A GENETIC ALGORITHM FOR THE MULTIPLE-CHOICE INTEGER PROGRAM

Atidel Ben Hadj-Alouane
James C. Bean

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

Technical Report 92-50

September 1992
Revised July 1993

A Genetic Algorithm for the Multiple-Choice Integer Program *

Atidel Ben Hadj-Alouane
James C. Bean

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

July 15, 1993

Abstract

We present a genetic algorithm for the multiple–choice integer program that finds an optimal solution with probability one (though it is typically used as a heuristic). General constraints are relaxed by a nonlinear penalty function for which the corresponding dual problem has weak and strong duality. The relaxed problem is attacked by a genetic algorithm with solution representation special to the multiple-choice structure. Nontraditional reproduction, crossover and mutation operations are employed. Extensive computational test for dual degenerate problem instances show that sub-optimal solutions can be obtained with the genetic algorithm within running times that are shorter than those of the OSL optimization routine.

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

1 Introduction

The multiple-choice integer program is a linear binary program in which the variables have been partitioned. For any feasible solution exactly one variable from each partitioning set must take the value one (all others are zero). The objective is to minimize a linear function while satisfying general linear constraints. This widely applicable structure includes as special cases the assignment problem, multiple-choice knapsack problem and generalized assignment problem and has been used for many real applications ([2], [4], [6], [11]). The multiple-choice integer program (MCIP) can be stated mathematically:

$$
(MCIP) \qquad \begin{aligned}
&\min cx \\
&s.\ to \quad Ax - b \geq 0 && (1) \\
&\qquad \sum_{j=1}^{n_i} x_{ij} = 1, \ for \ i = 1, 2, \ldots, m && (2) \\
&\qquad x_{ij} \in \{0, 1\}. && (3)
\end{aligned}
$$

For each set, $i = 1, \ldots, m$, equations (2) and (3) force exactly one variable in $\{x_{ij}\}_{j=1}^{n_i}$ to be one. Constraints (2) are called multiple-choice constraints or generalized upper bounds. The system of inequalities (1) are general linear constraints where A is a $k \times n$ ($n = \sum_{i=1}^{m} n_i$) coefficient matrix and b is a constant vector in $\Re^k$.

Solving the above problem is basically finding the correct choice of positive variable from each multiple-choice set. Finding such a choice may require investigating $\prod_{i=1}^{m} n_i$ combinations, which grows exponentially in terms of n in the worst case. The most successful techniques for solving $(MCIP)$ are branch-and-bound algorithms that use either linear programming ([13]), Lagrangian relaxation ([8] [20]), or variations of Lagrangian relaxation ([2], [6]) for bounding purposes.

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. A typical Lagrangian relaxation for $(MCIP)$ relaxes the general constraints,

(1), resulting in the following simple problem:

$$(PR_\lambda) \quad \begin{array}{ll} \min & cx - \lambda(Ax-b) \\ s.\ to & Ex = e_m \\ & x_{ij} \in \{0,1\}, \end{array}$$

where λ^T is a vector in $\Re^k$, $\lambda \geq 0$. Constraints $Ex = e_m$ are the multiple-choice constraints, where E is an $m \times n$ matrix and $e_m \in \Re^m$ is a vector of ones. The entries, E_{ij}, of E are given as follows,

$$E_{ij} = \begin{cases} 1 & if\ \sum_{k=1}^{i-1} n_k < j \leq \sum_{k=1}^{i} n_k \\ 0 & otherwise \end{cases}$$

This type of relaxation often fails to give reasonable solutions in dual degenerate problems [17], that is, problems with multiple optima. Such problems occur commonly in practice, for example, the real facility location problem in [5]. In this example there are many identical facilities to be placed. This ambiguity causes many solutions to be optimal.

In this paper we 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 (1). The penalty has the form

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

Figure 1 shows this function for a single constraint with $\lambda = 1$ and $\lambda = 5$. Adding $p_\lambda(x)$ to the objective function of $(MCIP)$ gives the following nonlinear integer program:

$$(PP_\lambda) \quad \begin{array}{ll} \min & cx + p_\lambda(x) \\ s.\ to & Ex = e_m \\ & x_{ij} \in \{0,1\}, \end{array}$$

where λ is a vector in $\Re^k$, $\lambda \geq 0$. Note that $p_\lambda(x)$ is convex and continuously differentiable. The relaxation (PP_λ) does not have the integrality property (see [8]), whereas (PR_λ) does. Below we show that multipliers exist such that (PP_λ) solves $(MCIP)$ exactly. This

