Application of a Hybrid Genetic Algorithm to Airline Crew Scheduling*

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

Scope and Purpose—Airline crew scheduling is a very visible and economically significant problem. Because of its widespread use, economic significance, and difficulty of solution, the problem has attracted the attention of the operations research community for over twenty-five years. The purpose of this paper was to develop a genetic algorithm for the airline crew scheduling problem, and to compare it with traditional approaches.

Abstract—This paper discusses the development and application of a hybrid genetic algorithm to airline crew scheduling problems. The hybrid algorithm consists of a steady-state genetic algorithm and a local search heuristic. The hybrid algorithm was tested on a set of forty real-world problems. It found the optimal solution for half the problems, and good solutions for nine others. The results were compared to those obtained with branch-and-cut and branch-and-bound algorithms. The branch-and-cut algorithm was significantly more successful than the hybrid algorithm, and the branch-and-bound algorithm slightly better.

1 Introduction

Genetic algorithms (GAs) are search algorithms that were developed by John Holland [17]. They are based on an analogy with natural selection and population genetics. One common application of GAs is for finding approximate solutions to difficult optimization problems. In this paper we describe the application of a hybrid GA (a genetic algorithm combined with a local search heuristic) to airline crew scheduling problems. The most common model for airline crew scheduling problems is the set partitioning problem (SPP)

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

*This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

subject to

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

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

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.

In airline crew scheduling, 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

$$a_{ij} = \begin{cases} 1 & \text{if flight leg } i \text{ is on rotation } j \\ 0 & \text{otherwise.} \end{cases} \tag{4}$$

Airline crew scheduling is an economically significant problem [1, 3, 11, 16] and often a difficult one to solve. One approximate approach (as well as the starting point for most exact approaches) is to solve the linear programming (LP) relaxation of the SPP. A number of authors [3, 11, 21] 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 [1, 3, 11].

Exact approaches are usually based on branch-and-bound, with bounding strategies such as linear programming and Lagrangian relaxation. Fischer and Kedia [10] used continuous analogs of the greedy and $3-opt$ methods to provide improved lower bounds. Eckstein 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 [9]. Desrosiers et al. developed an algorithm that uses a combination of Dantzig-Wolfe decomposition with restricted column generation [8]. Hoffman and Padberg report optimal solutions when they use branch-and-cut for a large set of real-world SPP problems [16].

Several motivations for applying genetic algorithms to the set partitioning problem exist. First, since a GA works directly with integer solutions, there is no need to solve the LP relaxation. Second, genetic algorithms can provide flexibility in handling variations of the model such as constraints on cumulative flight time, mandatory rest periods, or limits on the amount of work allocated to a particular base by modifying the evaluation function. More traditional methods may have trouble accommodating the addition of new constraints as easily. Third, at any iteration, genetic algorithms contain a population of possible solutions. As noted by Arabeyre et al. [2], "The knowledge of a family of good solutions is far more important than obtaining an isolated optimum." Fourth, the NP-completeness of finding feasible solutions in the general case [23] and the enormous size of problems of current industrial interest make the SPP a good problem on which to test the effectiveness of GAs.

2 The Hybrid Genetic Algorithm

Genetic algorithms work with a *population* of candidate solutions. In the original GAs of Holland [17] each candidate solution is represented as a string of bits, where the interpretation of the bit string is problem specific. Each bit string in the population is assigned a value according to a problem-specific fitness function. A "survival-of-the fittest" step selects strings from the old population randomly, but biased by their fitness. These strings recombine by using the crossover and mutation operators (see Section 2.4) to produce a new generation of strings that are (one hopes) more fit than the previous one.

2.1 Population Replacement

The generational replacement genetic algorithm (GRGA) replaces the entire population each generation by their offspring and is the traditional genetic algorithm defined by Holland [17]. The hope is that the offspring of the best strings carry the important "building blocks" [12] from the best strings forward to the next generation. The GRGA, however, allows the possibility that the best strings in the population do not survive to the next generation. Also, as pointed out by Davis [6], some 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.

The steady-state genetic algorithm (SSGA) is an alternative to the GRGA that replaces only a few individuals at a time, rather than an entire generation [26, 27]. In practice, the number of new strings created 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.

