Two-Level 0-1 Programming through Parallel Genetic Algorithms

Keiichi Niwa *, Ichiro Nishizaki †, Masatoshi Sakawa ‡

Abstract—This paper deals with a two-level 0-1 programming problem in which there is no coordination between the decision maker (DM) at the upper level and the decision maker at the lower level. The authors propose a modified computational method that solves problems related to computational methods for obtaining the Stackelberg solution. Specifically, in order to shorten the computational time of a computational method implementing a genetic algorithm (GA) proposed by the authors, a parallel genetic algorithm is introduced with respect to the upper level GA, which handles decision variables for the upper level DM. Parallelization of the upper level GA is also performed along with parallelization of the upper level GA. The proposed algorithm is also improved in order to eliminate unnecessary computation during operation of the lower level GA, which handles decision variables for the lower level DM. In order to verify the effectiveness of the proposed method, we propose a comparison with existing methods by performing numerical experiments to verify both the accuracy of the solution and the time required for the computation.

Keywords: Two-level 0-1 programming problem, Stackelberg solution, Parallel genetic algorithm

1 Introduction

In the real world, we can often encounter situations in which there are multiple decision makers (DMs) in hierarchically structured organizations, and decisions may be taken serially or simultaneously in order to optimize each of the objectives. This kind of problem has been formulated as a two-level programming problem [13]. In two-level programming problems, upper level DMs take their decisions first, and then, with a knowledge of the decisions of the upper level DMs, the lower level DMs make their decisions in order to optimize their own objective functions. This combination of decisions is known as a Stackelberg solution. In this paper, both the upper level and the lower level have one DM, and the problem is treated as a two-level 0-1 programming problem in which both DMs treat all of their decision variables as 0-1 variables.

As an overview of research dealing with two-level programming problems that include discrete variables, Bard et al. presented an algorithm based on the branch-and-bound approach in order to derive the Stackelberg solution for two-level 0-1 programming problems [4] and two-level mixed integer programming problems [3]. Wen et al. [14] have presented a computation method for obtaining the Stackelberg solution to two-level programming problems which have 0-1 parameters for the decision variables in the upper level and continuous parameters for the decision variables in the lower level.

On the other hand, the adaptive process of systems in the natural world has been explained, and genetic algorithms (GAs) which imitate the evolution occurring in living organisms have been receiving attention at international conferences related to GAs, publications by Goldberg [7], as have methodologies for optimization, adaptation and learning. GAs have also been adopted for a variety of combinatorial optimization problems, and their effectiveness has been reported [12].

An example of research related to two-level programming problems using a GA is given by Anandalingam, et al. [2] which presents a method for deriving a Stackelberg solution for two-level linear programming problems. In order to derive a Stackelberg solution for 0-1 programming problems related to two-level decentralized systems, the authors [10] have also proposed a computational method that adopts the double string proposed by Sakawa, et al as the individual representation. In order to improve the computational accuracy of the Stackelberg solution, the authors have proposed computational methods that implement sharing [9] and cluster analysis [11] methods. Use of these methods allows for the derivation of approximate Stackelberg solutions with relatively high precision and in a relatively short time, but there is still room for improvement, particularly with regards to calculation times.

This paper therefore focuses on two-level 0-1 programming problems, and proposes an improved computational
method that addresses problems related to the computational method proposed by the authors for deriving the Stackelberg solution. Specifically, in order to shorten the computational time of a computational method implementing a genetic algorithm (GA) proposed by the authors, a parallel genetic algorithm is introduced with respect to the upper level GA, which handles decision variables for the upper level DM. Parallelization of the lower level GA is also performed along with parallelization of the upper level GA. The proposed algorithm is also improved in order to eliminate unnecessary computation during operation of the lower level GA, which handles decision variables for the lower level DM. In order to verify the effectiveness of the proposed method, we propose a comparison with existing methods by performing numerical experiments to verify both the accuracy of the solution and the time required for the computation.

