A DUAL GENETIC ALGORITHM FOR BOUNDED INTEGER PROGRAMS

James C. Bean
Atidel Ben Hadj-Alouane

Department of Industrial and Operations Engineering
The University of Michigan
Ann Arbor, MI 48109-2117

Technical Report 92-53

October 1992
Revised June 1993

A Dual Genetic Algorithm for Bounded Integer Programs*

James C. Bean
Atidel Ben Hadj-Alouane

Department of Industrial and Operations Engineering
University of Michigan
Ann Arbor, MI 48109-2117

June 24, 1993

Abstract

We present an algorithm for bounded variable integer programs that finds an optimal solution with probability one (though it is typically used as a heuristic). Constraints are relaxed by a nonlinear penalty function for which the corresponding dual has weak and strong duality. The dual problem is attacked by a genetic algorithm. Nontraditional reproduction, crossover and mutation operations are employed. Computational tests compare the genetic algorithm with optimal solution by OSL on dual degenerate problems.

KEYWORDS: Integer programming, genetic algorithms, duality

Résumé

Nous proposons un algorithme pour la résolution des programmes linéaires en variables entières et bornées. Cet algorithme trouve une solution optimale avec probabilité égale à un (quoiqu'il soit utilisé comme une heuristique). Une relaxation des contraintes est pénalisée par une fonction nonlinéaire dont le problème correspondant possède les propriétés de faible et forte dualité. Le problème dual est résolu avec un algorithme génétique. La convergence vers une solution optimale est garantie grâce à des operations de reproduction, croisement et mutation non traditionelles. Une expérimentation sur des problèmes degenerés permet de comparer l'algorithme génétique avec les solutions optimales du logiciel OSL.

MOTS CLEFS: Programmation en nombres entiers, algorithmes génétiques, dualité.

---

*This work was supported in part by the National Science Foundation under Grants DDM-9018515 and DDM-9202849 to the University of Michigan.

1 Introduction

The bounded variable integer program can be stated mathematically:

$$\begin{array}{lll} & \min & cx \\ (P) & s.\ to & Ax - b \geq 0 \\ & & 0 \leq x_j \leq u_j, \text{ integer}, \end{array}$$

where A is a $k \times n$ coefficient matrix, b is a constant vector in $\Re^k$ and u is a finite vector in $\Re^n$. Note that problems with nonzero lower bounds can easily be translated to the form in (P).

If special structure is present, it is common to use Lagrangian relaxation ([5] [4]). Lagrangian relaxation drops some constraints from the problem while introducing to the objective a weighted linear penalty for constraint deviation. Choosing correct weights in this penalty function can result in good bounds or even optimal solutions to the original problem. An extreme Lagrangian relaxation relaxes all general constraints resulting in the simple problem:

$$\begin{array}{lll} & \min & cx - \lambda(Ax - b) \\ (PR_\lambda) & s.\ to & 0 \leq x_j \leq u_j, \text{ integer}, \end{array}$$

where $\lambda^T \geq 0$ is a vector in $\Re^k$. This type of relaxation typically fails to give reasonable solutions since, having the integrality property ([5]), it can only return solutions with $x_j = 0$ or $x_j = u_j$.

Genetic algorithm approaches to this type of problem include constraints elimination [9] and also penalty function techniques. In [3] we address the multiple-choice integer program, a special case of (P), and adapt a nonlinear penalty function, $p_\lambda(x)$, commonly used in continuous nonlinear programming (see [1]). This penalty consists of summing the weighted squares of violations of the constraints. The penalty has the form

$$p_\lambda(x) = \sum_{i=1}^{k} \lambda_i [\min(0, A_{i.}x - b_i)]^2.$$

In this paper we extend this approach to bounded variable integer programs. Figure 1 shows this function for a single constraint. Adding $p_\lambda(x)$ to the objective function gives

the nonlinear integer program:

$$(PP_\lambda) \quad \begin{array}{ll} \min & cx + p_\lambda(x) \\ s.\ to & 0 \leq x_j \leq u_j, \text{ integer} \end{array}$$

where $\lambda \geq 0$ is a vector in $\Re^k$. Note that $p_\lambda(x)$ is convex and continuously differentiable. The relaxation (PP_λ) does not have the integrality property. Below we show that multi-

