Distribution Category:
Mathematics and
Computer Science (UC-405)

ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, IL 60439

---

**ANL-94/23**

---

# A Parallel Genetic Algorithm for the Set Partitioning Problem

by

*David Levine*

Mathematics and Computer Science Division

May 1994

This work was supported by the Office of Scientific Computing, U.S. Department of Energy, under Contract W-31-109-Eng-38. It was submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science in the Graduate School of the Illinois Institute of Technology, May 1994 (thesis adviser: Dr. Tom Christopher).

# TABLE OF CONTENTS

LIST OF FIGURES

## LIST OF TABLES

| Table | | Page |
|---|---|---|

## LIST OF ABBREVIATIONS

| Abbreviation | Term |
|---|---|
| CPGA | Coarse-grained parallel genetic algorithm |
| FPGA | Fine-grained parallel genetic algorithm |
| GA | Genetic algorithm |
| GRGA | Generational replacement genetic algorithm |
| IMGA | Island model genetic algorithm |
| IP | Integer programming |
| LP | Linear programming |
| MIMD | Multiple-instruction multiple-data |
| OR | Operations research |
| PE | Processing element |
| PGA | Parallel genetic algorithm |
| SCP | Set covering problem |
| SIMD | Single-instruction multiple-data |
| SISD | Single-instruction single-data |
| SPP | Set partitioning problem |
| SSGA | Steady-state genetic algorithm |
| SUS | Stochastic Universal Selection |

## LIST OF SYMBOLS

| Symbol | Meaning |
|---|---|
| $a_{ij}$ | A binary coefficient of the set partitioning matrix. |
| $c_j$ | The cost coefficient of column $j$. |
| $f$ | The genetic algorithm evaluation function. |
| $i$ | A row (constraint) index. |
| $j$ | A column (variable) index. |
| $m$ | The number of rows (constraints) in the problem. |
| $n$ | The number of columns (constraints) in the problem. |
| $p_b$ | Probabilistic binary tournament selection parameter. |
| $p_c$ | Crossover probability. |
| $p_m$ | Mutation probability. |
| $p_u$ | Uniform crossover probability parameter. |
| $r_i$ | The set of columns such that $r_i \subseteq R_i$ and $x_j = 1$. |
| $\|r_i\|$ | The number of columns in the set $r_i$. |
| $t$ | A time index, usually the generation of the genetic algorithm. |
| $u$ | The genetic algorithm fitness function. |
| $x_j$ | A binary decision variable. |
| $\mathbf{x}$ | A vector of binary decision variables; also used as a bit string. |
| $z$ | The set partitioning objective function. |
| $B_i$ | The set of column indices that have their first one in row $i$. |
| $I$ | The set of row indices. |
| $J$ | The set of column indices. |
| $N$ | The genetic algorithm population size. |
| $P_j$ | The set of row indices that have a one in column $j$. |
| $\|P_j\|$ | The size of the set $P_j$. |
| $P_{AVG}$ | The average value of the $\|P_j\|$. |

| Symbol | Meaning |
|---|---|
| $P_{MAX}$ | The maximum value of the $\lvert P_j \rvert$. |
| $P(t)$ | The genetic algorithm population at time $t$. |
| $R_i$ | The set of column indices that have a one in row $i$. |
| $\lvert R_i \rvert$ | The size of the set $R_i$. |
| $R_{AVG}$ | The average value of the $\lvert R_i \rvert$. |
| $\lambda_i$ | Scalar multiplier of the evaluation function's penalty term. |
| $\Delta_j$ | The change in $z$ when complementing the value of column $j$. |
| $\Delta_{j1}$ | The change in $z$ when $x_j$ is set to one. |

## ABSTRACT

In this dissertation we report on our efforts to develop a parallel genetic algorithm and apply it to the solution of the set partitioning problem—a difficult combinatorial optimization problem used by many airlines as a mathematical model for flight crew scheduling. We developed a distributed steady-state genetic algorithm in conjunction with a specialized local search heuristic for solving the set partitioning problem. The genetic algorithm is based on an island model where multiple independent subpopulations each run a steady-state genetic algorithm on their own subpopulation and occasionally fit strings migrate between the subpopulations. Tests on forty real-world set partitioning problems were carried out on up to 128 nodes of an IBM SP1 parallel computer. We found that performance, as measured by the quality of the solution found and the iteration on which it was found, improved as additional subpopulations were added to the computation. With larger numbers of subpopulations the genetic algorithm was regularly able to find the optimal solution to problems having up to a few thousand integer variables. In two cases, high-quality integer feasible solutions were found for problems with 36,699 and 43,749 integer variables, respectively. A notable limitation we found was the difficulty solving problems with many constraints.

# CHAPTER I

# INTRODUCTION

In the past decade a number of new and interesting methods have been proposed for the solution of combinatorial optimization problems. These methods, such as genetic algorithms, neural networks, simulated annealing, and tabu search are based on analogies with physical or biological processes. During the same time period parallel computers have matured to the point where, at the high end, they are challenging the role of traditional vector supercomputers as the fastest computers in the world. On a different front, motivated primarily by significant economic considerations, but also by advances in computing and operations research technology, many major airlines have been exploring alternative methods for deciding how flight crews (pilots and flight attendants) should be assigned in order to satisfy flight schedules and minimize the associated crew costs. Our objective in this dissertation was to develop a parallel genetic algorithm and apply it to the solution of the set partitioning problem—a difficult combinatorial optimization problem that is used by many airlines as a mathematical model for assigning flight crews to flights.

This chapter introduces the major components of this work—the set partitioning problem, parallel computers, and genetic algorithms—and then discusses our goals. Chapter II describes the sequential genetic algorithm and local search heuristic used as the basis for the parallel genetic algorithm. Chapter III presents the parallel genetic algorithm and describes the computational experiments we performed. Chapter IV presents our conclusions. Chapter V suggests areas of further research.

The outline of this chapter is as follows. In the first section we describe the set partitioning problem. We give a mathematical statement of the problem, discuss its application to airline crew scheduling, and review previous solution approaches. The second section briefly discusses parallel computers. The third section describes genetic algorithms: their application to function optimization, previous approaches to constrained problems, and different parallel models. The last section discusses the motivation for pursuing this work and our specific goals.

## 1.1 The Set Partitioning Problem

**1.1.1 Mathematical Statement.** The set partitioning problem (SPP) may be stated mathematically as

$$\text{Minimize } z = \sum_{j=1}^{n} c_j x_j \tag{1.1}$$

subject to

$$\sum_{j=1}^{n} a_{ij} x_j = 1 \quad \text{for } i = 1, \ldots, m \tag{1.2}$$

$$x_j = 0 \text{ or } 1 \qquad \text{for } j = 1, \ldots, n, \tag{1.3}$$

where $a_{ij}$ is binary for all $i$ and $j$, and $c_j > 0$. The goal is to determine values for the binary variables $x_j$ that minimize the objective function $z$.

The following notation is common in the literature [24, 46]† and motivates the name "set partitioning problem." Let $I = \{1, \ldots, m\}$ be a set of row indices, $J = \{1, \ldots, n\}$ a set of column indices, and $P = \{P_1, \ldots, P_n\}$, where $P_j = \{i \in I | a_{ij} = 1\}$, $j \in J$. $P_j$ is the set of row indices that have a one in the $j$th column. $|P_j|$ is the cardinality of $P_j$. A set $J^* \subseteq J$ is called a *partition* if

$$\bigcup_{j \in J^*} P_j = I \tag{1.4}$$

$$j, k \in J^*, j \neq k \Rightarrow P_j \bigcap P_k = \emptyset. \tag{1.5}$$

Associated with any partition $J^*$ is a cost given by $\sum_{j \in J^*} c_j$. The objective of the SPP is to find the partition with minimal cost.

The following additional notation will be used later on. $R_i = \{j \in J | a_{ij} = 1\}$ is the (fixed) set of columns that intersect row $i$. $r_i = \{j \in R_i | x_j = 1\}$ is the (changing) set of columns that intersect row $i$ included in the current solution. $\Delta_{j_1}$ is the change in $z$ as a result of setting $x_j$ to one. $\Delta_j$ is the change in $z$ when complementing $x_j$. $\Delta_{j_1}$ and $\Delta_j$ measure both the cost coefficient, $c_j$, and the impact on constraint feasibility (see Section 2.4.3.)

**1.1.2 Applications.** Many applications of the SPP have been reported in the literature. A large number of these are scheduling problems where given a discrete, finite set of solutions, a set of constraints, and a cost function, one seeks the schedule that satisfies the constraints at minimum cost. A partial list of these applications includes crew scheduling, tanker routing, switching circuit design, assembly line balancing, capital equipment decisions, and location of offshore drilling platforms [6].

The best-known application of the SPP is airline crew scheduling. In this formulation each row ($i = 1, \ldots, m$) represents a flight leg (a takeoff and landing) that must be flown. The columns ($j = 1, \ldots, n$) represent legal round-trip rotations (pairings) that an airline crew *might* fly. Associated with each assignment of a crew to a particular flight leg is a cost, $c_j$.

The matrix elements $a_{ij}$ are defined by

$$a_{ij} = \begin{cases} 1 & \text{if flight leg } i \text{ is on rotation } j \\ 0 & \text{otherwise.} \end{cases} \tag{1.6}$$

†Numbers in square brackets refer to the numbered entries in the references.

Airline crew scheduling is a very visible and economically significant problem. The operations research (OR) literature contains numerous references to the airline crew scheduling problem [2, 3, 4, 7, 25, 36, 46, 47]. Estimates of over a billion dollars a year for pilot and flight attendant expenses have been reported [1, 7]. Even a small improvement over existing solutions can have a large economic benefit.

At one time solutions to the SPP were generated manually. However, airline crew scheduling problems have grown significantly in size and complexity. In 1981 problems with 400 rows and 30,000 columns were described as "very large" [47]. Today, problems with hundreds of thousands of columns are "very large," and one benchmark problem has been generated with 837 rows and 12,753,313 columns [9].

**1.1.3 Previous Algorithms.** Because of the widespread use of the SPP (and often the difficulty of its solution) a number of algorithms have been developed. These can be classified into two types: approximate algorithms which try to find "good" solutions quickly, and exact algorithms which attempt to solve the SPP to optimality.

An important approximate approach (as well as the starting point for most exact approaches) is to solve the linear programming (LP) relaxation of the SPP. In the LP relaxation, the integrality restriction on $x_j$ is relaxed, but the lower and upper bounds of zero and one are kept. A number of authors [7, 25, 47] have noted that for "small" SPP problems the solution to the LP relaxation is either all integer, in which case it is also the optimal integer solution, or has only a few fractional values that are easily resolved. However, in recent years it has been noted that as SPP problems grow in size, fractional solutions occur more frequently, and simply rounding or performing a "small" branch-and-bound tree search may not be effective [2, 7, 25].

Marsten [46] noted twenty years ago that for most algorithms in use at that time, solving the linear programming relaxation to the SPP was the computational bottleneck. This is because the LP relaxation is highly degenerate. The past several years have seen a number of advances in linear programming algorithms and the application of that technology to solving the LP relaxation of very large SPP problems [2, 9].

One of the oldest exact methods is implicit enumeration. In this method partial solutions are generated by taking the columns one at a time and exploring logical implications of their assignments. Both Garfinkel and Nemhauser [24] and Marsten [46] developed implicit enumeration algorithms. Another traditional method is the use of cutting planes (additional constraints) in conjunction with the simplex method. Balas and Padberg [6] note that cutting plane algorithms were moderately successful even while using general-purpose cuts and not taking advantage of the shape of the SPP polytope. A third method is column generation, where a specialized version of the simplex method produces a sequence of integer solutions that (one hopes) converge to the optimal integer solution. Applying a generic branch-and-bound program is also possible. Various bounding strategies have been used, including linear programming and Lagrangian relaxation. Fischer and Kedia [21] use continuous analogs of the greedy and $3-opt$ methods to provide improved lower bounds. Of recent interest is the work of Eckstein [20], who has developed a general-purpose mixed-integer programming system for use on the CM-5 parallel computer and applied it to, among other problems, set partitioning.

At the time of this writing the most successful approach appears to be the work of Hoffman and Padberg [36]. They present an exact approach based on the use of branch-and-cut—a branch-and-bound–like scheme where, however, additional preprocessing and constraint generation take place at each node in the search tree. An important component of their system is a high-quality linear programming package for solving the linear programming relaxations and a linear programming–based heuristic for getting good integer solutions quickly. They report optimal solutions for a large set of real-world SPP problems.

### 1.2 Parallel Computers

Traditionally, parallel computers are classified according to Flynn's taxonomy [22]. Flynn's classification distinguishes parallel computers according to the number of instruction streams and data operands being computed on simultaneously. There are three main classifications of interest: single-instruction single-data (SISD) computers, single-instruction multiple-data (SIMD) computers, and multiple-instruction multiple-data (MIMD) computers.

The SISD model is the traditional sequential computer. A single program counter fetches instructions from memory. The instructions are executed on *scalar* operands. There is no parallelism in this model.

In the SIMD model there is again a single program counter fetching instructions from memory. However, now the operands of the instructions can be one of two types: either scalar or array. If the instruction calls for execution involving only scalar operands, it is executed by the control processor (i.e., the central processing unit fetching instructions from memory). If, on the other hand, the instruction calls for execution using array operands, it is broadcast to the processing elements.

The processing elements (PEs) are separate computing devices. The PEs do not have their own program counter. Instead, they rely upon the control processor to determine the instructions they will execute. Each PE typically has its own, relatively small, memory in which are stored the unique operands the PE will execute the instruction broadcast by the control processor on. The parallelism arises from having multiple PEs (typically 4K–64K in recent commercial machines) executing the *same* instruction, but on different operands. This type of parallel execution is referred to as synchronous since each PE is always executing the same instruction as other PEs.

In a MIMD computer there exist multiple processors each of which has its own program counter. Processors execute independently of each other according to what-ever instruction the program counter points to next. MIMD computers are usually further subdivided according to whether the processors share memory or each has its own memory.

In a shared-memory MIMD computer both the program's instructions and the part of the program's data to be shared exist within a single shared memory. Additionally, some data may be private to a processor and not be globally accessible by other processors. The processors execute asynchronously of each other. In the most common programming model, they subdivide a computation that is performed on a large data structure in shared memory, each processor performing a part of the computation. Communication and synchronization between the processors are handled by having them each read or write a shared-memory location.

A distributed-memory MIMD computer consists of multiple "nodes." A node is essentially just a sequential computer, that is, a processor and its own (local) memory (and sometimes a local disk also). The processor's program counter fetches instructions from the local memory, and the instructions are executed on data that also resides in local memory. The nodes are connected together via some type of physical interconnection network that allows them to communicate with each other. Parallelism is achieved by having each processor compute simultaneously on the data in its local memory. Communication and synchronization are handled exclusively through the passing of messages (a destination address and the processor local data to be sent) over the interconnection network.

Currently, MIMD computers are more common than SIMD computers. Shared-memory computers are common when only a few processors are being integrated, such as in a multiprocessor workstation. Distributed-memory computers are more common when tens or hundreds of processors are being integrated. The tradeoffs involved are the (widely perceived) ease of use of shared-memory programming relative to distributed-memory programming versus the difficulty of cost-effectively scaling shared-memory computers to integrate more than a few tens of processors before memory access bottlenecks arise. It seems likely that in the next several years we will see the integration of both shared and distributed-memory as "nodes" in a distributed memory computer become themselves shared-memory multiprocessors.

Our interest in parallel computers is as an implementation vehicle for our algorithm. As we explain later, a parallel genetic algorithm is a *model* that can be *implemented* on both sequential and parallel computers. For the model of a parallel genetic algorithm we use, a distributed-memory MIMD computer is the most natural choice for implementation and the one we pursued.

### 1.3 Genetic Algorithms

Genetic algorithms (GAs) are search algorithms. They were developed by Holland [37] and are based on an analogy with natural selection and population genetics. One important use of GAs, and the one we studied, is for finding approximate solutions to

```
t ← 0
initialize P(t)
evaluate P(t)
foreach generation
      t ← t + 1
      select P(t + 1) from P(t)
      recombine P(t + 1)
      evaluate P(t + 1)
endfor
```

Figure 1.1. Simple Genetic Algorithm

difficult optimization problems. As opposed to other optimization methods, genetic algorithms work with a *population* of candidate solutions instead of just a single solution. In the original GAs of Holland, and the ones we use in this paper, each solution may be represented as a string of bits[†], where the interpretation of the meaning of the string is problem specific.

Genetic algorithms work by assigning a value to each string in the population according to a problem-specific fitness function. A "survival-of-the fittest" step selects strings from the *old* population, according to their fitness. These strings recombine using operators such as crossover or mutation to produce a new generation of strings that are (one hopes) more fit than the previous one. A generic genetic algorithm is shown in Figure 1.1.

Two important but competing themes exist in a GA search: the need for selective pressure so that the GA is able to focus the search on promising areas of the search space, and the need for population diversity so that important information (particular bit values) is not lost. Whitley notes [66]:

> Many of the various parameters that are used to "tune" genetic search are really indirect means of affecting selective pressure and population diversity. As selective pressure is increased, the search focuses on the top individuals in the population, but because of this "exploitation" genetic diversity is lost. Reducing the selective pressure (or using a larger population) increases "exploration" because more genotypes and thus more schemata are involved in the search.

[†]In this dissertation we use the terms bit, value, and string instead of the more common GA terminology gene, allele, and chromosome.

In the context of function optimization, strong selective pressure may quickly focus the search on the best individuals at the expense of population diversity, and the lack of diversity can lead the GA to prematurely converge on a suboptimal solution. Conversely, if the selective pressure is relaxed, a high diversity may be maintained, but the search may fail to improve values.

Three performance measures for genetic algorithms are in common use: online performance, offline performance, and best string found. The *online performance* is the average of all function evaluations up to and including the current trial. This measure gauges ongoing performance. The *offline performance* is the average of the best strings from each generation. The offline performance is a running average of all the best performance values to a particular time. The *best string found* is the value of the best string found so far in any generation and is the best metric to measure function optimization ability.

**1.3.1 Constrained Problems.** One trait common to many combinatorial problems solved by GAs is that feasible solutions are easy to construct. For some problems, however—the SPP in particular—generating a feasible solution that satisfies the problem constraints is itself a difficult problem. Three approaches to handling problem constraints have been discussed. In the first, solutions that violate a constraint(s) are infeasible and therefore are declared to have no fitness. This approach is impractical because many problems are tightly constrained and finding a feasible solution may be almost as difficult as finding the optimal one. Also, infeasible solutions often contain valuable information and should not be discarded outright. In the second approach, the GA operators are specialized for the problem, so that no constraints are violated. In the third, a penalty term is incorporated into the fitness function to penalize strings that violate constraints. The idea is to degrade the fitness of infeasible strings but not throw away valuable information contained in the cost term of the fitness function. Below we discuss some examples of the second and third approaches.

Jog, Suh, and Gucht [39] summarize many of the crossover operators used for the traveling salesperson problem (TSP). In general, these operators try to include as much of the parent strings as possible in the offspring, subject to the constraint that the offspring contain a valid tour. In the TSP, since all cities are connected to all other cities, it is relatively easy to 'fix up" an offspring that contains either an invalid tour or a partial tour, by adding missing cities and removing duplicate cities.

As an example, Muhlenbein [48] uses a specialized crossover operator for the TSP called the maximal preservative crossover operator (MPX). The idea is to retain as many valid edges from the parent strings as possible. MPX works by randomly selecting an arbitrary length string from one of the parents to initialize the offspring. Edges are then added from either parent to the offspring, starting at the last city in the offspring, as long as a valid tour is still possible. Otherwise, the next city in one of the parent strings is added. The aim of MPX is to preserve as much of the parent's subtours as possible.

Von Laszewsski and Muhlenbein [65] define a structural crossover operator for the graph partitioning problem that copies whole partitions from one solution to another. Since the copy process may violate the "equal size partition" constraint, a "repairing" operator is applied to "fix things up." For this problem, mutation may also create invalid solutions (mutation is defined as the exchange of two numbers in order to avoid infeasibilities).

Penalty methods allow constraints to be violated. Depending on the magnitude of the violation, however, a penalty that is proportional to the size of the infeasibility is incurred that degrades the objective function. If the cost is large enough, highly infeasible strings will rarely be selected for reproduction, and the GA will concentrate on feasible or near-feasible solutions. A generic evaluation function is of the form

$$c(\mathbf{x}) + p(\mathbf{x}),$$

where $c(\mathbf{x})$ is a cost term (often the objective function of the problem of interest) and $p(\mathbf{x})$ is a penalty term.

Richardson et al. [55] provide advice and experimental results for constructing penalty functions. The authors suggest not making the penalty too harsh, since infeasible solutions contain information that should not be ignored. As an example, they point out that if one removes a column from the optimal solution to a set covering problem, an infeasible solution results. This implies that the optimal solution is separated from infeasible solutions by a Hamming distance of one. Using a similar argument, they note that a single, one-bit mutation can produce the optimal solution from an infeasible one. They suggest that the cost of the penalty term reflect the cost of making an infeasible solution into a feasible one.

Siedlecki and Sklansky [56] use a dynamically calculated penalty coefficient in a GA applied to a pattern recognition problem. Two interesting properties of their problem are that (1) the minimum occurs on a boundary point of the feasible region, and (2) the penalty function is monotonically growing. They report that a variable penalty coefficient outperforms the fixed coefficient penalty.

Cohoon, Martin, and Richards [13] use a penalty term when solving the K-partition problem. The penalty is exponentially increasing with the degree of constraint violation. They observe that the GA tends to "exploit" the penalty term by concentrating its search in a particular part of the search space, willingly incurring a small penalty if the scalar multiplier of the penalty term is not too large.

Smith and Tate [57] suggest a dynamic penalty function for highly constrained problems. They apply this to the unequal area facility layout problem. The severity of their penalty varies and depends on the best solution and best feasible solution found so far. Their intent is to favor solutions that are near feasibility over solutions that are more fit but less feasible.

In conjunction with rank-based selection, Powell and Skolnick [53] scale the objective function for their problem so that all the feasible points always have higher fitness than the infeasible points. This approach avoids difficulties with choosing an appropriate penalty function, but still allows infeasible solutions into the population.

**1.3.2 Parallel Genetic Algorithms.** When referring to a parallel genetic algorithm (PGA) it is important to distinguish between the PGA as a particular *model* of a genetic algorithm and a PGA as a means of *implementing* a (sequential or parallel model of a) genetic algorithm. In a parallel genetic algorithm model, the full population exists in a distributed form; either multiple independent subpopulations exist, or there is one population but each population member interacts only with a limited set of neighbors.