One advantage of the SSGA is that it 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. In earlier empirical testing [20] we found the SSGA more effective than the GRGA, and the results reported in this paper all use the SSGA with one individual replaced each generation, and a population size of 100.

2.2 Problem Data Structures

A solution to the SPP 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 $x_j = 1(0)$ 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.

We also ordered the SPP matrix into block "staircase" form [24]. 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. Ordering the matrix in this manner is helpful in determining feasibility. In any block, at *most* one x_j may be set to one. We use this fact in our initialization scheme (see Section 2.6.)

2.3 Evaluation Function

The obvious way to evaluate the fitness of a bit string is as a minimizer of Equation (1), the SPP objective function. However, since just finding a feasible solution to the SPP is NP-complete [23], and many or most strings in the population may be infeasible, Equation (1) alone is insufficient because it does not take into account infeasibilities. Our approach to this problem is to incorporate a penalty term into the evaluation function to penalize strings that violate constraints.

Such *penalty methods* allow constraints to be violated. Depending on the magnitude of the violation, however, a penalty (in our case proportional to the size of the infeasibility) is incurred that degrades the objective function. 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 [25].

As part of the work in [20] we investigated several different penalty terms without any conclusive results. For the results reported in this paper we used the *linear penalty term*

$$\sum_{i=1}^{m} \lambda_i \sum_{j=1}^{n} |a_{ij} x_j - 1| \,. \tag{5}$$

Here, λ_i is a scalar weight that penalizes the violation of constraint i. 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 constraint's (in)feasibility. We know of no method to calculate an optimal value for λ_i. Therefore, we made the empirical choice of setting λ_i to the largest c_j from the columns that intersected row 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 in the *set covering problem* (the equality in Eq. (2) is replaced by "$\geq$"). In the case of set partitioning, however, the choice of λ_i provides no such bound, and the GA may find infeasible solutions more attractive than feasible ones.

2.4 GA Operators

The primary GA operators are selection, crossover, and mutation. We choose strings for reproduction via *binary* tournament selection [12, 13]. Two strings were chosen randomly from the population, and the fitter string was allocated a reproductive trial. To produce an offspring we held two binary tournaments, each of which produced one parent string. These two parent strings were then recombined to produce an offspring.