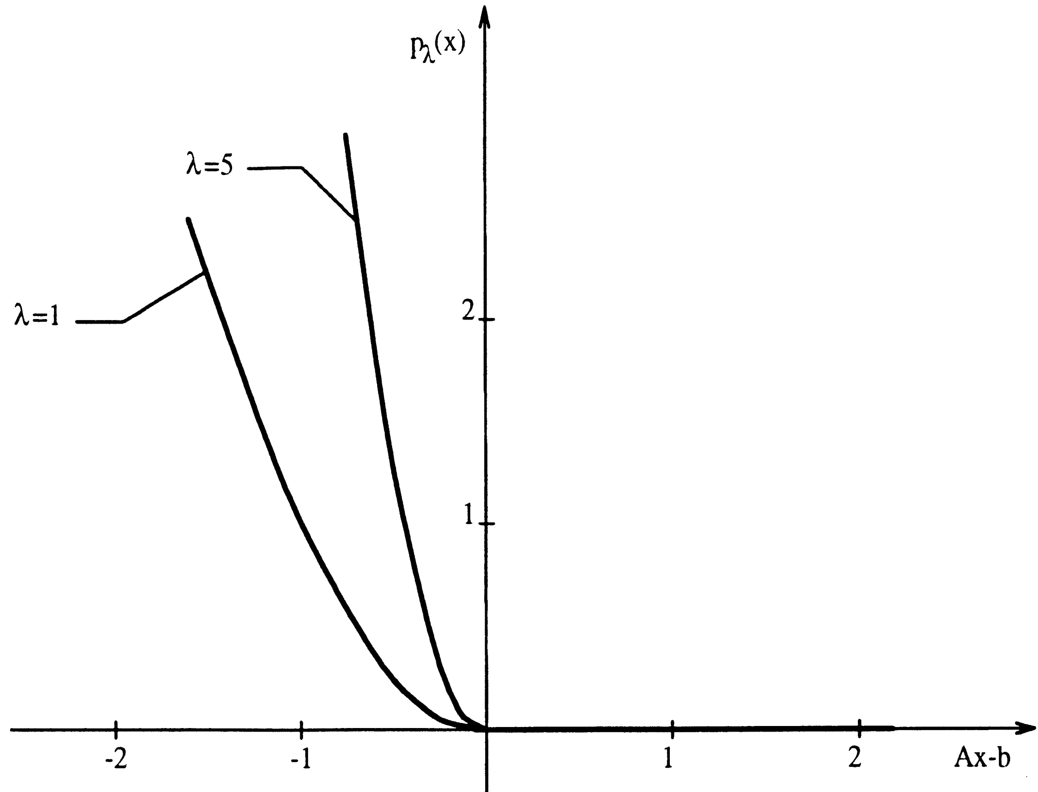

Figure 1: Nonlinear penalty function versus constraint violation

is a stronger result than is shown for nonlinear continuous programs or for traditional linear Lagrangian relaxation for integer programming.

Unfortunately, (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.

Genetic algorithms with penalty functions have been used and discussed in the literature [9] [18]. Other genetic algorithm approaches to constrained optimization appear in [16] and [3]. The genetic approach presented here appears to be the first to use a parametrized penalty function. That is, it is a primal/dual algorithm.

The remainder of this paper is organized as follows. The second Section motivates the difficulties caused by dual degeneracy and illustrates the nonlinear relaxation, (PP_λ). The third Section proves weak and strong duality for the dual corresponding to the nonlinear relaxation. Section 4 provides a brief overview of genetic algorithms and describes the genetic algorithm for (PP_λ). Section 5 evolves this into a genetic algorithm for solving

$(MCIP)$. The sixth Section presents computational results. Finally, Section 7 presents some extensions.

2 Dual Degeneracy

The problem $(MCIP)$ is said to be dual degenerate if it has multiple optima. Let $v(\cdot)$ be the optimal value of problem $(\cdot)$. The traditional, linear dual is $\max_{\lambda \geq 0} v(PR_\lambda)$. This dual can have a duality gap, that is $\max_{\lambda \geq 0} v(PR_\lambda) < v(MCIP)$.

In the dual degenerate case the dual has nongeneral characteristics. This is illustrated by the following example of a multiple-choice integer program formulation of a facility location problem involving two identical facilities.

Example 1

Consider the case of $m = 2$, and $n_1 = n_2 = 2$. The original problem is

$$
(MCIP) \quad \begin{array}{ll}
\min & x_{11} + 2x_{12} + x_{21} + 2x_{22} \\
s.\ to & -2x_{11} + 3x_{12} - 2x_{21} + 3x_{22} \geq 1 \\
& x_{11} + x_{12} = 1 \\
& x_{21} + x_{22} = 1 \\
& x_{11}, x_{12}, x_{21}, x_{22} \in \{0, 1\}
\end{array}
$$

The first multiple-choice set, containing the variables x_{11} and x_{12}, represents the first facility that will be located in one of two possible locations: location 1 ($x_{11} = 1$ and $x_{12} = 0$) or location 2 ($x_{11} = 0$ and $x_{12} = 1$). A similar interpretation can be given for the second multiple-choice set which contains x_{21} and x_{22}. In addition, the two facilities are identical since, for each location, the variables associated with the two facilities have same cost and constraint coefficients. The problem has two optimal solutions: $(x_{11}, x_{12}, x_{21}, x_{22}) = (1, 0, 0, 1)$ and $(x_{11}, x_{12}, x_{21}, x_{22}) = (0, 1, 1, 0)$. The linear Lagrangian relaxation is:

$$(PR_\lambda) \quad \begin{aligned} \min \quad & (1+2\lambda)x_{11} + (2-3\lambda)x_{12} + (1+2\lambda)x_{21} + (2-3\lambda)x_{22} \\ s.\ to \quad & x_{11} + x_{12} = 1 \\ & x_{21} + x_{22} = 1 \\ & x_{11}, x_{12}, x_{21}, x_{22} \in \{0,1\}. \end{aligned}$$

If $\lambda < 1/5$, the optimal solution to (PR_λ) is $(1,0,1,0)$, and if $\lambda > 1/5$, the optimal solution is $(0,1,0,1)$. Therefore, for any of these values of λ the optimal solution is either to put both facilities in location 1, which is not feasible in the original problem, or to put both facilities in location 2, which is the worst feasible solution to the original problem. At $\lambda = 1/5$ all four solutions are equivalent. For no value of λ is a good solution returned. Figure 2 shows the convex optimization problem formed by the dual; each line corresponds to values of $cx - \lambda(Ax - b)$ for a specific solution x. Note that all lines intersect at $\lambda = 1/5$. This corresponds to the classical concept of dual degeneracy in linear programming.

