A Parallel Genetic Algorithm for the Set Partitioning Problem*

David Levine
Argonne National Laboratory
Mathematics and Computer Science Division
9700 South Cass Avenue
Argonne, Illinois 60439, U.S.A.
levine@mcs.anl.gov

Abstract. This paper describes a parallel genetic algorithm developed for the solution of the set partitioning problem—a difficult combinatorial optimization problem used by many airlines as a mathematical model for flight crew scheduling. 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.

Keywords. Island model, combinatorial optimization, parallel computing, airline crew scheduling

*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.

1 Introduction

The past several years have seen an increasing number of reports of the successful application of genetic algorithms for solving optimization problems. 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 work was to unify these factors by developing a parallel genetic algorithm and applying it to the solution of the set partitioning problem—a difficult combinatorial optimization problem used by many airlines as a mathematical model for assigning flight crews to flights.

There were a number of motivations for developing a parallel genetic algorithm for the set partitioning problem (SPP). First is the particularly challenging nature of the SPP. The challenges include the NP-completeness of finding *feasible* solutions, and the enormous size of problems of current industrial interest. Second, because of its use as a model for crew scheduling by most major airlines, there is great practical value in developing a successful algorithm. Third, genetic algorithms can provide flexibility in handling variations of the SPP model that may be useful. The evaluation function can be easily modified to handle constraints such as cumulative flight time, mandatory rest periods, or limits on the amount of work allocated to a particular base not explicitly part of the SPP model. Fourth, 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." Finally, we believe genetic algorithms have great potential for scaling to take advantage of the larger and larger numbers of processors increasingly available on parallel computers.

The rest of this paper is laid out as follows. In Section 2 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. Section 3 describes the sequential genetic algorithm on top of which the parallel genetic algorithm was built. Section 4 describes the parallel genetic algorithm. Section 5 presents the parallel experiments we performed and discusses the results. Finally, Section 6 contains concluding remarks and suggests areas for further research.

2 The Set Partitioning Problem

The set partitioning problem (SPP) may be stated mathematically as

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

subject to

$$\sum_{j=1}^{n} a_{ij} x_j = 1 \quad \text{for } i = 1, \ldots, m \tag{2}$$

$$x_j = 0 \text{ or } 1 \qquad \text{for } j = 1, \ldots, n, \tag{3}$$

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 [12, 21] 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 jth column. $|P_j|$ is the cardinality of P_j. A set $J^* \subseteq J$ is called a *partition* if

$$\bigcup_{j \in J^*} P_j = I \tag{4}$$

$$j, k \in J^*, j \neq k \Rightarrow P_j \bigcap P_k = \emptyset. \tag{5}$$

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 in Sections 3.2 and 3.3. $R_i = \{j \in J | a_{ij} = 1\}$ is the (fixed) set of columns that intersect row i, while $r_i = \{j \in R_i | x_j = 1\}$ is the (changing) set of columns that intersect row i included in the current solution. Δ_{j_1} is the change in the evaluation function (see Section 3.2) as a result of setting x_j to one. Δ_j is the change in the evaluation function when complementing x_j. Δ_{j_1} and Δ_j measure both the cost coefficient, c_j, and the impact on constraint feasibility (see Section 3.2.)

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

$$a_{ij} = \begin{cases} 1 & \text{if flight leg } i \text{ is on rotation } j \\ 0 & \text{otherwise.} \end{cases} \tag{6}$$

Airline crew scheduling is a very visible and economically significant problem. Estimates of over a billion dollars a year for pilot and flight attendant expenses have been reported [1, 5]. 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" [22]. 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 [6].

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. Here we mention some of the more recent methods. See Balas and Padberg [4] for a survey of older methods.

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 [5, 13, 22] have noted that for "small" SPP problems the solution to the LP relaxation either is 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, 5, 13].

Branch-and-bound may be viewed as an exact approach if the algorithm runs until an integer solution (if one exists) is proven optimal, or as an approximate approach if the algorithm is terminated "early" with a "good" integer solution. Various bounding strategies have been used, including linear programming and Lagrangian relaxation. Fischer and Kedia [11] use continuous analogs of the greedy and $3-opt$ methods to provide improved lower bounds. Of recent interest is the work of Eckstein [10], 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. The most successful approach appears to be the work of Hoffman and Padberg. They present an exact approach based on the use of branch-and-cut—a branch-and-bound–like scheme with additional preprocessing and constraint generation at each node in the search tree. They report optimal solutions for a large set of real-world SPP problems [16].

3 The Sequential Genetic Algorithm

In this section we describe the sequential GA we used as the basis for the parallel genetic algorithm. The choice of algorithm, the selection of parameter settings, and the development of a local search heuristic to use with the sequential GA were the result of significant research and experimentation. Here, we summarize the sequential algorithm. The interested reader is referred to [18, 19] for additional details.

3.1 Problem Representation

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 $\mathbf{x}^*$ with the interpretation that x_j is one (zero) if bit j is one (zero) in the binary vector.

Representing an 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.