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,

```
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 1: Two-Point Crossover

new and promising areas of the search space will be explored. Figure 1 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.

Mutation is applied in the traditional GA sense; it is a background operator that provides a theoretical guarantee that no bit value is ever permanently fixed to one or zero in all strings. In our implementation of mutation we complement a bit with probability $1/n$.

In our algorithm we apply crossover *or* mutation. To do this we select two parent strings, and generate a random number $r \in [0, 1]$. If r is less than the crossover probability, $p_c = 0.6$, we create two new offspring via two-point crossover and randomly select one of them to insert in the new population. Otherwise, we randomly select one of the two parent strings, make a copy of it, and apply mutation to it. In either case we also test the new string to see whether it duplicates a string already in the population. If it does, it undergoes (possibly additional) mutation until it is unique.

2.5 Local Search Heuristic

There is mounting experimental evidence [6, 18, 22] that hybridizing a genetic algorithm with a local search heuristic is beneficial. It combines the GA's ability to widely sample a search space with a local search heuristic's hill-climbing ability. Our early experience with the GRGA [19], as well as subsequent experience with the SSGA [20], was that both methods had trouble finding optimal (often even feasible) solutions. 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.

To present this heuristic, we define the following notation. Let $J = \{1, \ldots, n\}$ be a set of column indices. $R_i = \{j \in J | a_{ij} = 1\}$ 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 in the current solution.

The heuristic we developed is called ROW (since it takes a row-oriented view of the problem). The basic outline is given in Figure 2. ROW works as follows. For *niters* iterations (a parameter of the heuristic), one of the m rows of the problem is selected (either randomly or according to the constraint with the largest infeasibility). 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 (every point in the neighborhood is evaluated and the one that most improves the current solution is accepted as the move) or first-improving (the first

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

Figure 2: The ROW Heuristic

move found that improves the current solution is made) strategy. If we are using best-improving, we apply one of the following rules.

1. $|r_i| = 0$: For each $j \in R_i$ calculate Δ_{j_1}, the change in z when $x_j \leftarrow 1$. Set to one the column that minimizes Δ_{j_1}.
2. $|r_i| = 1$: Let k be the unique column in r_i. For each $j \in R_i, j \neq k$ calculate $\Delta^k_{j_1}$, the change in z when $x_k \leftarrow 0$ and $x_j \leftarrow 1$. If $\Delta^k_{j_1} < 0$ for at least one j, set $x_k \leftarrow 0$ and $x_j \leftarrow 1$ for the j that minimizes $\Delta^k_{j_1}$.
3. $|r_i| > 1$: For each $j \in r_i$ calculate Δ_j, the change in z when $x_k \leftarrow 0, \forall k \in r_i, k \neq j$. Set to one the column that minimizes Δ_j.

Strictly speaking, this is not a best-improving heuristic, since in cases 1 and 3 we can move to neighboring solutions that degrade the current solution. Nevertheless, we allow this situation because 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. In case 1 we select a *random* column j from R_i and set $x_j \leftarrow 1$. In case 2 we set $x_k \leftarrow 0$ and $x_j \leftarrow 1$ as soon as we find *any* $\Delta^k_{j_1} < 0, j \in R_i$. In case 3 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 cases 1 and 3, since we have no guarantee we will find a "first-improving" solution but we know that we must change the current solution to become feasible, we make a random move that at least makes constraint i feasible, without weighing all the implications (cost component and (in)feasibility of other constraints).

2.6 Initialization

The initial GA population is usually generated randomly. The intent is to sample many areas of the search space and let the GA discover the most promising ones. We developed a modified random initialization scheme. Block random initialization, based on a suggestion of Gregory [14], 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 $\overline{P}$, we expect the number of $x_j = 1$ in the optimal solution to be approximately $m/\overline{P}$. We use the ratio of $m/\overline{P}$ 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 ≥ 1, we set to one a single column in every block.

3 Computational Results

3.1 Test Problems