One advantage of the PGA model is that traditional genetic algorithms tend to convergence prematurely, an effect that PGAs seem to be able to partially mitigate because of their ability to maintain more diverse subpopulations by exchanging "genetic material" between subpopulations. Also, in a traditional GA the expected number of offspring of a string depend on the string's fitness relative to *all* other strings in the population. This situation implies a global ranking that is unlike the way natural selection works.

Many GA researchers believe a PGA is a more realistic model of species in nature than a single large population; by analogy with natural selection, a population is typically many independent subpopulations that occasionally interact. Parallel genetic algorithms also naturally fit the model of the way evolution is viewed as occurring; a large degree of independence exists in the "global" population.

Parallel computers are an attractive platform for the implementation of a PGA. The calculations associated with the sequential GA that each subpopulation performs may be computed in parallel, leading to a significant savings in elapsed time. This is important since it allows the *global* population size, and hence the overall number of reproductive trials, to grow without much increase in elapsed computation time.

A parallel *implementation* of the traditional sequential genetic algorithm *model* is also possible. A simple way to do this is to parallelize the loop that creates the next generation from the previous one. Most of the steps in this loop (evaluation, crossover, mutation, and, if used, local search) can be executed in parallel. The selection step, depending on the selection algorithm, may require a global sum that can be a parallel bottleneck. When such an approach has been taken, it is often on a distributed-memory computer. However, unless function evaluation (or local search) is a time-consuming step, the parallel computing overheads associated with distributing data structures to processors, and synchronizing and collecting the results, can mitigate any performance improvements due to multiple processors. Instead, this type of parallel implementation is an obvious candidate for the "loop-level" parallelism common on shared-memory machines. This has important implications for anticipated future parallel computers. Such machines are expected to have multiple processors sharing memory on a node, and many such nodes in a distributed-memory configuration. It will be natural to map a PGA onto the distributed nodes, and speed the *sequential* GA at each node by using the multiple processors to parallelize the generation loop.

Parallel genetic algorithms can be classified according to the granularity of the distributed population, *coarse grained* vs. *fine grained*, and the manner in which the GA operators are applied [39]. In a coarse-grained PGA the population is divided into several subpopulations, each of which runs a traditional GA independently and in parallel on its own subpopulation. Occasionally, fit strings migrate from one subpopulation to another. In some implementations migrant strings may move only to geographically nearby subpopulations, rather than to any arbitrary subpopulation.

In a fine-grained PGA a single population is divided so that a single string is assigned to each processor. Processors select from, crossover with, and replace only strings in their *neighborhood*. Since neighborhoods overlap, fit strings will migrate throughout the population.

**1.3.2.1 Coarse-Grained Parallel Genetic Algorithms.** In a coarse-grained parallel genetic algorithm (CPGA), also referred to later as an island model, multiple processors each run a sequential GA on their own subpopulation. Processors exchange strings from their subpopulation with other processors. Some important choices in a CPGA are which other processors a processor exchanges strings with, how often processors exchange strings, how many strings processors exchange with each other, and what strategy is used when selecting strings to exchange.

Tanese [62] applied a CPGA to the optimization of Walsh-like functions using a 64-processor Ncube computer. Periodically, fit strings were selected and sent to neighboring processors for possible inclusion in their future generations. Exchanges took place only among a processor's neighbors in the hypercube. These exchanges varied over time, taking place over a different dimension of the hypercube each time. Tanese found that the CPGA was able to determine the global maximum of the function about as often as the sequential GA. Tanese reported near-linear speedup of the CPGA over the traditional GA for runs of 1,000 generations. In most cases Tanese's main metric, the average of which generation the global maximum was found on, preferred eight as the optimal number of subpopulations. Tanese also experimented with variable mutation and crossover rates among the subpopulations and found these results at least as good as earlier results.

In [63] Tanese experimented with the *partitioned* genetic algorithm (a CPGA with no migration between processors allowed). A total population size of 256 was partitioned into various power-of-two subpopulation sizes. In all cases the partitioned GA found a better "best fitness value" than the traditional GA, even with small subpopulations sizes such as eight or four. The average fitness of the population at the last generation, however, was consistently worse than that calculated with the traditional algorithm.

Experiments with migration found that a higher average fitness could be obtained if many migrants were sent infrequently or if only a few migrants were sent more frequently. Each processor generated extra offspring during a migration generation and selected migrants uniformly from among the "overfull" population. Often the partitioned GA found fitter strings than the CPGA with migration. Best results were achieved with a migration rate such as 20% of each subpopulation migrating every 20 iterations.

In [60] Starkweather, Whitley, and Mathias describe another CPGA. Each processor sent *copies* of its best strings to one of its neighbors, which replaced its worst string with these. A ring topology was used where, on iteration one, $p_0$ sends to $p_1$, $p_1$ sends to $p_2$, etc., and on iteration two, $p_0$ sends to $p_2$, $p_1$ sends to $p_3$, etc. All sends were done in parallel. In their tests the total population size was fixed, and they experimented with various-sized partitions of the total population among the processors. When no mutation was used, performance improved for two of the four problems as the number of subpopulations was increased, but degraded on the other two. When adaptive mutation was used, with the mutation probability increasing to some predefined maximum as the similarity of the two parents increased, the runs were more successful and achieved good results relative to the serial runs. The more distributed the GA, the more often adaptive mutation was invoked, since smaller subpopulations converge more rapidly than larger ones. Their experiments also indicate that migrating strings too often, or not often enough, degrade performance.

Cohoon, Martin, and Richards [13] applied the CPGA to the K-partition problem using a 16-processor hypercube. Each processor had its own subpopulation of eighty strings, and fifty iterations were run between migrations. An interesting feature of their work was the random choice of scaling coefficient for the penalty term in their fitness function $c(\mathbf{x}) + \lambda p(\mathbf{x})$. The scaling factor $\lambda$ influences how much weight infeasibilities have in evaluating a string's fitness. Two experiments were done. One used $\lambda = 1$ for each subpopulation. In the other, each processor chose a value for $\lambda$ uniformly on the interval (0,1). When the metric "best observed fitness" was applied, the runs with uniformly distributed $\lambda$ were consistently better than those with $\lambda$ fixed at one in each processor.

Kroger, Schwenderling, and Vornberger [41] used a CPGA on a network of 32 transputers to solve the two-dimensional bin packing problem. At "irregular intervals" a processor received strings from neighboring processors. A "parallel elitist strategy" was used whereby, whenever a processor improved upon the best string in its population, it sent a copy of that string to all other processors in its neighborhood. The best results were found with a "medium size" neighborhood and a local population of ten strings.

Pettey, Leuze, and Grefenstette [51] ran a CPGA on an Intel iPSC hypercube. Each generation each processor sent its best strings to each neighbor and received its neighbor's best strings. These were then inserted into each processor's subpopulation by using a replacement scheme. Subpopulation size was fixed at 50 strings; and 1, 2, 4, 8, and 16 processors were used. They believe their results indicate an increased likelihood of premature convergence. This work is at an extreme from most CPGAs, because strings are exchanged every generation and always with the same neighbors. These conditions explain the apparent increased likelihood of premature convergence.

Gordon and Whitley [28] compare eight different parallel genetic algorithms and a version of Goldberg's Simple Genetic Algorithm [26] on several function optimization test problems. Among their conclusions is that island models (CPGAs) perform well, particularly on the hardest problems in their test suite.

**1.3.2.2 Fine-Grained Parallel Genetic Algorithms.** In a fine-grained parallel genetic algorithm (FPGA) exactly one string is assigned to each processor. In the FPGA the model is of one population in which the strings have only local interactions and neighborhoods, as opposed to global ones. Choices in an FPGA include neighborhood size, processor connection topology, and string replacement scheme.

Muhlenbein [48] applied an FPGA to the traveling salesperson problem and the graph partitioning problem. Each string selected a mate from within a small neighborhood of its own processor. Within its neighborhood each processor performed selection, crossover, and mutation without any central control. In addition, each string attempted to improve itself by applying a local search heuristic.

Muhlenbein's objective was to avoid premature convergence by allowing only slow propagation of highly fit strings across the full population. This is dependent on the topology of the processor's neighborhood, which he calls the *population structure*. By choosing a population structure that takes a long time to propagate strings throughout the population, Muhlenbein claimed he avoided premature convergence. The topology used was a two-dimensional circular *ladder* with two strings per "step." A neighborhood size of eight was used by each string. Some overlap occurred among neighborhoods, enabling fit strings to propagate through the population.

In [65] an FPGA was applied to the graph partitioning problem. Strings were mapped to a 64-processor transputer system. Selection was done independently by each string within a small neighborhood of the two-dimensional population structure. The parent string was replaced if the offspring was at least as good as the worst string in the neighborhood. A small neighborhood size in conjunction with a large population size gave the best results.

In [29] Gorges-Schleuter implemented an FPGA on a 64-processor Parsytec transputer system using a sparse graph as the population topology. An elitist strategy was used whereby offspring are accepted for the next generation only if they were more fit than the local parent. A string's fitness was defined relative to other strings in its neighborhood, and neighborhoods could overlap. The algorithm was applied to the TSP problem, using a population size of 64 and a neighborhood size of eight. Results showed that, with a small neighborhood size, communication costs were negligible, and linear speedup was achieved.

**1.3.2.3 Other Parallel Genetic Algorithms.** Fogarty and Huang [23] used a transputer array for the parallel evaluation of a population of 250 strings applied to a real-time control problem. For this problem, evaluating the fitness of a member of the population takes a relatively long time. A host processor ran the main GA program and distributed strings for evaluation to the other transputer processors for evaluation. Maximum speedups in the range of 25–27 were obtained on 40–72 processors. The incremental improvement in speedup was slightly sublinear up to about 16–20 processors, but then fell off quickly.

Liepens and Baluja [44] used a parallel GA with a central processor phase. In parallel, 15 subpopulations of ten strings each run a GA on their own subpopulations. Next, during the central processor phase, the most fit string from each subpopulation is gathered along with an additional 15 randomly generated strings. Under the control of the central processor a recombination phase of these 30 strings occurs. The best string is then injected into the populations of one-third of the processors. Commenting about parallelism, Liepens and Baluja believe that smaller subpopulations remain more heterogeneous.

## 1.4 Thesis Methodology

In this section we explain the motivation and objectives of this thesis, and the performance metrics used.

**1.4.1 Motivation.** There were a number of motivations for applying (parallel) genetic algorithms to the set partitioning problem. One was the particularly challenging nature of the problem. The challenges include the NP-completeness of finding feasible solutions in the general case, and the enormous size of problems of current industrial interest. Also, because of its use as a model for crew scheduling by most major airlines, there is great practical value in developing a successful algorithm.

Genetic algorithms can provide flexibility in handling variations of the model that may be useful. The evaluation function can be easily modified to handle other constraints such as cumulative flight time, mandatory rest periods, or limits on the amount of work allocated to a particular base. More traditional methods may have trouble accommodating the addition of new constraints as easily. Also, at any iteration genetic algorithms contain a population of possible solutions. As noted by Arabeyre et al. [3],

> The knowledge of a family of good solutions is far more important than obtaining an isolated optimum.

This reality has been noted also by many operations research practitioners. Often, for political or other reasons, it is not possible to implement the best solution, but it may be desirable to find one with similar behavior. Traditional operations research algorithms do not maintain knowledge of solutions other than the current best, whereas GAs maintain the "knowledge of a family of good solutions" in the population.

Additionally, the problem has attracted the attention of the operations research community for over twenty-five years, and many real problems exist, so it is possible to compare genetic algorithms with a number of other algorithmic approaches. One advantage of a GA approach is that since it works directly with integer solutions there is no need to solve the LP relaxation.

Finally, as parallel computers move into mainstream computing. the challenge to researchers in all areas is to develop algorithms that can exploit the potential of these powerful new machines. The model of genetic algorithm parallelism we pursue in this dissertation has, we believe, great potential for scaling to take advantage of larger and larger numbers of processors. Since we believe the algorithm maps well to parallel computers, it motivates us to see whether this can help us to solve hard problems of practical interest.

**1.4.2 Thesis Objectives.** This thesis had several objectives which span the fields of genetic algorithms, operations research, and parallel computing. The primary objective was to determine whether a GA can solve real-world SPP problems. Current real-world SPP problems have been generated of almost arbitrary length. Even many smaller problems have posed significant difficulties for traditional methods. Also, in the general case, just finding a *feasible* solution to the SPP is NP-complete [49]. We wished to see how well a GA could perform on such a problem.

We also wished to identify characteristics of SPP problems that were hard for a genetic algorithm. The SPP is both tightly constrained and, in many cases, very large. It also has a natural bit string representation and so is an interesting problem on which to study the effectiveness of GAs. Most applications of GAs have traditionally been to problems with tens or hundreds of bits. We wished to see whether GAs could handle larger problems without the "disruption factor" hindering the search ability. Also, tightly constrained problems have not been the forte of genetic algorithms, and one of our objectives was to see how accurately this limitation carried over to the SPP problem.

Finally, we also wished to study aspects of the parallel genetic algorithm model. We wished to determine the role and influence of parameters such as migration frequency and how strings are selected to migrate or be replaced. We were interested in the algorithmic behavior with the addition of increasing numbers of subpopulations; whether there would be an improvement in the quality of the best solution found, or if it would be found faster, or both.

**1.4.3 Performance Metrics.** The main performance metric we used was the "quality" of the solution found. This was measured by how close to optimality the best solution found was. A second metric was the "efficiency" of the parallel genetic algorithm model we used. As we increased the number of subpopulations (and hence the total population size) we wished to determine whether the number of GA iterations required to find a solution decreased. The third metric of interest, "robustness", was the ability of the algorithm to perform consistently well on a wide range of problem types. This was studied by choosing a large set of test problems and trying to characterize on different "problem profiles" how well the GA performed. Finally, we also compared the parallel GA with traditional operations research methods to see which were more effective.

# CHAPTER II

# SEQUENTIAL GENETIC ALGORITHM

The motivation for the work presented in this chapter was to develop a sequential genetic algorithm that worked well on the set partitioning problem. This would then be used as a building block upon which to develop the parallel genetic algorithm. Although much theoretical work on GAs exists, and much more is currently being pursued by the GA community, there does not yet exist a complete theory for GAs that says which GA operators and their parameter values are best. Often when implementing a GA, practitioners rely upon a large body of empirical research that exists in the literature. In some cases this work is theoretically guided; in others it is the result of extensive experiments or specific application case studies. It is in this context that the work in this chapter was performed.

In Section 2.1 we discuss the test problems we use in this chapter. Section 2.2 discusses the basic genetic algorithm we tested. Section 2.3 discusses the local search heuristic we developed. Section 2.4 discusses specific components of the genetic algorithm and provides a complexity analysis. Finally, Section 2.5 summarizes the results.

## 2.1 Test Problems

The test problems used in this chapter are given in Table 2.1 where they are sorted by increasing number of columns. These problems are a subset of those used by Hoffman and Padberg in [36]. They are "real" set partitioning problems provided by the airline industry. The columns in this table are the test problem name, the number of rows and columns† in the problem, the total number of nonzeros in the $A$ matrix, the objective function value for the linear programming relaxation, and the objective function value for the optimal integer solution. By the standards of SPP problems solved by the airline industry today, these problems can be classified as small ( `nw41`, `nw32`, `nw40`, `nw08`, `nw15`, `nw20`), medium (`nw33`), and large (`aa04`, `nw18`), according to the number of rows and columns in the problem. This particular subset was selected so that we would have several smaller models and a few larger ones.

We can characterize how difficult the test problems are in several ways. First, we can look at the problem parameters, such as the number of rows, columns, and nonzeros. In general, we assume that the larger and more dense a problem is, the harder it is to solve. For the GA, this is justified from a complexity standpoint, since various components of the GA and local search heuristic we use have running time

†In the rest of this dissertation we use rows and columns interchangeably with constraints and variables.

Table 2.1 Sequential Test Problems

| Problem Name | No. Rows | No. Cols | No. Nonzeros | LP Optimal | IP Optimal |
|---|---|---|---|---|---|
| nw41 | 17 | 197 | 740 | 10972.5 | 11307 |
| nw32 | 19 | 294 | 1357 | 14570.0 | 14877 |
| nw40 | 19 | 404 | 2069 | 10658.3 | 10809 |
| nw08 | 24 | 434 | 2332 | 35894.0 | 35894 |
| nw15 | 31 | 467 | 2830 | 67743.0 | 67743 |
| nw20 | 22 | 685 | 3722 | 16626.0 | 16812 |
| nw33 | 23 | 3068 | 21704 | 6484.0 | 6678 |
| aa04 | 426 | 7195 | 52121 | 25877.6 | 26402 |
| nw18 | 124 | 10757 | 91028 | 338864.3 | 340160 |

Table 2.2 Sequential Test Problem Solution Characteristics

| Problem Name | LP Iters. | LP Nonzeros | LP Ones | IP Nodes |
|---|---|---|---|---|
| nw41 | 174 | 7 | 3 | 9 |
| nw32 | 174 | 10 | 4 | 9 |
| nw40 | 279 | 9 | 0 | 7 |
| nw08 | 31 | 12 | 12 | 1 |
| nw15 | 43 | 7 | 7 | 1 |
| nw20 | 1240 | 18 | 0 | 15 |
| nw33 | 202 | 9 | 1 | 3 |
| aa04 | >7428 | 234 | 5 | >1 |
| nw18 | >162947 | 68 | 27 | >62 |

of the order of the number of rows or columns, or the number of nonzeros in a row or column (see Section 2.4.7).

We can also gain some insight into the difficulty of the test problems by solving them with a traditional operations research algorithm.† The test problems have been solved using the public-domain `lp_solve` program [8]. `lp_solve` solves linear programming problems using the simplex method and solves integer programming (IP) problems using the branch-and-bound algorithm. The results are given in Table 2.2. The columns are the test problem name; the number of simplex iterations required to solve the LP relaxation, plus the additional simplex iterations when solving LP subproblems in the branch-and-bound tree; the number of variables in the solution to the LP relaxation that were not zero; the number of the nonzero variables in the solution to the LP relaxation that were one (i.e., not fractional); and the number of nodes searched in the branch-and-bound tree before an optimal solution was found.

`lp_solve` found optimal solutions for problems `nw41`, `nw32`, `nw40`, `nw08`, `nw15`, `nw20`, and `nw33`. `lp_solve` found the optimal solution to the LP relaxation for `nw18`, but not the optimal integer solution before a CPU time limit was reached. The large number of simplex iterations and nodes searched for this problem, relative to the others (except `aa04`), indicate (at least for `lp_solve`) it is a hard problem. `aa04` was the most difficult—`lp_solve` was not able to solve the associated LP relaxation and, in fact, aborted after over 7,000 simplex iterations. `aa04` seems to be a difficult problem for others as well [36]. We conclude that the seven smaller problems are "relatively easy," `nw18` is more difficult, and `aa04` is very difficult.

## 2.2 The Genetic Algorithm

One way to classify genetic algorithms is by the percentage of the population that is replaced each generation. Two choices, at extremes from each other, are common in the literature. The first, the *generational replacement* genetic algorithm (GRGA), replaces the entire population each generation and is the traditional genetic algorithm as defined by Holland [37] and popularized by Goldberg [26]. The second, the *steady-state* genetic algorithm (SSGA), replaces only one or two strings each generation and is a more recent development [61, 66, 69].

In the GRGA the entire population is replaced each generation by their offspring. The hope is that the offspring of the best strings carry the important "building blocks" [26] from the best strings forward to the next generation. The basic outline of the GRGA is given in Figure 1.1. The GRGA allows the possibility that the best strings in the population do not survive to the next generation. Also, as Davis points out [15], many of the best strings may not be allocated any reproductive trials. It is also possible that mutation or crossover destroy or alter important bit values so that they are not propagated into the next generation by the parent's offspring. Many

†We defer discussion of a comparison with Hoffman and Padberg to the next chapter.

Table 2.3 Comparison of the Use of Elitism in GRGA

| Problem | No Elitism | | | Elitism | | |
|---|---|---|---|---|---|---|
| Name | Opt. | Feas. | Trials | Opt. | Feas. | Trials |
| nw41 | 2 | 559 | 863 | 2 | 737 | 864 |
| nw32 | 0 | 412 | 840 | 0 | 562 | 841 |
| nw40 | 0 | 491 | 864 | 0 | 705 | 864 |
| nw08 | 2 | 23 | 860 | 0 | 35 | 861 |
| nw15 | 0 | 3 | 856 | 0 | 4 | 862 |
| nw20 | 0 | 267 | 863 | 0 | 440 | 863 |
| nw33 | 0 | 3 | 575 | 0 | 22 | 576 |
| aa04 | 0 | 0 | 859 | 0 | 0 | 858 |
| nw18 | 0 | 0 | 473 | 0 | 0 | 474 |

implementations of the GRGA use *elitism*; if the best string in the old population is not chosen for inclusion in the new population, it is included in the new population anyway. The idea is to avoid "accidentally" losing the best string found so far. GA practice has shown this is usually advantageous.

Table 2.3 compares the use of elitism in the GRGA. The column *Problem Name* is the name of the test problem. The subheadings *Opt.* and *Feas.* are the number of optimal and feasible integer solutions found, out of the number of trials given in the *Trials* column, respectively. In these experiments we varied several parameters at once (elitism, selection algorithm, penalty term, fitness function, crossover operator, crossover probability, and initialization strategy). The population size was fixed at 50 and the mutation rate at $1/n$. For each choice of parameter value or operator, we performed one computer "run" for each test problem[†]. In each run the random number generator was initialized by using the microsecond portion of the Unix `gettimeofday` system call as a seed.

Comparing the results using as the metric the number of feasible solutions found, we find with a $\chi^2$ test[‡] that elitism is beneficial on five of the problems ( `nw41`, `nw32`, `nw40`, `nw20`, `nw33`). However, the most obvious result from Table 2.3 is the *lack of optimal solutions found*, even on the smaller problems. The main difficulty was the population's premature convergence, so that all the strings in the population were duplicates and no new search was occurring (see also [43] for more on our earlier work). It was this that led us to pursue alternative GA approaches, and in the rest of this dissertation we will report results only for the steady-state genetic algorithm which we found more successful.

---

[†]Because of resource limits, scheduling conflicts, and system crashes, not all runs completed for all problems.

[‡]All $\chi^2$ tests reported in this dissertation use a significance level of 5 percent.

The steady-state genetic algorithm is an alternative to the GRGA that replaces only a few individuals at a time, rather than an entire generation. In practice, the number of new strings to create each generation is usually one or two. The new string(s) replace the worst-ranked string(s) in the population. In this way the SSGA allows both parents and their offspring to coexist in the same population (in fact, this is the usual case).