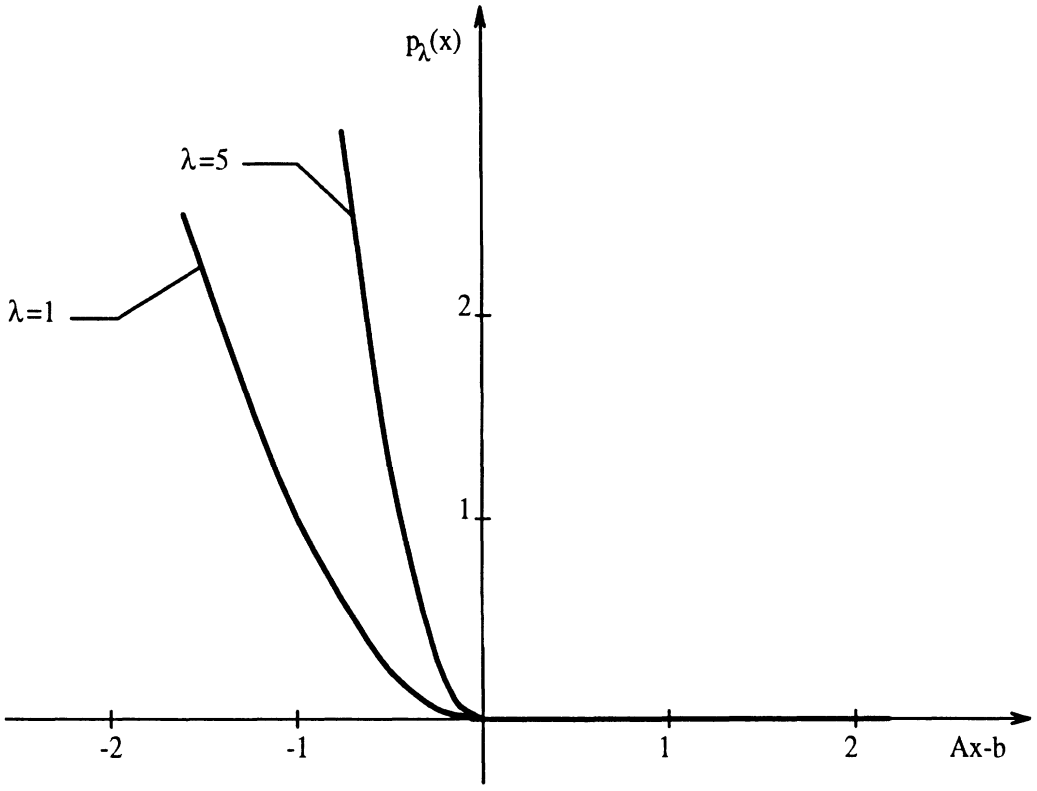

Figure 1: Nonlinear penalty function versus constraint violation

pliers exist such that (PP_λ) solves (P) exactly. This is a stronger result than is shown for nonlinear continuous programs or for traditional linear Lagrangian relaxation for integer programming.

However, (PP_λ) is nonlinear and, hence, more difficult to solve than (PR_λ). We present a genetic algorithm that handles this nonlinearity. Further, we note that it will solve (PP_λ) optimally with probability one.

The remainder of this paper is organized as follows. The second Section proves weak and strong duality for the dual corresponding to the nonlinear relaxation. Section 3

describes the genetic algorithm for (PP_λ). Section 4 evolves this into a genetic algorithm for solving (P). The fifth Section presents computational results. Finally, Section 6 presents a summary.

2 Validity of Nonlinear Relaxation

As in linear Lagrangian relaxation, the nonlinear relaxation provides some important theoretical tools that can be used to evaluate its solutions. The following results are extensions of results in [3]. Lemma 1 proves weak duality.

Lemma 1 (Weak Duality) $v(PP_\lambda) \leq v(P)$ *for all* $\lambda \geq 0$.

Proof: Let x^* be optimal to (P). Then x^* is feasible in (PP_λ) and $p_\lambda(x^*) = 0$ for all λ. Hence $v(PP_\lambda) \leq cx^* = v(P)$. ∎

The following Theorem describes the relationship between $v(PP_\lambda)$ and $v(P)$ for a fixed λ.

Theorem 1 (Fixed λ)

a- For a given $\lambda \geq 0$, *if* x *is optimal to* (PP_λ) *and* $Ax - b \geq 0$, *then* x *is optimal to* (P).

b- For a given $\lambda \geq 0$ *and* $\varepsilon > 0$, *if* x *is* ε*-optimal to* (PP_λ) *and* $Ax - b \geq 0$, *then* x *is* ε*-optimal to* (P).

