genetic algorithm for the NESCAC scheduling problem.

3.1 Representational Schemes

The first decision in designing a genetic algorithm for a specific real-world task is choosing a representation of solutions. One must determine what a chromosome will look like and how information

regarding the problem to be solved will be incorporated into a chromosome.

Binary strings are most often used to represent potential solutions to a problem. Binary strings are easy to create and manipulate, they do not require complex data structures or a large amount of space in memory, and all possible bit strings of a given length may, in theory, be created and examined. The simple nature of binary strings makes it easy to prove theorems regarding the algorithms that employ them [3]. When Holland first began working with genetic algorithms, he chose bit string chromosomes for this very reason, enabling him to prove the Schema Theorem about the performance of his newly discovered problem-solving technique [9].

Some researchers, such as David Goldberg, feel that a genetic algorithm should be a "black box" problem solver, void of any domain knowledge or other customization for a specific problem outside of the evaluation function [6]. Therefore, binary strings have been the traditional choice of encoding because they allow for the creation of a robust algorithm that can be applied to a wide variety of problems; it is possible to encode almost anything as a binary string. For example, for the problem of finding the maximum value for some n-dimensional mathematical function, binary numbers representing each of the n inputs may be concatenated together to form one large binary string. Once a solution is in this generic form, simple crossover and mutation operators may be applied, and the processes of natural selection may be carried out. The algorithm can be completely blind to the meaning of each bit in the chromosome. One standard genetic algorithm, with appropriate parameters and operators, can therefore be used to attempt to solve any number of problems [3].

Developing a multi-purpose "black box" genetic algorithm using a binary encoding scheme may be valuable for the research community, but it is problematic when approaching real-world applications. While it may be possible to represent just about any solution to any problem using a binary string, such an encoding scheme often leads to great difficulty in designing other aspects of the genetic algorithm. For example, in the Traveling Salesman Problem (TSP), a potential tour may be encoded as a binary string by joining together the binary representation of each numbered city in the order in which the cities are visited. Applying simple bit-changing operators, however, will likely lead to a binary string signifying a tour that is not a well-formed solution to the problem [19].

A more appropriate encoding for the TSP employs a permutation of the list of visited cities' numbers as a chromosome and uses special order-based operators that combine or mutate parents in such a way that only the order of the elements is changed, ensuring that the result is a valid ordering without any duplicates or missing values [19]. An encoding based on permutations of integers has several benefits. Aside from ensuring the validity of child chromosomes, these chromosomes are significantly easier for humans to comprehend, since a list of integers tends to be a more familiar solution representation than a binary string.

Some genetic algorithm researchers would object to modifying the basic encoding scheme because the Schema Theorem was based on binary string representations [19]. However, if new representations work more effectively and efficiently, then we should explore these possibilities. Alternate encodings and their complementary operators have been rather successful when applied to real-world problems in the past. Genetic algorithms with such variations also seem to function and manipulate schemata in ways comparable to their more traditional counterparts [14].

For scheduling problems, as discussed below, a schedule can take many forms, such as a list of events with corresponding assigned times or a matrix of scheduling data. Schedulers in the real world are likely to look at a calendar-like view of a schedule. These natural representations make the task of maintaining the feasibility of a solution during crossover and mutation exceedingly difficult. Thus an adequate representation must fit the problem well, but also be consistent with existing or easily adaptable genetic operators [3, 19].

3.1.1 Encodings for Scheduling Problems

There are two main ways to represent a schedule as a chromosome in a genetic algorithm: directly or indirectly.

Direct Encoding

In a direct encoding, a schedule itself is the chromosome. This schedule may take the form of a grid of events grouped by dates and times, or a list of all of the events in the schedule, each with an assigned time. In either case, the representation will mostly likely be fairly large and unwieldy,

requiring complex operators to work on it. A crossover operator, for example, would need to ensure that parts of two different schedules combine to form a feasible schedule. Likewise, mutating a direct representation of a schedule would require repair work to ensure that all events are still included in the modified schedule. In a complicated real-world task like scheduling, however, it may be advantageous to have a more direct encoding. Customized operators can access a schedule in its natural form and make beneficial swaps and corrections. For example, if a sports schedule were missing an important rival match-up at the end of the season, a greedy algorithm could scan the last few weeks of the schedule to find a low-priority game with which to substitute the high-priority one. With a chromosome closely representing the underlying schedule, this type of local improvement is possible, and good solutions may be found more rapidly than with other encodings [19].

Indirect Encoding

A schedule can alternatively be represented in an indirect fashion, for example as a permutation of teams to be scheduled. Rather than having all the information regarding the schedule in the chromosome itself, the list of teams serves as a blueprint for generating a schedule. A schedule-builder or chromosome interpreter creates the actual schedule. The ordering of the teams does not directly represent a schedule, but it has a direct influence on the schedule produced from it. This schedule can then be evaluated on the basis of the problem's constraints. The drawback of an indirect encoding is that it is difficult to predict how a change in the ordering of the elements in the chromosome affects the schedule itself. This makes creating operators to enact specific changes in a schedule near impossible [19]. Only a sophisticated schedule-building procedure would allow one to reverse the interpretation process and create a chromosome based on a directly manipulated schedule. Nevertheless, like the permutation-based encoding for the TSP, this simpler representation can be easily manipulated using existing order-based crossover and mutation operators.

After choosing an indirect encoding scheme, one must design a problem-specific way of decoding a chromosome into a schedule representation that can be evaluated based on the problem's constraints. The main consideration in creating any decoder is deciding how intelligent or complex to make it. While it must always create valid, legal solutions, it is possible to allow the decoder to

perform some degree of optimization or enforcement of certain constraints before returning a decoded version of a solution. On the other hand, it may be important to keep weak schedules in the population for added diversity. We will see that, in addressing the NESCAC problem, some hard constraints are best imposed at the time of schedule-building to prevent many infeasible schedules from being created, while other softer constraints are reserved for the evaluation function.

3.1.2 The NESCAC Encoding and Decoding Scheme

Initial experimentation with a direct representation for the NESCAC men's soccer schedule quickly demonstrated the inadequacy of this approach. Under this strategy, a chromosome looks much like a real schedule in print: a two-dimensional array, with columns corresponding to teams and rows corresponding to weeks in the season. Each element in the matrix refers to a game between a home team and an away team. Unfortunately, finding initial feasible schedules under this representation is as difficult as the entire scheduling problem. Games may be added to the schedule one at a time, but it becomes increasingly hard to insert the final games between teams that share no free dates. A greedy algorithm to build a feasible schedule, regardless of the soft constraints, thus requires a considerable amount of backtracking and consequently a great deal of time. Experiments with this approach did not generate any feasible schedules throughout many trials.

In addition, it is difficult to build genetic operators to manipulate this representation. For example, there is no easy way to combine the first half of one schedule with the second half of another, as the resulting schedule will inevitably contain some games twice and some not at all.

Having learned that an indirect encoding is better suited to represent a schedule, the short encoding developed by Leonard for a genetic algorithm to schedule the Australian Basketball League was modified for use with the current problem [11]. This approach first uses a simple deterministic algorithm to generate a set of games for each week. Once this timetable is created, prioritized rules are applied to choose where the games are played. These two steps result in a schedule that can easily be assessed by the evaluation function.

Schedule-Builder

The indirect encoding consists of two permutations of team identifiers in the range [0, $N-1$], where N is the number of teams [11]. For the NESCAC problem, $N = 10$. The search space thus contains $N! \times N!$ solutions

A schedule-building routine decodes the chromosome to form a schedule in two steps. First, it generates the game match-ups per round. Then it applies a set of constraint-based rules to choose the home team for each game.

Generating Games

The following algorithm, adapted from Leonard, generates a set of games for each week in the season [11]:

1. Use the first permutation in the chromosome to obtain an ordered list of teams, indexed from 0 to $N-1$. If there is an odd number of teams, then one "virtual" team is added at the start of the list.

2. Create one round of games by pairing up teams: team_0 vs. team_{N-1}, team_1 vs. team_{N-2}, etc.

3. Assign the current round to week w, where w is the position of team_1 in the second permutation.

4. Rotate teams between position 1 and position $N-1$, leaving team_0 in place. That is, team_1 becomes team_2, team_2 becomes team_3, ..., and team_{N-1} becomes team_1.

5. Repeat steps 2 through 4 until the original team in position 1 returns to position 1.

For $N = 10$, this algorithm creates nine rounds, each assigned to a week, such that each team plays every other team exactly once and only plays one game per week. Had there been a "virtual" team, its games would represent byes for the opposing team. The NESCAC men's soccer season is restricted to eight weeks, though, so week zero is considered to be a mid-week game played at

some point in the season determined either by the evaluation function or the teams involved. The resulting schedule thus satisfies Constraints 1, 2, and 3. (See Section 1.2.)