To test the hybrid algorithm we selected a subset of forty problems (most of the small- and medium-sized problems, and a few of the larger problems) from the Hoffman and Padberg test set [16]. These are real set partitioning problems provided by the airline industry. They are given in Table 1, where they have been sorted by increasing numbers of columns. 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. One reason we did not test all of the larger problems is that in practice (e.g., as in [16]) they are usually preprocessed by a matrix reduction feature that, for large problems, can significantly reduce the size of the problem. However, we did not have access to such a capability.

To try to gain some insight into the difficulty of the test problems, we solved them using the public-domain `lp_solve` program [4]. This program solves linear programming problems by using the simplex method and solves integer programming (IP) problems by using the branch-and-bound algorithm.

The columns in Table 1 are the name of the test problem, the number of rows and columns in the problem, the optimal objective function value for the LP relaxation, and the objective function value of the optimal integer solution, 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 1 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.

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. For the larger problems `lp_solve` results are mixed. On the `nw` problems (`nw07`, `nw06`, `nw11`, `nw18`, and `nw03`)

Table 1: Test Problems

Problem Name	No. Rows	No. Cols	LP Optimal	IP Optimal	LP Iters	LP Nonzeros	LP Ones	IP Nodes
nw41	17	197	10972.5	11307	174	7	3	9
nw32	19	294	14570.0	14877	174	10	4	9
nw40	19	404	10658.3	10809	279	9	0	7
nw08	24	434	35894.0	35894	31	12	12	1
nw15	31	467	67743.0	67743	43	7	7	1
nw21	25	577	7380.0	7408	109	10	3	3
nw22	23	619	6942.0	6984	65	11	2	3
nw12	27	626	14118.0	14118	35	15	15	1
nw39	25	677	9868.5	10080	131	6	3	5
nw20	22	685	16626.0	16812	1240	18	0	15
nw23	19	711	12317.0	12534	3050	13	3	57
nw37	19	770	9961.5	10068	132	6	2	3
nw26	23	771	6743.0	6796	341	9	2	11
nw10	24	853	68271.0	68271	44	13	13	1
nw34	20	899	10453.5	10488	115	7	2	3
nw43	18	1072	8897.0	8904	142	9	2	3
nw42	23	1079	7485.0	7656	274	8	1	9
nw28	18	1210	8169.0	8298	1008	5	2	39
nw25	20	1217	5852.0	5960	237	10	1	5
nw38	23	1220	5552.0	5558	277	8	2	7
nw27	22	1355	9877.0	9933	118	6	3	3
nw24	19	1366	5843.0	6314	302	10	4	9
nw35	23	1709	7206.0	7216	102	8	4	3
nw36	20	1783	7260.0	7314	74589	7	1	789
nw29	18	2540	4185.3	4274	5137	13	0	87
nw30	26	2653	3726.8	3942	2036	10	0	45
nw31	26	2662	7980.0	8038	573	7	2	7
nw19	40	2879	10898.0	10898	120	7	7	1
nw33	23	3068	6484.0	6678	202	9	1	3
nw09	40	3103	67760.0	67760	146	16	16	1
nw07	36	5172	5476.0	5476	60	6	6	1
nw06	50	6774	7640.0	7810	58176	18	2	151
aa04	426	7195	25877.6	26402	7428	234	5	>1
kl01	55	7479	1084.0	1086	26104	68	0	>37
aa05	801	8308	53735.9	53839	6330	202	53	>4
nw11	39	8820	116254.5	116256	200	21	17	3
aa01	823	8904	55535.4	56138	23326	321	17	>1
nw18	124	10757	338864.3	340160	162947	68	27	>62
kl02	71	36699	215.3	219	188116	91	1	>3
nw03	59	43749	24447.0	24492	4123	17	6	3

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.

3.2 ROW Heuristic Results

Creating a hybrid GA required selecting three parameters for the ROW heuristic. The first was the number of iterations to apply ROW. Since ROW can be computationally expensive, and empirical evidence in [20] showed no significant difference applying ROW for 1, 5, or 20 iterations, we fixed the number of iterations ROW was applied to one.

The two remaining parameters to select are the method for constraint selection and the search strategy (best-improving or first-improving). We studied this empirically using a subset of nine test problems from Table 1. This subset was selected so that we would have several smaller problems and a few larger ones.

Table 2 compares the best-improving (column *Best*) and first-improving (column *First*) strategies in conjunction with the method for constraint selection. *Random* means that one of the m constraints is selected randomly. *Max.* means that the constraint with the largest value of $|\sum_{j=1}^{n} a_{ij}x_j - 1|$ is selected. The columns *Opt.* and *Feas.* are the number of times out of ten independent trials an optimal or feasible solution was found.

The most obvious result in Table 2 is that no feasible solution was found on *any* run for *any* test problem with constraint selection via maximum violation. Random constraint selection helped significantly in finding feasible solutions, although not many optimal ones were found. It appears the randomness in cases one and three of the first-improving strategy helps escape from a locally optimal solution. The differences between the best-improving and first-improving strategies were not significant.

Table 2: ROW Heuristic: Best vs. First Improving

Problem Name	Best				First			
	Random		Max.		Random		Max.	
	Opt.	Feas.	Opt.	Feas.	Opt.	Feas.	Opt.	Feas.
`nw41`	0	10	0	0	0	10	0	0