Proof:

a- Since x is optimal to (PP_λ) then $v(PP_\lambda) = cx + p_\lambda(x)$. Since x is feasible in (P), then $cx \geq v(P)$ and $p_\lambda(x) = 0$ so $v(PP_\lambda) = cx$. By Lemma 1, $cx \leq v(P)$. Hence $v(P) = cx$.

b- Since x is ε-optimal to (PP_λ) and $Ax - b \geq 0$ then $cx - v(PP_\lambda) \leq \varepsilon$. By Lemma 1, $cx - v(P) \leq \varepsilon$.

That completes the proof. ∎

The following theorem establishes the existence of some value of the vector λ for which the hypotheses of Theorem 1-a are satisfied. This establishes strong duality for the dual $\max_{\lambda \geq 0} v(PP_\lambda)$. Denote S_λ as the set of optimal solutions to (PP_λ).

Theorem 2 (Strong Duality) *If (P) is feasible, then there exists $\bar{\lambda} \geq 0$ such that, for all $\lambda \geq \bar{\lambda}$, there exists $x \in S_\lambda$ such that $p_\lambda(x) = 0$ and x is optimal to (P).*

Proof: Let $X = \{x \in \Re^n : 0 \leq x_j \leq u_j, x_j \text{ integer}\}$, $Y = \{x \in \Re^n : Ax - b \geq 0\}$ and x^* be an optimal solution to (P). Let $\bar{Y}$ be the complement of Y.

$$\begin{aligned} v(PP_\lambda) &= \min_{x \in X}\{cx + p_\lambda(x)\} \\ &= \min\{\min_{x \in (X \cap Y)}(cx + p_\lambda(x)); \min_{x \in (X \cap \bar{Y})}(cx + p_\lambda(x))\} \\ &= \min\{cx^*; \min_{x \in (X \cap \bar{Y})}(cx + p_\lambda(x))\}. \end{aligned}$$

Note that, by assumption, X is a finite set and (PP_λ) and (P) are bounded. Define $\alpha(x) = \sum_{i=1}^{k}[\min(0, A_{i.}x - b_i)]^2$ and $\lambda_0 = \max_{x \in X, \alpha(x) \neq 0}\{\frac{cx^* - cx}{\alpha(x)}\} \in \Re$.
Choose $\bar{\lambda} = \max\{\lambda_0; 0\}e_k$, where $e_k = (1, 1, \ldots, 1)^T \in \Re^k$.
Then,

$$v(PP_{\bar{\lambda}}) = \min\{cx^*; \min_{x \in X, \alpha(x) \neq 0}(cx + \lambda_0 \alpha(x))\}.$$

For any $\lambda \geq \bar{\lambda}$ let $\bar{x}_\lambda = \arg\min_{x \in (X \cap \bar{Y})}(cx + p_\lambda(x))$. Since $\lambda \geq \bar{\lambda}$,

$$\begin{aligned} c\bar{x}_\lambda + p_\lambda(\bar{x}_\lambda) &\geq c\bar{x}_\lambda + \lambda_0 \alpha(\bar{x}_\lambda) \\ &\geq c\bar{x}_\lambda + (\frac{cx^* - c\bar{x}_\lambda}{\alpha(\bar{x}_\lambda)})\alpha(\bar{x}_\lambda) \\ &= cx^*. \end{aligned}$$

Therefore, $v(PP_\lambda) = cx^*$, i.e., $x^* \in S_\lambda$. Since $p_\lambda(x^*) = 0$, the proof is completed. ∎

The nonlinear relaxation is stronger than the analogous result from nonlinear programming. The latter eliminates duality gap only asymptotically. Further, it is stronger than the traditional Lagrangian relaxation due to the lack of a duality gap. The drawback is that the resultant problem, (PP_λ), is a nonlinear integer program.

This difficulty is overcome by the use of a genetic algorithm to solve (PP_λ). It solves the nonlinear problem with nearly the facility of a linear problem. The tradeoff is possible loss of optimality.

3 A Genetic Algorithm for (PP_λ)

Application of genetic algorithms to operations research problems has been limited due to the complex feasible domains. Given an optimization problem, often the hardest step in applying a genetic algorithm is encoding the solutions as strings so that crossovers of feasible solutions result in feasible solutions.