The SSGA has a "built-in" elitism since only the lowest-ranked string is deleted; the best string is automatically kept in the population. Also, the SSGA is immediately able to take advantage of the "genetic material" in a newly generated string without having to wait to generate the rest of the population as in a GRGA. A disadvantage of the SSGA is that with small populations some bit positions are more likely to lose their value (i.e., all strings in the population have the same value for that bit position) than with a GRGA. For this reason, SSGAs are often run with large population sizes to offset this.

SSGA practitioners advocate discarding a child string if it is a duplicate of a string currently in the population. By avoiding duplicate strings the population is able to maintain more diversity. In our implementation we do not discard a duplicate string, but repeatedly mutate it until it is unique. Not allowing duplicates turned out to be important. Before implementing a method to avoid duplicate strings, we found SSGA populations experienced a similar problem with premature convergence as did the GRGA. Avoiding duplicate strings had a noticeable effect in avoiding or delaying premature convergence.

Figure 2.1 presents the steady-state genetic algorithm we used. Here, we give a brief outline. Specific details of the operators follow in the next several sections. $P(t)$ is the population of strings at generation[†] $t$. Each generation one new string is inserted into the population. The first step is to pick a random string, $\mathbf{x}_{random}$, and apply a local search heuristic (Section 2.3) to it. Next, two parent strings, $\mathbf{x}_1$ and $\mathbf{x}_2$, are selected (Section 2.4.4), and a random number, $r \in [0, 1]$, is generated. If $r$ is less than the crossover probability, $p_c$, we create two new offspring via crossover (Section 2.4.6) and randomly select one of them, $\mathbf{x}_{new}$, to insert in the population. Otherwise, we randomly select one of the two parent strings, make a copy of it, and apply mutation to flip bits in the copy with probability $1/n$. In either case, the new string is tested to see whether it duplicates a string already in the population. If it does, it undergoes (possibly additional) mutation until it is unique. The least-fit string in the population is deleted, $\mathbf{x}_{new}$ is inserted, and the population is reevaluated. The experiments in this chapter all used a population size of 50.

To implement the genetic algorithm and local search heuristic, we wrote a program in ANSI C. It consists of approximately 10,000 lines of source code (including comments) and is portable and runs on all Unix systems it has been tested on. It

[†]We use generation and iteration interchangeably.

```
t ← 0
initialize P(t)
evaluate P(t)
foreach generation
     local_search (x_random ∈ P(t))
     select(x_1, x_2) from P(t)
     if( r < p_c ) then
          x_new = crossover(x_1, x_2)
     else
          x_new = mutate(x_1, x_2)
     endif
     delete (x_worst ∈ P(t))
     while (x_new ∈ P(t))
          mutate(x_new)
     P(t+1) ← P(t) ∪ x_new
     evaluate P(t+1)
     t ← t+1
endfor
```

Figure 2.1. Steady-State Genetic Algorithm

is capable of running on one or more processors. When run on one processor, it is functionally equivalent to a sequential program. For the experiments described in this chapter three different types of computers were used: Sun Sparc 2 workstations, IBM RS/6000 workstations, and an IBM SP1 parallel computer (for these experiments, the SP1 was used as if it were a collection of independent workstations—we ran multiple sequential jobs, each using one SP1 node with *no* interaction between the jobs). Details of the parallel aspects of the program are given in the next chapter.

## 2.3 Local Search Heuristic

A local search heuristic† attempts to improve a solution by moving to a better *neighbor* solution. Whenever the neighboring solution is better than the current solution, it replaces the current solution. When no better neighbor solution can be found, the search terminates.

Parker and Rardin [50] describe two important neighborhoods. In the $k$-change neighborhood, up to $k$ bits are complemented at a time. In the $k$-interchange neighborhood, up to $k$ bits are changed at a time, but in a complementary manner. Trade-offs exist between speed and solution quality; searching a large neighborhood will presumably lead to a better solution than searching a smaller one, but at an increased

†In the GA literature such methods often go by the name *hill-climbing*.

cost in solution time. A related issue is the extent of a given neighborhood that should be searched. At one extreme, every point in the neighborhood is evaluated and the one that improves the current solution the most accepted as the move. Alternatively, we can also make the first move found that improves the current solution. We refer to these two choices as best-improving and first-improving, respectively.

The experimental evidence of many researchers [15, 39, 40, 48] is that hybridizing a genetic algorithm with a local search heuristic is beneficial. It combines the GAs ability to widely sample a search space with a local search heuristic's hill-climbing ability. There are, however, theoretical objections to the use of a local search heuristic. An important one is that changing the "genetic material" in the population in a nonevolutionary manner will affect the schema represented in the population and undermine the GA. Gruau and Whitley [35] comment:

> Changing the coding of an offspring's bit string alters the statistical information about hyperplane subpartitions that is implicitly contained in the population. Theoretically, applying local optimization to improve each offspring undermines the genetic algorithm's ability to search via hyperplane sampling. The objection to local optimization is that changing inherited information in the offspring results in a loss of inherited schemata, and thus a loss of hyperplane information.
>
> Hybrid algorithms that incorporate local optimizations may result in greater reliance on hill-climbing and less emphasis on hyperplane sampling. This reliance could result in less global exploration of the search space because it is hyperplane sampling that is the basis for the claim that genetic algorithms globally sample a search space.

Our early experience with the GRGA [43], as well as subsequent experience with the SSGA, was that both methods had trouble finding optimal (sometimes even feasible) solutions (the SSGA was better than the GRGA, but still not satisfactory). This led us to develop a local search heuristic to hybridize with the GA to assist in finding feasible, or near-feasible, strings to apply the GA operators to.

A local search heuristic for the SPP must address the following. First, since the SPP is tightly constrained, an initial feasible solution may be difficult or impossible to construct. Second, in considering a $k$-change or $k$-interchange move, many of the possible moves may destroy or degrade the degree of feasibility. An effective local search heuristic for the SPP will most likely not be uniform in the size of the neighborhoods it explores, but will vary according to the context of the current solution. For example, if no column covers a row, the heuristic may pick a single column to set to one. For a row that is overcovered, however, the heuristic may try to set to zero all but one of the columns.

We developed a heuristic we call ROW (since it takes a row-oriented view of the problem). The basic outline is given in Figure 2.2. ROW works as follows. For some

```
foreach niters
      i = chose_row( random_or_max )
      improve (i, |r_i|, best_or_first)
endfor
```

Figure 2.2. ROW Heuristic

number of iterations (a parameter of the heuristic), one of the $m$ rows of the problem is selected (another parameter). For any row there are three possibilities: $|r_i| = 0$, $|r_i| = 1$, and $|r_i| > 1$. The action of ROW in these cases varies and also varies according to whether we are using a best-improving or first-improving strategy. In the case of best-improving we apply one of the following rules.

I. $|r_i| = 0$: For each $j \in R_i$ calculate $\Delta_{j_1}$. Set to one the column that minimizes $\Delta_{j_1}$.

II. $|r_i| = 1$: Let $k$ be the unique column in $r_i$. Calculate $\Delta'_j$, the change in $f$ when $x_k \leftarrow 0$ and $x_j \leftarrow 1, j \in R_i$. If $\Delta'_j < 0$ for at least one $j$, set $x_k \leftarrow 0$ and $x_l \leftarrow 1$, for $\Delta'_l < \Delta'_j,\ \forall j$.

III. $|r_i| > 1$: For each $j \in r_i$ calculate $\Delta''_j$, the change in $f$ when $x_k \leftarrow 0, \forall k \in r_i, k \neq j$. Set $x_k \leftarrow 0, \forall k \in r_i, k \neq j$, where $\Delta''_j < \Delta''_k, \forall k$.

We note that strictly speaking this is not a best-improving heuristic. The reason is that in cases I and III we can move to neighboring solutions that degrade the current solution. The reason we allow this is that we know that whenever $|r_i| = 0$ or $|r_i| > 1$, constraint $i$ is infeasible and we must move from the current solution even if neighboring solutions are less attractive. The advantage is that the solution "jumps out" of a locally optimal, but infeasible domain of attraction.

The first-improving version of ROW differs from the best-improving version in the following ways. If $|r_i| = 0$, we select a random column $j$ from $R_i$ and set $x_j \leftarrow 1$. If $|r_i| = 1$, we set $x_k \leftarrow 0$ and $x_j \leftarrow 1$ as soon as we find *any* $\Delta'_j < 0, j \in R_i$. Finally, if $|r_i| > 1$, we randomly select a column $k \in r_i$, leave $x_k = 1$, and set all other $x_j = 0, j \in r_i$. In the cases where $|r_i| = 0$ and $|r_i| > 1$, since we have no guarantee we will find a "first-improving" solution, but know that we must leave the current solution, we make a random move that makes constraint $i$ feasible, without measuring all the implications (cost component and (in)feasibility of other constraints).

We compared the different options for ROW. The results are given in Tables 2.4–2.7. In these runs we also varied the initialization scheme and penalty term used.

Table 2.4 compares the number of iterations (1, 5, and 20) of ROW that were applied to try to improve a string. A $\chi^2$ test shows no difference between these on any of the test problems. The explanation appears to be that ROW gets stuck in a local optimum and cannot escape within the neighborhood defined by the possible moves specified earlier.

Table 2.5 compares two methods for choosing the constraint to apply ROW to. *Random* means one of the $m$ constraints is selected randomly. *MaxViolation* means that the constraint with the largest value of $|\sum_{j=1}^{n} a_{ij}x_j - 1|$ is selected. The $\chi^2$ test shows that the results on four problems ( `nw41`, `nw32`, `nw15`, `nw33` ) are improved when the selected constraint is chosen randomly. In fact, the maximum violation strategy *never* found an optimal solution. The implication is that the use of randomness plays an important role in escaping local optima.

Table 2.6 compares the best-improving and first-improving strategies. The $\chi^2$ test shows that the first-improving strategy is significantly better on problems `nw41`, `nw40`, `nw15`, and `nw33`. It appears that the randomness in two of the steps of the first-improving strategy helps escape from a locally optimal solution.

Table 2.7 shows the hybrid of the SSGA used in combination with the ROW heuristic. We refer to this hybrid as SSGAROW. For six problems (`nw41`, `nw32`, `nw40`, `nw08`, `nw15`, `nw33`), the first-improving strategy performs significantly better according to the $\chi^2$ test. This table is interesting because we could argue that we would expect exactly the opposite result. That is, since the GA itself introduces randomness into the search we would expect to do better combining the best solution found by ROW rather than the first or a random one, which are presumably not as good. A possible explanation is that the GA has prematurely converged and so the only new search information being introduced is from the ROW heuristic. ROW, however, in its best-improving mode gets trapped in a local optimum, and so little additional search occurs.

Table 2.4 Number of Constraints to Improve in the ROW Heuristic

| Problem | 1 | | 5 | | 20 | |
|---|---|---|---|---|---|---|
| Name | Opt. | Trials | Opt. | Trials | Opt. | Trials |
| `nw41` | 12 | 288 | 8 | 288 | 10 | 284 |
| `nw32` | 1 | 288 | 4 | 287 | 2 | 286 |
| `nw08` | 0 | 285 | 0 | 287 | 0 | 282 |
| `nw15` | 40 | 142 | 40 | 142 | 36 | 141 |
| `nw20` | 2 | 259 | 0 | 257 | 0 | 258 |
| `nw33` | 1 | 280 | 3 | 277 | 5 | 264 |
| `aa04` | 0 | 225 | 0 | 220 | 0 | 213 |
| `nw18` | 0 | 276 | 0 | 277 | 0 | 267 |

Table 2.5 Choice of Constraint to Improve in the ROW Heuristic

| Problem | Random | | MaxViolation | |
|---|---|---|---|---|
| Name | Opt. | Trials | Opt. | Trials |
| `nw41` | 14 | 284 | 0 | 288 |
| `nw32` | 4 | 285 | 0 | 288 |
| `nw40` | 3 | 286 | 0 | 288 |
| `nw08` | 0 | 281 | 0 | 286 |
| `nw15` | 59 | 139 | 0 | 143 |
| `nw20` | 1 | 256 | 0 | 259 |
| `nw33` | 5 | 272 | 0 | 276 |
| `aa04` | 0 | 212 | 0 | 216 |
| `nw18` | 0 | 272 | 0 | 278 |

Table 2.6 Best Improving vs. First Improving in the ROW Heuristic

| Problem | Best | | First | |
|---|---|---|---|---|
| Name | Opt. | Trials | Opt. | Trials |
| `nw41` | 3 | 432 | 27 | 428 |
| `nw32` | 1 | 432 | 6 | 429 |
| `nw40` | 0 | 431 | 4 | 430 |
| `nw08` | 0 | 427 | 0 | 427 |
| `nw15` | 26 | 210 | 90 | 215 |
| `nw20` | 2 | 387 | 0 | 387 |
| `nw33` | 0 | 409 | 9 | 412 |
| `aa04` | 0 | 304 | 0 | 334 |
| `nw18` | 0 | 405 | 0 | 415 |

Table 2.7 Best Improving vs. First Improving in SSGAROW

| Problem | Best | | First | |
|---|---|---|---|---|
| Name | Opt. | Trials | Opt. | Trials |
| `nw41` | 21 | 212 | 53 | 213 |
| `nw32` | 8 | 214 | 34 | 211 |
| `nw40` | 3 | 214 | 16 | 213 |
| `nw08` | 4 | 215 | 15 | 212 |
| `nw15` | 21 | 211 | 47 | 210 |
| `nw20` | 2 | 213 | 4 | 213 |
| `nw33` | 0 | 195 | 7 | 189 |
| `aa04` | 0 | 209 | 0 | 209 |
| `nw18` | 0 | 152 | 0 | 154 |

```
typedef struct {
   int cost;
   int ncv;
   int *cover;
} AMATRIX;
```

Figure 2.3. Structure for Storing Row and Column Information

## 2.4 Genetic Algorithm Components

In this section we discuss some aspects of the genetic algorithm we examined.

**2.4.1 Problem Data Structures.** For solving the large SPP problems that arise in the airline industry [7, 9], data structures that are memory efficient and lend themselves to efficient computation are necessary. In the SPP, both the $A$ matrix and the solution vector are binary, and it is possible to devise special data structures that make efficient use of memory.

A solution to the SPP problem is given by specifying values for the binary decision variables $x_j$. The value of one (zero) indicates that column $j$ is included (not included) in the solution. This solution may be represented by a binary vector $\mathbf{x}^\dagger$ with the interpretation that $x_j = 1(0)$ if bit $j$ is one (zero) in the binary vector.

Representing a SPP solution in a GA is straightforward and natural. A bit in a GA string is associated with each column $j$. The bit is one if column $j$ is included in the solution, and zero otherwise. To make efficient use of memory, we had each bit in a computer word represent a column. Because most computers today are byte addressable, this approach improves storage efficiency by at least a factor of eight compared with integer or character implementations. It does, however, require the development of specialized functions to set, unset, and toggle a bit and to test whether a bit is set.

Since the SPP matrix is typically large and sparse and contains only the values zero and one, it is necessary only to store the indices of the rows and columns where $a_{ij} = 1$. At different points in the algorithm we require a list of the rows intersected by a particular column ($P_j$) or a list of the columns intersected by a particular row ($R_i$). We use the data structure shown in Figure 2.3 for both cases. In the column version, this structure holds $c_j$ in the `cost` field, $|P_j|$ in the `ncv` field, and $P_j$ in the `cover` array. The row version holds $\lambda_i$ in the `cost` field, $|R_i|$ in the `ncv` field, and $R_i$ in the `cover` array.

---

$^\dagger$We use $\mathbf{x}$ interchangeably as the solution to the SPP problem or as a bitstring in the GA population, that is, $\mathbf{x} \in P(t)$.

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 12 | 12 | 5 | 7 | 14 | 24 | 8 | 4 | 9 | 2 | 2 | 18 | 4 | 12 | 12 | 2 | 5 | 4 | 1 | 4 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |

Figure 2.4. Example $A$ Matrix before Sorting

**2.4.2 Block Column Form.** A useful initial step is the ordering of the SPP matrix into block "staircase" form [52]. Block $B_i$ is the set of columns that have their first one in row $i$. $B_i$ is defined for all rows but may be empty for some. Within $B_i$ the columns are sorted in order of increasing $c_j$.

Figures 2.4 and 2.5 show an example of an SPP matrix before and after sorting into block staircase format. The numbers at the top of the matrix are the column costs, $c_j$. The numbers at the bottom of the matrix are the column indices. In this example, $I = \{1, \ldots, 8\}$ and $B_1 = \{13, 8, 2\}, \ldots, B_7 = \{19, 11\}$, and $B_8 = \emptyset$.

Ordering the matrix in this manner is helpful in determining feasibility. In any block, at *most* one $x_j$ may be set to one. Our algorithm takes advantage of this ordering in two ways. First, one initialization scheme (randomly) sets at most one $x_j$ per block to one. Second, the block crossover operator defined in Section 2.4.6 takes advantage of the block column structure.

**2.4.3 Evaluation Function.** Three functions are of interest: the SPP objective function, the evaluation function, and the fitness function. It is the SPP objective function, $z$, that we wish to have the GA minimize. However, the difficulty with using $z$ directly is that it does not take into account whether a string is feasible. Therefore, we introduce an evaluation function to incorporate a cost term *and* a penalty term. Since GAs maximize fitness, however, we still must map the evaluation function (which is being minimized) to a nonnegavtive fitness value. This is the role of the fitness function.

The SPP objective function (Equation 1.1) is given by the definition of the problem. The evaluation function and the fitness function, however, are design choices we must make. Currently, no definitive theory exists to say which choice is best. For the

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 4 | 4 | 12 | 4 | 8 | 12 | 12 | 12 | 2 | 4 | 9 | 18 | 24 | 5 | 7 | 14 | 5 | 2 | 1 | 2 |
| | | | | | | | | | | | | | | | | | | | |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | | | | | | | | | | | | | | | | | | | |
| 13 | 8 | 2 | 20 | 7 | 14 | 15 | 1 | 10 | 18 | 9 | 12 | 6 | 3 | 4 | 5 | 17 | 16 | 19 | 11 |

Figure 2.5. Example $A$ Matrix after Sorting

evaluation function we investigated three choices, each reflecting a different penalty term.

The evaluation function measures "how good" a solution to the SPP problem a string is. This must take into account not just the cost of the columns included in the solution (the SPP objective function value), but also the degree of (in)feasibility of a string. Traditional OR algorithms restrict their search to feasible solutions, and so no additional term is included in the SPP objective function to penalize constraint violations. In the GA approach, however, the GA operators often produce infeasible solutions. In fact, since just finding a feasible solution to the SPP is NP-complete [49], it may be that many or most strings in the population are infeasible. Therefore, we need an evaluation function that takes into account the degree of infeasibility of the string. We used as the generic form of our evaluation function

$$f = c(\mathbf{x}) + p(\mathbf{x}), \tag{2.1}$$

where $f$ is the evaluation function; $c(\mathbf{x})$, the cost term, is the SPP objective function; and $p(\mathbf{x})$ is a penalty term. The choice of penalty term can be significant. If the penalty term is too harsh, infeasible strings that carry useful information but lie outside the feasible region will be ignored and their information lost. If the penalty term is not strong enough, the GA may search only among infeasible strings [55]. We investigated three penalty terms.

The *countinfz penalty term* is

$$\sum_{i=1}^{m} \lambda_i \Phi_i(\mathbf{x}), \tag{2.2}$$

where

$$\Phi_i(\mathbf{x}) = \begin{cases} 1 & \text{if constraint } i \text{ is infeasible,} \\ 0 & \text{otherwise.} \end{cases}$$

The countinfz penalty term indicates whether a constraint is infeasible, but does not measure the magnitude of the infeasibility.

In Equation (2.2) (and Equation (2.3) below), $\lambda_i$ is a scalar weight that penalizes the violation of constraint $i$. Choosing a suitable value for $\lambda_i$ is a difficult problem. In [55] Richardson et al. studied the choice of $\lambda_i$ for the set covering problem (SCP). In the SCP, the equality in Equation (1.2) is replaced by a $\geq$ constraint. The SCP, however, is *not* a highly constrained problem; in the SCP constraint $i$ is infeasible only if $|r_i| = 0$. However, it is easily made feasible by (even randomly) selecting an $x_j, j \in R_i$ to set to one. For the $|r_i| = 0$ case, however, such an approach will not work for the SPP, since any $x_j, j \in R_i$ set to one, while it will satisfy constraint $i$, may introduce infeasibilities into *other* currently feasible constraints. Similarly, if we try to make a constraint with $|r_i| > 1$ feasible by setting all but one of the $x_j, j \in r_i$ to zero, we may undercover other currently feasible constraints.

A good choice for $\lambda_i$ should reflect not just the "costs" associated with making constraint $i$ feasible, but also the impact on other constraints (in)feasibility. We know of no method to calculate an optimal value for $\lambda_i$. Therefore, we made the empirical choice of $\lambda_i = \max_j \{c_j | j \in R_i\}$. This choice is similar to the "P2" penalty in [55], where it provided an upper bound on the cost to satisfy the violated constraints of the SCP. In the case of set partitioning, however, the choice of $\lambda_i$ provides no such bound, and it is possible the GA may find infeasible solutions more attractive than feasible ones (for several problems discussed in the next chapter this situation did happen.)

The *linear penalty term* is

$$\sum_{i=1}^{m} \lambda_i \sum_{j=1}^{n} |a_{ij} x_j - 1| \,. \tag{2.3}$$

This penalty does measure the magnitude of constraint $i$'s infeasibility.

The *ST penalty term* ([57]) is

$$\sum_{i=1}^{m} (\Phi_i(\mathbf{x})/2) \left[z_{feas} - z_{best}\right]. \tag{2.4}$$

Here, $z_{feas}$ is the best *feasible* objective function value found so far, and $z_{best}$ is the best objective function value (feasible or infeasible) found so far. According to Smith and Tate [57]

> ...the explicit goal of our **penalty function** is to favor solutions which are near a feasible solution over more highly-fit solutions which are far from any feasible solution.

Following Smith and Tate, we used a distance-from-feasibility metric which was dependent on the number of violated constraints, but not the magnitude of their violations.

Table 2.8 compares the three penalty terms using SSGA. Only the *ST* penalty shows a significant result (problems `nw41` and `nw08`.) One point to note, is the paucity of optimal solutions found with any of the penalties using SSGA by itself. Table 2.9 contains a similar comparison using SSGAROW. Interestingly, the opposite effect is observed. Both the *countinfz* and *linear* penalty terms perform better than the *ST* penalty. Compared with each other, the only difference that showed up with the $\chi^2$ statistic was on `nw15` where the *linear* penalty term performed better.