As noted in the above example, when the Lagrangian relaxation is used, identical facilities in a solution are not treated separately, but rather as one unit that are located together. In order to avoid this problem while still benefiting from the idea of constraint relaxation we use the nonlinear relaxation (PP_λ).

The advantage of using the penalty function $p_\lambda(x)$ over the linear penalty function is that only infeasible solutions incur nonzero penalty. Therefore, if there is a value of λ for which the optimal solution to the relaxed problem is feasible to the original problem, it must be the optimal for $MCIP$. This is illustrated by the following example.

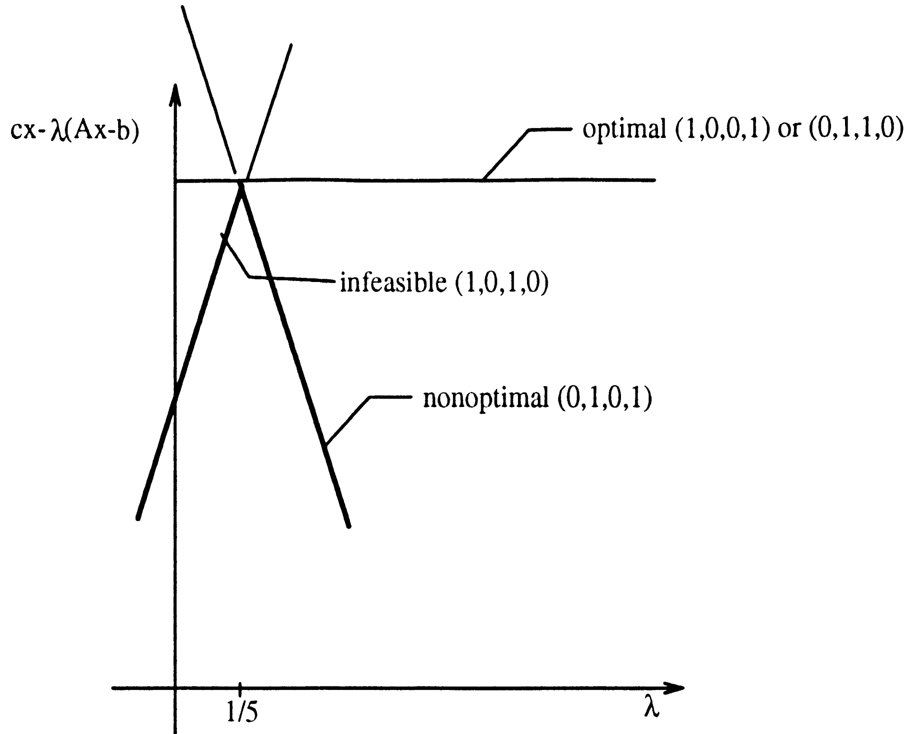

Figure 2: Lagrangian relaxation

Example 2

Consider the same problem of Example 1. The nonlinear relaxation is:

$$
(PP_\lambda) \quad \begin{aligned} \min \quad & x_{11} + 2x_{12} + x_{21} + 2x_{22} + p_\lambda(x) \\ s.\ to \quad & x_{11} + x_{12} = 1 \\ & x_{21} + x_{22} = 1 \\ & x_{11}, x_{12}, x_{21}, x_{22} \in \{0, 1\} \end{aligned}
$$

where $p_\lambda(x)$ is the penalty term as defined in (1). The optimal solutions to (PP_λ) are $(1,0,1,0)$ if $\lambda < 1/25$ and $(1,0,0,1)$ or $(0,1,1,0)$ if $\lambda > 1/25$. Figure 3 shows the convex optimization problem formed by this dual. Similarly to Figure 2, each line corresponds to values of $cx + p_\lambda(x)$ for a specific solution x.

From the above two examples, note that by not penalizing (or rewarding) the feasible solution $(0,1,0,1)$, the nonlinear relaxation allows for the optimal solutions to be discovered for a certain range of values of the parameter λ. However, in the case of Lagrangian