Choosing the Home Teams

The next step in the schedule-building routine is to choose the home team for each game. Adapted for the NESCAC problem from Leonard's work, this process applies a prioritized rule base [11]. For each game (between team one and team two) in each round, the following rules are applied in this order:

1. If the game has been assigned a location already because it is part of a triplet (see step 2a), continue to the next game.
2. {Constraint 7} If the game is between two teams in a triplet (i.e., Amherst vs. Williams):
   (a) If one of the other triplet games has been assigned a location (i.e., Amherst at Wesleyan), then set the home team of this game (i.e., Williams at Amherst) and the other game in the triplet (i.e., Wesleyan at Williams). This rule ensures that mutually exclusive triplets are always scheduled properly.
   (b) If none of the games in the triplet have been scheduled yet, continue to the next rule.
3. {Constraint 6} If team one requires a home game on this date, then make it the home team.
4. {Constraint 6} If team two requires a home game on this date, then make it the home team.
5. {Constraint 5} Let the number of consecutive away games for team one be c_1 and for team two be c_2. Also let c_{max} be the maximum number of consecutive away games allowed. For the NESCAC men's soccer schedule, $c_{max} = 2$.
   (a) If $c_1 > c_2$ and $c_1 \geq c_{max}$, then make team one the home team.
   (b) If $c_2 > c_1$ and $c_2 \geq c_{max}$, then make team two the home team.
   (c) Else, continue to the next rule.

6. {Constraint 4} Set the team that has fewer home games scheduled to be the home team. If both have an equal number of home games, arbitrarily assign team one to play at home.

These rules attempt to assign home teams to complete a schedule that satisfies the additional hard constraints. Given the prioritized nature of the rules, though, there is no guarantee that they will all be completely satisfied. As the first priority, enforcing a triplet condition may result in scheduling too many consecutive away games for one team, for example. Any such constraint violations will be recognized later by the evaluation function, and infeasible schedules will be eliminated.

3.2 Creating an Evaluation Function

The next major step in the development of a genetic algorithm for a real-world task is to build an evaluation function that can adequately identify strong and weak solutions. Within the metaphor of natural evolution, the evaluation function serves as the environment that determines how well each individual performs in the world of the problem. The genetic algorithm relies on fitness values of decoded chromosomes to dictate which parents are selected for creating the next generation of chromosomes. The function must embody knowledge that determines what makes a solution appear to be good or bad to the end user [19].

In traditional genetic algorithms, the evaluation function is often implicitly defined by the problem [12]. In the case of a function optimization problem, for example, the evaluation function is typically the function itself. However, even when there is a clear cost function, it is often necessary to incorporate further heuristics to make the evaluation useful. In the case of the Bin Packing problem, for example, where counting how many bins are used to "pack" all of the objects is the correct mathematical evaluation, Falkenauer argues that this measure does not adequately guide the search toward better solutions. He proposes a function that assesses not just how many bins, but which specific bins, are used in the current solution [5]. Most real-world problems have added constraints that make the development of a "perfect" evaluation function very challenging. Heuristic evaluation functions must capture the subtle techniques used by human experts when

distinguishing better from worse solutions. The preferences for a schedule are often described by humans in vague, hard to codify terms, making this task even more difficult [3]. Some problems also have multiple, sometimes conflicting, goals that need to be optimized simultaneously. Such multi-objective optimization tasks require substantial work in reconciling and assigning relative values to each of their goals [12].

3.2.1 Handling Infeasible Solutions

When designing an evaluation function, a decision must be made whether to allow infeasible solutions to stay in the pool. One way to handle infeasible solutions is to eliminate them from the population completely [12]. With this "death penalty" heuristic, infeasible solutions are not evaluated or processed in any way, thus simplifying the remaining parts of the algorithm. When the feasible search space is large relative to the entire search space, this technique may work well. In some cases, encodings and operators can be constructed to avoid ever producing infeasible solutions.

Alternatively, search problems may begin with a population of infeasible solutions that are gradually improved until they become feasible. Michalewicz argues that it is essential to consider infeasible solutions [12]. They introduce new variations and may include beneficial subparts which will recombine to form superior feasible solutions. Furthermore, the optimum solution may occur along the boundary between feasible and infeasible parts of the search space. In a heavily constrained problem, where the best solution often pushes the constraints to the absolute limit without violating them, this may be the case. Thus it is to the genetic programmer's advantage to be able to approach the optimum from both sides of the search space [12].

When infeasible solutions are allowed, a penalty function adds (or subtracts, depending on the direction of optimization) a penalty to the fitness value. A penalty value may be a constant, it may be based on the degree to which the solution is infeasible, or it may depend on the amount of effort needed to make the solution feasible [12]. Determining the latter two values can be quite difficult. One method of gauging the "amount" of infeasibility is to set penalty coefficients for each constraint. The penalty function is then the sum of each violation multiplied by the corresponding coefficient. This assigns a solution a penalty proportional to the number of violated constraints.

3.2.2 The NESCAC Evaluation Function

As we have already seen, the NESCAC algorithm uses a schedule-builder that ensures that most hard constraints are satisfied. With the encoding scheme and operators only permitting valid permutations, all chromosomes can be decoded to rounds of unique games, but choosing home teams may produce schedules that violate certain hard constraints. Thus the evaluation function must both assess the satisfaction of the soft constraints and penalize for any violated hard constraints. The function takes the decoded schedule as input and returns a fitness value ranging from near zero up to several hundred, depending on coefficient values.

The evaluation function is a composite of six ratings, determined as follows:

- Let r_1 be the Time Rating, evaluating Constraint 8. (See Appendix A for the NESCAC Travel Time Grid.)
  - Let $time_{ij}$ be the number of hours required to travel from team$_i$ to team$_j$.
  - Let avg_i be the average length of travel required by team$_i$ throughout the season:

$$avg_i = \frac{\sum_{j=0}^{N-1} time_{ij}}{2}$$

    since about half a team's games are away. These values range between 11.75 and 19.25.
  - Let act_i be the actual length of time traveled by team$_i$ during the current schedule:

$$act_i = \sum_{j \in A} time_{ij}$$

    where A is the set of team$_i$'s away opponents in this schedule.
  - Compute r_1 as:

$$r_1 = \sqrt{\frac{\sum_{i=0}^{N-1} (avg_i - act_i)^2}{N}} \tag{3.1}$$

- Let r_2 be the Consecutive Rating, evaluating Constraint 5.
  - Consider each team's games in order by round, and for each away game, increment a counter of the number of consecutive away games.
  - For every consecutive away game above the maximum number allowed, add 1 to r_2.
  - When a home game is found, reset the counter to 0.
- Let r_3 be the Violated Home Rating, which assesses Constraint 6.
  - Let H be the set of required home games.
  - Calculate r_3 as the number of games in H that are not scheduled for home.
- Let r_4 be the Location Balance Rating, evaluating Constraint 4.
  - Let $home_i$ be the number of home games played by team i.
  - Let $away_i$ be the number of away games played by team i.
  - r_4 = the number of teams for which $\mid away_i - home_i \mid \neq 1$
- Let r_5 be the Mid-Week Rating, which evaluates Constraint 9.
  - r_5 = the number of games in week 0 for which the travel time is greater than 2 hours, and the away team does not play any away games within 3 hours of the home team.
- Let r_6 be the Week Preference Rating, which evaluates Constraint 10.
  - Let P be the set of week preferences.
  - r_6 = the number of games in P that are not scheduled in the preferred range of weeks.

Each rating also has an associated coefficient f_i that dictates how much weight to place on it in the overall evaluation. (Each rating's coefficient value was initially set based on the relative importance of its corresponding constraint, as suggested by the league administration [16].)

The evaluation function E is thus defined as follows:

$$E = \sum_{i=1}^{6} f_i r_i \tag{3.2}$$

For a desirable schedule, $\sum_{i=2}^{6} r_i = 0$ because r_2 through r_6 each count violations which could theoretically be zero. Since r_1 is based on theoretical average travel times, it is not possible for this rating to evaluate to zero. Therefore, when all of the other constraints are satisfied, r_1 must be minimized to find a good solution.

After just a few sets of test runs using this formula, it became evident that only schedules with ratings two through six equal to zero ever emerge as the best solutions. As suggested earlier, infeasible solutions may contribute to better solutions. A schedule with too many consecutive away games, for example, may actually contain some positive schema. It would be unwise to disregard this schedule by giving this rating a very large weight. After a single crossover or mutation, the violating consecutive away games could be removed, and a highly desirable schedule could emerge. It was therefore decided that factors f_2 through f_6 should be equal to 10, or some number greater than a poor r_1 value. This allows schedules that violate one or two constraints in minor ways to survive and take part in the reproductive process. However, there is no way that a schedule with a major constraint violation will have an evaluation lower than one whose fitness is based on the Time Rating alone. The diversity provided by some infeasible solutions thus does not threaten the quality of the best solutions. By keeping these coefficients equal to 10, the positive benefits of maintaining some infeasible schedules are incorporated into this algorithm, but not the costs.