Table 2.8 Comparison of Penalty Terms in SSGA

| Problem | Linear | | Countinf | | ST | |
|---|---|---|---|---|---|---|
| Name | Opt. | Trials | Opt. | Trials | Opt. | Trials |
| nw41 | 3 | 284 | 4 | 284 | 14 | 286 |
| nw32 | 3 | 286 | 0 | 282 | 0 | 288 |
| nw40 | 0 | 284 | 0 | 287 | 1 | 288 |
| nw08 | 0 | 287 | 0 | 288 | 8 | 287 |
| nw15 | 1 | 286 | 0 | 285 | 0 | 286 |
| nw20 | 0 | 288 | 1 | 287 | 0 | 285 |
| nw33 | 0 | 288 | 0 | 286 | 0 | 288 |
| aa04 | 0 | 277 | 0 | 272 | 0 | 275 |
| nw18 | 0 | 276 | 0 | 276 | 0 | 279 |

Table 2.9 Comparison of Penalty Terms in SSGAROW

| Problem | Linear | | Countinf | | ST | |
|---|---|---|---|---|---|---|
| Name | Opt. | Trials | Opt. | Trials | Opt. | Trials |
| nw41 | 35 | 143 | 28 | 141 | 11 | 141 |
| nw32 | 18 | 142 | 24 | 142 | 0 | 141 |
| nw40 | 8 | 143 | 9 | 142 | 2 | 142 |
| nw08 | 6 | 142 | 8 | 143 | 5 | 142 |
| nw15 | 42 | 139 | 24 | 143 | 2 | 139 |
| nw20 | 4 | 141 | 2 | 142 | 0 | 143 |
| nw33 | 2 | 128 | 5 | 129 | 0 | 127 |
| aa04 | 0 | 144 | 0 | 136 | 0 | 138 |
| nw18 | 0 | 103 | 0 | 103 | 0 | 100 |

**2.4.4 Fitness and Selection.** The fitness function is used during the selection phase to determine the expected number of reproductive trials to allocate to a string. Genetic algorithms require that the fitness function be nonnegative and that the more highly fit a string, the larger its fitness function value (although see [42] for a discussion of the use of a nonmonotonic fitness function). For the SPP this requires a mapping from the evaluation function to the fitness function. As has been pointed out, however, the evaluation function value itself is not an "exact" measure of fitness [66]. The mapping from the evaluation function to the fitness function "should be considered a design parameter of the genetic algorithm, not a feature of the optimization problem" [31]. In general, the fitness function is given by $u(\mathbf{x}) = g(f(\mathbf{x}))$.

If selection is done via a binary tournament (see below), any fitness function that reflects the monotonicity of the evaluation function will suffice. If selection is to be done by calculating the expected number of reproductive trials and then sampling those, however, the choice of fitness function can play a significant role. We tested two choices for the fitness function.

A *dynamic linear fitness function* [26, 30] is given by

$$u(\mathbf{x}) = af(\mathbf{x}) + b(t).$$

We used $a = -1$ and $b(t) = 1.1 \cdot \max\{f(\mathbf{x})|\mathbf{x} \in P(t)\}$. De la Maza and Tidor [16] point out that the choice of $b(t)$ can significantly affect the selective pressure. Our choice of $b(t)$ is intended to interfere with the selective pressure as little as possible, while still converting the minimization of $f$ into the maximization of $u$.

We also tested a *linear rank fitness function* [5, 66] given by

$$u(\mathbf{x}) = Min + (Max - Min)\frac{rank(\mathbf{x}, t) - 1}{N - 1}, \tag{2.5}$$

where $rank(\mathbf{x}, t)$ is the index of $\mathbf{x}$ in a list sorted in order of decreasing evaluation function value. Ranking requires that $1 \leq Max \leq 2$, and $Min + Max = 2$. We used $Max = 2$. The advantage of ranking over other methods, when selection is proportional to a string's fitness, is that ranking is less prone to premature convergence caused by a super-individual.

Tables 2.10 contains the results of experiments we did comparing the dynamic fitness function to ranking using SSGAROW. Sampling was done using both stochastic universal selection and tournament selection. The $\chi^2$ test shows that neither method performed significantly better than the other on any of the test problems.

The selection phase allocates reproductive trials to strings on the basis of their fitness. Depending on the type of GA, strings selected from the old generation are either included directly in the new generation or become the parents of new strings created by the GA recombination operators. We compared two choices for the selection algorithm: stochastic universal selection and tournament selection.

Baker's stochastic universal selection (SUS) is an optimal sampling algorithm [5]. SUS may be thought of as constructing a roulette wheel using fitness proportionate selection and then spinning the wheel once, where the number of equally

Table 2.10 Comparison of Fitness Techniques in SSGAROW

| Problem | Cmax | | Ranking | |
| --- | --- | --- | --- | --- |
| Name | Opt. | Trials | Opt. | Trials |
| nw41 | 40 | 213 | 34 | 212 |
| nw32 | 19 | 210 | 23 | 215 |
| nw40 | 11 | 213 | 8 | 214 |
| nw08 | 8 | 214 | 11 | 213 |
| nw15 | 30 | 210 | 38 | 211 |
| nw20 | 5 | 210 | 1 | 216 |
| nw33 | 5 | 196 | 2 | 188 |
| aa04 | 0 | 209 | 0 | 209 |
| nw18 | 0 | 152 | 0 | 154 |

spaced markers on the wheel is equal to the population size. This method guarantees that each string is allocated $\lfloor expectedvalue \rfloor$ reproductive trials and no more than $\lceil expectedvalue \rceil$.

In *binary* tournament selection [26, 27] two strings are chosen randomly from the population. The more fit string is then allocated a reproductive trial. In order to produce an offspring, two binary tournaments are held, each of which produces one parent string. These two parent strings then recombine to produce an offspring. A variation of binary tournament selection is *probabilistic* binary tournament selection where the more fit string is selected with a probability $p_b$, $.5 \leq p_b < 1$. [54] Probabilistic binary tournament selection does allow for the possibility that the best string in the population may be lost. Its advantage is a reduction in the selective pressure.

Table 2.11 contains the results comparing SUS to tournament selection using the SSGAROW. The $\chi^2$ test again shows that neither method performs better than the other on any of the problems tested.

**2.4.5 Initialization.** We tested a total of six initialization schemes. Two are random, three are heuristics, and one uses the solution to the LP relaxation. The two random schemes are applied directly to all strings in the population. For the nonrandom methods we initialize a single string via the method being used and then randomly modify it to initialize the rest of the population.

Heuristic initialization violates the "usual" GA strategy of trying to achieve a highly diverse solution space search by random initialization. For quite a while we had trouble finding *feasible* solutions, however. Heuristic initialization was an attempt to bias the search in a more favorable direction. Below we describe the different methods we tested.

Table 2.11 Comparison of Selection Schemes in SSGAROW

| Problem | SUS | | Tournament | |
|---|---|---|---|---|
| Name | Opt. | Trials | Opt. | Trials |
| nw41 | 39 | 214 | 35 | 211 |
| nw32 | 21 | 212 | 21 | 213 |
| nw40 | 8 | 212 | 11 | 215 |
| nw08 | 6 | 215 | 13 | 212 |
| nw15 | 29 | 210 | 39 | 211 |
| nw20 | 4 | 210 | 2 | 216 |
| nw33 | 1 | 195 | 6 | 189 |
| aa04 | 0 | 207 | 0 | 211 |
| nw18 | 0 | 165 | 0 | 141 |

$J_{Chavatal} = \emptyset$
**do until** $(P_j = \emptyset, \forall j)$
  $k = \min_j \{\Delta_{j1}/|P_j| \,|\, x_j = 0\}$
  $J_{Chavatal} = J_{Chavatal} \bigcup k$
  $P_j = P_j - P_k$
**enddo**

Figure 2.6. Modified Chavatal Heuristic

**2.4.5.1 Modified Chavatal Heuristic.** This method is a modification of a heuristic proposed by Chavatal [12] for the set covering problem. For the set covering problem Chavatal notes:

> Intuitively, it seems the desirability of including $j$ in an optimal cover increases with the ratio $|P_j|/c_j$ which counts the number of points covered by $P_j$ per unit cost.

Our modification was to use $\Delta_{j1}/|P_j|$ as the quantity to minimize. The algorithm calculates a set of column indices, $J_{Chavatal}$, and is given in Figure 2.6.

**2.4.5.2 Greedy Heuristic.** The greedy heuristic is similar to the modified Chavatal heuristic. The difference is that the criterion used to decide which column to next set to one in Figure 2.6 is to use $\min_j \{\Delta_{j1} | x_j = 0\}$ instead of $\min_j \{\Delta_{j1}/|P_j| \,|\, x_j = 0\}$.

**2.4.5.3 Gregory's Heuristic.** Gregory's heuristic [32] is a generalization of the Vogel approximation method for generating a starting solution to a Hitchcock

$$
\begin{array}{l}
\textbf{while}(\ \exists\ i \text{ s.t. } r_i = \emptyset\ ) \\
\quad \textbf{for}(\ i = 1, m) \\
\quad\quad \textbf{if}(\ r_i = \emptyset\ ) \\
\quad\quad\quad \Delta_{k_1} = \min_j \{\Delta_{j_1} | j \in R_i\} \\
\quad\quad\quad \Delta_{l_1} = \min_j \{\Delta_{j_1} | j \in R_i, j \neq k\} \\
\quad\quad\quad d_i \leftarrow \Delta_{l_1} - \Delta_{k_1} \\
\quad\quad \textbf{end if} \\
\quad \textbf{end for} \\
\quad q = \min_i \{d_i < d_j, \forall j \text{ s.t. } r_j = \emptyset\} \\
\quad x_q \leftarrow 1 \\
\textbf{end while}
\end{array}
$$

Figure 2.7. Gregory's Heuristic

transportation model. For each row $i$ with $r_i = \emptyset$, the idea is to find the two columns that minimize $\Delta_{j_1}, j \in R_i$, calculate their difference, and find the minimum difference over all such rows. The algorithm is given in Figure 2.7.

**2.4.5.4 Random Initialization.** Random initialization sets $x_j \leftarrow 1$, for all columns $j$, with probability 0.5.

**2.4.5.5 Block Random Initialization.** Block random initialization, based on a suggestion of Gregory [32], uses information about the expected structure of an SPP solution. A solution to the SPP typically contains only a few "ones" and is mostly zeros. We can use this knowledge by randomly setting to one approximately the same number of columns estimated to be one in the final solution. If the average number of nonzeros in a column is $P_{AVG}$, we expect the number of $x_j = 1$ in the optimal solution to be approximately $m/P_{AVG}$.

We use the ratio of $m/P_{AVG}$ to the number of nonnull blocks as the "probability" of whether to set to one some $x_j$ in block $B_i$. If we do choose some $j \in B_i$ to set to one, that column is chosen randomly. If the "probability" is $\geq 1$, we set to one one column in every block.

Table 2.12 contains a comparison of four initialization strategies: the three heuristics (Chavatal, Gregory, and Greedy) and block random initialization using the SSGA. Since the SSGA algorithm by itself was unable to find many optimal solutions, it is not possible to make meaningful comparisons. However, the results suggested that Gregory's heuristic and block random initialization were the two most promising approaches. These were further compared using SSGAROW; the results are shown in Table 2.13. The new results are more meaningful; the $\chi^2$ comparison shows that block random initialization out performs Gregory's heuristic on five problems ( `nw41`, `nw40`, `nw15`, `nw20`, `nw33`) and is outperformed on one (`nw32`). We conclude that by giving the GA a wider selection of points in the search space to sample from, it does a better job than if we try to guide it.

In additional testing of block random initialization versus random initialization, we observed that with random initialization SSGA by itself faired poorly. This result is explained as follows. Approximately half the initial string will be one bits; however, a feasible SPP string has only a few one bits. SSGA alone has only mutation to "zero out" the one bits or crossover to combine "building blocks" of zero bits, and these processes are too slow.

When we compared block random initialization versus random initialization using SSGAROW, the results from the two methods were about the same. In this case the large neighborhood moves ROW makes when $|r_i| > 1$, which is true for most constraints initially, quickly zeros out most of the one bits. After a few generations the number of one bits left in a randomly initialized string quickly approaches the same number found in a block randomly initialized string.

Table 2.12 Comparison of Initialization Strategies in SSGA

| Problem | Chavatal | | Gregory | | Greedy | | Brandom | |
|---|---|---|---|---|---|---|---|---|
| Name | Opt. | Trials | Opt. | Trials | Opt. | Trials | Opt. | Trials |
| `nw41` | 1 | 214 | 13 | 214 | 0 | 212 | 7 | 214 |
| `nw32` | 0 | 214 | 2 | 214 | 0 | 214 | 1 | 214 |
| `nw40` | 0 | 215 | 0 | 215 | 0 | 215 | 1 | 214 |
| `nw08` | 0 | 215 | 8 | 215 | 0 | 216 | 0 | 216 |
| `nw15` | 0 | 215 | 0 | 214 | 0 | 214 | 1 | 214 |
| `nw20` | 0 | 214 | 0 | 215 | 0 | 215 | 1 | 216 |
| `nw33` | 0 | 216 | 0 | 216 | 0 | 214 | 0 | 216 |
| `aa04` | 0 | 205 | 0 | 208 | 0 | 202 | 0 | 209 |
| `nw18` | 0 | 207 | 0 | 208 | 0 | 206 | 0 | 210 |

**2.4.5.6 Linear Programming Initialization.** We also tried initializing the population using the solution to the LP relaxation of the test problem. The results in Table 2.14 were all obtained using the solution to the LP relaxation to initialize the population. This was done in a manner similar to the way the other heuristics were applied. First, the first string in the population was initialized using the LP relaxation and then was randomly perturbed to seed the rest of the population. Since the LP solution can be fractional, we experimented with three ways to "integerize" it. One was to use the nonzero value of a variable in the LP solution as a probability; if the value of a random number, $0 \leq r \leq 1$, was less than the variable's value, we set the corresponding bit to one, otherwise to zero. This is column *Flip*. In the second case, we set a bit to one if the corresponding value in the solution to the LP relaxation was $\geq 0.5$, otherwise to zero. This is column *Round*. In the third case, we set to one any bits whose corresponding variable in the solution to the LP relaxation was nonzero, otherwise to zero. This is column *Ceil*.

Table 2.13 Comparison of Initialization Strategies in SSGAROW

| Problem | Gregory | | Brandom | |
|---|---|---|---|---|
| Name | Opt. | Trials | Opt. | Trials |
| nw41 | 11 | 215 | 63 | 210 |
| nw32 | 36 | 213 | 6 | 212 |
| nw40 | 0 | 216 | 19 | 211 |
| nw08 | 7 | 212 | 12 | 215 |
| nw15 | 24 | 211 | 44 | 210 |
| nw20 | 0 | 213 | 6 | 213 |
| nw33 | 0 | 195 | 7 | 189 |
| aa04 | 0 | 212 | 0 | 206 |
| nw18 | 0 | 155 | 0 | 151 |

Between themselves *Ceil* outperforms *Flip* and *Round* on two problems (nw41, nw08) but otherwise none of the other results are significant at the 5 percent level of the $\chi^2$ test. A direct comparison of the the *Ceil* results with the block random results in Table 2.13 is not appropriate since the ten trials in Table 2.14 all used a particular set of parameters, whereas those in Table 2.13 were varied. However, we do note that for the smaller problems LP initialization does well, but for the larger ones (aa04, nw18) it was unable to help SSGAROW find a *feasible* solution to either. Since one of our motivations was to see whether we could develop an algorithm for the SPP that did not need to solve the LP relaxation as a starting point, we did not pursue LP initialization further.

Table 2.14 Linear Programming Initialization in SSGAROW

| Problem | Flip | | Round | | Ceil | |
|---|---|---|---|---|---|---|
| Name | Opt. | Trials | Opt. | Trials | Opt. | Trials |
| nw41 | 3 | 9 | 3 | 9 | 8 | 10 |
| nw32 | 4 | 9 | 3 | 9 | 5 | 9 |
| nw40 | 2 | 9 | 3 | 8 | 2 | 10 |
| nw08 | 4 | 9 | 6 | 9 | 10 | 10 |
| nw15 | 8 | 9 | 6 | 8 | 10 | 10 |
| nw20 | 1 | 8 | 5 | 9 | 2 | 10 |
| nw33 | 2 | 8 | 6 | 8 | 6 | 10 |
| aa04 | 0 | 7 | 0 | 8 | 0 | 9 |
| nw18 | 0 | 9 | 0 | 9 | 0 | 9 |

**2.4.6 Crossover.** The crossover operator takes bits from each parent string and combines them to create child strings. The motivating idea is that by creating new strings from substrings of fit parent strings, new and promising areas of the search space will be explored. Figure 2.8 illustrates the classical one-point crossover

| Parent Strings | Child Strings |
| --- | --- |
| `a a a a a a a a` | `a a b b b b b b` |
| `b b b b b b b b` | `b b a a a a a a` |

Figure 2.8. One-Point Crossover

| Parent Strings | Child Strings |
| --- | --- |
| `a a a a a a a a` | `a a b b b a a a` |
| `b b b b b b b b` | `b b a a a b b b` |

Figure 2.9. Two-Point Crossover

operator. Starting with two parent strings of length $n = 8$, a crossover site $c = 3$ is chosen at random. Two new strings are then created; one uses bits 1–2 from the first parent string and bits 3–8 from the second parent string; the other string uses the complementary bits from the two parent strings.

In the past several years, however, GA researchers have preferred either two-point or uniform crossover. It is these, along with a specialized two-point "block crossover" we developed for the SPP problem, that we compared.

**2.4.6.1 Two-Point Crossover.** Booker [10] cites DeJong [17] who noted that one-point crossover is really a special form of *two-point crossover* where the second "cut" point is always fixed at the zero location. Figure 2.9 illustrates two-point crossover. Starting with two parent strings of length $n = 8$, two crossover sites $c_1 = 3$ and $c_2 = 6$ are chosen at random. Two new strings are then created; one uses bits 1–2 and 6–8 from the first parent string and bits 3–5 from the second parent string; the other string uses the complementary bits from each parent string.

Two-point crossover (and one-point crossover) are special cases of $n$-point crossover operators. In the $n$-point crossover operators, more than one crosspoint is selected, and several substrings from each parent may be exchanged. Experiments by Booker [10] showed a significant improvement in off-line performance at the expense of on-line performance when using two randomly generated crossover points. In the case of function optimization, off-line performance is the more important measure.

**2.4.6.2 Two-Point Block Crossover.** We experimented with a modification of two-point crossover designed to take advantage of the block staircase form we sorted the SPP problem into. We define *two-point block crossover* to be crossover such that the crossover columns, $c_1$ and $c_2$, $c_1 < c_2$, are always selected to be the first columns of two blocks, $B_{i_1}$ and $B_{i_2}$.

Block crossover was developed as an attempt to preserve feasibility (or at least not make a solution more infeasible.) From the definition of block $B_i$ we know

```
Parent Strings                  Child Strings
a a a a a a a a                 b a a b a b b a
b b b b b b b b                 a b b a b a a b
```

Figure 2.10. Uniform Crossover

that all columns in $B_i$ have their first one in row $i$. It follows that at most one column in any block can be set to one in a feasible solution. The intent of two-point block crossover was to avoid introducing additional infeasibilities in the blocks that contain the crossover columns, since all columns in that block come from only one parent. Two-point block crossover can, however, still introduce infeasibilities into other blocks.

**2.4.6.3 Uniform Crossover.** One way to think of *uniform crossover* is as randomly generating a bit-mask that indicates from which parent string to take the next bit when creating the offspring [61]. Figure 2.10 illustrates uniform crossover. Starting with two parent strings of length $n = 8$, the bit-mask `01101001` is randomly generated. This mask is applied to the parent strings such that a "`1`" bit indicates that the next bit for the first child string should be taken from the first parent string, and a "`0`" bit indicates that the next bit for the first child string should be taken from the second parent string. The bit-string is then complemented and the process repeated to create the second child string.

Spears and DeJong [59] and Syswerda [61] give evidence to support the claim that uniform crossover has a better recombination potential—the ability to combine smaller building blocks into larger ones—than do other crossover operators. Testing by Syswerda showed that uniform crossover performed significantly better than one- or two-point crossover on most problems. DeJong and Spears [19] present empirical results on a set of $n$-peak problems (those with one global optima, but $n-1$ local optima) comparing two-point and uniform crossover with varying population sizes. Their results show that uniform crossover is better than two-point crossover for smaller values of $n$ and for smaller values of the population size $N$. In [58], however, Spears and DeJong note just the opposite effect as both $n$ and $N$ increase:

> This suggests a way to understand the role of multi-point crossover. With smaller populations, more disruptive crossover, such as uniform or $n$-point ($n \gg 2$) may yield better results because they help overcome the limited information capacity of smaller populations and the tendency for more homogeneity. However, with larger populations, less disruptive crossover operators (*two*–point) are more likely to work better, as suggested by the theoretical analysis.

Syswerda [61] notes that uniform crossover replaces the need for the inversion operator. Inversion moves bits around so that related sets of bits are less likely to be disrupted and more likely to be grouped with similar bit groupings. Because uniform crossover chooses bits randomly to mask, however, it does not have the same disrupting effect on long defining length schemata that $n$-point crossover does, and so inversion is not necessary. Thus uniform crossover may be advantageous for SPP problems because the long strings associated with large problems may make the interruption of long defining length schemata a serious problem.

Table 2.15 contains the results of our tests to compare all three crossover operators using SSGAROW. The $\chi^2$ test showed no significant difference between *any* of the crossover operators on *any* of the problems.

Table 2.15 Comparison of Crossover Operators Using SSGAROW

| Problem | Two-Point | | Uniform | | Two-Point Block | |
|---|---|---|---|---|---|---|
| Name | Opt. | Trials | Opt. | Trials | Opt. | Trials |
| nw41 | 24 | 142 | 24 | 141 | 26 | 142 |
| nw32 | 15 | 139 | 16 | 144 | 11 | 142 |
| nw40 | 5 | 142 | 8 | 142 | 6 | 143 |
| nw08 | 8 | 140 | 6 | 143 | 5 | 144 |
| nw15 | 25 | 140 | 22 | 140 | 21 | 141 |
| nw20 | 1 | 141 | 1 | 141 | 4 | 144 |
| nw33 | 3 | 141 | 1 | 143 | 3 | 100 |
| aa04 | 0 | 133 | 0 | 143 | 0 | 142 |
| nw18 | 0 | 143 | 0 | 141 | 0 | 22 |

Spears and DeJong [59] suggest parameterizing uniform crossover with a parameter $p_u$ that is the probability of swapping two parents bit values. Normally in uniform crossover $p_u = 0.5$, however, Spears and DeJong note that with $p_u = 0.1$, uniform crossover is less disruptive than two-point crossover with no defining length bias. They believe this is useful in being able to achieve a proper balance between exploration and exploitation. Table 2.16 shows the results of experiments we did to compare three values of $p_u$ (0.6, 0.7, and 0.8) with 0.5. The $\chi^2$ test showed no significant differences among any of the results.