In [2], a special encoding technique called random keys is shown to be successful for a variety of sequencing and optimization problems. It represents a solution with random numbers which are used as sort keys to decode the solution. For (PP_λ), a more direct encoding can be used. A solution can be represented by a string of length equal to the number of variables. Each position, j, of the string can take any integer in $\{0, \ldots, u_j\}$. This is illustrated by the following example.

Example 1 *Let $n = 4$ and $u_j = 3$ for all j. Then one string is an element of $\times_{j=1}^{4}\{0,1,2,3\}$. The particular instance $(0,3,2,1)$ represents the solution $x_1 = 0, x_2 = 3, x_3 = 2, x_4 = 1$.*

This large alphabet encoding (see [6]), also referred to as the real coding [7] or floating point coding [8], is an alternative to binary encodings. The experimental study in [8] praises the floating point coding, especially when used with special genetic operators, since it has the advantage of shorter representations. In our problem, it also has the characteristic that, after relaxation of the general constraints, there are no interactions between genes other than through the value function. Results in section 5 suggest that this is quite effective. A broader study is underway into the reasons for and limitations of this effectiveness.

In this section a genetic algorithm is applied to the problem (PP_λ). Since (PP_λ) has only upper bound constraints, using the encoding above, each string in a randomly

generated generation is guaranteed to be a feasible solution. Basic crossover or mutation operations produce feasible solutions.

There are many variations of genetic algorithms formed by using different reproduction, crossover and mutation operators [6]. The genetic algorithm employed here has proved to be very successful for many integer programs. It is virtually identical to that in [3] but is restated here for completeness. Given a current generation, the next generation is created as follows.

1. *Clone* N_r top solutions. N_r is usually fixed to 10% of the population size. This approach, called elitist reproduction [6], replaces the traditional probabilistic reproduction. The advantage of using an elitist reproduction is that the best solution is monotonically improving from one generation to another.

2. Bernoulli Crossover: As illustrated by the example below, first, randomly select two strings (parents) from the old population. Next, create two offspring by tossing a biased coin for each locus to decide which parent will contribute the allele. Then include in the new generation only the offspring with the better objective value. This is similar to the uniform crossover described in [6], and is equivalent to the parameterized uniform crossover introduced in [12]. We prefer the term *Bernoulli crossover*. Experiments have shown that for many integer programs this operator is computationally better than the one-point or two-point crossover. Intuitively, the success of this operator can be explained as follows. On one hand, it introduces a diversification in the search space since it is basically similar to the multi-point crossover; on the other hand, it conserves "good" substrings (building blocks), in the long run, since the two mates do not have the same probability of contributing alleles.

3. Create *mutations* by randomly generating a small number of entire solutions (N_m = 1%) and including them in the new generation. This operation is clearly different from the gene by gene mutation since it involves bringing new members to the population. In [2] this is referred to as "immigration" and is shown to have an

important role in preventing premature convergence of the population, especially when it is used with the elitist reproduction as opposed to the probabilistic reproduction. While this is not a typical mutation operator, it has a similar effect due to the Bernoulli crossover. When, in a subsequent generation, an immigrant is crossed over with another element of the population, its randomly generated alleles are inserted at some genes as if they had mutated. It also prevents loss of genetic material as in traditional mutation while strongly opposing convergence.

Example 2 *Let $n = 4$ and $u_j = 3$ for all j as in Example 1. Two particular instances are*

$$(0, 3, 2, 1); \quad (2, 3, 1, 2).$$

To cross over these individuals as described above, toss a coin for each element of the string. Suppose a heads selects from the first and a tails from the second to form the first offspring and the remaining alleles form the other offspring. If the outcome of the coins are H, T, H, T then the two offspring are

$$(0, 3, 2, 2); \quad (2, 3, 1, 1).$$

Based purely on the sampling of the immigration operator we can state the following convergence result. The proof is trivial and is omitted. Let $\bar{x}^k$ be the best solution found through k generations.

Theorem 3 *If there is immigration of at least one member of the population each generation, then $\lim_{k\to\infty} P(\bar{x}^k \in S_\lambda) = 1$. That is, if the genetic algorithm above is run long enough, the probability that one finds an optimal solution approaches one.*

4 Varying λ to Solve (P)

Combining Theorems 2 and 3 we have the following theoretical algorithm.