3.3 Selection and Reproduction

Compared to the encoding and evaluation function, the selection process is not as closely tied to the specific problem to be solved. However, how one samples the population in choosing parents is important to the overall performance of the algorithm. Effective evolution can only take place if successful, yet sufficiently diverse, parents are chosen. Parent selection must give more reproductive opportunities to the fittest individuals in the population. At the same time, less fit members of

the population should be given some chance to contribute to the genetic composition of the next generation.

3.3.1 Proportional Selection

Some selection techniques choose parents proportionally based on their fitness values. *Roulette wheel parent selection*, for example, simulates the spinning of a roulette wheel with sections sized in proportion to chromosomes' fitnesses. Therefore, fitter members are selected with higher probability. Roulette wheel selection seeks to directly promote the reproduction and reuse of fitter chromosomes in an attempt to improve the overall population. While roulette wheel selection offers a bias toward fitter members of the population, it can lead to sampling the same very good chromosomes with too much frequency. This selection procedure may thus lead to premature convergence around a local optimum [15].

3.3.2 Tournament Selection

Another approach to selection is based on chromosomes' rankings relative to one another. *Tournament selection* is a basic alternative to proportional selection in which t individuals in the population are selected uniformly at random, and the fittest of these t competing individuals is chosen as a parent. This method ensures that the selected individual is at least better than $t-1$ other individuals. Binary tournament selection, in which $t = 2$, is the most commonly used variation of this approach. Selection thus favors higher quality individuals, but not in the same way that proportional selection methods do. Weaker members still have a good chance for taking part in reproduction, since the selection is only relative to the other randomly selected competitors. For example, an individual fitter than the others in the tournament could still easily be below average overall. This approach is less prone to premature convergence than proportional methods [7]. By pitting individuals against each other for the ability to reproduce, this technique appears more comparable to the natural "survival of the fittest" process than roulette wheel selection [7].

3.3.3 Elitism and Steady-State Reproduction

It has thus far been assumed that the entire new generation is created by replacing all of the parents with their offspring. This is known as *generational replacement* [3]. Under this reproductive plan, genetic material transfers to the next generation exclusively through the application of operators to the parents selected. However, chromosomes selected with the best fitness values may combine in an ineffectual way to create new chromosomes with low fitness values. Moreover, the stochastic nature of these methods precludes any guarantee that strong chromosomes are selected for the reproduction process at all. Beneficial genetic material thus may be lost for future generations.

Elitism aims to reduce this problem by automatically copying the best chromosomes from the current population to the next. The remainder of the next generation is created through a generational replacement scheme. The main problem with this approach is that elite individuals and many identical or similar copies may soon dominate the population. However, the elitist strategy appears to be an improvement over a non-elitist one, as it provides a guarantee that the strongest members of the population are never lost [3]. In conjunction with elitism, some reproductive techniques *cull*, or eliminate, some number of very weak individuals to prevent offspring from inheriting their poor genetic material.

An extreme form of elitism, *steady-state reproduction* keeps much of the population constant and only replaces one or two individuals in the population in each new generation [3]. This gives better individuals multiple opportunities to be chosen for reproduction and to interact with one another.

"Steady-state without duplicates" is a further enhancement to prevent the population from quickly becoming saturated with duplicate copies of the same individual chromosomes [3]. In this strategy, any children identical to current members of the population are discarded, and reproduction continues until the population is filled with unique individuals. This makes more efficient use of the limited number of chromosomes in the population and reduces the risk of premature convergence around one chromosome. Because the time required to check for duplicates is negligible compared to the time spent evaluating solutions, the added benefit of being able to consider more

unique individuals generally outweighs any added cost [3].

3.3.4 The NESCAC Reproductive Plan

The NESCAC algorithm uses steady-state reproduction without duplicates, coupled with elitism and culling weak members from the population. The next generation is formed by keeping an elite portion of the population steady and by mating selected individuals from a non-weak portion of the population. Let n be the number of individuals in the population. If e is the elite percentage, $\lfloor n * e \rfloor$ individuals are considered elite. For example, if $n = 100$ and $e = .30$, then the top 30 individuals of the population are automatically promoted to the next generation. If c is the culled percentage, $\lfloor n * c \rfloor$ individuals are made unavailable for reproduction. If $n = 100$ and $c = .40$, then the bottom 40 individuals are eliminated. $\lceil n(1-e) \rceil$ new individuals are created by mating or mutating parent chromosomes selected from the top $\lceil n(1-c) \rceil$ individuals using a binary tournament selection algorithm. Once the selected chromosomes are either mutated or crossed over, only unique offspring are inserted into the next generation.

The complete reproduction process runs as follows:

1. Promote the $\lfloor n * e \rfloor$ elite individuals to the next generation.
2. Loop until $\lceil n(1-e) \rceil$ unique offspring are created:
   (a) Select one parent from the top $\lceil n(1-c) \rceil$ chromosomes using binary tournament selection.
   (b) For each permutation in the chromosome:
      i. Choose to apply either crossover or mutation based on predetermined weights.
      ii. If crossover is chosen, select an additional parent chromosome.
      iii. For the chosen type of operator, randomly choose one of the available operators.
      iv. Apply this operator to the current permutation of the selected chromosome(s).
   (c) If the newly created offspring is a duplicate, discard it. If it is unique, add it to the new generation.
3. The next generation will now have n individuals.

3.4 Operators

Crossover and mutation operators use pieces of the parent chromosomes to build new chromosomes. Crossover operators work on two parents, while mutation operators may simply alter one parent. It is a challenging task to determine how to combine or mutate solutions in a meaningful way. In addition to designing the actual operators' functionality, a genetic algorithm practitioner must determine the frequency with which each operator is applied and decide how and if multiple operators should be applied to the same parent. For example, it is possible to crossover two parents and then apply a mutation operator to each of the resulting offspring. Several approaches to crossover and mutation exist, yet we will focus primarily on those designed for permutation-based encodings.

3.4.1 Crossover Operators

Crossover operators, some would argue, are the fundamental reason why genetic algorithms are successful [4]. Genetic algorithms' ability to combine solutions distinguishes them from other heuristic methods that consider and manipulate individual solutions. Crossover operators, therefore, need to be designed in such a way that high-quality schemata from two parents can be integrated into one offspring. It is not necessary to understand how these schemata can be found or even to find them at all. Rather, a crossover operator should permit the possibility that strong schemata be combined to form fitter offspring chromosomes.

When Holland introduced genetic algorithms and began his experiments on bit strings, he used a simple *one-point crossover* operator [9]. It exchanges the pieces of the parent chromosomes on each side of a randomly chosen crossover point. In two 10-bit chromosomes, for example, if the crossover point is between the fifth and sixth bits, the first offspring inherits the first through fifth bits of the first parent and the sixth through tenth bits of the second parent. The second offspring receives the remaining halves of the parents. This process closely resembles genetic recombination in real DNA.

One-point crossover is useful in some simple cases, but it is usually inadequate in combining chromosomes' pieces in a meaningful way [3]. *Two-point crossover* exchanges bits around two

crossover points, allowing a greater number of possible schemata to be combined. *Uniform crossover* creates a template of random bits to dictate which parent bits transfer to which offspring bits [19]. Syswerda believes that the added capability of uniform crossover to join schemata located anywhere in parent chromosomes overshadows its threat of destroying meaningful patterns [19].

3.4.2 Mutation Operators

While crossover operators attempt to combine already examined good solutions, mutation operators introduce diversity into the population and allow the search to consider new solutions that the crossover operators alone might not be able to create. Crossover can only lead to the construction of a limited set of chromosomes based on the parents. By applying a small mutation to a chromosome, new areas of the search space can be explored.

The most common type of mutation with bit string encodings is simple *bit mutation* in which each bit in a parent is changed to a random one or zero if a probability test is passed. The probability test is based on a mutation rate parameter that is usually quite low [15, 3]. If chromosomes contain few bits, it is likely that the mutation operator will leave many whole chromosomes unchanged.

3.4.3 Operators for Order-Based Chromosomes

The traditional crossover and mutation operators discussed above are intended for use with bit strings. When a permutation encodes solutions to a problem, performing crossover and mutation becomes more complex, as offspring chromosomes must also be valid permutations. For example, if a one-point crossover were performed on the chromosomes (4 1 2 6 3 5) and (5 2 4 3 6 1) with the crossover point between the fourth and fifth digits, the offspring would be (4 1 2 6 6 1) and (5 2 4 3 3 5). Clearly, these are both malformed permutations. A mutation operator must also be adapted for permutations, as random changes are likely to produce permutations with duplicated or absent numbers.