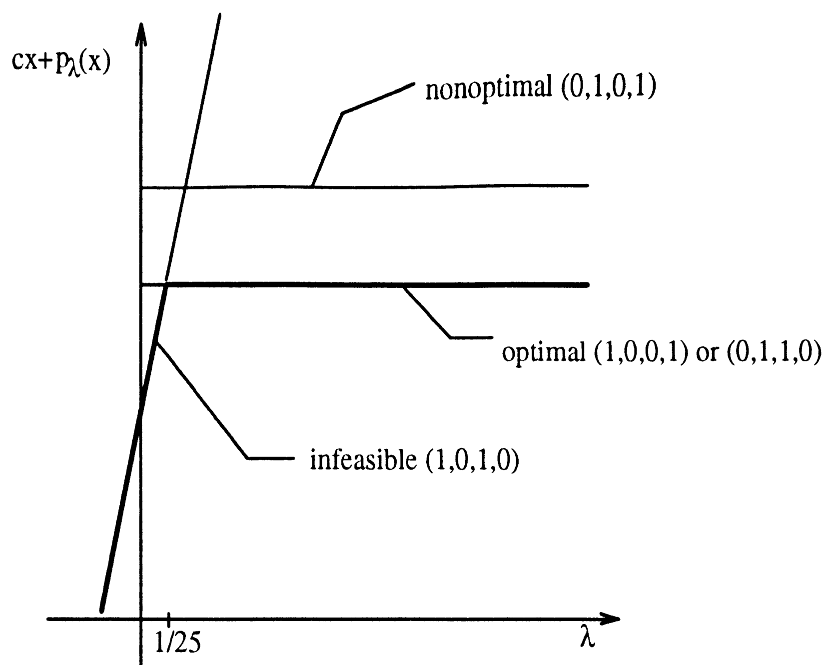

Figure 3: Nonlinear relaxation

relaxation, a degeneracy occured at $\lambda = 1/5$.

3 Validity of Nonlinear Relaxation

Analogous to the 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 on penalty function methods for continuous nonlinear programming [1]. Lemma 1 proves weak duality.

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

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

The following Theorem describes the relationship between $v(PP_\lambda)$ and $v(MCIP)$ 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 $(MCIP)$.

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 $(MCIP)$.

Proof:

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

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

That completes the proof. ∎

The following theorem establishes the existence of some value of the vector λ for which Theorem 1-a can be applied. This establishes strong duality for the dual $\max_{\lambda \geq 0} v(PP_\lambda)$.

Theorem 2 (Strong Duality) *Let S_λ be the set of optimal solutions to (PP_λ). If $(MCIP)$ 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 $(MCIP)$.*

Proof: Let $X = \{x \in \Re^n : Ex = e_m, x \in \{0,1\}\}$, $Y = \{x \in \Re^n : Ax - b \geq 0\}$ and x^* be an optimal solution to $(MCIP)$. 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, since X is a finite set, (PP_λ) and $(MCIP)$ are bounded. Define $\alpha(x) = \sum_{i=1}^{k}[\min(0, A_{i\cdot}x - b_i)]^2$ 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 $e_k = (1, 1, \ldots, 1)^T \in \Re^k$.

Therefore,

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

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. ∎

This nonlinear relaxation is stronger here than in continuous nonlinear programming. In the latter case, strong duality is attained 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 heuristically solve (PP_λ). It solves the nonlinear problem with nearly the facility of a linear problem. The tradeoff is possible loss of optimality.

4 A Genetic Algorithm Approach

4.1 Introduction to Genetic Algorithms

Genetic algorithms are random search techniques that mimic processes observed in natural evolution. They combine survival of the fittest (or best) among string structures (solutions) with a structured yet randomized information exchange [9][7]. Genetic algorithms differ from traditional optimization techniques in many aspects. They work with an encoding of the variables (typically as strings) rather than the variables themselves, and use probabilistic transition rules to move from one population of solutions to another rather than a single solution to another. The most important and interesting characteristic of genetic algorithms is that they use only objective function evaluations.

That is, they do not use any information on differentiability, convexity or other auxiliary characteristics. This property makes genetic algorithms easy to use and implement for a wide variety of optimization problems, including nonlinear integer problems.

A simple genetic algorithm works by randomly generating an initial population of solutions (a generation), then moves from one generation to another by breeding new solutions. The traditional breeding process involves objective function evaluation and three operators. The first operator is *reproduction* where strings (solutions) are copied to the next generation with some probability based on their objective function value. The second operator is *crossover* where randomly selected pairs of strings are mated, creating new strings. The crossover operation is described in detail in [3] and [9]. The third operator, *mutation*, is the occasional random alteration of the value at a string position. It plays a secondary role in genetic algorithms since, in practice, it is performed with a very small probability (on the order of 1/1000). Mutation diversifies the search space and protects from loss of genetic material that can be caused by reproduction and crossover.

4.2 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 the encoding the solutions as strings so that crossovers of feasible solutions result in feasible solutions.

The techniques for encoding solutions vary by problem and, in most cases, involve a certain amount of art. For general $0-1$ integer programs, solutions are typically represented by a string of bits of length equal to the total number of variables in the problem, n, where each string position can take the value 0 or 1 (in genetic terminology, the string position is called gene, and the position value is called allele). This results in a search space of cardinality 2^n, exponential in the number of variables [12].

Recently, a special encoding technique, called random keys [3], has been introduced and shown to be successful for a variety of sequencing and optimization problems. It consists of representing 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 multiple-choice sets. The allele, for a given set, takes the value of the index of the element of that set taking the value one. All other variables in that set take zero. Hence, each position (gene), i, of the string can take any integer in $\{1, \ldots, n_i\}$, where n_i is the number of variables in multiple-choice set i. This is illustrated by the following example.

Example 3 *let $m = 2$ and $n_i = 3$ for $i = 1, 2$. One solution that satisfies the multiple-choice constraints is $x_{12} = x_{23} = 1$ and all remaining four variables equal zero. Using the proposed representation, this solution can be represented by the following string* $(2, 3)$.