2 Two-level 0-1 programming problem

For the sake of brevity, we denote the upper and lower level DMs by DM1 and DM2, respectively. The vectors of decision variables for DM1 and DM2 are $x = (x_1, \ldots, x_{n_1})^T$, and $y = (y_1, \ldots, y_{n_2})^T$. The superscript $T$ indicates transposition. The objective functions of DM1 and DM2 are written as $z_1(x, y)$, and $z_2(x, y)$. The coefficient vectors of the objective functions are denoted by $e_1 = (e_{11}, \ldots, e_{1n_1})$, $d_1 = (d_{11}, \ldots, d_{1n_2})$, $e_2 = (e_{21}, \ldots, e_{2n_1})$, and $d_2 = (d_{21}, \ldots, d_{2n_2})$. The coefficient matrices in the constraints are the $m \times n_1$ matrix $A$, and the $m \times n_2$ matrix $B$. The vector of constants on the right hand side of the constraints is written as $b = (b_1, \ldots, b_m)^T$. The two-level 0-1 programming problem may now be formulated as the following.

$$\begin{align*}
& \text{maximize } \quad z_1(x, y) = c_1 x + d_1 y \\
& \text{where } y \text{ solves} \\
& \text{maximize } \quad z_2(x, y) = c_2 x + d_2 y \\
& \text{subject to} \quad A x + B y \leq b \\
& x \in \{0, 1\}^{n_1}, y \in \{0, 1\}^{n_2}
\end{align*}$$  

(1)

For the sake of simplicity, in this paper, it is assumed that each component of $A$, $B$, $c_1$, $c_2$, $d_1$, and $d_2$ is positive.

It is possible to express the process for choosing the Stackelberg solution for a two-level 0-1 programming problem in the following manner. Each decision maker completely knows objective functions and constraints of the opponent and self, and DM1 first makes a decision and then DM2 makes a decision so as to minimize the objective function with full knowledge of the decision of DM1. That is to say, when the decision by DM1 is denoted $\tilde{x}$, DM2 solves the 0-1 programming problem (2) with parameters $\tilde{x}$, choosing the optimal solution $y(\tilde{x})$ as the rational reaction to $\tilde{x}$.

$$\begin{align*}
& \text{maximize } \quad z_2(\tilde{x}, y) = d_2 y + c_2 \tilde{x} \\
& \text{subject to} \quad B y \leq b - A \tilde{x} \\
& \quad y \in \{0, 1\}^{n_2}
\end{align*}$$  

(2)

Under this premise, DM1 also determines $x$ by choosing the value which minimizes its own objective function. For problems which adopt the Stackelberg solution to conceptualize their solution, it is assumed that there is no consensus among DMs that might mutually constrain decisions. Putting it another way, their relationship may be described as non-cooperative.

3 GA based computational method

In this section, we present the derivation of the Stackelberg solution to the two-level 0-1 programming problem. A computational method based on a genetic algorithm is explained.

3.1 Coding and decoding

When solving 0-1 programming problems using GAs, binary strings are usually adopted to express individuals [8, 7]. However, under this representation, it is possible that infeasible individuals that do not satisfy the constraints may be generated, so there is a danger that the performance of the GA may degrade. Thus, in this paper, in order to derive only feasible solutions, a double string [12] is used which is composed of the substring corresponding to the decision of DM1, $x$, and the substring corresponding to the decision of DM2, $y$, as shown in Fig.1. The decisions of DM1 and DM2 are handled by performing genetic operators on each sub-individual. In this paper, the GA handling the decision of DM1 is called the upper level GA, and the GA handling the decision of DM2 is called the lower level GA.

![Figure 1: Double string](image)

For the sake of simplicity, in this paper, it is assumed that each component of $A$, $B$, $c_1$, $c_2$, $d_1$, and $d_2$ is positive.

It is possible to express the process for choosing the Stackelberg solution for a two-level 0-1 programming problem in the following manner. Each decision maker completely knows objective functions and constraints of the opponent and self, and DM1 first makes a decision and then DM2 makes a decision so as to minimize the objective function with full knowledge of the decision of DM1. That is to say, when the decision by DM1 is denoted $\tilde{x}$, DM2 solves the 0-1 programming problem (2) with parameters $\tilde{x}$, choosing the optimal solution $y(\tilde{x})$ as the rational reaction to $\tilde{x}$.
3.2 Reproduction

We first describe the reproduction operator of the lower level GA. In the lower level GA the value of \( y \) obtained by decoding individuals in the lower level GA, and the given values of the decision variables in the upper level GA, \( x \), are substituted into the objective function of DM2, \( z_2(x, y) \), and the value of the evaluation function for each individual is thus obtained. Next, the fitness value for each individual is derived using linear scaling, and the individuals remaining in the next generation are determined by applying elitist expected value selection.