- Set λ large enough to ensure that (PP_λ) solves (P) optimally.

- Run the genetic algorithm long enough to find an optimal solution to (PP_λ), and hence, (P).

The difficulties with implementing this procedure are 1) we do not know how long the genetic algorithm must be run, and, 2) it is usually difficult to determine a "good" value of λ, for a particular problem. Further, it has been seen empirically that for large values of λ, the search is limited only to feasible solutions, since infeasible solutions are highly penalized. This usually leads to an inefficient genetic algorithm, especially in case of problems with complex feasible domains. We describe below an implementation that seeks good solutions to (P) by concurrently adjusting the λ vector while running the genetic algorithm. If run long enough, the procedure will find an optimal solution to (P) with probability one. However, it is typically stopped heuristically. This algorithm is nearly identical to that in [3]. It is repeated here for completeness.

A sequence of increasing λ vectors is employed, i.e. we fix λ at a relatively small value, run the genetic algorithm for a certain number of generations, N_f, then halt to check whether the top solution has had a zero or nonzero penalty for these N_f generations. Typically, if λ is small enough, the top solution will have nonzero penalty (infeasible). In this case, λ is increased, then all solutions of the current generation are re-evaluated with the new λ, and the algorithm continues similarly. Our experience has shown that the rate at which this sequence is increased is crucial: a slow rate would improve the quality of solutions at the expense of rate of improvement; a fast rate may result in an inefficient genetic evolution. We adopt the following pragmatic strategy: use a sequence of initially increasing, then alternately decreasing and increasing, λ. In order to avoid cycling, it is important that the increasing and decreasing rates be different. Typically, we use an increasing rate (β_1) that is larger than the decreasing rate (β_2), to allow for a fast improvement at the early stages of the algorithm.

It is important to note that, in some cases, the initial value of λ is not small enough, and all the N_f consecutive top solutions may have zero penalties (feasible). In this situation, λ is decreased to allow for infeasible solutions to "compete" with feasible solutions, and hence diversify the search.

A pseudocode that describes this procedure is given below.

Initialization. Choose two scalars $\beta_1 > \beta_2 > 1$. Choose an initial vector, λ_1 and a frequency, N_f, for altering λ.
Randomly generate a population of solutions and evaluate objective values. Set $k = 1$.

Main Step. While (not stop)
BEGIN
Create a new generation using the genetic algorithm described above.
If the last N_f consecutive generations have top solution with nonzero penalty, let $\lambda_{k+1} = \beta_1 \lambda_k$, and re-evaluate current generation with λ_{k+1}.
If the last N_f consecutive generations have top solution with zero penalty, let $\lambda_{k+1} = \lambda_k / \beta_2$, and re-evaluate current generation with λ_{k+1}.
Otherwise $\lambda_{k+1} = \lambda_k$.
$k = k + 1$.
Stop if target value or generation count is reached.
END

In the above algorithm, the value of the frequency, N_f, is important. To allow for the genetic algorithm to reach a certain equilibrium for a given value of λ, a practical value is $N_f = \max\{50, \max_j u_j\}$.

Also note that different starting values of λ may result in different genetic algorithm runs and may effect its performance. However, the robustness of the above procedure comes from its ability to adjust the value of λ whether its initial value is small or large.

There is an alternative strategy to the varied λ; it consists of finding a "good" estimate of λ that results in hybrid populations (contain feasible and infeasible solutions), and use only that value to run the genetic algorithm. In [10], the problem of constructing adequate penalty functions is discussed and some guidelines are given. In our case, the adequacy of the penalty function is dependent on the value of λ. Theorem 2 suggests that this value may be estimated using $\bar{\lambda} = \max\{\lambda_0; 0\} e_k$, where $\lambda_0 = \max_{x \in X, \alpha(x) \neq 0} \{\frac{cx^* - cx}{\alpha(x)}\}$, as defined previously. Several ways can be employed to estimate λ_0, e.g. using the

linear programming dual solution, and/or some averaging techniques. This alternative procedure is a subject of an ongoing research and further details will appear elsewhere.

Computational tests of this procedure have been very successful. The results of this work are reported in the next section.

5 Computational Results

We present computational results for four sets of problems. The first two sets contain 10 zero-one integer programs and 10 general bounded variable integer programs (with $\max_i u_i = 10$); all 20 problems have 50 variables and 5 general constraints. The next two sets are constructed similarly. They contain 20 larger problems of 250 variables and 25 general constraints. A detailed description is presented in Table 1.