In genetic terminology, the string position is called gene, and the position value is called allele. With this representation, the size of the search space is reduced from 2^n to $\prod_{i=1}^{m} n_i$. The computational complexity resulting from this representation can still be exponential. However, if growth in variables takes place within existing multiple-choice sets the genetic algorithm has polynomial complexity.

This large alphabet encoding (see [9]), also referred to as the real coding [10] or floating point coding [15], is an alternative to binary encodings. The experimental study in [15] 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 6 suggest that this is quite effective. A broader study is underway into the reasons for and limitations of this effectiveness.

In this paper, a genetic algorithm is applied to the problem (PP_λ). Since (PP_λ) has only multiple-choice 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 [9]. The genetic algorithm employed here has

proved to be very successful for the MCIP, particularly large scale, dual degenerate MCIP's. Given a current generation, the next generation is created as follows.

1. *Copy* the N_c top solutions. The solutions of the current generation are sorted by increasing order of the objective function value, then the top N_c solutions are copied into the next generation. N_c is usually fixed to 10% of the population size. This approach, called elitist reproduction [9], replaces the traditional probabilistic reproduction. The advantage of using elitist reproduction is that the best solution is monotonically improving from one generation to another.

2. *Mate* random pairs: first, randomly select two strings (parents) from the entire current generation. Next, create two offspring as follows. For each gene, a biased coin is tossed, and the outcome determines whether or not to interchange the alleles of the parents. Formally, this operation can be described as follows. Consider the parent strings $P_1 = p_{11}p_{12}\ldots p_{1m}$ and $P_2 = p_{21}p_{22}\ldots p_{2m}$, where p_{ij} is the allele of the jth gene of the string P_i. Similarly, let $O_1 = \mathbf{o}_{11}\mathbf{o}_{12}\ldots\mathbf{o}_{1m}$ and $O_2 = \mathbf{o}_{21}\mathbf{o}_{22}\ldots\mathbf{o}_{2m}$ be the offspring created by mating P_1 and P_2. Note that, here, the bold style is used for $\mathbf{o}_{ij}$ since the latter denotes the gene of O_i and not the allele. Then, the probabilistic interchange of the m alleles can be modeled as m independent random variables, say $X_1, X_2, \ldots X_m$, having each the Bernoulli distribution with a parameter not equal to 1/2. These random variables are thus defined as follows.
   For $j = 1, \ldots, m$,

   $$X_j = \begin{cases} 1 & \textit{if alleles } p_{1j} \text{ and } p_{2j} \textit{ are interchanged} \\ 0 & \textit{otherwise} \end{cases}$$

   By performing these m independent trials, the genes of the two offspring are built. Therefore, The offspring genes are functions of the random variables X_j's, given as follows.
   For $j = 1, \ldots, m$,

   $$\mathbf{o}_{1j} = (1 - X_j)p_{1j} + X_j p_{2j}$$

   $$\mathbf{o}_{2j} = X_j p_{1j} + (1 - X_j)p_{2j}$$

An illustration of this operation is given in Example 4. Once the offspring are created, they are evaluated, and only the one with better objective value is included in the new generation. This mating operation is similar to the uniform crossover described in [9], and the parametrized uniform crossover introduced in [19]. Experiments have shown that, for the MCIP, this operator is computationally better than the one-point or two-point crossover.

3. Create *mutations* by randomly generating a small number of entirely new solutions ($N_m = 1\%$ of the population size) and including them in the new generation. A random solution consists of a string of m random numbers, say

   $r_1 r_2 \ldots r_m$, where r_i is uniformly distributed on the integer interval $[1, n_i]$, for each $i = 1, \ldots, m$. This operation is clearly different from the gene by gene mutation since it involves bringing new members to the population. In [3] 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.

Example 4 *Let $m = 4$ and $n_i = 3$ for all i. Then one string is an element of $\times_{i=1}^{4}\{1, 2, 3\}$. Two particular solutions are*

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

The former represents the solution $x_{11} = x_{22} = x_{33} = x_{41} = 1$, all other variables equal 0. To crossover 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

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

The genetic algorithm approach described above can be summarized by the following pseudocode. Note that the best feasible solution found is updated at each new generation.

Initialization. Choose a population size, N (experiments showed that $N \in [m, 3m]$ works well for our test problems). Choose a stopping criteria (an objective value bound or maximum number of generations). Set best objective value so far to infinity. Randomly generate and evaluate N solutions as described above in the immigration operation. Let G_1 be the set of these solutions. Set $k = 1$.

Main Step. While (not stop)

BEGIN

Set $G_{k+1} = \emptyset$

1. Sort the solutions in G_k by increasing order of objective value. Include the first N_c solutions in G_{k+1}. If top solution in G_k is feasible, update best feasible solution so far.

2. While $(|G_{k+1}| < (N - N_m))$

   BEGIN (Crossover)

   - Randomly (uniformly) select two solutions P_1 and P_2 from G_k.
   - Mate P_1 and P_2 to produce offspring O_1 and O_2.
   - Evaluate O_1 and O_2, and include in G_{k+1} the one with lower objective value.

   END (Crossover)

3. Randomly generate N_m solutions and include them in G_{k+1}.

If the first solution in G_k is feasible, update best feasible solution so far. $k = k + 1$. Stop if stopping criteria is met.