We now describe the reproduction operator of the upper level GA. In the upper level GA, the value of \( x \) obtained by decoding individuals in the upper level GA and the value of the rational reaction obtained by applying the lower level GA, \( y(x) \), are substituted into the objective function of DM1, \( z_1(x, y(x)) \), and the value of the evaluation function for each individual is thus obtained. Next, the fitness value for each individual is calculated by applying linear scaling and adopting a clustering method. The individuals remaining in the next generation are determined by applying elitist expected value selection based on these fitness values.

3.3 Crossover and mutation

For double string individuals, if standard one-point or multi-point crossovers are performed then there is a possibility that infeasible individuals may be generated because the indexes occurring in the offspring, \( i_x(m) \), \( i_x(m') \), \( i_y(m) \), \( i_y(m') \), may have the same number. This kind of difficulty has been identified as occurring when genetic algorithms are applied to problems such as that of the traveling salesman problem, or the scheduling problem. Partially matched crossovers (PMX) have been devised to overcome this difficulty. In this paper, a modified version of PMX is used in order to handle the double strings proposed by Sakawa et al. [12]. Also, when determining whether or not to apply the crossover operator, a probability \( p_c \) is used. Its value is set in advance.

The mutation operator is thought to fulfill the role of a local random search in genetic algorithms. For double strings, the index string expresses the priority of the parameters. For binary strings, since the value of the 0-1 parameters themselves are expressed, strings with differing properties coexist in a single string, and it is necessary to apply mutations to each string. In this paper, the mutation operator is applied to each string, and inversion is used for additional strings. For binary strings, bit-reverse is introduced. When applying the mutation operator to individuals, it is first determined whether or not the mutation operator will be applied to an individual according to the mutation probability \( p_m \). In the case that mutation is applied, it is then determined whether to apply inversion or bit-reverse according to the mutation selection constant \( M_{pm} \).

3.4 Application of the parallel genetic algorithm

In the computational method proposed by the authors [11], an algorithm for eliminating unnecessary computations during operation of the GA is added, and the effectiveness of such is indicated by numerical experiments. However, while the proposed computational method succeeds in reducing the amount of computational time by approximately 30%, there is still likely much more room for improvement. In this study, therefore, we aim for further reductions in computational time, and consider parallelization of the upper level GA and the lower level GA implemented by the computational method proposed by the authors.

In genetic algorithms, it is possible to perform parallel processing in the greater part of the operations included in the algorithm. In reproduction operations, however, because it is necessary to calculate evaluation values for each individual in a population, and based on that value determine the fitness of each individual, direct application of parallel processing is difficult. Research related to the parallelization of genetic algorithms started with improvements to such barriers to the implementation of parallelization, and a variety of types of models have been proposed and their effectiveness noted by numerous researchers [5]. Today, genetic algorithms that implement parallel processing have come to be called parallel genetic algorithms.

Broadly classified, there are three types of parallel genetic algorithm:

1) Single-population master-slave GAs In these models, the population is not divided, and selection and crossover are performed globally, with only individual evaluation performed on multiple processors.

2) Multiple-population coarse-grained GAs In these models, the population is divided into multiple partial populations, and the partial populations are assigned to multiple processors. Selection, crossover, and mutation are then performed on the assigned processors, and at some fixed period an operation called migration is performed to swap individuals among populations. These models are also known as distributed GA.

3) Single-population fine-grained GAs In these models, each processor is assigned one or a very small number of individuals.

Of the three models described above, the single-population master-slave GAs performs parallel processing by not dividing the population, and using multiple
processors only for the evaluation of individuals [1, 6]. This model can be used for the sole purpose of distributing calculation load, and is used in situations in which the calculation of individual fitness values is extremely complex and so requires substantial calculation times. Another characteristic is that it also implements a simpler algorithm than the other two models, and because there are no changes to the population due to parallelization, there will be no effects on the search for a solution.

The computational method proposed by the authors includes an upper level GA and a lower level GA, and parallelization of each must be considered. We will first describe the parallelization of the lower level GA. A lower level GA is used to obtain the rational reaction \( y(x) \) to a given upper level GA individual \( x \). For each individual \( x \) of the upper level GA therefore there is an independently operating lower level GA, and so it is possible to divide the lower level GA operations and assign them across multiple processors.