`nw32`	0	10	0	0	1	10	0	0
`nw40`	0	10	0	0	0	10	0	0
`nw08`	0	4	0	0	0	10	0	0
`nw15`	0	2	0	0	3	10	0	0
`nw20`	0	10	0	0	0	10	0	0
`nw33`	0	10	0	0	0	10	0	0
`aa04`	0	0	0	0	0	0	0	0
`nw18`	0	0	0	0	0	0	0	0

Table 3 compares the best-improving and first-improving strategies using the hybrid of SSGA

in combination with the ROW heuristic (which we refer to as SSGAROW). In Table 3 constraint selection was done randomly. The results show the first-improving strategy was superior at finding optimal solutions. This is an interesting result since we could argue that we would expect exactly the opposite. That is, since the GA itself introduces randomness into the search, we would expect to do better applying crossover and mutation to the best solution found by ROW rather than the first, which is presumably not as good. A possible explanation is that the GA population has 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 3: SSGAROW: Best vs. First Improving

Problem	Best		First	
Name	Opt.	Feas.	Opt.	Feas.
nw41	6	10	9	10
nw32	0	10	4	10
nw40	2	10	7	10
nw08	0	2	0	4
nw15	1	10	5	10
nw20	0	10	1	10
nw33	0	10	1	10
aa04	0	0	0	0
nw18	0	0	0	0

3.3 Hybrid Genetic Algorithm Results

Table 4 compares the SSGA by itself, the ROW heuristic by itself, and the SSGAROW hybrid. SSGA and ROW both perform poorly with respect to finding optimal solutions. SSGA finds *no* optimal solutions, and ROW finds only four. SSGAROW, however, outperforms both ROW and SSGA. In this case, ROW makes good local improvements, and SSGA's recombination ability allows these local improvements to be incorporated into other strings, thus having a global effect.

Table 5 contains the results of applying SSGAROW to the test problems in Table 1. In these runs the ROW heuristic was applied to one randomly selected string each generation, a constraint was randomly selected, and a first-improving strategy was used. A run was terminated either when the optimal solution (which for the test problems was known) was found or after 10,000 iterations (in previous informal testing we had observed that most of the progress is made early in the search and the best solution found rarely changed after about 10,000 iterations).

To perform these experiments, for each problem in Table 1 we made ten independent runs. Each run was made on a single node of an IBM Scalable POWERparallel (SP) computer. For the IBM SP system used, a node consists of an IBM RS/6000 Model 370 workstation processor, 128 MB of memory, and a 1 GB disk.

The *No. Opt.* and *No. Feas.* columns are the number of times the optimal or feasible solution was found. The *Best Feas.* column is either the objective function value of the best feasible

Table 4: Comparison of Algorithms

Problem	SSGA		ROW		SSGAROW	
Name	Opt.	Feas.	Opt.	Feas.	Opt.	Feas.
nw41	0	10	0	10	9	10
nw32	0	10	1	10	4	10
nw40	0	10	0	10	7	10
nw08	0	0	0	10	0	4
nw15	0	1	3	10	5	10
nw20	0	10	0	10	1	10
nw33	0	4	0	10	1	10
aa04	0	0	0	0	0	0
nw18	0	0	0	0	0	0

solution found, or an "X" if no feasible solution was found. The first *% Opt.* column is the percentage from optimality of the best feasible solution. The entry is "O" if the best feasible solution found was optimal, the percentage from optimality if the best feasible solution was suboptimal, or "X" if no feasible solution was found. The *Avg. Feas.* column is the average objective function value of all the feasible solutions found. The second *% Opt.* column is the percentage from optimality of the average feasible solutions.

The results in Table 5 can be divided into two groups: those for which SSGAROW found feasible solution(s), and those for which it did not. The infeasible problems were nw08*, nw10, nw09, aa04, aa05, nw11, aa01, and nw18.

For the problems in the first group, feasible solutions were found almost every run. Optimal solutions were found on average one fifth of the time. Except for four problems (nw12, nw06, kl02, and nw03), the best feasible solution was within three percent of optimality. The average feasible solution varied, but was typically within five percent of optimality for the smaller problems, within ten percent of optimality for medium-sized problems, and above ten percent of optimality for the larger problems.

For the second group of problems, SSGAROW was unable to find any feasible solutions. These problems can be subdivided into those where SSGAROW never found a better (infeasible) solution than the optimal one, and those where it did. In the former class are nw09, aa04, aa05, aa01, and nw18. One point to note from Table 1 is the large number of constraints in aa01, aa04, aa05, and nw18. Also, we note from Table 1 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 the three aa problems, on average at the end of a run the best (infeasible) solution found had an evaluation function value approximately four times that of the optimal solution. For nw09 and nw18, however, the best (infeasible) solution found was within five to ten percent of the optimal solution, but still no feasible solutions were found. For these five problems between