END

Since each solution can be selected with nonzero probability by the immigration operator, we can conclude that, if the genetic algorithm above is run long enough, it will find an optimal solution with probability one.

5 A Genetic Algorithm Approach for $(MCIP)$

Combining Theorem 2 and the proposed genetic algorithm, we have the following theoretical algorithm.

- Set λ large enough to ensure that (PP_λ) solves $(MCIP)$ optimally.
- Run the genetic algorithm long enough to find an optimal solution to (PP_λ), and hence, $(MCIP)$.

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 the case of problems with complex feasible domains. We describe below an implementation that seeks good solutions to $(MCIP)$ by adjusting the λ vector while running the genetic algorithm. Initially, we fix λ at a certain value (the choice of this initial value is discussed later in this section), and run the genetic algorithm for a certain number of generations, N_f. We halt to check whether the top solution has had a zero or nonzero penalty for each of these N_f generations. If λ is small enough, the top solution will have nonzero penalty (infeasible). In this case, λ is increased. However, if λ is relatively large, solutions with zero penalty (feasible) remain on the top of the population. In this situation, λ is decreased. In both cases, the solutions of the current generation are re-evaluated with the new λ, and the algorithm continues. In case where the top solution alternates at least once between feasible and infeasible (or vice versa), we continue the algorithm with the same value of λ.

In this approach, we increase λ by the multiplicative factor β_1, and decrease by the factor β_2, where β_1 and β_2 are real constants chosen empirically. In order to avoid cycling, it is important that the increasing and decreasing rates of λ be different. In our experiment, we empirically initialize λ at a small value; therefore 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.

In general, the ideal value of λ is the one that results in hybrid populations (contain feasible and infeasible solutions) so that feasible solutions can be improved by mating with infeasible solutions that may have "good" genetic material, in this case, the top solution usually alternates between feasible and infeasible.

The initial value of λ is calculated based on the theoretical result given in Theorem 2, i.e. $\bar{\lambda} = \max\{\lambda_0; 0\}e_k$, where $\lambda_0 = \max_{x \in X, \alpha(x) \neq 0}\{\frac{cx^* - cx}{\alpha(x)}\}$. The following approximation of λ_0 is used.

$$\hat{\lambda}_0 = 1/\bar{N}_1 \sum_{i \in G_1, \alpha(x_i) \neq 0} (\frac{c\bar{x}^* - cx_i}{\alpha(x_i)}),$$

where G_1 is the set of solutions in the initial generation, $\bar{N}_1$ is the number of solutions, in G_1, with nonzero penalty (i.e with $\alpha(x_i) \neq 0$), and $\bar{x}^*$ is the optimal solution of the linear programming relaxation of $(MCIP)$.

If run long enough, the procedure described above will find an $(MCIP)$ with probability one. However, it is typically stopped heuristically by using lower bounds and setting a limit to the number of generations created. A pseudocode that describes this heuristic procedure is given below.

Initialization. Choose two scalars $\beta_1 > \beta_2 > 1$. Typical values are $\beta_1 = 4$ and $\beta_2 = 2.8$.

Set $\lambda_1 = \max\{\hat{\lambda}_0; \epsilon\}e_k$, where $\epsilon > 0$ and arbitrarily small.

Choose a frequency, N_f, for altering λ.

Choose a value for N_{max}, the maximum number of generations to be created. Set Lb to a lower bound, if available.

Randomly generate a population of solutions (as described in Step 3 of the genetic approach for (PP_λ)) and evaluate objective values.

Set $k = 1$.

Main Step. While (not stop)

BEGIN

Create a new generation using the genetic algorithm described in Section 4.

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 best solution found is no greater than Lb or if $k > N_{max}$.

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, we use $N_f = 2 \max_i n_i$ (twice the maximum number of variables in any multiple-choice set) which works well for our test problems. 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 it is initially small or large. This robustness is also supported by a small experiment on the rate by which λ increases or decreases, i.e. different values of β_1 and β_2. Results and discussion of this experiment are reported in Section 6.

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

6 Computational Results

We present computational results for two sets of problems: randomly generated problems and real facility location problems. Programming was done in C and the computation below is reported in seconds on an IBM RS/6000–730.

6.1 Randomly Generated Problems

Computational experience was carried out for three sets of randomly generated problems of different sizes: small, medium and large. Each set contains 10 problem instances. For each problem, the number of multiple–choice sets, m, is larger than the number of variables within each set, n_i. Computational testing has also been carried out for problems of the same sizes, and m smaller than n. Results for these problems were comparable to those reported here, and hence, were omitted. These test problem instances were all made dual degenerate by creating a random number of duplicate multiple–choice sets.

Table 1 presents the results from a total of 300 runs of the genetic algorithm on 30 different problem instances. Each line shows the name and size of the problem instance, and reports the outcome of 10 runs, each with a different random seed. For all these tests, the genetic algorithm runs were terminated when a solution with value within 5% of the optimal is discovered (if optimal is not available, the linear relaxation bound is used). However, there is a maximum of 2000 generations not to be exceeded for all tests. Population sizes of 30, 100 and 150 were used for small, medium and large problems, respectively. Each line also reports the time required for IBM's OSL package [14] to find heuristic solutions with the same accuracy as the genetic algorithm, as well as the time it requires to solve the problem optimally, if possible. All OSL runs were also carried out on an IBM RS/6000–730.