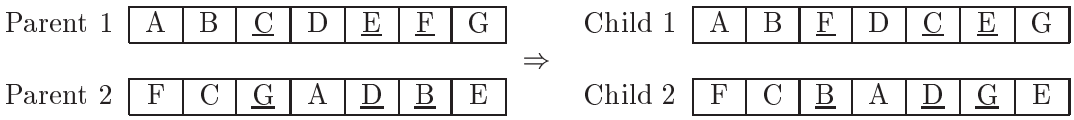

Figure 3.1: Order-Based Crossover example in which underlining denotes selected positions

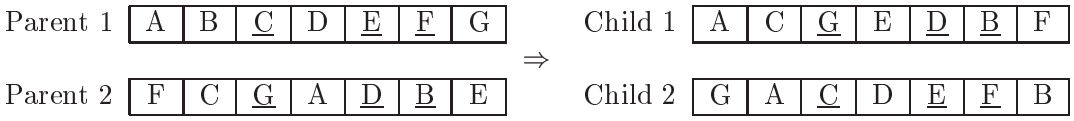

Figure 3.2: Position-Based Crossover example in which underlining denotes selected positions

Crossover Operators for Permutations

Permutation encodings demand specialized crossover operators that maintain the validity of the chromosomes as permutations, while combining information from both parents [3].

In *order-based crossover*, the order of elements in selected positions of one parent is imposed on the elements in the same positions of the other parent [19]. We see in Figure 3.1 that Parent 1 has C, E, and F in underlined, selected positions, and they are rearranged in Child 1 based on their relative ordering in Parent 2: F, C, E. In Parent 2, the corresponding elements are G, D, and B, reordered in Child 2 based on their order in Parent 1: B, D, G.

In *position-based crossover*, the absolute positioning of elements in selected positions of one parent is imposed on the corresponding elements of the other parent [19]. Consider the example in Figure 3.2. Elements C, E, and F are swapped with G, D, and B, maintaining their absolute positions. The remaining unused elements are arranged in each child based on their order in the corresponding parent. After obtaining G, D, and B, Child 1 is missing A, C, E, and F, so these elements are placed in the empty slots in the order in which they appear in Parent 1.

Finally, *uniform order-based crossover* adapts bit-string uniform crossover for order-based chromosomes [3]. It operates with a binary template of random ones and zeros. Child 1 inherits the values in Parent 1 at the positions marked by ones in the template, and Child 2 inherits those marked by zeros. The elements missing from each offspring are inserted based on the order in which they appear in the opposite parent. The template maintains some of the parent chromosomes' position-

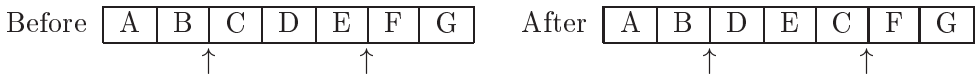

Figure 3.3: Scramble Sublist Mutation by permuting letters between second and fourth positions.

ing information, yet the inserting of the remaining elements transfers relative ordering information from one parent to the other [3].

For each of these crossover operators, a crossover rate parameter determines how much of a chromosome is affected by the operation. For order-based and position-based crossover, for example, this setting dictates how many positions are selected.

Mutation Operators for Permutations

Mutation can occur in permutations several ways. One method, known as *position-based mutation*, consists of removing an element from one position and inserting it into the chromosome at a different position, shifting other elements accordingly. Another approach, *order-based mutation*, simply swaps elements in two positions in a chromosome. An alternative approach that Davis finds superior in practice is known as *scramble sublist mutation* [3]. This technique creates a mutated child by permuting the elements in a contiguous subsection of a parent chromosome. It leaves the other elements as they appeared in the parent. See Figure 3.3 for an example.

For each of these mutation operators, a parameter regulates the degree of mutation. For the first two, this may dictate how many reinserts or swaps occur. For the third approach, the mutation rate may determine the size of the sublist to permute.

3.4.4 The NESCAC Operators

Several order-based operators are used to manipulate the permutation-based chromosomes in the NESCAC algorithm. When the reproduction process selects to use a crossover operator, either the order-based crossover operator or the position-based crossover operator is used. The choice of a specific operator is made at random. The same applies for mutation operators. Either a position-based, order-based, or scramble sublist mutation operator is randomly selected whenever the algorithm chooses to perform a mutation operation. Each has been shown to be effective in

some applications and in achieving its desired effect on the permutations themselves [3, 19]. Notice that, given the indirect nature of the encoding used here, it is difficult to see how changes made in the chromosomes by these operators will be reflected in the decoded schedule.

3.5 Parameter Values

Parameter values controlling the precise mechanics of the selection procedure and operators for a real-world optimization task must be fine-tuned to maximize the algorithm's effectiveness in finding good solutions quickly. Schaffer et al. notes that parameters, such as population size, crossover rate, and mutation rate, are well known to impact genetic algorithms' performance significantly, yet the theory behind this approach does little to suggest how to select their values [17]. It can be difficult to find robust parameter settings. Explaining further difficulties, Davis notes that, "Genetic algorithms are stochastic, and the same parameter settings used on the same problems by the same genetic algorithm generally yield different results [3]." Thus, finding the consistently best parameter values may require running many trials and could take as much work as solving the problem itself [3].

3.5.1 Population Size

Population size is one of the critical parameters of a genetic algorithm. If the population is too small, the algorithm may converge on a solution quickly, but not enough distinct schemata within the search space may have been considered. If a very large population is used, the algorithm may investigate a wide area of the search space, but significant improvements may require much time. It seems logical that a genetic algorithm with a larger population, though slower, will eventually reach better solutions. However, experience dictates that a population size depends greatly on the specific problem to be solved and how the other components of the algorithm are designed [19]. Well-designed operators, for example, may permit sufficient search space exploration when using a small population. In addition, while wary of drawing solid conclusions from limited empirical research, Goldberg believes that relatively small population sizes (30-200) work best with serial genetic algorithm implementations, while larger populations may work better when implementations utilize

parallelism across multiple processors [6]. Furthermore, Syswerda asserts that a fixed number of evaluations can be allocated to one run with a large population or several runs with a smaller population [19]. In applications like scheduling, where each run of the algorithm is likely to produce a very different answer, it may be advantageous to use the latter approach and simply use the best solution out of the several runs. This is precisely the technique applied here.

3.5.2 Crossover and Mutation Weights

Davis writes that, for any problem, there exists some ratio of crossover to mutation, as dictated by the crossover and mutation weights, that will produce the best results [3]. Crossover is generally used more frequently than mutation, as crossover is regarded as the fundamental cause of genetic algorithms' success, but an algorithm will not perform well if mutation is eliminated altogether [4]. In order to benefit from genetic recombination, yet also explore diverse segments of the search space, both types of operation must be used. The best ratio, though, appears to change throughout the course of a population's evolution. When the population is mostly random in the initial generations, crossover is necessary to pull together the best schemata out of these diverse individuals. When the population converges, mutation becomes more important, allowing the search to explore regions beyond the best solutions found so far. Davis has found that increasing the mutation probability while decreasing the crossover probability, while the algorithm runs, tends to improve performance [3]. Davis has also tested reactive mechanisms that adapt the operators' probabilities at each step based on the current performance of the operator's offspring [2]. For example, if the crossover operator were producing very fit offspring in one generation, then crossover would be used more in the next round of reproduction. However, if mutation began to lead to better solutions than crossover, mutation would be given a higher probability.

3.5.3 Methods for Finding Good Parameter Settings

Researchers have used several approaches to find good parameter values. De Jong's initial research used hand optimization to derive operator probabilities, while Grefenstette utilized a meta genetic algorithm to evolve the best parameter values for use in the genetic algorithm of interest [3]. In this

case, the meta algorithm's evaluation function measures the effectiveness of the latter algorithm using the parameter values encoded in the meta algorithm's chromosomes. Schaffer et al. used a brute-force approach to test every realistic combination of settings. They investigated parameter values for a set of function minimization tasks by performing a full factorial design with six different population sizes, ten crossover rates, seven mutation rates, and two types of crossover operators [17]. Finally, as discussed above, operator probabilities and other settings can be programmed to adjust themselves while the algorithm runs [3].

3.5.4 Parameters for the NESCAC Genetic Algorithm

The current algorithm has several parameters that control its behavior. The *population size* determines how many individuals are in the population at any time. The *elite percentage* (E) determines the number of top individuals to promote automatically each generation, and the *culled percentage* (C) is used to decide how many individuals at the bottom of the population are ignored when selection occurs. The *mutation weight* (W) gives the probability that the mutation operator is chosen during reproduction. The probability that crossover occurs is simply one minus the mutation weight. The *mutation rate* (M) sets the probability that any number in a chromosomes' two permutations is modified, and the *crossover rate* (X) dictates the percentage of a chromosome that is crossed over. Finally, the *number of generations* determines how many times the entire evolutionary process occurs.