We will next describe the parallelization of upper level GAs. Though there will be some variability according to the computing environment, where some parallelization of the upper level GA must be considered along with that of the lower level GA as described above, models 1) or 2) may be implemented. In the case where model 2) is employed there may be a decrease in the level of precision of the solutions obtained as compared to existing methods, due for example to an increase in the number of parameters that must be set as a result of the introduction of the distributed GA, or due to the splitting of the population. In this paper, we will employ a single-population master-slave GAs in which calculation times are reduced for operations related to individual fitness calculations, as well as recombination of individuals due to crossover operator and mutation operator, by assigning such operations to multiple processors. By employing such a model, it should be possible to reduce calculation times while using existing computational methods to obtain approximate Stackelberg solutions without loss of calculation precision.

### 3.5 Improvement of lower level GA avoidance procedures

In the computational method for obtaining Stackelberg solutions proposed by the authors, by introducing a lower level GA avoidance procedure it was possible to reduce the number of unneeded rational reaction calculations, and so calculation times were reduced. That method, however, had problems related to the structure of storage regions for saving individuals belonging to the upper level GA and associated individuals belonging to the lower level GA indicating rational reactions, and to algorithms for avoidance procedures for lower level GAs. In this study, we made some improvements to those regions.

We will first discuss the region used for saving an upper level GA individual \( x \) and the rational reaction \( y(x) \) obtained by a lower level GA. Under previous methods, when multiple \( x \) having the same value within the population of a single generation are saved in the storage region, or when an upper level GA individual \( x^t \) is saved within the population of a differing generation, a threshold value \( y_{\text{max}} \) is set as the upper limit for the number of times that the rational reaction \( y(x) \) may be searched for, and the rational reaction was obtained by applying the lower level GA up to that number of times. Because of that, storage sufficient to save up to a number of rational reactions \( y_i(x), i = 1, \ldots, y_{\text{max}} \) equal to the threshold value was allocated. The only data required for application of the upper and lower level GAs, however, is that combination \((x, y_{\text{local}}(x))\) from among \( x \) and the rational reaction \( y_i(x) \) that maximizes the value of the objective function \( z_j(x, y) \) of the upper level DM. In this paper, therefore, we have introduced a storage region that eliminates unnecessary storage space, structured as shown below (Fig. 2).

![Figure 2: Storage for saving \( x \) and \( y(x) \)](image)

Here, \( x^t, i = 1, \ldots, x_{\text{max}} \) indicates those values of \( x \) that were used in the past for handling individuals of the upper level GA, and \( y^t(x^t), i = 1, \ldots, x_{\text{max}} \) indicates the values of the rational reactions associated with \( x^t \) obtained by the lower level GA. \( x^t_{\text{counter}} \in \{1, 2, \ldots, y_{\text{max}}\}, i = 1, \ldots, x_{\text{max}} \) indicates the number of times that the lower level GA was used to find the rational reaction \( y^t(x^t) \) for \( x^t \). \( x_{\text{max}} \) indicates the maximum number of DM1 decisions \( x \) saved, and \( y_{\text{max}} \) indicates the maximum number of times that the lower level GA can be repeatedly used to find the rational reaction \( y^t(x^t) \) for \( x^t \). \( z_1(x^t, y^t(x^t)) \) and \( z_2(x^t, y^t(x^t)) \) are stored \( x^t, y^t(x^t) \) values used in place of DM1 and DM2 objective functions.

By furthermore using an algorithm like the following, the number of applications of the lower level GA is reduced, and unnecessary calculation times eliminated.
Storage of the rational reaction $y(x)$ and lower level GA avoidance procedures

Step 1 If there exists in $x^i$ an upper level GA individual $x$, proceed to Step 2. If one does not exist, first check if the number of $x^i$ has reached $x_{max}$, and if so continue on to Step 3. If not, proceed to Step 4.

Step 2 If $x_{counter}^{'}$ has reached $y_{max}$, then the saved $y(x^i)$ is returned to the upper level GA as the rational reaction and the algorithm terminates. If not reached, proceed to Step 4.

Step 3 Select the least of the values $z_1(x^i, y(x^i))$ from the saved $x^i$, and take that $x^i$ value as $x^i$. After applying the lower level GA and thus obtaining the rational reaction $y(x)$ for $x$, if $z_1(x^i, y(x^i)) \leq z_1(x, y(x))$, save $x', y(x')$, $z_1(x, y(x'))$, $z_2(x, y(x'))$ in the storage region $x^k$, and terminate the algorithm.