Studies of crossover rate suggest that a high rate, which disrupts many strings selected for reproduction, is important in a small population. Further studies show a decreasing crossover rate as the population size increases. Some classical results using *generational replacement* GAs have suggested $N$ = 50-100 and $p_c = 0.6$ (DeJong [17]), and $N = 80$ and $p_c = 0.45$ (Grefenstette [30]) as good values for offline performance. More recently, steady-state GAs have become prominent; but no set of parameter values is yet a default.

To try to determine a good crossover rate, we tested three crossover probabilities, 0.3, 0.6, and 0.9, in conjunction with the three crossover operators described earlier. The results are shown in Table 2.17. The $\chi^2$ test shows little conclusive evidence; 0.6

Table 2.16 Parameterized Uniform Probability Using SSGAROW

| Problem | $p_b = 0.5$ | | $p_b = 0.6$ | | $p_b = 0.7$ | | $p_b = 0.8$ | |
|---|---|---|---|---|---|---|---|---|
| Name | Opt. | Trials | Opt. | Trials | Opt. | Trials | Opt. | Trials |
| nw41 | 7 | 10 | 7 | 8 | 6 | 10 | 9 | 10 |
| nw32 | 4 | 10 | 0 | 10 | 3 | 9 | 0 | 7 |
| nw40 | 5 | 10 | 4 | 10 | 2 | 6 | 2 | 6 |
| nw08 | 1 | 9 | 2 | 10 | 3 | 8 | 3 | 10 |
| nw15 | 5 | 8 | 8 | 10 | 4 | 9 | 6 | 8 |
| nw20 | 0 | 8 | 1 | 9 | 1 | 10 | 0 | 10 |
| nw33 | 1 | 9 | 3 | 9 | 4 | 10 | 3 | 10 |
| aa04 | 0 | 8 | 0 | 8 | 0 | 6 | 0 | 1 |
| nw18 | 0 | 9 | 0 | 8 | 0 | 8 | 0 | 10 |

superior to 0.9 on two problems (nw41, nw08) and 0.3 superior to 0.6 and 0.9 on one problem (nw40).

Table 2.17 Comparison of Crossover Probabilities in SSGAROW

| Problem | 30% | | 60% | | 90% | |
|---|---|---|---|---|---|---|
| Name | Opt. | Trials | Opt. | Trials | Opt. | Trials |
| nw41 | 23 | 141 | 33 | 142 | 18 | 142 |
| nw32 | 13 | 144 | 14 | 139 | 15 | 142 |
| nw40 | 12 | 144 | 4 | 140 | 3 | 143 |
| nw08 | 7 | 142 | 10 | 144 | 2 | 141 |
| nw15 | 24 | 141 | 24 | 141 | 20 | 139 |
| nw20 | 2 | 140 | 1 | 142 | 3 | 144 |
| nw33 | 3 | 144 | 1 | 136 | 3 | 104 |
| aa04 | 0 | 136 | 0 | 141 | 0 | 141 |
| nw18 | 0 | 116 | 0 | 95 | 0 | 95 |

### 2.4.7 Computational Complexity.

Here we give a complexity analysis for the average cost per iteration for the algorithm given in Figure 2.1. We note that the analysis is particular to specific operator choices we made (e.g., uniform crossover vs. two-point crossover) and also reflects the particular data structures being used. For the test problems used in the next chapter, $N$ was 100, $m$ varied from 17 to 823 and was typically 20–40, and $n$ varied from 197 to 43,749 and was typically 600–3000.

The ROW heuristic is applied to one randomly selected string each generation, and one constraint is randomly selected to try to improve. A first-improving strategy is used. We define $P_{MAX} = \max_j\{|P_j|\} \leq K < m$. That is, $P_{MAX}$ is the largest number of nonzeros in a column, and is bounded by a constant $K$, less than the number of rows. We will use $P_{MAX}$ below as an upper bound on $|P_j|$. We define $R_{AVG}$ to be the average number of nonzeros in a row. Since the choice of constraint is equally likely, we use $R_{AVG}$ when determining complexity terms dependent on the number of nonzeros in a row. For the test problems used in the next chapter, $P_{MAX}$ was typically 7–17, and $R_{AVG}$ was typically 150–200.

If $|r_i| = 0$, a single column is randomly selected in constant time, and an $O(P_{MAX})$ step follows to update the count of how many columns cover each row. If $|r_i| = 1$, we must first determine which column covers this row in time $O(R_{AVG})$. Next, we loop over each $j \in R_i$ ($O(R_{AVG})$) and consider a 1-*interchange* move with the column currently one. Each such comparison requires evaluations of the cost to add and delete the respective columns. Each of these requires a loop over all the rows covered by that column ($O(P_{MAX})$) so the total complexity is $O(R_{AVG}P_{MAX})$. If $|r_i| > 1$, a single column is randomly selected in constant time. Next, to determine the set of columns in $|r_i|$ requires a search through $R_i$, at complexity $O(R_{AVG})$, to see which columns *can* cover this row. These are then set to zero, which takes time $O(|r_i|)$. We conclude that the complexity of ROW is $O(R_{AVG}P_{MAX})$.

Selection was done using a binary tournament. This requires randomly selecting two parents and may be done in constant time. We chose to use uniform crossover, which requires a bit mask for every bit position; its complexity is $O(n)$. Mutation is also $O(n)$, since we call a random number generator for each bit to determine whether we should flip it. Determining the string to delete required looking through the whole population, which takes time $O(N)$.

Not allowing duplicate strings requires comparing the new offspring to each string in the population ($O(N)$). Each of these comparisons requires comparing each bit position ($O(n)$). Therefore, the total complexity of the comparisons is $O(nN)$. If we do find a duplicate, we go through an unknown number of mutate steps, each of which takes time ($O(n)$), until the string is no longer a duplicate.

Function evaluation is done twice each generation, once for the newly created offspring of the GA, and once for the string that ROW was applied to. Evaluating the function requires determining the cost and penalty terms. Calculating the cost component is $O(n)$, since we must test each bit to see which $c_j$ to include in the cost term. To determine the penalty term, we must first determine $|r_i|$ for each $i \in I$. To do this, we loop over each column $j$ ($O(n)$) and, if $x_j = 1$, update $r_i$ for all $i \in P_j$. So the complexity to calculate the penalty term is $O(nP_{MAX})$. Using the up-to-date $|r_i|$'s, we loop over each constraint $O(m)$ to determine the total penalty term. Once we have the evaluation function values, we calculate the fitnesses by searching through the population $O(N)$ to find the least fit string and then calculating Equation (2.4.4) for each string ($O(N)$).

Collecting the largest terms the cost of an average iteration is

$$C_{AVG} = O(nN) + O(R_{AVG}P_{MAX}) + O(m).$$

We make the following empirical observations. First, as described in Section 2.4.1, our implementation works directly with the bits stored in a computer word. If the word length is `WL`, in many cases steps in the algorithm that have complexity $O(n)$ can be done in time $O(n/\mathtt{WL})$, since we can often test for equality or nonzero bits at the word level rather than the individual bit level. Since an SPP solution is mostly zero bits, in practice this will usually be advantageous. Second, we did not keep a sorted list of evaluation function values. However, this could be used to reduce the complexity to determine the string to delete and calculate the fitness values. Third, it is possible to use some hashing of the indices of the one bits in a string to make testing for duplicates more efficient.

### 2.5 Discussion

One early conclusion we reached was that the generational replacement GA, even with elitism, was not very good at finding solutions to SPP problems. In fact, even finding feasible solutions to relatively small problems proved a difficult challenge. The primary cause of this was premature convergence. The SSGA proved more successful, particularly at finding feasible solutions. However, the SSGA still had considerable difficulty finding optimal solutions. This situation motivated us to develop a local search heuristic to hybridize with the SSGA.

The ROW heuristic we developed is specialized for the SPP. ROW has three parameters: how many iterations it is applied, how to select the constraint to apply it to, and how to select a move to make. In general, the most successful approach with ROW seems to be to "work quicker, not harder." We found that applying ROW to just one constraint, choosing this constraint randomly, and using a first-improving strategy (which also introduces randomness when a constraint is infeasible) is more successful than attempts to apply ROW to the most infeasible constraint or find the best-improving solution.

The advantages of ROW relative to a best-improving 1-*opt* heuristic we also implemented [43] are its ability to make moves in large neighborhoods such as when

$|r_i| > 1$, its willingness to move downhill to escape infeasibilities, and the randomness introduced by the first-improving strategy. Even with ROW we detected a convergence in the population after some period of time. When all constraints are feasible, ROW no longer introduces any randomness since in the case $|r_i| = 1$ it is in a "true" first-improving strategy mode. When most constraints are feasible, the $1 - interchange$ moves examined degrade the current solution, so ROW remains trapped in a local optima.

Table 2.18 compares the SSGA, the ROW heuristic, and the SSGAROW hybrid. SSGA and ROW are not much different. Using the $\chi^2$ test, SSGA outperforms ROW on problem `nw08`, and ROW outperforms SSGA on `nw15` and `nw33`. SSGAROW, however, outperforms both ROW and SSGA on five and seven of the test problems, respectively. The search heuristic is able to make good local improvements to the strings, and the GA's recombination ability allows these local improvements to be incorporated into other strings and thus have a global effect.

We tested several operator and parameter value choices. In most cases we concluded that the different options we compared all worked about the same. More specifically, the *linear* amd *countinfz* penalty terms performed about the same. There was no significant difference between either fitness techniques or selection method. The different crossover operators and crossover probabilities we tested also all behaved about the same. An exception was our attempt to initialize the population using some type of heuristic method. We found the wide sampling of the initial search space provided by block random initialization was preferred.

Table 2.18 Comparison of Algorithms

| Problem | SSGA | | ROW | | SSGAROW | |
|---|---|---|---|---|---|---|
| Name | Opt. | Trials | Opt. | Trials | Opt. | Trials |
| `nw41` | 21 | 854 | 30 | 860 | 74 | 425 |
| `nw32` | 3 | 856 | 7 | 861 | 42 | 425 |
| `nw40` | 1 | 859 | 4 | 861 | 19 | 427 |
| `nw08` | 8 | 862 | 0 | 854 | 19 | 427 |
| `nw15` | 1 | 857 | 116 | 425 | 68 | 421 |
| `nw20` | 1 | 860 | 2 | 774 | 6 | 426 |
| `nw33` | 0 | 862 | 9 | 821 | 7 | 384 |
| `aa04` | 0 | 824 | 0 | 649 | 0 | 418 |
| `nw18` | 0 | 831 | 0 | 820 | 0 | 306 |

# CHAPTER III
# PARALLEL GENETIC ALGORITHM

In this chapter we discuss the parallel genetic algorithm we developed. First, we give an overview of the island model that is the basis for the parallel genetic algorithm. Next, we discuss several parameters of the island model and experiments we carried out to try and determine good ones. Third, we describe the hardware and software environment in which the experiments were performed. Fourth, we present the results of our experiments applying the parallel genetic algorithm to a test suite of set partitioning problems. We conclude with a discussion of our results.

## 3.1 The Island Model Genetic Algorithm

In population genetics an island model is one where separate and isolated *subpopulations* evolve independently and in parallel. It is believed that multiple distributed subpopulations, with *local* rules and interactions, are a more realistic model of species in nature than a single large population.

The island model genetic algorithm (IMGA) is analogous to the island model of population genetics. A GA population is divided into several subpopulations, each of which is randomly initialized and runs an independent sequential GA on its own subpopulation. Occasionally, fit strings migrate between subpopulations.

The migration of strings between subpopulations is a key feature of the IMGA. First, it allows the distribution and sharing of above average schemata via the strings that migrate. This serves to increase the overall selective pressure since additional reproductive trials are allocated to those strings that are fit enough to migrate [67]. At the same time, the introduction of migrant strings into the local population helps to maintain genetic diversity, since the migrant string arrives from a different subpopulation which has evolved independently.

The IMGA may be subject to premature convergence pressure if too many copies of a fit string migrate too often, and to too many subpopulations. It is possible that after a certain number of migration steps each subpopulation contains a copy of the globally fittest individual, and copies of this string (and only this string) migrate between subpopulations. In fact, this occurred often in our early experiments when we were not checking to see whether the *arriving* string was a duplicate of one already in the subpopulation. The "fix" was to extend the test for duplicate strings (see Section 2.2) to the arriving string.

The IMGA is itself a logical model. By this we mean that the underlying computer hardware used for the implementation is not specified, only the high-level model. For example, an IMGA can be executed on a sequential computer by time-sharing the processor over the computations associated with each subpopulation's sequential GA. However, the most natural computer hardware on which to implement an IMGA is a distributed-memory parallel computer. In this case each island is mapped to a node, and the processor on that node runs the sequential GA on its subpopulation. Since the nodes execute in parallel, it is possible to perform more reproductive trials in a fixed (elapsed) time period as processors are added, assuming the parallel computing overheads associated with communicating migrating strings do not increase the computational effort significantly. Because selection and other GA operators are applied locally, no global synchronization is required. Finally, strings migrate relatively infrequently, and the amount of data sent is usually small. The result is a very low (attractive) communication to computation ratio.

A word about terminology. Since we always maintain a one to one mapping of subpopulations to processors, in the rest of this thesis we will use the words processor, node, and subpopulation interchangeably. That is, when we say node or processor, we mean the subpopulation that resides on that node or processor.

The IMGA is programmed using a single-program multiple-data (SPMD) programming model; each processor is executing the same program, but on different data (their respective subpopulations). "Synchronization" occurs between processors only when strings are exchanged. A generic IMGA is shown in Figure 3.1. The difference between Figure 3.1 and Figure 2.1 is the addition of a test to see whether on this iteration a string is to be migrated. If so, the neighboring subpopulation to migrate the string to is determined, and the string to migrate, $\mathbf{x}_{migrate}$, is selected and sent to the neighbor. A migrant string, $\mathbf{x}_{recv}$, is then received from a neighboring population, and the string to delete, $\mathbf{x}_{delete}$ is determined and replaced by $\mathbf{x}_{recv}$.

### 3.2 Parameters of the Island Model

An IMGA is characterized by several choices: the type of sequential GA to run on each node, how many strings to migrate and how often to migrate them, how to choose the string(s) to migrate and the string(s) to replace, the logical topology the subpopulations are arranged in, and which subpopulations communicate on a migration step. From our work in the previous chapter, we concluded that a steady-state genetic algorithm in conjunction with the ROW heuristic was an effective choice for the sequential GA. For the other choices, however, a number of possibilities existed.

The choice of "communication" parameters in the IMGA echoes the competing themes of selective pressure and population diversity noted in sequential GAs. Frequently migrating many fit strings and deleting the least fit strings serve to increase the selective pressure, but decrease the population diversity. The choice of logical topology and neighbors to communicate with will affect how "fast" fit strings may migrate among subpopulations.

We chose to fix the number of strings to migrate to one. There were two reasons for this choice. First, it seemed intuitively appealing in conjunction with a SSGA; integrating a single arriving migrant string is similar to how the SSGA integrates its own newly created offspring. The primary differences are that the migrant string arrives from a different subpopulation and is presumably of above-average fitness. The second reason was simply to cut down on the size of the parameter space being explored and to focus on choices for the other parameters. For a similar reason to the latter, we also chose to fix the logical topology of the subpopulations to a two-dimensional toroidal mesh. Each processor exchanged strings with its four neighbors, alternating between them each migration generation (i.e., north, east, west, south, north, . . . ).

To determine suitable values for the other parameters, we performed a set of experiments, similar in philosophy to those described in the preceding chapter. Each of these experiments was performed using eight processors on the IBM SP1. Each processor ran the SSGAROW algorithm on its own subpopulation of size 50. Each run was terminated either when an optimal solution was found or when an iteration limit of 50,000 was reached. Except for the population size and limit on the number of iterations, all other parameters used in these tests were the same as those used in the main experiments described in more detail in Section 3.5.

We restricted these experiments to the seven smaller problems used in our sequential tests. Our intention was to reduce the computational effort required. For each of these seven test problems we ran a total of 72 trials. On each trial we varied one of the parameters: the string to migrate, the string to delete, and the migration frequency. Each trial was randomly initialized as described in Section 3.3.

**3.2.1 String to Migrate.** There are two reasons to send a string to another subpopulation. One is to increase the fitness of the other subpopulation. The other is to help the other subpopulation maintain diversity. As in the sequential GA, the competing themes of selective pressure and diversity arise. If a subpopulation consistently and frequently receives similar, highly fit strings, these strings become predominant in the population, and the GA will focus its search on them at the expense of lost diversity. If, on the other hand, random strings are received, diversity may be maintained, but the subpopulation's fitness will likely not improve.

We compared two ways to choose the string to migrate. In the first, the fittest string in a subpopulation was sent to a neighbor. This strategy tends to increase the selective pressure. In the second case, the string to migrate was selected via a probabilistic binary tournament with parameter $p_b = 0.6$. The second choice serves to reduce the selective pressure while still attempting to migrate strings with above-average fitness.

Table 3.1 compares the two strategies. The results in the *Tournament* column used a probabilistic binary tournament to select the string to migrate. The results in the *Best* column selected the best string in the subpopulation to migrate. The column labeled *OptIter* is an average, over all runs where an optimal solution was found, of the iteration in which the optimal solution was found. The $\chi^2$ test shows no significant difference between either strategy using the number of optimal solutions found as the comparison metric. From the *OptIter* column we note that the strategy

```
t ← 0
initialize P(t)
evaluate P(t)
foreach generation
      local_search (x_random ∈ P(t))
      select(x_1, x_2) from P(t)
      if( r < p_c ) then
            x_new = crossover(x_1, x_2)
      else
            x_new = mutate(x_1, x_2)
      endif
      delete (x_worst ∈ P(t))
      while (x_new ∈ P(t))
            mutate(x_new)
      P(t + 1) ← P(t) ∪ x_new
      if ( migration generation) then
            to = neighbor(myid, gen)
            x_migrate = string_to_migrate(P(t+1))
            send_string(to, x_migrate)
            x_recv = recv_string ( )
            x_delete = string_to_delete(P(t+1))
            replace_string(x_delete, x_recv, P(t + 1))
      endif
      evaluate(P_{t+1})
      t ← t + 1
endfor
```

Figure 3.1. Island Model Genetic Algorithm

Table 3.1 Migrant String Selection Strategies

| Problem Name | Tournament Opt. | Tournament Trials | Tournament OptIter | Best Opt. | Best Trials | Best OptIter |
|---|---|---|---|---|---|---|
| nw41 | 36 | 36 | 601 | 36 | 36 | 586 |
| nw32 | 24 | 36 | 4865 | 22 | 36 | 5172 |
| nw40 | 29 | 36 | 6596 | 30 | 36 | 4147 |
| nw08 | 34 | 36 | 5375 | 31 | 36 | 8135 |
| nw15 | 36 | 36 | 986 | 36 | 36 | 942 |
| nw20 | 18 | 36 | 10601 | 18 | 36 | 5677 |
| nw33 | 24 | 36 | 6807 | 31 | 36 | 4148 |

that is the fastest at finding an optimal solution varies by problem, although `nw40`, `nw20`, and `nw33` show the tournament strategy is significantly slower, most likely implying less selective pressure. We conclude that both the tournament and best strategy are effective and that the choice is not significant as long as above-average fitness strings are being migrated.

**3.2.2 String to Delete.** We tested two strategies for determining the string to delete. The first was to delete the least fit string in the subpopulation. The other was to hold a probabilistic binary tournament with parameter $p_b = 0.4$ and delete the "winner." Deleting the worst-ranked string more aggressively enforces the selective pressure.

Table 3.2 compares the two strategies. The column labeled *Tournament* was defined previously. The column labeled *Worst* refers to selecting the least fit string in the subpopulation to be deleted. The $\chi^2$ test shows the tournament strategy performs significantly better on three problems (`nw32`, `nw40`, and `nw33`). From the *OptIter* column we see that the tournament strategy is again significantly slower at finding the optimal solution. Here, however, the reduction in selective pressure pays dividends, as this strategy is more successful at finding the optimal solution.

The result on `nw20` is interesting. More optimal solutions are found using the worst strategy, although it is just below the 5 percent significance level of the $\chi^2$ test. From Table 2.2 we note that of the seven smaller test problems, `nw20` is in some ways the hardest; it had an all fractional solution to the linear programming relaxation and required the most nodes to be searched in the branch-and-bound tree. It would seem deleting the worst-ranked subpopulation member more severely enforces selective pressure than the choice of string to migrate leading to results similar to what has been observed for sequential GAs; the population converges to a solution faster, but the solution is not necessarily as good as can be found by moderating some of the selective pressure. The increased selective pressure may be necessary on more difficult problems, however.

Table 3.2 String Deletion Strategies

| Problem Name | Tournament Opt. | Trials | OptIter | Worst Opt. | Trials | OptIter |
|---|---|---|---|---|---|---|
| nw41 | 36 | 36 | 638 | 36 | 36 | 549 |
| nw32 | 28 | 36 | 5611 | 18 | 36 | 4080 |
| nw40 | 34 | 36 | 5857 | 25 | 36 | 4661 |
| nw08 | 33 | 36 | 7285 | 32 | 36 | 6079 |
| nw15 | 36 | 36 | 978 | 36 | 36 | 950 |
| nw20 | 14 | 36 | 8926 | 22 | 36 | 7638 |
| nw33 | 32 | 36 | 6202 | 23 | 36 | 4065 |

Table 3.3 Comparison of Migration Frequency

| Problem | No Migration | | 100 | | 1000 | | 5000 | |
|---|---|---|---|---|---|---|---|---|
| Name | Opt. | Trials | Opt. | Trials | Opt. | Trials | Opt. | Trials |
| nw41 | 24 | 24 | 24 | 24 | 24 | 24 | 24 | 24 |
| nw32 | 15 | 24 | 15 | 24 | 17 | 24 | 14 | 24 |
| nw40 | 22 | 24 | 17 | 24 | 20 | 24 | 22 | 24 |
| nw08 | 0 | 24 | 21 | 24 | 23 | 24 | 21 | 24 |
| nw15 | 8 | 24 | 24 | 24 | 24 | 24 | 24 | 24 |
| nw20 | 14 | 24 | 14 | 24 | 11 | 24 | 11 | 24 |
| nw33 | 15 | 24 | 21 | 24 | 16 | 24 | 18 | 24 |

**3.2.3 Frequency of Exchange.** We tested three string migration frequencies and *no* migration. The results, given in Table 3.3, are not conclusive. The only significant result with the $\chi^2$ test was that all three migration choices performed better on `nw08` and `nw15` than no migration. Even without migration, however, the GA still found a number of optimal solutions.

As an example of an ambiguous result, we note that for `nw20`, which we earlier described as possibly the most difficult of the seven test problems, the most optimal solutions were found *both* by migrating as frequently as possible and by not migrating at all.