All test problems were randomly generated. The values of c and A were independently generated using the uniform distribution over the integer interval $[0, 1000]$. The uniform distribution over $[1, max_j u_j]$ was used to generate the upper bounds on the variables. The right hand sides were calculated as

$$b_i = \sum_{j=1}^{n} (u_j a_{ij})/2, \qquad i = 1, \ldots, k.$$

Similar experimental design can be found in [11]. These problems were made dual degenerate by creating a random number of duplicate variables. See [3] for a discussion of dual degeneracy. Programming was done in C and the computation below is reported in seconds on an IBM RS/6000-730.

Tables 2 through 3 each present the results from a total of 100 runs of the genetic algorithm on 10 different problems of a given size. Every line reports the outcomes of 10 runs, each with a different seed, terminated when a solution with value within 2% of the optimal is discovered, or a generation count limit of 5000 is reached. For all these problems, a population size of 100 was used. Each line also reports the time required for IBM's OSL package to solve the problems optimally.

Table 4 and Table 5 each present the results for 100 runs on 10 larger problems (250 variables). Each line reports the outcome of 10 runs using different seeds. Since OSL

Table 1: **Description of test problems**

Problems	#variables (n)	k	$\max_j u_j$
ip50 - ip59	50	5	1
ip60 - ip69	50	5	10
ip250–ip259	250	25	1
ip260–ip269	250	25	10

failed to solve these relatively large, highly degenerate problems (e.g. for problem ip250, OSL shut down after about 11 hours without finding an optimal solution), and since no good solutions were known a priori, solutions of the linear programming relaxations were used as lower bounds in the genetic algorithm. Each run was terminated when a solution with value within 2% of the lower bound is discovered, or a generation count limit of 5000 is reached. A population of size 400 was used for these problems.

The computational results show two important points. First, the computation times are superior to OSL's, though note that we compare a heuristic to an optimal algorithm. For some of the 50–variable problems OSL failed to find optimal solutions in a reasonable amount of time (e.g. after 9.5 hours of system time, OSL failed to find an optimal solution for problem $ip51$). Second, the problems with general bounded variables ($u_i > 1$) were easier to solve with the genetic algorithm than zero–one problems. This is due to the difference in scaling of the objective function values, since all data is in the same range.

6 Summary and Extensions

We present a nonlinear relaxation technique for solving bounded integer programs. This relaxation uses a nonlinear penalty function that results in a problem (PP_λ) for which the dual has weak and strong duality. This is a stronger result than is shown for this penalty function in continuous nonlinear programming or for the traditional Lagrangian

relaxation. It is an extension of results for the multiple-choice integer program in [3].

The relaxed problem is solved by a genetic algorithm that uses nontraditional reproduction, crossover and mutation operators. The mutation operator, referred to as immigration, guarantees convergence to an optimal solution.

An approach for varying λ is presented in Section 4. While it appears to work well for these problems, the best variation process is not known. This a subject of current research.

Section 5 shows excellent computational results for 40 randomly generated, dual degenerate problems. However, further testing with different ranges of c and A should be carried out. Current research seeks to implement the algorithm on a massively parallel machine. In this implementation, several populations with different seeds can be run simultaneously and the best chosen.

References

[1] M. S. Bazaraa and C. M. Shetty. *Nonlinear Programming Theory and Algorithms*, chapter 9. John Wiley & Sons, Inc, New York, 1979.

[2] J. C. Bean. Genetics and random keys for sequencing and optimization. Technical Report 92-43, Department of Industrial and Operations Engineering, University of Michigan, 1992.

[3] A. Ben Hadj-Alouane and J. C. Bean. A genetic algorithm for the multiple–choice integer program. Technical Report 92-50, University of Michigan, 1992.

[4] M. L. Fisher. Lagrangean relaxation method for solving integer programming problems. *Management Science*, 27:1–18, 1981.

[5] A. M. Geoffrion. Lagrangean relaxation for integer programming. *Mathematical Programming*, pages 82–114, 1974. Study 2.

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

[7] D. E. Goldberg. Real-coded genetic algorithms, virtual alphabets, and blocking. Technical Report 90001, University of Illinois at Urbana-Champain, September 1990.