Step 4 After obtaining the rational reaction $y(x)$ for $x$ by applying the lower level GA, save $x$, $y(x)$, $z_1(x, y(x))$, $z_2(x, y(x))$, and terminate the algorithm.

Implementation of the algorithm described above improved upon previous methods.

3.6 The algorithm for the improved computational method

The following is a summary of the algorithm used in the computational method after improvement. Here, the number of processors used in the experiment is taken to be $N_p$.

Step 1 Taking the generation of the upper level GA as $t_n := 0$, $N_u$ initial individuals are randomly generated.

Step 2 For each individual $x$ in the upper level GA determine whether or not to apply the lower level GA, and find the number of lower level GAs to apply $N_{al}$. For those individuals to which the lower level GA will be applied, calculate $N_{al}/N_p$, and based on that value allocate the upper level GA individuals to the available processors so that the number of individuals on each is approximately even. For each individual allocated to a processor, apply the lower level GA operations in Step 2-1 through Step 2-4, and obtain the rational reaction $y(x)$. For those $N_p - N_{al}$ individuals to which the lower level GA will not be applied, take the saved $y(x)$ as the rational reaction, and proceed to Step 4.

Step 2-1 Set $t_i := 0$. Randomly generate $N_l$ lower level GA individuals $y$, and take these as the initial population of the lower level GA. Proceed to Step 2-2.

Step 2-2 Use the $x$ given as the upper level GA individual and the $y$ generated by the lower level GA to calculate the DM2 objective function value, and after applying linear scaling use that value to generate an individual. Proceed to Step 2-3.

Step 2-3 If $t_1$ has exceeded the previously defined a maximum number of generation $M_1$, take the individual with the best fitness value as the optimal individual $y(x)$, and proceed to Step 2-4. Otherwise, apply crossover operator and mutation operator to each lower level GA individual, let $t_1 = t_1 + 1$, and proceed to Step 2-2.

Step 2-4 For each processor, if application of the lower level GA for each assigned individual $x$ is not complete, return to Step 2-1. If it is complete, perform synchronization between the processors, and proceed to Step 3.

Step 3 Calculate the values for the DM1 and DM2 objective functions using the lower level rational reaction $y(x)$ obtained by operation of the lower level GA, and the individual $x$ of the upper level GA. When doing so, each processor is assigned $N_{al}/N_p$ calculation. After performing synchronization between the processors, perform the procedures required to save $x$ and its rational reactions $y(x)$ to the storage region, and proceed to Step 4.

Step 4 Calculate the DM1 objective function value for each upper level GA individual $x$, and after performing linear scaling apply the clustering method to measure the level of convergence of the individuals. Depending upon the degree of convergence, calculate the fitness value of each individual. Proceed to Step 5.

Step 5 If $t_n$ has exceeded the previously set a maximum number of generation $M_n$, then terminate the algorithm. In that case, the individual obtained up to that generation with the best fitness value is taken as the optimal individual $(x, y)$. Otherwise, proceed to Step 6.

Step 6 Reproduction operator is performed using the fitness values of each individual of the upper level GA. After determining the pairs upon which crossover operator will be performed, assign $N_u/N_p$ pairs to each processor, and perform crossover operator and mutation operator. Next, perform synchronization between the processors, and return to Step 2 with $t_n := t_n + 1$.

4 Conclusion

This paper has focused on a two-level 0-1 programming problem in which there is not coordination between the
decision maker (DM) at the upper level and the decision maker at the lower level. The authors proposed a modified computational method that solves problems related to computational methods for obtaining the Stackelberg solution. Specifically, in order to shorten the computational time of a computational method implementing a genetic algorithm (GA) proposed by the authors, a parallel genetic algorithm is introduced with respect to the upper level GA, which handles decision variables for the upper level DM. Parallelization of the lower level GA is also performed along with parallelization of the upper level GA. The proposed algorithm is also improved in order to eliminate unnecessary computation during operation of the lower level GA, which handles decision variables for the lower level DM. In order to verify the effectiveness of the proposed method, we propose a comparison with existing methods by performing numerical experiments to verify both the accuracy of the solution and the time required for the computation.

References