### 3.3 Computational Environment

The IBM SP1 parallel computer used to run the multiple independent sequential trials described in Chapter II was used in a tightly coupled mode for the parallel experiments described in this chapter. The IBM SP1 we used had 128 nodes, each of which consisted of an IBM RS/6000 Model 370 workstation processor, 128 MB of memory, and a 1 GB disk. Each node ran its own copy of the AIX operating system. The SP1 makes use of a high-performance switch for connecting the nodes.

The parallel program was initially developed on Unix workstations making use of the message passing capabilities of the `p4` [11] parallel programming system. For the parallel experiments on the SP1, the code was ported to the Chameleon [34] message-passing system. Chameleon is designed to provide a portable, high-performance message-passing system. Chameleon runs on top of many other message passing systems, both vendor-specific and third party, allowing widespread portability. In our case Chameleon's `p4` interface allowed us to continue development on workstations and, at the same time, begin experiments on the SP1 where we used Chameleon's EUIH interface. EUIH is an experimental low-overhead version of IBM's External User Interface message passing transport layer. The primary advantage of EUIH is its efficiency for applications that need high-speed communications. Although we do not consider the PGA such an application, since small amounts of data are communicated relatively infrequently, EUIH is the standard transport layer in use on the SP1 system that we used at Argonne National Laboratory.

The parallel program itself is based on the single-program multiple-data (SPMD) model in common use today on distributed-memory computers. It uses explicit sends and receives for communicating strings between processors. Broadcasts from processor zero to other processors handle various initialization tasks. A number of statistical calculations, not part of the algorithm but used for periodic report writing, are handled by collective (global) operations.

Random number generation was done using an implementation of the universal random number generator proposed by Marsaglia, Zaman, and Tseng [45], and translated to C from James' version [38]. Each time a parallel run was made, all subpopulations were randomly seeded. This was done by having processor zero get and broadcast the microsecond portion of the Unix `gettimeofday` system call. Each processor then added its processor id to the value returned by the Unix `gettimeofday` and used this unique value as its random number seed. For the random number generator in [45] each unique seed gives rise to an independent sequence of random numbers of size $\approx 10^{30}$ [38].

### 3.4 Test Problems

To test the parallel genetic algorithm we selected a subset of forty problems from the Hoffman and Padberg test set [36]. This included the nine problems used in Chapter II and thirty-one others. The test problems are given in Table 3.4, where they have been sorted according to increasing numbers of columns. The columns in this table are the test problem name, the number of rows and columns in the problem, the number of nonzeros in the $A$ matrix, the optimal objective function value for the LP relaxation, and the objective function value of the optimal integer solution.

Table 3.5 gives attributes of the solution to the LP relaxation and results from solving the integer programming problem with the lp_solve[†] program. The columns in this table are the name of the test problem, the number of simplex iterations required by lp_solve to solve the LP relaxation plus the additional simplex iterations required to solve LP subproblems in the branch-and-bound tree, the number of variables in the solution to the LP relaxation that were not zero, the number of the nonzero variables in the solution to the LP relaxation that were one (rather than having a fractional value), and the number of nodes searched by lp_solve in its branch-and-bound tree search before an optimal solution was found.

The optimal integer solution was found by lp_solve for all but the following problems: aa04, kl01, aa05, aa01, nw18, and kl02, as indicated in Table 3.5 by the ">" sign in front of the number of simplex iterations and number of IP nodes for these problems. For aa04 and aa01, lp_solve terminated before finding the solution to the LP relaxation. For aa05, kl01, and kl02, lp_solve found the solution to the LP relaxation but terminated before finding any integer solution. A nonoptimal integer solution was found by lp_solve for nw18 before it terminated. Termination occurred either because the program aborted or because a user-specified resource limit was reached.

Many of these problems are "long and skinny", that is, they have few rows relative to the number of columns (it is common in the airline industry to generate subproblems of the complete problem that contain only a subset of the flight legs the airlines are interested in, solve the subproblems, and try to create a solution to the complete problem by piecing together the subproblems). Of these test problems, all but two of the first thirty have fewer than 3000 columns (nw33 and nw09 have 3068 and 3103 columns, respectively). The last ten problems are significantly larger, not just because there are more columns, but also because there are more constraints.

For lp_solve many of the smaller problems are fairly easy, with the integer optimal solution being found after only a small branch-and-bound tree search. There are, however, some exceptions where a large tree search is required (nw23, nw28, nw36, nw29, nw30). These problems loosely correlate with a higher number of fractional values in the LP relaxation than many of the smaller problems, although this correlation does not always hold true (e.g., nw28 with few fractional values requires a "large" tree search, while nw33 with "many" fractional values does not). For the larger problems lp_solve results are mixed. On the nw problems (nw07, nw06, nw11, nw18, and nw03) the results are quite good, with integer optimal solutions found for all but nw18. Again, the size of the branch-and-bound tree searched seems to correlate loosely with the degree of fractionality of the solution to the LP relaxation. On

[†]We note that as a public-domain program lp_solve should not be used as the standard by which to judge the effectiveness of linear and integer programming solution methodology. Our interest here was in being able to characterize the solution difficulty of the test problems and to make a "ballpark" comparison against traditional operations research methodology. For this purpose we believe lp_solve was adequate.

the `kl` and `aa` models, `lp_solve` has considerably more difficulty and does not find any integer solutions.

### 3.5 Parallel Experiments

Our hypothesis was that a parallel genetic algorithm could be developed that would solve real-world set partitioning problems and, further, that the effectiveness of the parallel GA would improve as the number of subpopulations increased.

Our work in Chapter II concentrated on finding a sequential GA that worked well on the SPP. The work in Section 3.2 concentrated on finding a good set of "communication" parameters to use with the IMGA. While we do not claim to have found the optimal set of values in either case, we do believe we have made reasonably good choices.

All the results to be presented were made with the following operators and parameter settings. The sequential GA used was steady-state, with one new individual generated each generation. Fitness was calculated using a dynamic linear fitness function. The penalty term used in the evaluation function was the *countinfz* penalty term (Equation (2.2)). In order to generate a new individual, two strings were selected by holding two binary tournaments. A random number, $0 \leq r \leq 1$, was generated to decide whether a string should undergo crossover *or* mutation. If $r \leq p_c = 0.6$, uniform crossover (with $p_u = 0.7$) was performed, and one of the two offspring was randomly selected to insert into the population. If $r > p_c$, one of the two parent strings was randomly selected, a clone of that parent string was made, and the clone underwent mutation. The mutation rate was fixed and set to the reciprocal of the string length. The least fit string in the population was selected to be deleted. Before inserting a new string into the population, it was first tested to see whether it was a duplicate of a string already in the population. If so, mutation was applied to the string until it was no longer a duplicate of any string in the population.

The ROW heuristic was applied to one randomly selected string each generation. A constraint was randomly selected, and ROW attempted to improve the string with respect to that constraint. The first-improving strategy was used. A run was terminated either when the optimal solution was found† or when all subpopulations had performed 100,000 iterations.

For the communication parameters, the best string in a subpopulation was selected to migrate to a neighboring subpopulation every 1,000 iterations. The string to delete was selected by holding a probabilistic binary tournament (with $p_b = 0.4$). Note that the probabilistic deletion strategy allows a chance that the best string

†For these tests, the value of the (known) optimal solution was stored in the program which tested the best feasible solution found each iteration against the optimal solution and stopped if they were the same.

Table 3.4 Parallel Test Problems

| Problem Name | No. Rows | No. Cols | No. Nonzeros | LP Optimal | IP Optimal |
|---|---|---|---|---|---|
| nw41 | 17 | 197 | 740 | 10972.5 | 11307 |
| nw32 | 19 | 294 | 1357 | 14570.0 | 14877 |
| nw40 | 19 | 404 | 2069 | 10658.3 | 10809 |
| nw08 | 24 | 434 | 2332 | 35894.0 | 35894 |
| nw15 | 31 | 467 | 2830 | 67743.0 | 67743 |
| nw21 | 25 | 577 | 3591 | 7380.0 | 7408 |
| nw22 | 23 | 619 | 3399 | 6942.0 | 6984 |
| nw12 | 27 | 626 | 3380 | 14118.0 | 14118 |
| nw39 | 25 | 677 | 4494 | 9868.5 | 10080 |
| nw20 | 22 | 685 | 3722 | 16626.0 | 16812 |
| nw23 | 19 | 711 | 3350 | 12317.0 | 12534 |
| nw37 | 19 | 770 | 3778 | 9961.5 | 10068 |
| nw26 | 23 | 771 | 4215 | 6743.0 | 6796 |
| nw10 | 24 | 853 | 4336 | 68271.0 | 68271 |
| nw34 | 20 | 899 | 5045 | 10453.5 | 10488 |
| nw43 | 18 | 1072 | 4859 | 8897.0 | 8904 |
| nw42 | 23 | 1079 | 6533 | 7485.0 | 7656 |
| nw28 | 18 | 1210 | 8553 | 8169.0 | 8298 |
| nw25 | 20 | 1217 | 7341 | 5852.0 | 5960 |
| nw38 | 23 | 1220 | 9071 | 5552.0 | 5558 |
| nw27 | 22 | 1355 | 9395 | 9877.0 | 9933 |
| nw24 | 19 | 1366 | 8617 | 5843.0 | 6314 |
| nw35 | 23 | 1709 | 10494 | 7206.0 | 7216 |
| nw36 | 20 | 1783 | 13160 | 7260.0 | 7314 |
| nw29 | 18 | 2540 | 14193 | 4185.3 | 4274 |
| nw30 | 26 | 2653 | 20436 | 3726.8 | 3942 |
| nw31 | 26 | 2662 | 19977 | 7980.0 | 8038 |
| nw19 | 40 | 2879 | 25193 | 10898.0 | 10898 |
| nw33 | 23 | 3068 | 21704 | 6484.0 | 6678 |
| nw09 | 40 | 3103 | 20111 | 67760.0 | 67760 |
| nw07 | 36 | 5172 | 41187 | 5476.0 | 5476 |
| nw06 | 50 | 6774 | 61555 | 7640.0 | 7810 |
| aa04 | 426 | 7195 | 52121 | 25877.6 | 26402 |
| kl01 | 55 | 7479 | 56242 | 1084.0 | 1086 |
| aa05 | 801 | 8308 | 65953 | 53735.9 | 53839 |
| nw11 | 39 | 8820 | 57250 | 116254.5 | 116256 |
| aa01 | 823 | 8904 | 72965 | 55535.4 | 56138 |
| nw18 | 124 | 10757 | 91028 | 338864.3 | 340160 |
| kl02 | 71 | 36699 | 212536 | 215.3 | 219 |
| nw03 | 59 | 43749 | 363939 | 24447.0 | 24492 |

Table 3.5 Solution Characteristics of the Parallel Test Problems

| Problem Name | LP Iters | LP Nonzeros | LP Ones | IP Nodes |
|---|---|---|---|---|
| nw41 | 174 | 7 | 3 | 9 |
| nw32 | 174 | 10 | 4 | 9 |
| nw40 | 279 | 9 | 0 | 7 |
| nw08 | 31 | 12 | 12 | 1 |
| nw15 | 43 | 7 | 7 | 1 |
| nw21 | 109 | 10 | 3 | 3 |
| nw22 | 65 | 11 | 2 | 3 |
| nw12 | 35 | 15 | 15 | 1 |
| nw39 | 131 | 6 | 3 | 5 |
| nw20 | 1240 | 18 | 0 | 15 |
| nw23 | 3050 | 13 | 3 | 57 |
| nw37 | 132 | 6 | 2 | 3 |
| nw26 | 341 | 9 | 2 | 11 |
| nw10 | 44 | 13 | 13 | 1 |
| nw34 | 115 | 7 | 2 | 3 |
| nw43 | 142 | 9 | 2 | 3 |
| nw42 | 274 | 8 | 1 | 9 |
| nw28 | 1008 | 5 | 2 | 39 |
| nw25 | 237 | 10 | 1 | 5 |
| nw38 | 277 | 8 | 2 | 7 |
| nw27 | 118 | 6 | 3 | 3 |
| nw24 | 302 | 10 | 4 | 9 |
| nw35 | 102 | 8 | 4 | 3 |
| nw36 | 74589 | 7 | 1 | 789 |
| nw29 | 5137 | 13 | 0 | 87 |
| nw30 | 2036 | 10 | 0 | 45 |
| nw31 | 573 | 7 | 2 | 7 |
| nw19 | 120 | 7 | 7 | 1 |
| nw33 | 202 | 9 | 1 | 3 |
| nw09 | 146 | 16 | 16 | 1 |
| nw07 | 60 | 6 | 6 | 1 |
| nw06 | 58176 | 18 | 2 | 151 |
| aa04 | >7428 | 234 | 5 | >1 |
| kl01 | >26104 | 68 | 0 | >37 |
| aa05 | >6330 | 202 | 53 | >4 |
| nw11 | 200 | 21 | 17 | 3 |
| aa01 | >23326 | 321 | 17 | >1 |
| nw18 | >162947 | 68 | 27 | >62 |
| kl02 | >188116 | 91 | 1 | >3 |
| nw03 | 4123 | 17 | 6 | 3 |

in the population is replaced. The logical topology was fixed to a two-dimensional toroidal mesh as described earlier in Section 3.2.

Each problem was run once using 1, 2, 4, 8, 16, 32, 64, and 128 subpopulations. Each subpopulation was of size 100. As additional subpopulations were added to the computation, the total number of strings in the *global* population increased. Our assumption was that even though we were doubling the computational effort required whenever we added subpopulations, by mapping each subpopulation to an SP1 processor, the total elapsed time would remain relatively constant (except for the parallel computing overheads associated with string migration, which we felt would be relatively small).

The results of our experiments are summarized in Tables 3.6–3.9. Table 3.6 shows the percent from optimality of the best solution found in any of the subpopulations as a function of the number of subpopulations. An entry of "O" in the table indicates the optimal solution was found. An entry of "X" in the table means no integer feasible solution was found by any of the subpopulations. A numerical entry is the percent from the optimal solution of the best *feasible* solution found by any subpopulation after the 100,000 iteration limit was reached. A blank entry means that the test was not made (usually because of a resource limit or an abort). The solution values themselves are given in Table 3.7. Table 3.8 contains the first iteration on which some subpopulation found a feasible solution. Table 3.9 is similar except that it contains the first iteration on which some subpopulation found an optimal solution. In Table 3.9 an entry of "F" means a nonoptimal integer feasible solution was found.

Entries in the tables marked with a superscript [a] did not complete. If an entry is given, it is from a partially completed run. We give the specific results here. Since output statistics were reported only every 1,000 iterations, that is the resolution with which results are reported in Table 3.8. `nw10` aborted at 37,000 iterations when run using 128 subpopulations. `nw12` aborted at 11,000 iterations when run using 128 subpopulations. `nw09` aborted at 63,000 iterations when run using 64 subpopulations. `kl01` aborted at 76,000 iterations when run using 128 subpopulations. `kl02` aborted at 76,000 iterations when run using 1 subpopulation, and at 76,000 iterations when run using 16 subpopulations. `nw03` aborted at 24,000 iterations when run using 1 subpopulation, at 50,000 iterations when run using 2 subpopulations, and at 24,000 iterations when run using 4 subpopulations.

### 3.6 Discussion

One way of looking at Table 3.6 is to consider it as consisting of four parts (recall that the rows of the table are sorted by increasing numbers of columns in the test problems). The first two parts are defined by the rows between and including `nw41` and `nw06` (the first thirty two problems). We can think of dividing this rectangle into two triangular parts by drawing a diagonal line from the upper left part of the table (`nw41` with one subpopulation) to the bottom right (`nw06` with 128 subpopulations). Most of the results in the "upper triangle" are "O," indicating that an optimal

Table 3.6 Percent from Optimality vs. No. Subpopulations

| Problem | Number of Subpopulations | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Name | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 |
| nw41 | O | O | O | O | O | O | O | O |
| nw32 | 0.0006 | O | 0.0006 | O | O | O | O | O |
| nw40 | O | O | 0.0036 | O | O | O | O | O |
| nw08 | X | 0.0219 | O | O | O | O | O | O |
| nw15 | O | O | O | 0.0001 | 4.4285 | O | O | O |
| nw21 | 0.0037 | 0.0037 | O | O | O | O | O | O |
| nw22 | 0.0735 | 0.0455 | 0.0252 | O | O | O | O | O |
| nw12 | 0.1375 | 0.0912 | 0.0332 | 0.0218 | 0.0094 | O | O | 0.0246[a] |
| nw39 | 0.0425 | O | O | O | O | O | O | O |
| nw20 | 0.0091 | O | O | O | O | O | O | O |
| nw23 | O | O | O | O | 0.0006 | O | O | O |
| nw37 | O | 0.0163 | O | O | O | O | O | O |
| nw26 | 0.0011 | O | O | O | O | O | O | O |
| nw10 | X | X | X | X | X | X | X | X[a] |
| nw34 | 0.0203 | 0.0214 | O | O | O | O | O | O |
| nw43 | 0.0831 | 0.0626 | 0.0350 | O | O | O | O | O |
| nw42 | 0.2727 | 0.0229 | O | O | O | O | O | O |
| nw28 | 0.0469 | O | O | O | O | O | O | O |
| nw25 | 0.1040 | 0.1137 | O | O | O | O | O | O |
| nw38 | 0.0323 | O | O | O | O | O | O | O |
| nw27 | 0.0818 | 0.0567 | O | 0.0039 | O | O | O | O |
| nw24 | 0.0826 | 0.0215 | O | 0.0015 | 0.0038 | O | O | O |
| nw35 | 0.0770 | O | 0.0171 | O | O | O | O | O |
| nw36 | 0.0038 | 0.0010 | 0.0194 | 0.0010 | 0.0019 | O | O | O |
| nw29 | 0.0580 | O | O | 0.0116 | O | O | O | O |
| nw30 | 0.1116 | O | O | O | O | O | O | O |
| nw31 | 0.0069 | 0.0069 | O | O | O | O | O | O |
| nw19 | 0.1559 | 0.1332 | 0.0715 | 0.0880 | 0.0148 | O | O | O |
| nw33 | 0.0128 | O | O | O | O | O | O | O |
| nw09 | 0.0398 | X | 0.0363 | 0.0231 | 0.0155 | 0.0151 | 0.154[a] | O |
| nw07 | 0.3089 | O | O | O | O | O | O | O |
| nw06 | 2.0755 | 0.2532 | O | 0.1779 | 0.0448 | 0.0291 | O | O |
| aa04 | X | X | X | X | X | | | |
| kl01 | 0.0524 | 0.0359 | 0.0368 | 0.0303 | 0.0239 | 0.0184 | 0.0082 | 0.0092[a] |
| aa05 | X | X | X | | X | | | |
| nw11 | X | X | X | X | X | X | X | X |
| aa01 | X | X | X | X | X | X | | |
| nw18 | X | X | X | X | X | X | X | X |
| kl02 | 0.1004[a] | 0.1004 | 0.0502 | 0.0593 | 0.0593[a] | | 0.0410 | 0.0045 |
| nw03 | 0.2732 | 0.1125[a] | 0.1371[a] | | | | | 0.0481 |

[a] See text for discussion.

Table 3.7 Best Solution Found vs. No. Subpopulations

| Problem | Number of Subpopulations | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Name | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 |
| nw41 | 11307 | 11307 | 11307 | 11307 | 11307 | 11307 | 11307 | 11307 |
| nw32 | 14886 | 14877 | 14886 | 14877 | 14877 | 14877 | 14877 | 14877 |
| nw40 | 10809 | 10809 | 10848 | 10809 | 10809 | 10809 | 10809 | 10809 |
| nw08 | X | 36682 | 35894 | 35894 | 35894 | 35894 | 35894 | 35894 |
| nw15 | 67743 | 67743 | 67743 | 67755 | 67746 | 67743 | 67743 | 67743 |
| nw21 | 7436 | 7436 | 7408 | 7408 | 7408 | 7408 | 7408 | 7408 |
| nw22 | 7498 | 7302 | 7160 | 6984 | 6984 | 6984 | 6984 | 6984 |
| nw12 | 16060 | 15406 | 14588 | 14426 | 14252 | 14118 | 14118 | 14466[a] |
| nw39 | 10509 | 10080 | 10080 | 10080 | 10080 | 10080 | 10080 | 10080 |
| nw20 | 16965 | 16812 | 16812 | 16812 | 16812 | 16812 | 16812 | 16812 |
| nw23 | 12534 | 12534 | 12534 | 12534 | 12542 | 12534 | 12534 | 12534 |
| nw37 | 10068 | 10233 | 10068 | 10068 | 10068 | 10068 | 10068 | 10068 |
| nw26 | 6804 | 6796 | 6796 | 6796 | 6796 | 6796 | 6796 | 6796 |
| nw10 | X | X | X | X | X | X | X | X[a] |
| nw34 | 10701 | 10713 | 10488 | 10488 | 10488 | 10488 | 10488 | 10488 |
| nw43 | 9644 | 9462 | 9216 | 8904 | 8904 | 8904 | 8904 | 8904 |
| nw42 | 9744 | 7832 | 7656 | 7656 | 7656 | 7656 | 7656 | 7656 |
| nw28 | 8688 | 8298 | 8298 | 8298 | 8298 | 8298 | 8298 | 8298 |
| nw25 | 6580 | 6638 | 5960 | 5960 | 5960 | 5960 | 5960 | 5960 |
| nw38 | 5738 | 5558 | 5558 | 5558 | 5558 | 5558 | 5558 | 5558 |
| nw27 | 10746 | 10497 | 9933 | 9972 | 9933 | 9933 | 9933 | 9933 |
| nw24 | 6836 | 6450 | 6314 | 6324 | 6338 | 6314 | 6314 | 6314 |
| nw35 | 7772 | 7216 | 7340 | 7216 | 7216 | 7216 | 7216 | 7216 |
| nw36 | 7342 | 7322 | 7456 | 7322 | 7328 | 7314 | 7314 | 7314 |
| nw29 | 4522 | 4274 | 4274 | 4324 | 4274 | 4274 | 4274 | 4274 |
| nw30 | 4382 | 3942 | 3942 | 3942 | 3942 | 3942 | 3942 | 3942 |
| nw31 | 8094 | 8094 | 8038 | 8038 | 8038 | 8038 | 8038 | 8038 |
| nw19 | 12598 | 12350 | 11678 | 11858 | 11060 | 10898 | 10898 | 10898 |
| nw33 | 6764 | 6678 | 6678 | 6678 | 6678 | 6678 | 6678 | 6678 |
| nw09 | 70462 | X | 70222 | 69332 | 68816 | 68784 | 68804[a] | 67760 |
| nw07 | 7168 | 5476 | 5476 | 5476 | 5476 | 5476 | 5476 | 5476 |
| nw06 | 24020 | 9788 | 7810 | 9200 | 8160 | 8038 | 7810 | 7810 |
| aa04 | X | X | X | X | X | | | |
| kl01 | 1143 | 1125 | 1126 | 1119 | 1112 | 1106 | 1095 | 1096[a] |
| aa05 | X | X | X | | X | | | |
| nw11 | X | X | X | X | X | X | X | X |
| aa01 | X | X | X | X | X | X | | |
| nw18 | X | X | X | X | X | X | X | X |
| kl02 | 241[a] | 241 | 230 | 232 | 232[a] | | 228 | 220 |
| nw03 | 31185 | 27249[a] | 27852[a] | | | | | 25671 |

[a] See text for discussion.