*We use $\mathbf{x}$ interchangeably as the solution to the SPP problem or as a bitstring in the GA population as in, for example, Figure 2.

3.2 Evaluation Function

The evaluation function measures "how good" a solution to the SPP problem a string is. This function needs to 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. However, the GA operators often produce infeasible solutions. In fact, since just finding a feasible solution to the SPP is NP-complete [23], it may be that many or most strings in the population are infeasible.

We used for our evaluation function

$$\sum_{j=1}^{n} c_j x_j + \sum_{i=1}^{m} \lambda_i \Phi_i(\mathbf{x}), \tag{7}$$

where

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

The first term is the SPP objective function, and the second term is the penalty function. The penalty function indicates whether a constraint is infeasible, but does not measure the magnitude of the infeasibility. The term λ_i is a scalar weight that penalizes constraint i's infeasibility.

Choosing a suitable value for λ_i is a difficult problem. A good choice for λ_i should reflect not just the "costs" associated with making constraint i feasible, but also the impact on other constraints (in)feasibility. In [25] Richardson et al. studied the choice of λ_i for the set covering problem (SCP). In the SCP, the equality in Equation (2) is replaced by a $\geq$ constraint. Unlike the SPP, however, the SCP 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. On the other hand, such an approach will not work with $|r_i| = 0$ 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.

We know of no method to calculate an optimal value for λ_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 [25], where it provided an upper bound on the cost to satisfy the violated constraints of the SCP. In the case of the SPP, however, the choice of λ_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 section this situation did happen.)

3.3 The ROW Heuristic

Our early experience with a generational replacement genetic algorithm [18], as well as subsequent experience with a steady-state genetic algorithm [19], was that both had trouble finding optimal (often even feasible) solutions. This result 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.

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

Figure 1: ROW Heuristic

The heuristic we developed is called ROW (since it takes a row-oriented view of the problem). The basic outline is given in Figure 1. ROW works as follows. For some number of iterations (the parameter *niters*), one of the m rows of the problem is selected by `choose_row` (either randomly or according to the largest infeasibility). For any row there are three possibilities: $|r_i| = 0$, $|r_i| = 1$, and $|r_i| > 1$. The action of `improve` in these cases varies and also varies according to whether we are using a best-improving or first-improving strategy. In the case of a best-improving strategy we apply one of the following rules.

1. $|r_i| = 0$: For each $j \in R_i$ calculate Δ_{j_1}. Set to one the column that minimizes Δ_{j_1}.
2. $|r_i| = 1$: Let k be the unique column in r_i. Calculate Δ'_j, the change in the evaluation function 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 $\Delta'_l < \Delta'_j$, $\forall j$.
3. $|r_i| > 1$: For each $j \in r_i$ calculate Δ''_j, the change in the evaluation function 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 $\Delta''_j < \Delta''_k, \forall k$.

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 $j \in R_i$ 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* modify the current solution to get feasible, we make a random move that makes constraint i feasible, without measuring all the implications (cost component and (in)feasibility of other constraints).

For the results presented in this paper we used the following settings for ROW. The number of iterations of ROW that were applied to try to improve a string was one. Choosing the constraint to apply ROW to was done randomly. A first-improving selection strategy was used.

3.4 Hybrid Steady-State Genetic Algorithm

After much experimentation [18, 19] we settled on an algorithm that hybridized the ROW heuristic with a steady-state genetic algorithm (SSGA). We call the hybrid algorithm SSGAROW. Figure 2 presents the specific implementation we used.

$P(t)$ is the population of strings at generation[†] t. Each generation one new string is inserted

[†]We use generation and iteration interchangeably.

```
t ← 0
initialize P(t)
evaluate P(t)
foreach generation
    ROW (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: Hybrid Steady-State Genetic Algorithm

into the population. The first step is to pick a random string, $\mathbf{x}_{random}$, and apply the ROW heuristic to it. Next, two parent strings, $\mathbf{x}_1$ and $\mathbf{x}_2$, are selected by holding two binary tournaments, and a random number, $r \in [0, 1]$, is generated. If r is less than the crossover probability of 0.6, we create two new offspring via uniform crossover with parameter 0.7 [27], 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 complement 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, $\mathbf{x}_{worst}$, is deleted, $\mathbf{x}_{new}$ is inserted, and the population is reevaluated.

4 The Parallel Genetic Algorithm

The parallel genetic algorithm we used is based on an island model. In population genetics an island model is one where separate and isolated *subpopulations* evolve independently and in parallel. 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 increases the overall selective pressure since additional reproductive trials are allocated to those strings that are fit enough to migrate [29]. 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.

An IMGA is characterized by several choices: the type of sequential GA run on each subpopulation, 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, and the logical topology the subpopulations are arranged in. The choice of "communication" parameters in the IMGA echoes the competing themes of selective pressure and population diversity in sequential GAs. Frequently migrating many fit strings and deleting the least fit strings 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 fixed 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, 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, ...). The sequential GA run on each subpopulation was SSGAROW.

To determine suitable values for the other communication parameters, we performed a limited set of experiments, described in [19]. To summarize, 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 parameter 0.4).

The IMGA we used is shown in Figure 3. The difference between Figure 3 and Figure 2 is the addition of the **if** block to determine whether a string is to be migrated this iteration. 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 $\mathbf{x}_{recv}$.

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. To test this, we implemented the algorithm described in Sections 3 and 4 and tested it on a parallel computer on a set of real-world SPP problems.