*Although a feasible solution was found for nw08 in one run, the performance characteristics are most closely related to nw10 and nw11, two infeasible problems.

Table 5: Hybrid Genetic Algorithm Results

Problem Name	No. Opt.	No. Feas.	Best Feas.	% Opt.	Avg. Feas.	% Opt.
nw41	10	10	11307	0	11307	.000
nw32	1	10	14877	0	15238	.024
nw40	0	10	10848	.004	10872	.006
nw08	0	1	37078	.033	37078	.033
nw15	8	10	67743	0	67743	.000
nw21	1	10	7408	0	7721	.042
nw22	0	10	7060	.011	7830	.121
nw12	0	10	15110	.063	15645	.108
nw39	4	10	10080	0	10312	.023
nw20	0	10	16965	.009	17618	.048
nw23	6	10	12534	0	12577	.003
nw37	5	10	10068	0	10257	.019
nw26	1	10	6796	0	7110	.046
nw10	0	0	X	X	X	X
nw34	5	10	10488	0	10616	.012
nw43	0	10	9146	.027	9620	.080
nw42	2	10	7656	0	8137	.062
nw28	6	10	8298	0	8509	.025
nw25	1	10	5960	0	6709	.125
nw38	6	10	5558	0	5597	.007
nw27	1	10	9933	0	10750	.082
nw24	1	10	6314	0	7188	.138
nw35	2	10	7216	0	7850	.088
nw36	0	10	7336	.003	7433	.016
nw29	0	10	4378	.024	4568	.069
nw30	4	10	3942	0	4199	.065
nw31	3	10	8038	0	8598	.070
nw19	0	10	11060	.015	12146	.115
nw33	1	10	6678	0	7128	.067
nw09	0	0	X	X	X	X
nw07	1	10	5476	0	6375	.164
nw06	0	5	9802	.255	16679	1.136
aa04	0	0	X	X	X	X
kl01	0	10	1110	.022	1134	.044
aa05	0	0	X	X	X	X
nw11	0	0	X	X	X	X
aa01	0	0	X	X	X	X
nw18	0	0	X	X	X	X
kl02	0	9	232	.059	241	.100
nw03	0	9	26622	.087	33953	.386

twelve and twenty-three percent of the constraints remained infeasible.

For `nw08`, `nw10`, and `nw11`, SSGAROW was able to find infeasible strings with lower evaluation function values than the optimal solution and had concentrated its search on those strings. For these problems the penalty term used in the evaluation function was not strong enough, and the GA exploited that.

Table 6 compares the solutions to the test problems found by `lp_solve`, the branch-and-cut algorithm of Hoffman and Padberg [16] (column *HP*), and the SSGAROW algorithm. The entries are "O" if the optimal solution was found, the percentage from optimality if the best feasible solution was suboptimal, or an "X" if no feasible solution was found. The `lp_solve` results were obtained on an IBM RS/6000 Model 590 workstation. Hoffman and Padberg's results are from Table 3 in [16] where the runs were made on an IBM RS/6000 Model 550 workstation. For SSGAROW, the entries are the fifth column from Table 5.

The branch-and-cut algorithm obtained the best results, solving *all* problems to optimality. For these results, however, a matrix reduction preprocessing step was applied to the test problems to reduce their size (in addition to the matrix reduction done as part of the branch-and-cut algorithm itself), that was not available to either `lp_solve` or SSGAROW. `lp_solve` found optimal (or in the case of `nw18` near-optimal) solutions to all but five of the larger problems (for which no feasible solutions were found). SSGAROW found optimal solutions to twenty problems, and solutions within five percent of optimality for nine others. Of the other eleven problems, feasible solutions greater than five percent of optimality were obtained for four of them, and for the seven others, no feasible solution was found.

We have not reported the actual solution times, however, since each algorithm was run on a different model IBM workstations. However, if we take into account the relative performance of the different processors [20] we can make some general comments about computational performance. The branch-and-cut algorithm was significantly faster than both `lp_solve` and SSGAROW. For the problems where both `lp_solve` and SSGAROW found optimal solutions, the `lp_solve` solution times were slightly faster than SSGAROW when the faster processor `lp_solve` ran on is factored in.

4 Conclusions

The SSGA alone was sometimes successful at finding feasible solutions, but not at finding optimal solutions. This motivated us to develop the ROW heuristic to hybridize with the SSGA. Two important parameters of ROW are how constraint selection is performed, and how to select a move to make. We found random constraint selection and a first-improving strategy (which also introduces randomness) were more successful than attempts to apply ROW to the most infeasible constraint or find the best-improving solution. Key points about ROW are its ability to make moves in large neighborhoods, its willingness to move downhill to escape infeasibilities, and the randomness introduced by random constraint selection and the first-improving strategy. However, when all constraints are feasible ROW no longer introduces any randomness, since it is in a "true" first-improving mode and all moves examined degrade the current solution, so ROW remains trapped at a local optima.