Results show that the genetic algorithm is about ten times faster than OSL's heuristic branch–and–bound for the small problems, and three times faster for the medium problems. For the large problems, the genetic algorithm is twice as fast as OSL. However, the code size and simplicity, the robustness of the genetic algorithm, and the ease with which it can be adopted to various platforms make it more appealing than branch-and-bound as a heuristic. Moreover, the data structures used in the genetic algorithm are simple (basically arrays), whereas branch-and-bound requires more complex data structures which can also be complex to maintain (e.g. memory size, allocation/deallocation and structuring overhead). Note that all large problems and even some of the medium problems could not be optimally solved by OSL, typically due to the lack of storage.

The results of the genetic algorithm also show two important points. First, computation time increases with problem size in a reasonable manner. Second, there is little variance across problem instances and across the random seeds. The algorithm appears to be very robust to these parameters.

6.2 Real Facility Location Problems

Three facility location problem instances are considered. They consist of determining locations and sizes of different categories of stores (or facilities) in a shopping mall consisting of scattered empty spaces and existing stores, so as to maximize a revenue function. All data is derived from a real economic and optimization study [5], which resulted in dual degenerate problems. Table 2 shows the sizes of these problems (*Pb1, Pb2* and *Pb3*), the results of 30 runs of the genetic algorithm with different random seeds, and the time required for OSL to find a heuristic solution, and the time it takes to solve the problems optimally. The population sizes used for these tests are 30 for both *Pb1* and *Pb2*, and 100 for *Pb3*. These results show three important points. First, note the small computational time of the genetic algorithm compared to OSL's. Second, note the small variance across the random seeds. Third, note the scalability of the genetic algorithm as opposed to OSL. For the latter, the results were unpredictable since it took about five times longer to heuristically solve the small problem, *Pb1*, than the medium problem *Pb2*, and nearly twice as much time to optimally solve *Pb1* as it did to solve the much larger problem instance, *Pb3*.

6.3 Experiment on Different Variation Rate of λ

In the previous computational experiment, the values of β_1 and β_2, the rates by which λ varies, were fixed to $\beta_1 = 4$ and $\beta_2 = 0.7\beta_1$. The following small experiment is designed to test the robustness of the genetic algorithm to these choices.

One problem of each size category, from the randomly generated problems, are selected to be used in this experiment. Keeping the same relation between β_1 and β_2, and allowing

β_1 to change, the genetic algorithm is run on these test problems, for 10 different seeds. The results are reported on Table 3. Although a more extensive computational testing is needed to draw general conclusions, this small experiment supports the robustness of our genetic algorithm with respect to the strategy for the variation of λ since there is no significant change in the results by varying the value of β. The results, however, suggest that by making the rate of variation large enough, i.e. giving a large increase (or decrease) to λ, there is a faster convergence. Intuitively, this means, for example in the case of an increase, infeasible solutions which are at the top of the population due to a small penalty become highly penalized and, hence, are replaced with feasible solutions. However, this relation between the convergence rate and high variation rates cannot be generalized for all problems. Seeking the best strategy and more conclusive results is the subject of ongoing research and results will be reported elsewhere.

7 Extensions

The genetic algorithm approach presented in this paper is addressed specifically to the multiple-choice integer program. However, all theoretical results as well as the solution approach are applicable to any integer program with bounded variables, since such a program can be transformed to a multiple–choice integer program. One transformation is to express each variable, say x, as a linear combination of $0-1$ variables: e.g.
$x = 0x_0 + 1x_1 + 2x_2 + \ldots + ux_u$, with

$$x_0 + x_1 + x_2 + \ldots + x_u = 1,$$

where $0 \leq x \leq u$. By doing this transformation, the number of multiple–choice constraints created is equal to the number of bounded variables. However, no general constraints are added. Thus, the duality theory of Section 3 is valid for a wide class of problems.

Section 6 shows excellent computational results for 330 runs on 30 randomly generated, dual degenerate multiple-choice problems, and 3 real facility location problems,

also dual degenerate. Further speed-up of the algorithm can be achieved by taking advantage of parallel computation. 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. Moreover, for each seed, processing of the population itself can be parallelized by having several processors working on sub-populations of solutions simultaneously.

Acknowledgment

The authors would like to thank Jennifer Merchant for her help in carrying out the computational experiments and valuable suggestions regarding choices of parameters. We would also like to thank an anonymous referee for many helpful comments.

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. A Lagrangian algorithm for the multiple choice integer program. *Operations Research*, 32:1185–1193, 1984.

[3] J. C. Bean. Genetics and random keys for sequencing and optimization. Technical Report 92-43, University of Michigan, 1992.

[4] J. C. Bean, J. R. Birge, J. Mittenthal, and C. Noon. Matchup scheduling with multiple resources, release dates and disruptions. *Operations Research*, 39:470–483, 1991.

[5] J. C. Bean, C. E. Noon, S. M. Ryan, and G. J. Salton. Selecting tenants in a shopping mall. *Interfaces*, 18:1–9, March-April 1988.

[6] J. C. Bean, C. E. Noon, and G. J. Salton. Asset divestiture at Homart Development Company. *Interfaces*, 17:48–64, January-February 1987.