```
t ← 0
initialize P(t)
evaluate P(t)
foreach generation
    ROW (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: Island Model Genetic Algorithm

5.1 Computational Environment

The parallel computer we used for our experiments was an IBM SP1 with 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 uses a high-performance switch for connecting the nodes. The SP1 supports the distributed-memory programming model.

Our code was written in C and used the `Chameleon` [15] message-passing library. `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` ran on top of IBM's EUI-H message-passing software.

Random number generation was done using an implementation of the universal random number generator proposed by Marsaglia, Zaman, and Tseng [20], and translated to C from James' version [17]. Each time a parallel run was made, all subpopulations were randomly seeded. This was done by having one processor get and broadcast to all the other processors the microsecond portion of the value returned by the Unix `gettimeofday` system call. Each processor then added its processor id to this value and used the resulting unique value as its random number seed. For the random number generator in [20] each unique seed gives rise to an independent sequence of random numbers of size $\approx 10^{30}$ [17].

Each test 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). A run was terminated either when the optimal solution was found[‡] or when all subpopulations had performed 100,000 iterations.

5.2 Test Problems

To test the parallel genetic algorithm, we selected a subset of forty problems from the test set used by Hoffman and Padberg [16]. The test problems are given in Table 1, 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 2 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

[‡]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.

[§]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

Table 1: 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 2: 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
vnw19	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

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 2 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 3,000 columns (`nw33` and `nw09` have 3,068 and 3,103 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 the `kl` and `aa` models, `lp_solve` has considerably more difficulty and does not find any integer solutions.

5.3 Experimental Results

The results of our experiments are summarized in Tables 3–6. Table 3 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

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.

themselves are given in Table 4. Table 5 contains the first iteration on which some subpopulation found a feasible solution. Table 6 is similar except that it contains the first iteration on which some subpopulation found an optimal solution. In Table 6 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 5. `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.

One way of looking at Table 3 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 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 improved 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 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 1 is the large number of constraints in `aa01`, `aa04`, `aa05`, and `nw18` (we will return to `nw18` in a moment). We note from Table 2 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 7 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

Table 3: 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 4: 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 5: 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 6: 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.

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 7: 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†	84.5	62.2		56.2		
`aa01`	70.1	66.0	75.2	70.0	53.0	54.6	

Table 5 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. With 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 6 is similar to Table 5; 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's 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 8: 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.

Table 8 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 [16] (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 [16]. 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 8 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 8 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 8 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 5 that it was able to find other 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 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 8 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 8 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 [14] 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 [8, 9]. 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 section, 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.

6 Conclusions and Future Work

The SPP is a difficult problem for a genetic algorithm. The primary reason is that the SPP is highly constrained and a GA has difficulties finding feasible solutions. This is true for both

the generational replacement GA and the steady-state GA. A hybrid algorithm combining the steady-state GA with the SPP-specific ROW heuristic was more effective than either algorithm by itself and was able to find feasible (and sometimes optimal) solutions to the smaller SPP test problems.

The ROW heuristic is parameterized according to how much effort it should spend trying to improve a solution. In general, the most successful approach was to "work quicker, not harder" and to make random choices whenever possible. The ROW heuristic is effective at making local improvements, particularly with respect to infeasibilities, and the SSGA propagates these improvements to other strings thus having a global effect.

Using the hybrid SSGAROW algorithm 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.

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.

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 both 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.

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 [30]. Several researchers [7, 28] make the case for a high mutation rate when mutation is separated from crossover, as it is in our implementation.

We found that the random choice of columns to add or delete to the current solution 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.

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 [24], or the dynamic penalty of Smith and Tate [26] for which we had mixed results [19]. However, for the `aa` problems, we are less optimistic. Table 7 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.

Acknowledgments

A number of people helped in various ways during the course of this work. I thank Greg Astfalk, Bob Bulfin, Tom Canfield, Tom Christopher, Remy Evard, John Gregory, Bill Gropp, Karla Hoffman, John Loewy, Rusty Lusk, Jorge Moré, Bob Olson, Gail Pieper, Paul Plassmann, Nick Radcliffe, Xiaobai Sun, David Tate, and Stephen Wright. This paper is based on my Ph.D. thesis at Illinois Institute of Technology.

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

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

[6] 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.

[7] 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.

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

[9] 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.

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

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

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

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

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

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

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

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

[18] 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.

[19] D. Levine. *A Parallel Genetic Algorithm for the Set Partitioning Problem.* PhD thesis, Illinois Institute of Technology, Chicago, 1994. Department of Computer Science.

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

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

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

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

[24] 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.

[25] 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.

[26] 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.

[27] 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, San Mateo, 1991. Morgan Kaufmann.

[28] 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.

[29] 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.

[30] 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.