Table 6: Comparison of Solution Time

Problem	lp_solve	HP	SSGAROW
nw41	O	O	O
nw32	O	O	O
nw40	O	O	.004
nw08	O	O	.033
nw15	O	O	O
nw21	O	O	O
nw22	O	O	.011
nw12	O	O	.063
nw39	O	O	O
nw20	O	O	.009
nw23	O	O	O
nw37	O	O	O
nw26	O	O	O
nw10	O	O	X
nw34	O	O	O
nw43	O	O	.027
nw42	O	O	O
nw28	O	O	O
nw25	O	O	O
nw38	O	O	O
nw27	O	O	O
nw24	O	O	O
nw35	O	O	O
nw36	O	O	.003
nw29	O	O	.024
nw30	O	O	O
nw31	O	O	O
nw19	O	O	.015
nw33	O	O	O
nw09	O	O	X
nw07	O	O	O
nw06	O	O	.255
aa04	X	O	X
kl01	X	O	.022
aa05	X	O	X
nw11	O	O	X
aa01	X	O	X
nw18	.011	O	X
kl02	X	O	.059
nw03	O	O	.087

We ran ten independent trials of SSGAROW on each of forty real-world SPP problems. We found the optimal solution at least once for half of these problems, and solutions within five percent of optimality for nine others. For eight problems, we were unable to find any feasible solutions. In three of these cases the penalty term was not strong enough and the GA exploited this by concentrating its search among infeasible strings with lower evaluation function values than the optimal solution.

We compared SSGAROW with branch-and-cut and branch-and-bound algorithms. The branch-and-cut algorithm was the most successful, solving all problems to optimality, and in much less time than either SSGAROW or branch-and-bound. SSGAROW was also outperformed by branch-and-bound, but not as significantly. In fact, SSGAROW found feasible solutions for two of the larger test problems when branch-and-bound did not.

It is not surprising that genetic algorithms are outperformed by traditional operations research algorithms. GAs are general-purpose tools that will usually be outperformed when specialized algorithms for a problem exist [6, 7]. However, several points are worth noting. First, the `nw` models are relatively easy to solve with little branching and may not by indicative of current SPP problems airlines would like to solve [15]. Second, genetic algorithms map naturally onto parallel computers and we expect them to scale well with large numbers of processors. Our parallel computing experience [20] is very promising in this regard.

Several areas for possible enhancement exist. First, for several problems the penalty function was not strong enough and the GA exploited this by searching among infeasible strings with lower evaluation function values than feasible strings. Research into stronger penalty terms or other methods for solving constrained problems with genetic algorithms is warranted. Second, using an adaptive mutation rate or simulated-annealing-like move in the ROW heuristic to maintain diversity in the population in order to sustain the search warrants further investigation. Recently, Chu and Beasley [5] have shown that preprocessing the constraint matrix to reduce its size, as well as modifications to the fitness definition, parent selection method, and population replacement scheme lead to improved performance solving SPP problems.

Acknowledgments

Parts of this paper are based on my Ph.D. thesis at Illinois Institute of Technology, 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. I also thank two anonymous referees for their helpful comments.

References

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

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

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

[4] 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`.

[5] P. Chu and J. Beasley. *A Genetic Algorithm for the Set Partitioning Problem.* Technical report, Imperial College, 1995.

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

[7] 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.

[8] J. Desrosiers, Y. Dumas, M. Solomon, and F. Soumis. *The Airline Crew Pairing Problem.* Technical Report G-93-39, Université de Montréal, 1993.

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

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

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

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

[13] 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.

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

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

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

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

[18] 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.

[19] 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.

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

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

[22] 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.

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

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

[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] 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.

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