Chapter 4

Experimental Results and Discussion

In this chapter, we examine the success of the genetic algorithm in achieving its goal: finding desirable NESCAC soccer schedules in a relatively short period of time. First, results of a series of tests conducted to determine the best parameter settings are presented. We then consider the performance of the algorithm when using the best and worst parameter configurations and compare these results to a completely random search. Finally, the desirability and optimality of the best schedule found in these tests is discussed.

4.1 Implementation and Specific Constraints Used

The genetic algorithm for the NESCAC problem was implemented in Java and run on a Linux workstation with dual Intel® XEON™ 2.4 GHz processors. While Constraints 1, 2, 3, 4, and 5 are consistent from year to year, the specifics of the remaining constraints vary and must be provided as input to the algorithm. The home games required by Constraint 6 are different each season. The rivalries relevant to Constraints 7 and 10 may also fluctuate. Finally, Constraints 8 and 9 require that a travel grid be input, as travel routes and field locations may change. See Appendix A for the current travel data. These constraints are automatically read from files and parsed by the program for use by the schedule-builder and evaluation function.

Several discussions with NESCAC Administrative Director Andrea Savage revealed a set of constraints for the 2003 men's soccer season that were used in all test runs of the algorithm. Since only the homecoming dates for Amherst (week five), Williams (week eight), and Wesleyan (week

seven) follow a known pattern from year to year, and next season's requests are still unknown, two other required home dates were added to make the set of constraints more realistic: Bates (week four) and Tufts (week five). Current rival triplets are Amherst, Williams, and Wesleyan, and Bates, Bowdoin, and Colby. For week preferences, Amherst must play Williams between weeks seven and nine, and all other rival games must occur in the second half of the season, weeks five through nine.

4.2 Finding the Best Parameters

The implemented software requires that the parameters discussed in Section 3.5.4 be provided as input. A full factorial design with five of these parameters, two with three possible values and three with two possible values, was used to find the combination of parameter values that lead the algorithm to the best schedules. Table 4.1 lists the parameters and values that were tested. There are 72 ways these parameter values can be combined, producing 72 *configurations*. Two sets of tests were conducted in which each possible configuration was used as input to the algorithm. Both used the relatively small population size of 50, based on Goldberg [6], Syswerda [19], Schaffer et al. [17]. In Test Suite 1, each run lasted 100 generations. To see if the influence of each parameter changes over the course of the algorithm, the runs in Test Suite 2 spanned 200 generations. To counterbalance the stochastic nature of the algorithm, each configuration was run 100 times, and analyses considered mean fitness values over all replications of a given configuration. As the high degree of randomness in a genetic algorithm can cause a good combination of parameter settings to lead occasionally to poor results, considering the average best fitnesses allows us to observe the

Parameter	**Values Tested**	c_1**: Best**	c_2**: Worst**	c_3**: Random Search**
E: Elite Percentage	0.2, 0.4	0.2	0.4	0.02
C: Culled Percentage	0.2, 0.4	0.4	0.2	0.0
W: Mutation Weight	0.2, 0.4	0.2	0.4	1.0
X: Crossover Rate	0.2, 0.4, 0.6	0.2	0.6	0.0
M: Mutation Rate	0.1, 0.3, 0.5	0.1	0.5	1.0

Table 4.1: Parameters and values tested. Configurations c_1 and c_2: parameter values found to lead to the best and worst behavior in Test Suites 1 and 2. Configuration c_3: parameter values that cause the algorithm to behave like a random search.

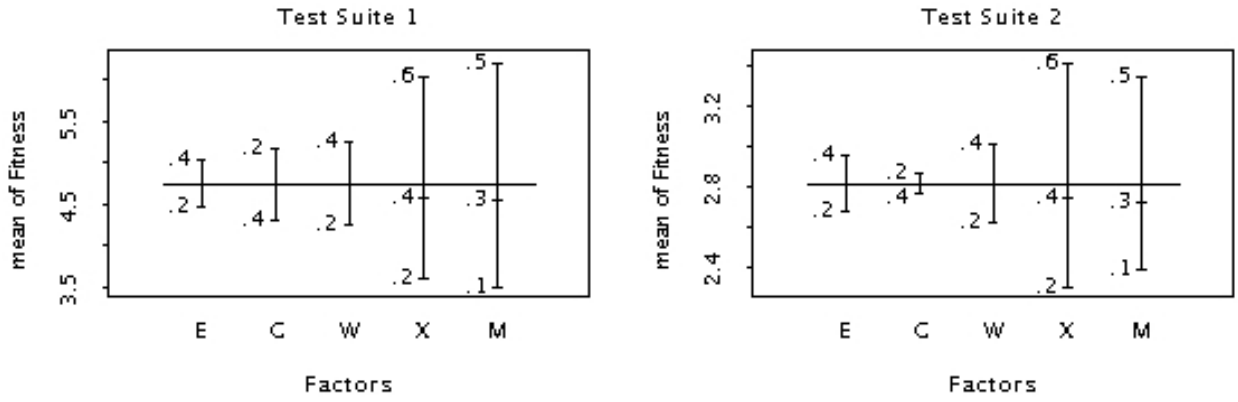

Figure 4.1: Sample means for the 72 parameter configurations run in Test Suites 1 and 2, with the fitness for each marginal parameter setting averaged over 100 trials each lasting 100 and 200 generations, respectively.

general trend of behavior with a given configuration.

Figure 4.1 presents the sample means for each factor in each test suite and clearly shows which parameter values tend to produce lower fitness scores. For example, in Test Suite 1, 3600 runs used an elite percentage of 0.2 (100 replications of half of the configurations) and ended with a mean fitness of 4.4515. The other 3600 runs used an elite percentage of 0.4 and ended with a mean fitness of 5.0272. This suggests that a value of 0.2 for the elite percentage should be preferred over 0.4. To examine more precisely the statistical significance in the observed sample means, an ANOVA analysis was performed on the each of test suites' findings. These results are provided in Appendix B. The ANOVA run on Test Suite 1 reveals that all five sample mean differences are statistically significant. With $P \ll 0.001$ for each factor, the likelihood that these differences appeared by chance is very small. For Test Suite 2, the same preferred values are observed for each parameter, and four of the five mean differences are statistically significant, as their P values are well below 0.001. Only the difference in sample means for the culled percentage is not statistically significant in this second set of tests. This suggests that this parameter has a greater impact in earlier generations, but as the population fills up with primarily good solutions over more generations, a lower culled percentage may help stave off the elimination of good solutions that may contribute to even better solutions. While one value is not statistically better than the other, a culled percentage of 0.4

led to finding a lower mean fitness in both test suites. Thus this value and the other four better values are accepted as preferable. Though even worse values might exist, the values that led to higher fitness values are considered undesirable in the remaining tests. Table 4.1 lists which of the parameter values tested were found to lead to the best and worst behavior in both test suites.

4.3 Performance Using the Best and Worst Parameters

Having investigated parameter settings and found preferable and undesirable values, a second round of testing was conducted to study the behavior of the algorithm over a longer number of generations. Using the best and worst parameter settings listed in Table 4.1, the algorithm was run for 1000 generations with a population size of 50. End-users would probably only run the program a few times in search of a schedule, but 100 trials were run in order to discover a general trend and counteract the stochastic nature of the algorithm. It was hoped that highly evolved and desirable schedules would emerge out of the 100 trials with configuration c_1. At the same time, the tests sought to further illustrate the negative impact of configuration c_2. Figure 4.2 shows the change in mean fitness across generations for the algorithm run with c_1 and c_2. While it is likely that some trials using c_2 ended with a very low mean fitness, it is clear that, averaged over 100 trials lasting 1000 generations, the preferable parameter values, c_1, continue to lead to better solutions than the undesirable settings, c_2. The mean fitness at each generation is lower for c_1, and the mean final fitness for the runs using c_1 is 1.5331, versus 1.6721 for those with c_2.