[8] C. Z. Janikow and Z. Michalewicz. An experimental comparison of binary and floating point representation in genetic algorithms. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 31–36, 1991.

[9] Z. Michalewicz and C. Z. Janikow. Handling constraints in genetic algorithms. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 151–157, 1991.

[10] J. T. Richardson, M. R. Palmer, G. Liepins, and M. Hilliard. Some guidelines for genetic algorithms with penalty functions. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 191–197, 1989.

[11] P. Sinha and A. Zoltners. The multiple choice knapsack problem. *Operations Research*, 28:503–515, 1979.

[12] W. M. Spears and K. A. De Jong. On the virtues of parametrized uniform crossover. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 230–236, 1991.

Table 2: **Problems with** $(n,k,\max_j u_j)$**=(50,5,1)**

Computation required to get within 2% of the optimal solution over 10 random seeds

Problem	Generations			Seconds			Seconds
	min	median	max	min	median	max	OSL
ip50	129	746	999	6.68	38.90	51.72	107.80
ip51	23	490	1015	1.22	25.44	52.51	†
ip52	23	752	1201	1.27	39.09	61.66	1397.10
ip53	10	524	1074	0.55	27.63	56.65	†
ip54	42	366	1349	2.21	19.22	69.60	1770.91
ip55	69	501	1143	2.66	25.97	58.86	345.93
ip56	47	404	951	2.53	20.98	49.62	148.35
ip57	63	373	956	3.28	19.26	50.04	55.00
ip58	20	164	803	1.08	8.55	41.61	398.61
ip59	38	749	1346	2.05	39.32	70.28	203.89

† Shut down after about 9 hours without finishing the branch-and-bound. It had not found the optimal solution. LP optimal was used as lower bound for the genetic algorithm.

Table 3: **Problems with** $(n,k,\max_j u_j)$**=(50,5,10)**
Computation required to get within 2% of lower bound
over 10 random seeds

Problem	Generations			Seconds			Seconds
	min	median	max	min	median	max	OSL
ip60	147	205	711	7.32	10.17	35.37	723.07
ip61	104	645	1182	5.16	32.07	58.96	†
ip62	63	195	1151	3.14	9.66	56.87	†
ip63	105	335	580	5.25	16.71	28.99	†
ip64	99	215	1164	4.98	10.82	59.08	†
ip65	102	206	519	5.09	10.23	25.71	9.54
ip66	109	302	1607	5.49	15.06	80.55	†
ip67	37	87	757	1.86	4.31	37.68	†
ip68	128	563	1295	6.39	28.08	64.41	†
ip69	49	110	351	2.46	5.47	17.38	†

† Shut down after about 9 hours without finishing the branch-and-bound. It had not found the optimal solution. LP optimal was used as lower bound for the genetic algorithm.

Table 4: **Problems with** $(n,k,\max_j u_j)$**=(250,25,1)**
Computation required to get within 2% of the LP optimal solution
over 10 random seeds

Problem	Generations			Seconds		
	min	median	max	min	median	max
ip250	1	814	3016	3.78	249.06	7076.63
ip251	1	1	3078	3.75	3.77	7096.62
ip252	133	965	2824	313.11	1229.43	6109.29
ip253	24	1132	3266	57.52	1870.13	5566.50
ip254	15	513	4181	36.03	1177.02	2534.06
ip255	124	1171	2760	272.52	836.40	2137.87
ip256	32	513	2073	75.44	268.34	4802.35
ip257	26	1193	2444	61.45	1139.83	1489.46
ip258	71	922	3336	165.94	1169.65	6339.41
ip259	1	1	3626	3.71	3.75	3199.92

Table 5: **Problems with $(n,k,\max_j u_j)$=(250,25,10)**
Computation required to get within 2% of the LP optimal solution
over 10 random seeds

Problem	Generations			Seconds		
	min	median	max	min	median	max
ip260	43	101	1006	96.59	224.73	2231.13
ip261	1	1	1	3.51	3.55	3.56
ip262	5	39	2404	12.31	87.32	3850.65
ip263	1	3	10	3.58	6.91	23.44
ip264	1	1	1	3.53	3.60	3.64
ip265	57	162	238	127.02	356.82	525.24
ip266	876	1176	4431	1151.18	2491.55	3353.75
ip267	36	48	138	80.87	106.05	305.53
ip268	1	1	1	3.53	3.56	3.58
ip269	34	86	2261	76.30	190.83	4690.81