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

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

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

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

[11] R. Haessler, P. Sweeney, and F. Talbot. A separable linear programming model for retail planning. In *Proceedings of the 14th Annual Meeting*, San Francisco, 1992. American Institute for Decision Sciences.

[12] W. E. Hart and R. K. Belew. Optimizing an arbitrary function is hard for the genetic algorithm. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 190–195, 1991.

[13] W. C. Healy, Jr. Multiple choice programming. *Operations Research*, 12:122–138, 1964.

[14] IBM Corporation. *Optimization Subroutine Library Guide and Reference.*

[15] 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.

[16] 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.

[17] K. G. Murty. *Linear Programming.* John Wiley and Sons, 1983.

[18] 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.

[19] 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.

[20] D. J. Sweeney and R. A. Murphy. Branch-and-bound methods for multi-item scheduling. *Operations Research*, 29:853–864, 1981.

Table 1: **Randomly Generated Problems**

Computational effort over 10 random seeds

Problem	Size	Generations			Seconds			Seconds	Seconds
	(m,n_i,k)	min	median	max	min	median	max	heur. OSL	opt. OSL
sma1	(10,5,1)	8	17	45	0.02	0.04	0.10	0.40	0.89
sma2	(10,5,1)	10	13	102	0.02	0.03	0.24	0.39	0.56
sma3	(10,5,1)	7	18	33	0.01	0.04	0.08	1.70	2.76
sma4	(10,5,1)	14	19	37	0.03	0.04	0.09	0.70	1.81
sma5	(10,5,1)	22	60	325	0.06	0.14	0.73	0.49	0.77
sma6	(10,5,1)	12	18	30	0.03	0.04	0.07	0.52	1.03
sma7	(10,5,1)	3	11	19	0.01	0.04	0.04	0.59	2.49
sma8	(10,5,1)	7	18	98	0.02	0.04	0.22	0.42	0.44
sma9	(10,5,1)	12	64	253	0.03	0.15	0.58	0.36	0.41
sma10	(10,5,1)	8	18	34	0.02	0.04	0.09	0.54	1.12
mda1	(40,20,10)	53	85	267	4.62	7.47	22.25	14.21	1635.68
mda2	(40,20,10)	56	86	168	4.88	7.48	14.34	41.49	*
mda3	(40,20,10)	69	91	146	6.01	7.48	12.73	30.66	114.53
mda4	(40,20,10)	68	83	146	5.90	7.28	12.37	20.77	630.36
mda5	(40,20,10)	67	83	147	5.81	7.26	12.52	18.41	1358.18
mda6	(40,20,10)	67	82	128	5.81	7.20	11.50	29.90	4238.00
mda7	(40,20,10)	73	88	128	6.27	7.89	12.38	35.57	1489.11
mda8	(40,20,10)	67	78	128	5.83	6.77	10.96	22.73	7195.00
mda9	(40,20,10)	44	84	201	3.85	7.50	16.85	10.13	*
mda10	(40,20,10)	60	105	223	5.18	9.17	18.77	25.57	*
lga1	(100,50,20)	154	186	223	131.27	158.46	190.03	293.57	*
lga2	(100,50,20)	156	190	210	131.88	159.58	180.03	288.43	*
lga3	(100,50,20)	119	162	187	100.45	175.35	158.21	253.04	*
lga4	(100,50,20)	127	160	228	109.55	140.17	198.45	363.48	*
lga5	(100,50,20)	162	197	229	139.70	169.64	197.77	363.48	*
lga6	(100,50,20)	110	142	156	95.71	122.85	134.71	340.47	*
lga7	(100,50,20)	131	141	174	113.23	122.47	150.59	308.00	*
lga8	(100,50,20)	148	175	241	127.20	151.09	207.03	317.91	*
lga9	(100,50,20)	146	157	193	124.91	134.50	165.80	349.66	*
lga10	(100,50,20)	148	166	228	125.13	140.40	193.35	310.05	*

* Shut down after at least 20 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 and heuristic OSL.

Table 2: **Facility Location Problems**

Computational effort required to get within 2.5% of the optimal solution

over 10 random seeds

Problem	Size	Generations			Seconds			Seconds	Seconds
	(m,n_i,k)	min	median	max	min	median	max	heur. OSL	opt. OSL
Pb1	(11,3,1)	2	7	13	0.01	0.02	0.03	0.62	105.23
Pb2	(11,21,7)	11	12	13	0.04	0.05	0.06	0.14	0.14
Pb3	(41,71,24)	24	28	42	4.13	4.79	7.25	12.51	54.80

Table 3: **Effect of the Variation rate of λ**

Problem	Size	β_1	Generations			Seconds		
	(m,n_i,k)		min	median	max	min	median	max
sma5	(10,5,1)	2	34	56	347	0.08	0.13	0.78
		4	22	60	325	0.06	0.14	0.73
		8	19	32	112	0.04	0.09	0.26
mda5	(40,20,10)	2	67	83	219	5.89	7.36	18.28
		4	67	83	147	5.81	7.26	12.52
		8	67	83	127	5.91	7.48	11.03
lga5	(100,50,20)	2	162	197	229	137.87	167.40	194.48
		4	162	197	229	139.70	169.64	197.77
		8	162	197	229	139.70	169.64	197.77