4.4 Better Than Random Search?

One issue that often arises in genetic algorithms research is that of demonstrating an algorithm's superiority to a completely random search, one in which many solutions are considered at random until a sufficiently good one is found. To compare the effectiveness of the current algorithm to that of a purely random search, the algorithm was run using the parameter configuration c_3 in Table 4.1. These mutation weight and mutation rate settings dictate that, at each reproductive step, only mutation occurs, and each chromosome is completely rearranged. Thus each successive

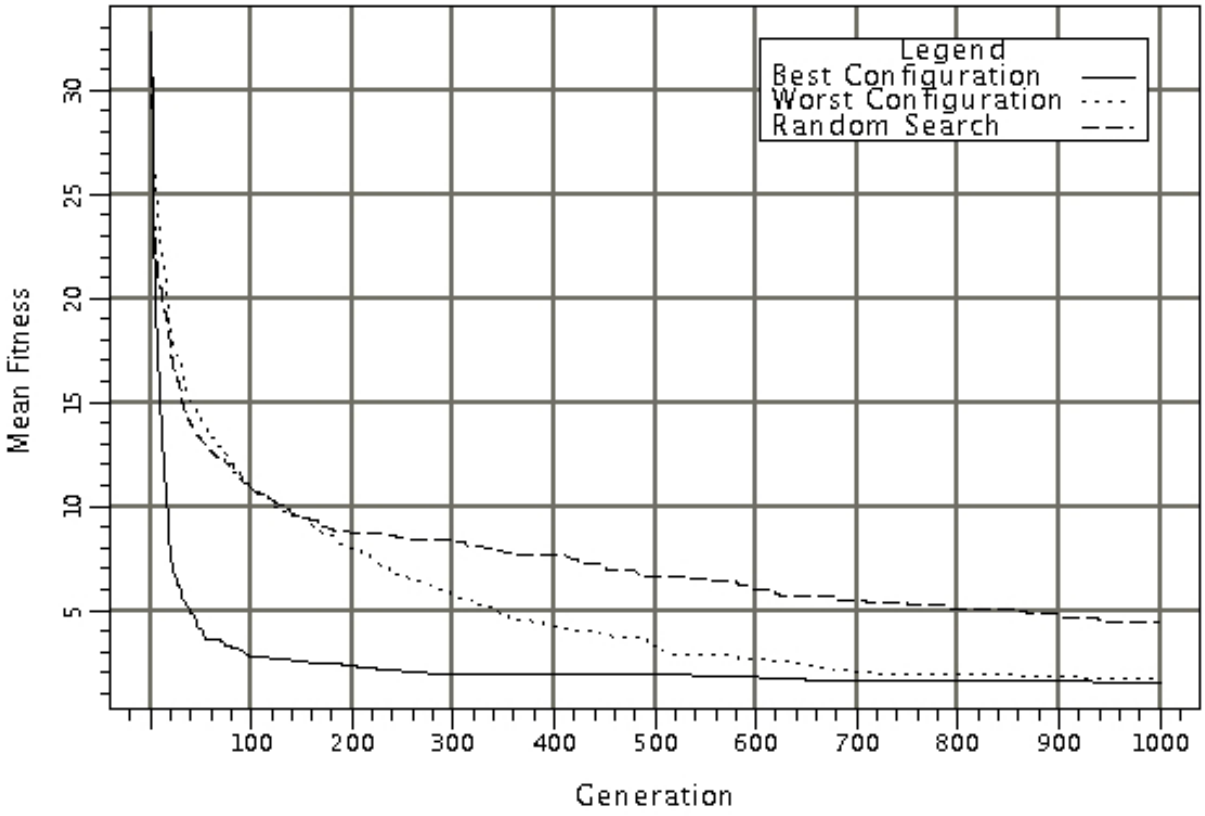

Figure 4.2: Means of the best fitnesses obtained in each generation over 100 trials.

population consists of entirely random, new individuals. The crossover rate and culled percentage are irrelevant. In order to keep track of the single best solution found at any point in this random search, an elite percentage of 0.02 is used with a population of 100. With c_3, the algorithm simply considers a set of unique, random chromosomes in each generation and saves the best individual found so far. Figure 4.2 compares the behavior of this configuration, averaged over 100 trials, to that of the two test configurations discussed in the previous section. Clearly, a random search takes considerably more time to find solutions with low fitness values than the genetic search, with either good or bad parameter settings, does. After 1000 generations, the mean fitness of the best schedules found with configuration c_3 had a fitness of 4.3822, whereas configurations c_1 and c_2 ended with

mean fitnesses of 1.5331 and 1.6721, respectively. Note that the average run-time for each set of 100 trials was 1151.76 user seconds (about 19.20 minutes or 0.3199 hours). An individual run of the algorithm thus took only about 11.52 user seconds.

4.5 Example Highly Desirable Schedule

Out of all trial runs using c_1, the best schedule found has a fitness value of 0.7925, signifying that all constraints but the time balancing constraint are fully satisfied. This schedule, presented in Table 4.2, is considered highly desirable and was presented to the NESCAC administrators for consideration. At this time, they have not yet provided feedback.

4.5.1 Constraint Satisfaction

This is a highly desirable schedule in terms of the constraints outlined by the NESCAC administrators. (See Sections 1.2 and 4.1.) Recall that some constraints are strictly enforced by the schedule-builder and therefore could never be violated. Violations of the remaining constraints, except Constraint 8 regarding travel time, are penalized with a factor of 10 in the evaluation function. Therefore, with a fitness value of less than 10, this schedule cannot possibly violate any constraint other than Constraint 8. Clearly, each team plays each other team once within an eight week season consisting of eight weekend games and one mid-week game. Each team only plays one game at any single time. All teams play either four home games and five away games or five home games and four away games. No team ever plays more than two consecutive away games. Teams requiring home games on certain dates are scheduled to play at home, and all rival match-ups that must occur during a certain part of the season do so. For example, traditionally a huge rivalry, the Amherst versus Williams game is scheduled at Amherst in Week 7, Amherst's homecoming date. The two rival triplet conditions are also satisfied, and all the games involved take place in the second half of the season. For example, Amherst plays at Wesleyan in week 6, Wesleyan plays at Williams in week 5, and Williams plays at Amherst in week 7. Two of the mid-week games require teams to travel less than two hours: Trinity at Amherst (1.25 hours) and Connecticut College at Wesleyan (.75 hours). These games can be scheduled by the two competing teams for any available

	Amh	**Bat**	**Bow**	**Col**	**Con**	**Mid**	**Tri**	**Tuf**	**Wes**	**Wil**
Week 1	Tuf	@Con	@Wes	Tri	Bat	Wil	@Col	@Amh	Bow	@Mid
Week 2	@Mid	Wes	Tuf	@Wil	@Tri	Amh	Con	@Bow	@Bat	Col
Week 3	Bat	@Amh	@Tri	@Con	Col	@Wes	Bow	Wil	Mid	@Tuf
Week 4	@Con	Tri	@Wil	Wes	Amh	Tuf	@Bat	@Mid	@Col	Bow
Week 5	Col	@Bow	Bat	@Amh	@Mid	Con	@Tuf	Tri	@Wil	Wes
Week 6	@Wes	@Mid	@Col	Bow	Tuf	Bat	Wil	@Con	Amh	@Tri
Week 7	Wil	Col	Con	@Bat	@Bow	@Tri	Mid	@Wes	Tuf	@Amh
Week 8	@Bow	@Tuf	Amh	Mid	@Wil	@Col	Wes	Bat	@Tri	Con
Mid-week	Tri	Wil	Mid	@Tuf	@Wes	@Bow	@Amh	Col	Con	@Bat
avg_i	11.875	15.625	15.25	18.125	13.375	19.25	12.375	11.75	13.0	15.125
act_i	10.5	16.25	14.75	19.25	14	19	12.75	12.5	12.75	14
$\lvert avg_i - act_i \rvert$	1.375	0.625	0.5	1.125	0.625	0.25	0.375	0.75	0.25	1.125

Table 4.2: The best schedule found using configuration c_1 (fitness = 0.7925). The "@" sign denotes away games.

weekday. The other three mid-week games can be played during the same weekends as another nearby away games. For example, Colby is scheduled to play at Tufts in a non-weekend slot, but 3.25 hours is too far to travel on a weekday. This game can be scheduled for the Sunday following Colby's game at Amherst, though, since Tufts and Amherst are only 2 hours apart.

Since all of the other constraints are satisfied, the requirement of balancing travel time is the determinant of this schedule's fitness value. Below the timetable of games, Table 4.2 lists each team's theoretical average travel time (in hours), the actual travel time required in this schedule, and the difference between these two measures. (See Section 3.2.2.) The sum of the squared differences is equal to 6.28125. Plugging this number into Equation 3.1 yields a value of 0.7925 (rounded to four decimal places). It is plain to see that, in addition to satisfying all other constraints, this schedule requires each team to travel an amount very close to its own theoretical average, thus making it a highly desirable schedule.

4.5.2 Is This Schedule Optimal?

Since the best possible schedule is unknown, there really is no way of knowing if a better possible schedule may be found in future tests. Through all of the tests conducted, this schedule best fulfills the constraints that were included in the schedule-builder and evaluation function. The only room for improvement lies in further balancing travel times. Based on theoretical average travel times, r_1 in the evaluation function can never be exactly 0, but it may be possible to find a schedule which reduces this value further. Enacting a greater balance in travel time, however, might lead to violating one of the other constraints. Thus a small r_1 value might be outweighed by another rating in the evaluation function. Because of this difficulty in calculating the lowest possible evaluation value, we can only conclude that this schedule is the best found over several hundred trials using fine-tuned parameters, and it is therefore "near optimal" in terms of the constraints used here.