Table 3.8 First Feasible Iteration vs. No. Subpopulations

| Problem | Number of Subpopulations | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Name | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 |
| nw41 | 676 | 299 | 393 | 353 | 233 | 127 | 310 | 89 |
| nw32 | 185 | 590 | 520 | 562 | 415 | 373 | 257 | 145 |
| nw40 | 376 | 710 | 434 | 384 | 204 | 223 | 211 | 275 |
| nw08 | X | 5893 | 33876 | 8067 | 6669 | 8393 | 6167 | 4819 |
| nw15 | 2031 | 1233 | 1019 | 1228 | 766 | 767 | 501 | 624 |
| nw21 | 786 | 813 | 618 | 584 | 654 | 627 | 471 | 392 |
| nw22 | 860 | 597 | 540 | 504 | 466 | 426 | 143 | 235 |
| nw12 | 3308 | 2007 | 2379 | 2586 | 1615 | 1963 | 1847 | 2000[a] |
| nw39 | 1017 | 755 | 923 | 516 | 530 | 347 | 447 | 325 |
| nw20 | 1128 | 895 | 912 | 893 | 380 | 619 | 316 | 324 |
| nw23 | 2291 | 2089 | 1686 | 1498 | 525 | 1178 | 1249 | 956 |
| nw37 | 734 | 384 | 620 | 544 | 196 | 502 | 361 | 165 |
| nw26 | 1055 | 978 | 971 | 881 | 760 | 331 | 423 | 474 |
| nw10 | X | X | X | X | X | X | X | X[a] |
| nw34 | 1336 | 672 | 865 | 505 | 354 | 436 | 462 | 295 |
| nw43 | 1036 | 989 | 1025 | 736 | 636 | 675 | 320 | 437 |
| nw42 | 1178 | 936 | 774 | 540 | 460 | 500 | 323 | 361 |
| nw28 | 784 | 372 | 494 | 71 | 289 | 199 | 228 | 13 |
| nw25 | 474 | 731 | 788 | 221 | 328 | 315 | 356 | 369 |
| nw38 | 875 | 1040 | 873 | 662 | 693 | 418 | 311 | 398 |
| nw27 | 874 | 726 | 516 | 658 | 313 | 540 | 437 | 403 |
| nw24 | 1020 | 772 | 898 | 763 | 749 | 670 | 456 | 507 |
| nw35 | 1505 | 1263 | 1084 | 926 | 721 | 893 | 812 | 634 |
| nw36 | 696 | 625 | 493 | 400 | 390 | 361 | 286 | 104 |
| nw29 | 1070 | 604 | 441 | 556 | 424 | 558 | 342 | 294 |
| nw30 | 500 | 622 | 584 | 649 | 481 | 498 | 377 | 356 |
| nw31 | 1447 | 1118 | 1029 | 675 | 358 | 369 | 580 | 236 |
| nw19 | 1656 | 807 | 933 | 1020 | 857 | 812 | 602 | 616 |
| nw33 | 986 | 550 | 815 | 645 | 538 | 493 | 296 | 281 |
| nw09 | 20787 | X | 18414 | 11324 | 11593 | 11737 | 8000[a] | 9025 |
| nw07 | 1132 | 1278 | 589 | 1307 | 928 | 777 | 636 | 677 |
| nw06 | 7472 | 10036 | 5658 | 3920 | 2846 | 3440 | 1788 | 2385 |
| aa04 | X | X | X | X | X | | | |
| kl01 | 3095 | 5146 | 3641 | 4836 | 3324 | 3299 | 3573 | 4000[a] |
| aa05 | X | X | X | | X | | | |
| nw11 | X | X | X | X | X | X | X | X |
| aa01 | X | X | X | X | X | X | | |
| nw18 | X | X | X | X | X | X | X | X |
| kl02 | 6000[a] | 4436 | 6626 | 4721 | 4000[a] | | 4840 | 4521 |
| nw03 | 10563 | 9000[a] | 7000[a] | | | | | 3944 |

[a] See text for discussion.

Table 3.9 First Optimal Iteration vs. No. Subpopulations

| Problem | Number of Subpopulations | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Name | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 |
| nw41 | 3845 | 1451 | 551 | 623 | 758 | 402 | 398 | 362 |
| nw32 | F | 1450 | F | 3910 | 2740 | 2697 | 2054 | 1006 |
| nw40 | 540 | 1597 | F | 1658 | 2268 | 958 | 979 | 696 |
| nw08 | X | F | 34564 | 8955 | 14760 | 10676 | 8992 | 10631 |
| nw15 | 4593 | 17157 | 5560 | F | F | 929 | 692 | 1321 |
| nw21 | F | F | 7875 | 3929 | 4251 | 1818 | 1868 | 2514 |
| nw22 | F | F | F | 29230 | 3370 | 3037 | 2229 | 1820 |
| nw12 | F | F | F | F | F | 62976 | 34464 | F[a] |
| nw39 | F | 2345 | 3738 | 1079 | 1396 | 900 | 1232 | 913 |
| nw20 | F | 2420 | 3018 | 5279 | 27568 | 2295 | 2282 | 1654 |
| nw23 | 2591 | 6566 | 3437 | 3452 | F | 1723 | 2125 | 1477 |
| nw37 | 75737 | F | 1410 | 1386 | 1443 | 1370 | 835 | 779 |
| nw26 | F | 84765 | 52415 | 24497 | 13491 | 1660 | 1512 | 2820 |
| nw10 | X | X | X | X | X | X | X | X[a] |
| nw34 | F | F | 2443 | 1142 | 1422 | 1110 | 1417 | 843 |
| nw43 | F | F | F | 11004 | 3237 | 21069 | 4696 | 3296 |
| nw42 | F | F | 2702 | 3348 | 1070 | 1223 | 1187 | 724 |
| nw28 | F | 903 | 1897 | 1232 | 776 | 718 | 371 | 191 |
| nw25 | F | F | 2634 | 70642 | 4351 | 5331 | 1024 | 1896 |
| nw38 | F | 68564 | 27383 | 1431 | 1177 | 1093 | 603 | 514 |
| nw27 | F | F | 610 | F | 2569 | 1669 | 3233 | 2135 |
| nw24 | F | F | 908 | F | F | 11912 | 2873 | 4798 |
| nw35 | F | 3659 | F | 3182 | 1876 | 1224 | 1158 | 634 |
| nw36 | F | F | F | F | F | 3367 | 2739 | 4200 |
| nw29 | F | 17212 | 5085 | F | 17146 | 1368 | 2243 | 795 |
| nw30 | F | 3058 | 1777 | 1154 | 1650 | 846 | 866 | 949 |
| nw31 | F | F | 1646 | 3085 | 1287 | 1890 | 1682 | 732 |
| nw19 | F | F | F | F | F | 79125 | 27882 | 37768 |
| nw33 | F | 1670 | 1659 | 7946 | 1994 | 2210 | 829 | 873 |
| nw09 | F | X | F | F | F | F | F[a] | 71198 |
| nw07 | F | 29033 | 7459 | 4020 | 4831 | 1874 | 2543 | 1935 |
| nw06 | F | F | 51502 | F | F | F | 48215 | 19165 |
| aa04 | X | X | X | X | X | | | |
| kl01 | F | F | F | F | F | F | F | F[a] |
| aa05 | X | X | X | | X | | | |
| nw11 | X | X | X | X | X | X | X | X |
| aa01 | X | X | X | X | X | X | | |
| nw18 | X | X | X | X | X | X | X | X |
| kl02 | F[a] | F | F | F | F[a] | | F | F |
| nw03 | F | F[a] | F[a] | | | | | F |

[a] See text for discussion.

solution was found. For these problems the hybrid SSGAROW algorithm was able to find the optimal solution to all but one problem. For approximately two thirds of these problems only four subpopulations were necessary before the optimal solution was found. For the other one third of the problems, additional subpopulations are necessary in order to find the optimal solution. For numerical entries in the "lower triangle," we observe that in general the best solution found improves as additional subpopulations participate, even if the optimal solution was not reached. Using 64 subpopulations, the optimal solution was found for 30 of the first 32 test problems. `nw06`, with 6,774 columns, was the largest problem for which we found an optimal solution.

The next two parts of Table 3.6 are defined by rows `aa04` to `nw18` (`kl01` is similar to `kl02` and `nw03` in that increasingly better integer feasible solutions were found as additional subpopulations were added, and so we "logically" group `kl01` with `kl02` and `nw03`) and by the last two problems `kl02` and `nw03`. The first of these, `aa04` through `nw18`, define the group of problems we were not able to solve. For these problems we were unable to find *any* integer feasible solutions. One obvious point to note from Table 3.4 is the large number of constraints in `aa01`, `aa04`, `aa05`, and `nw18` (we will return to `nw18` in a moment). We note from Table 3.5 that these problems have relatively high numbers of fractional values in the solution to the LP relaxation and that they were difficult for `lp_solve` also.

For these problems, Table 3.10 summarizes the average number of infeasible constraints across all strings in all subpopulations as a function of the number of subpopulations. One trend is the general decrease in the average number of infeasible constraints as additional subpopulations are added. For the `aa` problems the incremental improvement, however, appears to be decreasing.

For `nw11` and `nw18` (and also `nw10` for which no feasible solution was found), the GA was able to find infeasible strings with higher fitness than feasible ones and had concentrated its search on those strings. For these problems the best (infeasible) string had an evaluation function value approximately half that of the optimal integer solution. In this case the GA has little chance of ever finding a feasible solution. This is, of course, simply the GA exploiting the fact that for these problems the penalty term used in the evaluation function is not strong enough. For the three `aa` problems this is not the case. On average, near the end of a run an (infeasible) solution has an evaluation function value approximately twice that of the optimal integer solution.

The last two problems, `kl02` and `nw03`, have many columns and an increasing number of constraints. However, the GA was able to find integer feasible solutions on all runs we tried and a very good one for `kl02` with 128 subpopulations. The trend here is similar to all but the infeasible problems. We *conjecture* that with "enough" subpopulations the GA would compute optimal solutions to these problems also. We caution, however, that this is speculation.

Table 3.10 No. of Infeasible Constraints vs. No. Subpopulations

| Problem | Number of Subpopulations | | | | | | |
|---|---|---|---|---|---|---|---|
| Name | 1 | 2 | 4 | 8 | 16 | 32 | 64 |
| `nw11` | 1.6 | 1.7 | 2.7 | 2.1 | 2.1 | 2.4 | 2.4 |
| `nw18` | 17.7 | 12.4 | 14.5 | 15.2 | 14.5 | 14.1 | 14.2 |
| `aa04` | 26.3 | 22.9 | 25.5 | 17.9 | 16.3 | | |
| `aa05` | $95.0^{\dagger}$ | 84.5 | 62.2 | | 56.2 | | |
| `aa01` | 70.1 | 66.0 | 75.2 | 70.0 | 53.0 | 54.6 | |

Table 3.8 shows the first iteration when a feasible solution was found by one of the subpopulations. If we recall that the migration frequency is set to 1,000, we see that even on one processor, over one fourth of the problems find feasible solutions before any migration takes place. The number of problems for which this occurs grows as subpopulations are added. Using 128 subpopulations 27 problems have feasible solutions before the first migration occurs. The ones that do not are the problems where the penalty term was not strong enough, no feasible solution was ever found, or they are the largest problems we tried. The implication is that the ROW heuristic does a good job of decreasing the infeasibilities; and by simply running enough copies of a sequential GA, the likelihood of one of them "getting lucky" increases. The excessive iterations `nw08` takes to get feasible is, again, due to the fact that the penalty term is not strong enough. In this case, however, the penalty is "almost strong enough"; hence, less fit feasible solutions eventually are found "in the neighborhood" of the best (infeasible) strings in the population. A similar problem occurred with `nw09`.

Table 3.9 is similar to Table 3.8; here it is the iteration when an *optimal* solution was found by one of the subpopulations that is shown. Again, we see a general trend of the first optimal iteration occurring earlier as we increase the number of subpopulations. With one subpopulation an optimal solution was found for only one problem (`nw40`) before migration occurred. With 128 subpopulations the optimal solution was found for 13 problems before migration occurred. Several problems show significant decrease in the iteration count as the number of subpopulations increases. As an example, by the time 128 subpopulations are being used to solve `nw37`, `nw38`, and `nw29`, which initially take tens of thousands of iterations to find the optimal solution, the optimal solution has been found before any string migration has occurred.

Table 3.11 compares the solution value found (the subcolumn *Result*) and time in CPU seconds (the subcolumn *Secs.*) of `lp_solve`, the work of Hoffman and Padberg [36] (the column *HP*), and our work (the column *SSGAROW*). The subcolumn *Result* contains a "O" if the optimal solution was found, a numerical entry which is the percentage from optimality of the best suboptimal integer feasible solution found, or an "X" if no feasible solution was found.

The timings for `lp_solve` were made on an IBM RS/6000 Model 590 workstation using the Unix `time` command which had a resolution of one second. These times include the time to convert from the standard MPS format used in linear programming to `lp_solve`'s input format. The timings for Hoffman and Padberg's work are from Tables 3 and 8 in [36]. These runs were made on an IBM RS/6000 Model 550 workstation. The results for SSGAROW are the CPU time charged to processor zero in a run that used the number of processors given in the *Nprocs* column. This is the best solution time achieved where an optimal solution was found. If the entry is numerical, it is the percentage from optimality of the best solution found and the number of processors used for that run. If no feasible solution was found, it is the time and number of processors used. When either `lp_solve` or SSGAROW did not find the optimal solution, the time is prefaced with a $>$.

We offer the comparative results in Table 3.11 with the following caveats. All the timings were done using a heavily instrumented, unoptimized version of our program that performed many global operations to collect statistics for reporting. A number of possible areas for performance improvement exist. Additionally, as noted above, the timings in Table 3.11 are all from different model IBM RS/6000 workstation processors. As such, the reader should adjust them accordingly (depending on the benchmark used, the Model 590 is between a factor of 1.67 and 5.02 times faster than the Model 370, and between a factor of 3.34 and 5.07 times faster than a Model 550). Nevertheless, we include Table 3.11 in the interest of providing some "ballpark" timings to complement the algorithmic behavior.

For many of the first thirty-two problems, where all three algorithms found optimal solutions for all problems (except SSGAROW on `nw10`), we observe that the branch-and-cut solution times are approximately an order of magnitude faster than the branch-and-bound times, and the branch-and-bound times are themselves an order of magnitude faster than SSGAROW. For problems where the penalty term was "not strong enough," but the optimal solution was still found (`nw08`, `nw12`, `nw09`) SSGAROW performs poorly. In two other cases (`nw19`, `nw06`) the search simply takes a long time, the problems have larger numbers of columns (2,879 and 6,774, respectively), and the complexity of the steps in the algorithm that involve $n$ become quite noticeable. There are also some smaller problems for which, if we adjust the times according to the performance differences due to the hardware, SSGAROW seems competitive with branch-and-bound as implemented by `lp_solve`.

On the larger problems we observe that branch-and-cut solved all problems to optimality, in most cases quite quickly. Both `lp_solve` and SSGAROW had trouble with the `aa` problems, neither found a feasible solution to any of the three problems. For the two `kl` problems, SSGAROW was able to find good integer feasible solutions while `lp_solve` did not find any feasible solutions. Although SSGAROW's `kl` computations take much more time than is allotted to `lp_solve`, we note from Table 3.8 that is was able to find less good feasible solutions much earlier in its search. For the larger `nw` problems, `lp_solve` did much better than SSGAROW, proving two optimal (`nw11`, `nw03`) and finding a good integer feasible solution to the other. SSGAROW

Table 3.11 Comparison of Solution Time

| Problem | lp_solve | | HP | | SSGAROW | | |
|---|---|---|---|---|---|---|---|
| Name | Result | Secs.[b] | Result | Secs.[b] | Result | Secs.[b] | Nprocs |
| nw41 | O | 1 | O | 0.1 | O | 4 | 4 |
| nw32 | O | 2 | O | 0.2 | O | 8 | 2 |
| nw40 | O | 3 | O | 0.2 | O | 1 | 1 |
| nw08 | O | 2 | O | 0.1 | O | 135 | 8 |
| nw15 | O | 3 | O | 0.1 | O | 14 | 1 |
| nw21 | O | 1 | O | 0.3 | O | 43 | 32 |
| nw22 | O | 1 | O | 0.3 | O | 65 | 64 |
| nw12 | O | 1 | O | 0.1 | O | 1188 | 64 |
| nw39 | O | 1 | O | 0.2 | O | 16 | 8 |
| nw20 | O | 1 | O | 0.6 | O | 17 | 2 |
| nw23 | O | 6 | O | 0.3 | O | 9 | 1 |
| nw37 | O | 1 | O | 0.2 | O | 16 | 4 |
| nw26 | O | 2 | O | 0.3 | O | 41 | 32 |
| nw10 | O | 1 | O | 0.1 | X | >431 | 1 |
| nw34 | O | 2 | O | 0.3 | O | 18 | 8 |
| nw43 | O | 2 | O | 0.4 | O | 73 | 16 |
| nw42 | O | 3 | O | 1.0 | O | 23 | 16 |
| nw28 | O | 6 | O | 0.4 | O | 8 | 2 |
| nw25 | O | 3 | O | 0.6 | O | 36 | 64 |
| nw38 | O | 4 | O | 1.4 | O | 23 | 128 |
| nw27 | O | 3 | O | 0.3 | O | 7 | 4 |
| nw24 | O | 4 | O | 0.6 | O | 12 | 4 |
| nw35 | O | 4 | O | 0.5 | O | 33 | 128 |
| nw36 | O | 237 | O | 3.7 | O | 128 | 64 |
| nw29 | O | 29 | O | 1.0 | O | 49 | 128 |
| nw30 | O | 20 | O | 0.8 | O | 33 | 8 |
| nw31 | O | 10 | O | 1.4 | O | 34 | 4 |
| nw19 | O | 9 | O | 0.5 | O | 1727 | 64 |
| nw33 | O | 26 | O | 1.5 | O | 25 | 2 |
| nw09 | O | 8 | O | 0.5 | O | 5442 | 128 |
| nw07 | O | 16 | O | 0.7 | O | 129 | 32 |
| nw06 | O | 589 | O | 10.4 | O | 2544 | 128 |
| aa04 | X | >3600 | O | 139337 | X | >1848 | 1 |
| kl01 | X | >1000 | O | 35.4 | .0092 | >11532 | 128 |
| aa05 | X | >1200 | O | 215.3 | X | >3014 | 2 |
| nw11 | O | 27 | O | 2.1 | X | >2548 | 1 |
| aa01 | X | >600 | O | 14441 | X | >2126 | 1 |
| nw18 | .0110 | >3600 | O | 62.5 | X | >2916 | 1 |
| kl02 | X | >3600 | O | 134.4 | .0045 | >43907 | 128 |
| nw03 | O | 375 | O | 24.0 | .0481 | >64994 | 128 |

[b] See text for discussion.

has "penalty troubles" with two of these and takes a long time on `nw03` to compute an integer feasible, but suboptimal solution.

We stress that the times given in Table 3.11 are not just when the optimal solution was found using either the branch-and-bound or branch-and-cut algorithms, but when it was *proven* to be optimal. In the case of SSGAROW we have "cheated" in the sense that for the test problems the optimal solution values are known and we took advantage of that knowledge to specify our stopping criteria. This was advantageous in two ways. First, we knew when to stop (or when to keep going). Second, we knew when a solution was optimal, even though SSGAROW inherently provides no such mathematical tools to determine this. For use in a "production" environment the optimal solutions are typically not known, and an alternative stopping rule would need to be implemented. Conversely, however, we believe that if we had implemented a stopping rule, then in the case of many of the problems we would have given up the search earlier when it "became clear" that progress was not being made.

From Table 3.11 we note that the branch-and-cut work of Hoffman and Padberg clearly provides the best results in all cases. Comparing SSGAROW with `lp_solve`, we see that neither can solve the `aa` problems: `lp_solve` does better than SSGAROW on most (but not all) of the `nw` problems, and SSGAROW does better than `lp_solve` on the two `kl` problems. John Gregory has suggested [33] that the `nw` models, while "real world," are not indicative of the SPP problems most airlines would like to be able to solve, in that they are relatively easy to solve with little branching and that more difficult models may be in production use now, being "solved" by heuristics rather than by exact methods.

In conclusion, it is clear that the branch-and-cut approach of Hoffman and Padberg is superior to both `lp_solve` and SSGAROW in all cases. With respect to genetic algorithms this is not surprising; several leading GA researchers have pointed out that GAs are general-purpose tools that will usually be outperformed when specialized algorithms for a problem exist [15, 18]. Comparing SSGAROW with the branch-and-bound approach as implemented by `lp_solve`, we find that `lp_solve` fares better for many but not all of the test problems. However, the expected scalability we believe SSGAROW will exhibit on larger numbers of processors and the more difficult models that may be in production usage suggest that the parallel genetic algorithm approach may still be worthy of additional research.

In closing this discussion, we offer the following caution about the results we have presented. Each result is stochastic; that is, it depends on the particular random number seed used to initialize the starting populations. Ideally, we would like to be able to present the results as averages for each entry obtained over a large number of samples. However, at the time we did this work, computer time on the IBM SP1 was at a premium, and we were faced with the choice of either running a large number of repeated trials on a restricted set of test problems (which itself would raise the issue of which particular test problems to use) or running only a single test at each data point (test problem and number of subpopulations), but sampling over a larger set of test problems. We believe the latter approach is more useful.

CHAPTER IV

# CONCLUSIONS

The main conclusions of this thesis are the following.

I. The generational replacement genetic algorithm performed poorly on the SPP, even with elitism. Difficulties were experienced just finding feasible solutions to SPP problems, let alone optimal ones. The primary cause was premature convergence. The SSGA proved more successful, particularly at finding feasible solutions. However, the SSGA still had considerable difficulties finding optimal solutions. This situation motivated us to develop a local search heuristic to hybridize with the SSGA.