An important question remains, though: Is this really close to the best schedule in the eyes of the NESCAC administrators? Scheduling a perfect season, one that will succeed in pleasing all players, coaches, and college administrators alike, generating the most revenue in ticket sales and media coverage, and creating the fairest competition possible, is a task that can only be performed

by experienced scheduling experts [13]. Many of the heuristics employed may be difficult to grasp and convey to a researcher who is trying to encode them in a computer algorithm. Therefore, while the schedule presented may appear highly desirable to an untrained eye, someone who has been working with the same sports league for many years may immediately be able to spot a severe problem due to an overlooked constraint. Thus the most we may conclude is that the algorithm succeeds in finding what appears to be a desirable solution to the problem, as internally defined by the schedule-builder and evaluation function. In practice, though, this and other similar schedules may require some manual adjustment before it is usable by the NESCAC. Section 5.2 discusses ways to make the current scheduling process more realistic.

Chapter 5

Topics for Further Research and Concluding Remarks

5.1 Algorithmic Enhancements

The algorithm developed here has demonstrated its ability to evolve highly desirable and near optimal sports schedules for the NESCAC men's soccer season. Further research into the design of each piece of the algorithm may lead to finding even better schedules. However, since the algorithm is only needed to run once or a few times each year, improving speed is not a high priority.

5.1.1 Representational Scheme

A more direct representation might allow for more beneficial recombination and manipulation of genetic material, leading to the creation of better schedules. With the current encoding scheme, a chromosome is fairly removed from an actual schedule, making it near impossible to predict a change in a decoded schedule based on a change in a chromosome. A more natural encoding, similar in structure and appearance to a real sports schedule, could permit local changes and manipulation of specific games. For example, a schedule might be very good except for the fact that a few of its away games violate home game requirements. If it were possible, at the chromosomal level, to change these games to home games and propagate any other necessary changes throughout the schedule, desirable schedules could be evolved more quickly. This is what Costa has done in his work to schedule the National Hockey League using a memetic algorithm that combines the

evolutionary, parallel nature of a genetic algorithm with the slightly more intelligent methods of tabu search [1]. With the current encoding using chromosomes that must be decoded to form real schedules, it would be impossible to design operators to carry out a tabu search in place of mutation at each reproductive step.

5.1.2 Evaluation Function

Further research could also investigate building an evaluation function that gauges a more precise level of schedule desirability. Further collaboration with the individuals who currently schedule the NESCAC men's soccer season may result in discovering new constraints and factors. Also, this could result in being able to fine-tune the existing factors f_2 through f_6, thus leading to faster convergence to a highly desirable solution.

5.1.3 Selection Procedure

As discussed in Section 3.3.1, alternative approaches to the current tournament selection procedure exist. A roulette wheel parent selection algorithm could be integrated with the rest of the current implementation to give chromosomes a probability of selection proportional to their fitness. Currently, the fitness value (equal to the output of the evaluation function) is being minimized. Using a proportional selection method would require converting small evaluation values into large fitness values, so better schedules would be selected with higher probability.

5.1.4 Operators

The current operators are completely blind to the meaning of the chromosomes upon which they act and the details of the problem to be solved. As mentioned above, with a more direct representation, there is the potential for customized operators to enact problem-specific changes on a given solution. A third operator could also be introduced which performs a type of hill-climbing operation to try to move from a current solution to one slightly different and hopefully slightly better. Designing such a problem-specific operator would require great care in preventing infeasible solutions from emerging.

5.1.5 Parameter Values

Further research could explore a wider range of values for each parameter setting and study the interactions between the various parameter values. For example, a mutation weight of .2 may work best with a mutation rate of .8, whereas these values may not lead to very good behavior when used independently. More substantial statistical analyses on the current data or another round of testing with a larger array of potential values may reveal a configuration that will surpass the best one found here. In addition, other methods of finding parameter values, such as installing an adaptive mechanism or running a meta genetic algorithm to evolve a set of values, could possibly lead to finding better configurations.

5.2 Making the System More Realistic and Interactive

The algorithm presented here is able to find very good schedules for the NESCAC men's soccer season independently of other sports teams' schedules and without regard for past years' schedules. The real scheduling problem is not this simple. College sports conferences often schedule several sports simultaneously to ensure field availability, coordinate busing, and attract large numbers of fans, for example [16]. NESCAC often attempts to base the schedule of fall sports teams like men's soccer on that of the football team because larger crowds will come out for a home soccer game if the football team is playing home the same day. Similarly, if fans travel a long distance to see Amherst play Trinity, for example, in football, it would be nice if Amherst were playing at Trinity in other sports on the same day. Scheduling also usually takes into account past schedules. One year's schedule should not be too similar to the last, as no team should have to repeat the same string of historically strong opponents [16]. Also, teams generally prefer the last game of the season to be played at home. Reference to past seasons is necessary to balance out the fulfillment of this request between teams over multiple years.

Placing the men's soccer season in a greater context would make the scheduling algorithm more applicable to the real-world problem. Simultaneous scheduling of multiple sports could be achieved using the current algorithm by expanding the chromosomes to represent several sports' schedules

and by implementing constraints that apply to the set of schedules as a whole. Several sport-specific schedule-builders could decode each pair of permutations, and a set of evaluation functions could assess the decoded schedules based on the specific requirements for each sport. A meta evaluation function could then assess the schedules in relation to one another. For example, one rating in this function might count the number of times a school's football team plays at home, but the soccer team is away. Minimizing this and other similar ratings, while also minimizing ratings for individual schedules, would produce a set of good schedules that favorably interact. Furthermore, to incorporate historic context, an additional rating could be added to each individual evaluation function that judges the similarity between the current schedule and previous years' schedules for each sport. Such a rating could consider teams' rankings in past years to determine which series of games should not be repeated.

Since an automated scheduling program should be easy for a human scheduler to operate, future work could also explore making the system more interactive and user-friendly. A graphical user interface, including the ability to alter the algorithm's settings while it runs, would be useful. It would be helpful if the user could view each of the schedules in the population at any given time to ensure that the search is heading toward a schedule that is likely to be accepted. Subtleties in certain schedules could nullify their strong evaluation and render them useless in reality. An evaluation function may not be able to incorporate these intangible qualities that are easily observed by a human expert. Thus, while some bad chromosomes are needed for diversity, a user's ability to interact with the algorithm may avert searches that would end in unusable schedules. The exact amount of interaction allowed would have to be investigated. One possibility would be to allow the user to designate a small number of schedules as elite, regardless of their fitness values. This would allow some level of human guidance without defeating the purpose of the evolutionary mechanisms.

5.3 Extending the Algorithm

While the current algorithm was designed for the NESCAC men's soccer season, it could be used in other contexts. Aside from the decoding procedure and evaluation function, the algorithm's

genetic components are problem-neutral. The permutation-based representation, the selection and reproductive procedure, the crossover and mutation operators, and the parameter settings lack any specific reference to the NESCAC problem or sports scheduling. The algorithm should scale well to accommodate larger sports leagues with different constraints. Given a permutation of any number of team identifiers, the game generating routine can create all of the necessary match-ups for a complete season. Only the evaluation function and the schedule-builder's prioritized rule-base to assign home teams would need to be reconstructed to incorporate a different league's preferences. To apply this general evolutionary framework to another problem domain besides scheduling, one would only need to design a new decoding process and evaluation function. This robust, portable nature might be weakened, however, if the improvements suggested above, which make the algorithm more problem-specific, are implemented.

5.4 Conclusion

The task of scheduling a sports league is complex, requiring a balance between many potentially conflicting interests and limited resources. The combinatorics of creating a timetable in which several teams must play each other once in a given time-frame and within certain constraints makes the problem one of exponentially increasing difficulty. The process may take hours by hand, and manually produced schedules often fail to fulfill all of the requirements [13]. A poorly designed sports schedule may result in dissatisfaction by players and coaches, poor attendance by fans, and low ticket revenue. Thus computer automated schedulers that perform this task better than human experts can be extremely valuable. This type of optimization task usually requires the use of heuristics to guide a search toward a desirable solution in a short amount of time. Genetic algorithms achieve this goal by applying the concept of evolution, apparently successful in nature, to solve problems by computer. With this "survival of the fittest" approach, the only knowledge needed from a problem domain is the ability to distinguish good solutions from bad solutions. Examining whole populations of evolving solutions in parallel, genetic algorithms can provide a more efficient search for a near optimal schedule than a human scheduler. A permutation-based

genetic algorithm using a problem-specific scheduler-builder has demonstrated its ability to "evolve" highly desirable schedules for the NESCAC men's soccer season. With carefully selected parameter values, the algorithm regularly finds schedules that satisfy all of the league administrators' specified requirements, including a near optimal balance between each team's amount of travel. Further constraints and considerations may need to be incorporated to find schedules that are truly usable by the league; however, this work suggests that evolutionary mechanisms can effectively find solutions to the problem of scheduling a highly constrained sports season.

Bibliography

[1] Daniel Costa. An evolutionary tabu search algorithm and the NHL scheduling problem. *INFOR*, 33(3):161–178, 1995.

[2] Lawrence Davis. Adapting operator probabilities in genetic algorithms. In J. David Schaffer, editor, *Proceedings of the 3rd International Conference on Genetic Algorithms*, pages 61–69, George Mason University, June 1989. Morgan Kaufmann.

[3] Lawrence Davis. A genetic algorithms tutorial. In Lawrence Davis, editor, *Handbook of Genetic Algorithms*, pages 1–101. Van Nostrand Reinhold, New York, 1991.

[4] Emanuel Falkenauer. Applying genetic algorithms to real-world problems. In Lawrence David Davis, Kenneth De Jong, Michael D. Vose, and L. Darrell Whitley, editors, *Evolutionary Algorithms*, pages 65–88. Springer, New York, 1999.

[5] Emanuel Falkenauer and Alain Delchambre. A genetic algorithm for bin packing and line balancing. In *Proceedings of the IEEE 1992 International Conference on Robotics and Automation*, pages 1186–1192, Nice, France, 1992.

[6] David E. Goldberg. Zen and the art of genetic algorithms. In J. David Schaffer, editor, *Proceedings of the 3rd International Conference on Genetic Algorithms*, pages 80–85, George Mason University, June 1989. Morgan Kaufmann.

[7] John J. Greffenstette and James E. Baker. How genetic algorithms work: A critical look at implicit parallelism. In J. David Schaffer, editor, *Proceedings of the 3rd International*

Conference on Genetic Algorithms, pages 20–27, George Mason University, June 1989. Morgan Kaufmann.

[8] Jean-Philippe Hamiez and Jin-Kao Hao. Solving the sports league scheduling problem with tabu search. *Lecture Notes in Computer Science*, 2148:24–36, 2001.

[9] John H. Holland. *Adaptation in Natural Artificial Systems.* University of Michigan Press, Ann Arbor, 1975.

[10] Phil Husbands. Genetic algorithms for scheduling. *AISB Quarterly*, 89., 1994.

[11] Jason Leonard. Interactive game scheduling with genetic algorithms. Master's thesis, Royal Melbourne Institute of Technology University, Melbourne, Australia, 1998.

[12] Zbigniew Michalewicz. The significance of the evaluation function in evolutionary algorithms. In Lawrence David Davis, Kenneth De Jong, Michael D. Vose, and L. Darrell Whitley, editors, *Evolutionary Algorithms*, pages 151–166. Springer, New York, 1999.

[13] George L. Nemhauser and Michael A. Trick. Scheduling a major college basketball conference. *Operations Research*, 46:1–8, 1998.

[14] Jackie Rees and Gary J. Koehler. An investigation of GA performance results for different cardinality alphabets. In Lawrence David Davis, Kenneth De Jong, Michael D. Vose, and L. Darrell Whitley, editors, *Evolutionary Algorithms*, pages 191–206. Springer, New York, 1999.

[15] Colin R. Reeves. *Modern Heuristic Techniques for Combinatorial Problems.* John Wiley & Sons, New York, NY., 1995.

[16] Andrea Savage. NESCAC administrative director. Personal communication, 2002-3.

[17] J. David Schaffer, Richard A. Caruana, Larry J. Eshelman, and Rajarshi Das. A study of control parameters affecting online performance of genetic algorithms for function optimiza-

tion. In J. David Schaffer, editor, *Proceedings of the 3rd International Conference on Genetic Algorithms*, pages 51–60, George Mason University, June 1989. Morgan Kaufmann.

[18] Kathleen Steinhöfel, Andreas Albrecht, and Chak-Kuen Wong. Convergence analysis of simulated annealing-based algorithms solving flow shop scheduling problems. *Lecture Notes in Computer Science*, 1767:277–290, 2000.

[19] Gilbert Syswerda. Schedule optimization using genetic algorithms. In Lawrence Davis, editor, *Handbook of Genetic Algorithms*, pages 332–349. Van Nostrand Reinhold, New York, 1991.

Appendix A

NESCAC Travel Time Grid

The following grid contains the number of hours required to travel between every pair of teams in the NESCAC men's soccer league [16]. The bottom row contains each team's theoretical average travel time, computed by dividing the sum of each column in half, since about half of each team's games will be played away. School abbreviations are listed below.

	Amh	**Bat**	**Bow**	**Col**	**Con**	**Mid**	**Tri**	**Tuf**	**Wes**	**Wil**
Amh		4.0	3.75	4.5	2.0	3.25	1.25	2.0	1.5	1.5
Bat	4.0		0.5	1.0	4.0	5.0	4.5	2.75	4.5	5.0
Bow	3.75	0.5		1.0	4.0	5.0	4.25	2.5	4.5	5.0
Col	4.5	1.0	1.0		4.75	5.75	5.0	3.25	5.25	5.75
Con	2.0	4.0	4.0	4.75		5.25	1.0	2.0	0.75	3.0
Mid	3.25	5.0	5.0	5.75	5.25		4.0	3.75	4.25	2.25
Tri	1.25	4.5	4.25	5.0	1.0	4.0		2.0	0.5	2.25
Tuf	2.0	2.75	2.5	3.25	2.0	3.75	2.0		2.25	3.0
Wes	1.5	4.5	4.5	5.25	0.75	4.25	0.5	2.25		2.5
Wil	1.5	5.0	5.0	5.75	3.0	2.25	2.25	3.0	2.5	
Average	11.875	15.625	15.25	18.125	13.375	19.25	12.375	11.75	13.0	15.125

Amh: Amherst College
Bat: Bates College
Bow: Bowdoin College
Col: Colby College
Con: Connecticut College
Mid: Middlebury College
Tri: Trinity College
Tuf: Tufts University
Wes: Wesleyan University
Wil: Williams College

Appendix B

Parameter Test Suite ANOVA Results

The following test results were obtained using S-PLUS Version 6.1.2 Release 2 for Linux 2.2.12.

Analysis of Variance Model for Test Suite 1

```
Short Output:
Call:
aov(formula = Fitness ~ E + C + W + X + M, data = test1.df, qr = T,
    na.action = na.exclude)

Terms:
                        E        C        W        X        M Residuals
Sum of Squares      596.5   1311.9   1770.1   7229.7   8818.1  133958.3
Deg. of Freedom         1        1        1        2        2      7192

Residual standard error: 4.315787
Estimated effects are balanced

            Df Sum of Sq  Mean Sq  F Value        Pr(F)
         E   1     596.5  596.504  32.0253 1.580303e-08
         C   1    1311.9 1311.855  70.4313 0.000000e+00
         W   1    1770.1 1770.057  95.0314 0.000000e+00
         X   2    7229.7 3614.846 194.0751 0.000000e+00
         M   2    8818.1 4409.067 236.7154 0.000000e+00
Residuals 7192  133958.3   18.626

Tables of means
 Grand mean
 4.7394
 E
     .2     .4
 4.4515 5.0272
 C
     .2     .4
 5.1662 4.3125
```

```
 W
     .2     .4
 4.2436 5.2352
 X
     .2     .4     .6
 3.6066 4.5683 6.0432
 M
     .1     .3     .5
 3.4949 4.5397 6.1835

Standard errors for differences of means
                 E          C          W          X          M
           0.10172    0.10172    0.10172    0.12459    0.12459
replic. 3600.00000 3600.00000 3600.00000 2400.00000 2400.00000
```

Analysis of Variance Model for Test Suite 2

```
Short Output:
Call:
aov(formula = Fitness ~ E + C + W + X + M, data = test2.df, qr = T,
    na.action  = na.exclude)

Terms:
                        E        C        W        X        M Residuals
 Sum of Squares    145.84    18.90   264.29  1486.14  1138.30  70028.75
Deg. of Freedom         1        1        1        2        2      7192

Residual standard error: 3.120422
Estimated effects are balanced

            Df Sum of Sq  Mean Sq  F Value     Pr(F)
        E    1    145.84 145.8429 14.97816 0.0001097
        C    1     18.90  18.8970  1.94073 0.1636323
        W    1    264.29 264.2912 27.14288 0.0000002
        X    2   1486.14 743.0701 76.31380 0.0000000
        M    2   1138.30 569.1502 58.45210 0.0000000
Residuals 7192  70028.75   9.7370

Tables of means
 Grand mean
 2.8147
 E
     .2     .4
 2.6724 2.9570
 C
     .2     .4
 2.8659 2.7634
 W
     .2     .4
 2.6231 3.0063
 X
     .2     .4     .6
 2.3006 2.7380 3.4055
 M
     .1     .3     .5
 2.3826 2.7191 3.3424

Standard errors for differences of means
                 E          C          W          X          M
        7.3549e-02 7.3549e-02 7.3549e-02 9.0079e-02 9.0079e-02
replic. 3.6000e+03 3.6000e+03 3.6000e+03 2.4000e+03 2.4000e+03
```