II. The local search heuristic we developed (ROW) is specialized for the SPP. We found that ROW was about as effective as the SSGA in finding (feasible or optimal) solutions. We found that in many cases ROW was more effective with a "work quicker, not harder" approach. We found that applying ROW to just one constraint, choosing this constraint randomly, and using a first-improving strategy (which also introduces randomness when a constraint is infeasible) was more successful than attempts to apply ROW to the most infeasible constraint or find the best-improving solution. One reason ROW was relatively successful was its willingness to degrade the current solution in order to satisfy infeasible constraints. However, when all constraints were feasible, ROW no longer introduced any randomness and was often trapped in a local optimum.

III. A hybrid algorithm that combines the SSGA and ROW heuristic was more effective than either one by itself (a combination we called SSGAROW). The ROW heuristic is effective at making local improvements, particularly with respect to infeasibilities, and the SSGA helps to propagate these improvements to other strings and thus have a global effect.

IV. Performance of the hybrid algorithm was relatively insensitive to a large number of operator choices and parameter values tested. In most cases performance remained essentially unaffected. An exception was the attempts to initialize the population using heuristic methods. We concluded that we were better off initializing the population randomly and letting SSGAROW take advantage of the wider distribution of points to sample from and make its own way through the search space. Also, we found that not allowing duplicate strings in the population was important in avoiding or delaying premature convergence.

V. The island model genetic algorithm has several parameters related to string migration. On a limited set of tests we found that, overall, migration was preferable to no migration (although on some problems no migration was just as effective as migration), but that the migration interval itself made no significant difference. To determine the string to migrate (delete), we compared the choice of the best- (worst-) ranked string with holding a probabilistic binary tournament. For the choice of string to migrate, we found both choices performed about the same. For the choice of string to delete, we found holding a probabilistic binary tournament worked best. Deleting the worst-ranked string seemed to significantly increase the selective pressure and sometimes led to premature convergence.

VI. Running the hybrid SSGAROW algorithm on each subpopulation in an island model was an effective approach for solving real-world SPP problems of up to a few thousand integer variables. For all but one of the thirty-two small and medium-sized test problems the optimal solution was found. For several larger problems, good integer feasible solutions were found. We found two limitations, however. First, for several problems the penalty term was not strong enough. The GA exploited this by concentrating its search on infeasible strings that had (in some cases significantly) better evaluations than a feasible string would have had. For these problems, either no feasible solution was ever found or the number of iterations and additional subpopulations required to find the optimal solution was much larger than for similar problems for which the penalty term worked well. A second limitation was the fact that three problems had many constraints. For these problems, even though the penalty term seemed adequate, SSGAROW was never able to find a feasible solution.

VII. Adding additional subpopulations (which increase the global population size) was beneficial. When an optimal solution was found, it was usually found on an earlier iteration. In cases where the optimal solution was not found, but a feasible one was (i.e., on the largest test problems), the quality of the feasible solution improved as additional subpopulations were added to the computation. Also notable was the fact that, as additional subpopulations were added, the number of problems for which the optimal solution was found before the first migration occurred continued to increase.

VIII. We compared SSGAROW with implementations of branch-and-cut and branch-and-bound algorithms, looking at the quality of the solutions found and the time taken. Branch-and-cut was clearly superior to SSGAROW and branch-and-bound, finding optimal solutions to all test problems in less time. Both SSGAROW and branch-and-bound found optimal solutions to the small and medium-sized test problems. On larger problems the results were mixed, with both branch-and-bound and SSGAROW doing better than each other on different problems. The branch-and-bound results seem to correlate with how close to integer feasible the solution to the linear programming relaxation was. In many cases branch-and-bound took less time, but we note that the implementation of SSGAROW used was heavily instrumented.

In conclusion, as a proof of concept, we have demonstrated that a parallel genetic algorithm can solve small and medium-sized real-world set partitioning problems. A number of possible areas for further research exist and are discussed in the next chapter.

# CHAPTER V
# FUTURE WORK

A number of interesting areas for future research exist. These include algorithmic enhancements, performance improvements, exploitation of operation research methods, and planning for the next generation of parallel computers.

I. Most of the progress made by SSGAROW occurs early in the search. Profiles of many runs show that the best solution found rarely changes after about 10,000 iterations. This observation seems to hold true irrespective of the number of subpopulations. More subpopulations lead to a more effective early search, but do not help beyond that. We believe that both an adaptive mutation rate and further work on the ROW heuristic can help.

   Currently, the mutation rate is fixed at the reciprocal of the string length, a well-known choice from the GA literature where it plays the role of restoring lost bit values, but does not itself act as a search operator. One possibility is to use an adaptive mutation rate that changes based on the value of some GA statistic such as population diversity or the Hamming distance between two parent strings [68]. Several researchers [14, 64] make the case for a high mutation rate when mutation is separated from crossover, as it is in our implementation. A high mutation rate may also be more successful in an SSGA since, although it may disrupt important schemata in the offspring, those schemata remain intact in the parent strings that remain in the population [64].

   We found that the random choice of variables to add or delete to the current string that the ROW heuristic made when constraints were infeasible helped the GA sample new areas of the search space. However, when all constraints are feasible, ROW no longer introduces any randomness. This is because when all constraints are feasible, all of the alternative moves ROW considers degrade the current solution. Therefore no move is made and ROW remains trapped in a local optimum. We believe some type of simulated annealing-like move in this case would help sustain the search.

II. One limitation of the SSGAROW algorithm was its inability to find feasible solutions for six problems. For three of those, and several others for which optimal solutions were found but with degraded performance, the penalty function was not strong enough. A number of possibilities exist for additional research in this area, including stronger penalty terms (e.g., quadratic), the ranking approach of Powell and Skolnick [53], and revisiting the $ST$ penalty term for which we had mixed results. However, for the `aa` problems, we are less optimistic. Table 3.10 appears to indicate diminishing returns with respect to the reduction in infeasibilities in these problems as additional subpopulations are added to the computation. Much further work on penalties remains to be done.

III. In order to be of practical value, an effective termination strategy is needed. Currently, we stop after either a specified number of iterations or, in the cases of the test set, when we find the *known* optimal solution; such an approach is not viable in practice. One approach might be to stop when the evaluation function value has not changed in a specified number of iterations. A more GA-like approach might use some measure of population similarity such as the average Hamming distance as a convergence test.

IV. Additional work in determining good choices for the parameters of the island model is another area for further research. Selecting the string to migrate or delete seems closest to what has been studied for sequential genetic algorithms (see, for example, Goldberg and Deb's [27] comments about deleting the worst-ranked string in Genitor, and compare that with our empirical findings in Section 3.2.2). However, the appropriate choice of migration interval remains an open question. In fact, the results in Table 3.9, where increasing numbers of problems are solved before before any migration occurs as subpopulations are added, raise the questions of whether migration is necessary or even beneficial. Finally, although we have not explored it here, the choice of logical topology for the subpopulations warrants investigation. For example, is it better for a subpopulation to communicate with many other subpopulations or with the same one?

V. The performance of SSGAROW is not currently optimized. We believe performance improvements are available in several areas. For example, implementation improvements would include incremental updating of certain population statistics that are currently recomputed in full each generation, hashing to make the search for duplicates more efficient, or a faster random number generator. As an example of an algorithmic improvement, uniform crossover requires $O(n)$ calls to a random number generator to determine the bit mask to apply to the parent strings, whereas two-point crossover requires only two calls. Also, in the case where a constraint is feasible, it is computationally desirable to have ROW make a move in constant time, rather than incurring an expensive $O(R_{AVG}P_{MAX})$ cost.

VI. We might also be able to take advantage of operations research work. One example might be to revisit the use of the solution to the LP relaxation to initialize (perhaps just some of) the population. Both Fischer and Kedia [21] and Hoffman and Padberg [36] suggest heuristics for finding integer solutions to SPP problems that might also be incorporated in the initial population. A number of methods for preprocessing a set partitioning problem and using logical reductions to reduce the number of constraints and/or variables have been suggested. These make the problem smaller and (intuitively we assume) easier for the GA to solve.

VII. The current implementation of the IMGA is synchronous. By this we mean that after a string has been migrated from a subpopulation, that subpopulation does not continue executing the sequential GA until it receives a migrant string from a different subpopulation. An asynchronous implementation is also possible. In that case a processor periodically checks its message queue for migrant strings that have been sent from other subpopulations. If any are found, they can be integrated into the subpopulation in the usual manner. If the message queue is empty, the processor continues running the sequential GA on its subpopulation and periodically continues checking its message queue. The advantage of this approach is that the processor is not idle while waiting to receive new strings from neighboring processors, but is instead improving the fitness of its subpopulation.

VIII. We believe the next important class of parallel computer will be distributed-memory MIMD machines, where each node is a shared-memory multiprocessor. From a GA implementation perspective, this raises the question of how best to take advantage of such hardware. "Loop-level" parallelism is available in the generational replacement genetic algorithm when creating generation $t+1$ from $t$ that can exploit such hardware. For the steady-state genetic algorithm, however, because only one new string is created each generation, no such "outer" loop exists. Perhaps in that case, for long enough strings, a fine-grained approach that exploits parallelism within an individual GA operation (e.g., mutation, function evaluation) would be appropriate.

ACKNOWLEDGMENTS

I thank my adviser, Tom Christopher, for allowing me the independence to pursue this work. I thank the members of my committee, Graham Campbell, Peter Greene, Rusty Lusk, and Nick Thomopoulos, for their interest. I am grateful to Argonne National Laboratory and the Mathematics and Computer Science Division for financial support, for access to their computing facilities, and for the stimulating environment they provide their employees.

Although this work is my own, thanks are due to a number of people who helped me in various ways. Thanks to Greg Astfalk for supplying the airline crew scheduling problems. Thanks to Bob Bulfin who introduced me to research and academia many years ago. Thanks to Tom Canfield for statistical advice. Thanks to Remy Evard for help with C macros. Thanks to John Gregory for solving the test problems with a branch-and-bound program, suggestions for initialization, numerous helpful discussions and advice, and a long and valuable friendship. Thanks to Bill Gropp for LaTeX wizardry and for writing wonderfully useful software tools such as the Chameleon library. Thanks to Karla Hoffman for discussions about her branch-and-cut work. Thanks to John Loewy for encouragement and statistical advice. Thanks to Rusty Lusk for many useful suggestions and help with `p4`. Thanks to Jorge Moré for discussions about penalty methods in nonlinear optimization. Thanks to Bob Olson for patient and helpful answers to numerous Perl questions. Thanks to Gail Pieper for her usual outstanding job of technical editing. Thanks to Paul Plassmann for advice and numerous one liners. Thanks to Nick Radcliffe for helpful answers to many genetic algorithms queries. Thanks to the computer support group in the MCS Division at Argonne, a small, but dedicated group who keep a complex computing environment working. Thanks to Xiaobai Sun and Stephen Wright for timely help with child care. Thanks to David Tate for discussing his penalty function with me.

Finally, and most importantly thanks to my parents, Bernard and Sylvia, for their love, support, and encouragement through the years. I hope I can be as good a parent to my children as they have been to theirs.

D.L.

REFERENCES


[1] R. Anbil, E. Gelman, B. Patty, and R. Tanga. Recent Advances in Crew Pairing Optimization at American Airlines. *INTERFACES*, 21:62–74, 1991.

[2] R. Anbil, R. Tanga, and E. Johnson. A Global Approach to Crew Pairing Optimization. *IBM Systems Journal*, 31(1):71–78, 1992.

[3] J. Arabeyre, J. Fearnley, F. Steiger, and W. Teather. The Airline Crew Scheduling Problem: A Survey. *Transportation Science*, 3(2):140–163, 1969.

[4] E. Baker and M. Fisher. Computational Results for Very Large Air Crew Scheduling Problems. *OMEGA*, 9(6):613–618, 1981.

[5] J. Baker. Reducing bias and inefficiency in the selection algorithm. In J. Grefenstette, editor, *Proceedings of the Second International Conference on Genetic Algorithms and Their Applications*, pages 14–21, Hillsdale, New Jersey, 1987. Lawrence Erlbaum Associates.

[6] E. Balas and M. Padberg. Set Partitioning: A Survey. *SIAM Review*, 18(4):710–760, 1976.

[7] J. Barutt and T. Hull. Airline Crew Scheduling: Supercomputers and Algorithms. *SIAM News*, 23(6), 1990.

[8] M. Berkelaar. `lp_solve`, 1993. A public domain linear and integer programming program. Available by anonymous `ftp` from `ftp.es.ele.tue.nl` in directory `pub/lp_solve`, file `lp_solve.tar.Z`.

[9] R. Bixby, J. Gregory, I. Lustig, R. Marsten, and D. Shanno. *Very Large-Scale Linear Programming: A Case Study in Combining Interior Point and Simplex Methods.* Technical Report CRPC, Rice University, 1991.

[10] L. Booker. Improving Search in Genetic Algorithms. In *Genetic Algorithms and Simulated Annealing*, pages 61–73. Pitman Publishing, London, 1987.

[11] R. Butler and E. Lusk. Monitors, Messages, and Clusters: The p4 Parallel Programming System. *Parallel Computing*, 20, 1994.

[12] V. Chavatal. A Greedy Heuristic for the Set Covering Problem. *Mathematics of Operations Research*, 4(3):233–235, 1979.

[13] J. Cohoon, W. Martin, and D. Richards. Genetic algorithms and punctuated equilibria in VLSI. In H. Schwefel and R. Manner, editors, *Parallel Problem Solving from Nature*, pages 134–144, Berlin, 1991. Springer-Verlag.

[14] L. Davis. Adapting operator probabilities in genetic algorithms. In J. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 61–69, San Mateo, 1989. Morgan Kaufmann.

[15] L. Davis. *Handbook of Genetic Algorithms.* Van Nostrand Reinhold, New York, 1991.

[16] M. de la Maza and B. Tidor. An analysis of procedures with particular attention paid to proportional and Boltzmann selection. In S. Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 124–131, San Mateo, 1993. Morgan Kaufmann.

[17] K. DeJong. *An Analysis of the Behavior of a Class of Genetic Adaptive Systems.* PhD thesis, University of Michigan, Ann Arbor, 1975. Department of Computer and Communication Sciences.

[18] K. DeJong. Genetic algorithms are NOT function optimizers. In D. Whitley, editor, *Foundations of Genetic Algorithms -2-*, pages 5–17. Morgan Kaufmann, San Mateo, 1993.

[19] K. DeJong and W. Spears. An analysis of the interacting roles of population size and crossover in genetic algorithms. In H. Schwefel and R. Manner, editors, *Parallel Problem Solving from Nature*, pages 38–47, New York, 1991. Springer-Verlag.

[20] J. Eckstein. *Parallel Branch-and-Bound Algorithms for General Mixed Integer Programming on the CM-5.* Technical Report TMC-257, Thinking Machines Corp., 1993.

[21] M. Fischer and P. Kedia. Optimal Solution of Set Covering/Partitioning Problems Using Dual Heuristics. *Management Science*, 36(6):674–688, 1990.

[22] M. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, 21:948–960, 1972.

[23] T. Fogarty and R. Huang. Implementing the genetic algorithm on transputer based parallel processing systems. In H. Schwefel and R. Manner, editors, *Parallel Problem Solving from Nature*, pages 145–149, Berlin, 1991. Springer-Verlag.

[24] R. Garfinkel and G. Nemhauser. *Integer Programming.* John Wiley & Sons Inc., New York, 1972.

[25] I. Gershkoff. Optimizing Flight Crew Schedules. *INTERFACES*, 19:29–43, 1989.

[26] D. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Publishing Company, Inc., New York, 1989.

[27] D. Goldberg and K. Deb. A comparative analysis of selection schemes used in genetic algorithms. In G. Rawlins, editor, *Foundations of Genetic Algorithms*, pages 69–93. Morgan Kaufmann, San Mateo, 1991.

[28] S. Gordon and D. Whitley. Serial and parallel genetic algorithms as function optimizers. In S. Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 177–183, San Mateo, 1993. Morgan Kaufmann.

[29] M. Gorges-Schleuter. Explicit parallelism of genetic algorithms through population structures. In H. Schwefel and R. Manner, editors, *Parallel Problem Solving from Nature*, pages 150–159, New York, 1991. Springer-Verlag.

[30] J. Grefenstette. Optimization of Control Parameters for Genetic Algorithms. *IEEE Transactions on Systems, Man, and Cybernetics*, 16(1):122–128, 1986.

[31] J. Grefenstette and J. Baker. How genetic algorithms work: A critical look at implicit parallelism. In J. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 20–27, San Mateo, 1989. Morgan Kaufmann.

[32] J. Gregory. Private communication, 1991.

[33] J. Gregory. Private communication, 1994.

[34] W. Gropp and B. Smith. *Chameleon Parallel Programming Tools Users Manual.* Technical Report ANL-93/23, Argonne National Laboratory, 1993.

[35] F. Gruau and D. Whitley. Adding Learning to the Cellular Development of Neural Networks: Evolution and the Baldwin Effect. *Evolutionary Computation*, 1(3):213–233, 1993.

[36] K. Hoffman and M. Padberg. Solving Airline Crew-Scheduling Problems by Branch-and-Cut. *Management Science*, 39(6):657–682, 1993.

[37] J. Holland. *Adaption in Natural and Artificial Systems.* The University of Michigan Press, Ann Arbor, 1975.

[38] F. James. A Review of Pseudorandom Number Generators. *Computer Physics Communication*, 60:329–344, 1990.

[39] P. Jog, J. Suh, and D. Gucht. *Parallel Genetic Algorithms Applied to the Traveling Salesman Problem.* Technical Report No. 314, Indiana University, 1990.

[40] T. Kido, H. Kitano, and M. Nakanishi. A hybrid search for genetic algorithms: Combining genetic algorithms, tabu search, and simulated annealing. In S. Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, page 614, San Mateo, 1993. Morgan Kaufmann.

[41] B. Kroger, P. Schwenderling, and O. Vornberger. Parallel genetic packing of rectangles. In H. Schwefel and R. Manner, editors, *Parallel Problem Solving from Nature*, pages 160–164, Berlin, 1991. Springer-Verlag.

[42] T. Kuo and S. Hwang. A genetic algorithm with disruptive selection. In S. Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 65–69, San Mateo, 1993. Morgan Kaufmann.

[43] D. Levine. A genetic algorithm for the set partitioning problem. In S. Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 481–487, San Mateo, 1993. Morgan Kaufmann.

[44] G. Liepins and S. Baluja. *apGA: An Adaptive Parallel Genetic Algorithm.* Technical report, Oak Ridge National Laboratory, 1991.

[45] G. Marsaglia, A. Zaman, and W. Tseng. *Stat. Prob. Letter*, 9(35), 1990.

[46] R. Marsten. An Algorithm for Large Set Partitioning Problems. *Management Science*, 20:774–787, 1974.

[47] R. Marsten and F. Shepardson. Exact Solution of Crew Scheduling Problems Using the Set Partitioning Model: Recent Successful Applications. *Networks*, 11:165–177, 1981.

[48] H. Muhlenbein. Parallel Genetic Algorithms and Combinatorial Optimization. In O. Balci, R. Sharda, and S. Zenios, editors, *Computer Science and Operations Research*, pages 441–456. Pergamon Press, 1992.

[49] G. Nemhauser and L. Wolsey. *Integer and Combinatorial Optimization.* John Wiley & Sons, New York, 1988.

[50] R. Parker and R. Rardin. *Discrete Optimization.* Academic Press, San Diego, 1988.

[51] C. Pettey, M. Leuze, and J. Grefenstette. A parallel genetic algorithm. In J. Grefenstette, editor, *Proceedings of the Second International Conference on Genetic Algorithms and Their Applications*, pages 155–161, Hillsdale, New Jersey, 1987. Lawrence Erlbaum Associates.

[52] J. Pierce. Application of Combinatorial Programming to a Class of All-Zero-One Integer Programming Problems. *Management Science*, 15:191–209, 1968.

[53] D. Powell and M. Skolnick. Using genetic algorithms in engineering design optimization with non-linear constraints. In S. Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 424–431, San Mateo, 1993. Morgan Kaufmann.

[54] N. Radcliffe. Private communication, 1993.

[55] J. Richardson, M. Palmer, G. Liepins, and M. Hilliard. Some Guidelines for Genetic Algorithms with Penalty Functions. In J. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 191–197, San Mateo, 1989. Morgan Kaufmann.

[56] W. Siedlecki and J. Sklansky. Constrained genetic optimization via dynamic reward-penalty balancing and its use in pattern recognition. In J. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 141–150, San Mateo, 1989. Morgan Kaufmann.

[57] A. Smith and D. Tate. Genetic optimization using a penalty function. In S. Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 499–505, San Mateo, 1993. Morgan Kaufmann.

[58] W. Spears and K. DeJong. An Analysis of Multi-Point Crossover. In G. Rawlins, editor, *Foundations of Genetic Algorithms*, pages 301–315. Morgan Kaufmann, San Mateo, 1991.

[59] W. Spears and K. DeJong. On the virtues of parameterized uniform crossover. In R. Belew and L. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 230–236. Morgan Kaufmann, 1991.

[60] T. Starkweather, D. Whitley, and K. Mathias. Optimization Using Distributed Genetic Algorithms. In H. Schwefel and R. Manner, editors, *Parallel Problem Solving from Nature*, pages 176–185, New York, 1991. Springer-Verlag.

[61] G. Syswerda. Uniform crossover in genetic algorithms. In J. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 2–9, San Mateo, 1989. Morgan Kaufmann.

[62] R. Tanese. Parallel genetic algorithms for a hypercube. In J. Grefenstette, editor, *Proceedings of the Second International Conference on Genetic Algorithms and Their Applications*, pages 177–183, Hillsdale, New Jersey, 1987. Lawrence Erlbaum Associates.

[63] R. Tanese. Distributed genetic algorithms. In J. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 434–440, San Mateo, 1989. Morgan Kaufmann.

[64] D. Tate and A. Smith. Expected allele coverage and the role of mutation in genetic algorithms. In S. Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 31–37, San Mateo, 1993. Morgan Kaufmann.

[65] G. von Laszewsski and H. Muhlenbein. Partitioning a graph with a parallel genetic algorithm. In H. Schwefel and R. Manner, editors, *Parallel Problem Solving from Nature*, pages 165–169, Berlin, 1991. Springer-Verlag.

[66] D. Whitley. The GENITOR algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In J. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 116–121, San Mateo, 1989. Morgan Kaufmann.

[67] D. Whitley. An executable model of a simple genetic algorithm. In D. Whitley, editor, *Foundations of Genetic Algorithms -2-*, pages 45–62. Morgan Kaufmann, San Mateo, 1993.

[68] D. Whitley and T. Hanson. Optimizing neural networks using faster, more accurate genetic search. In J. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 391–396, San Mateo, 1989. Morgan Kaufmann.

[69] D. Whitley and J. Kauth. GENITOR: A different genetic algorithm. In *Rocky Mountain Conference on Artificial Intelligence*, pages 118–130, Denver, 1988.