EVOLUTIONARY COMPUTATION AND CONSTRAINED OPTIMIZATION

Ruhul Sarker, Thomas P. Runarsson* and Charles Newton
School of Computer Science
UC, University of New South Wales, ADFA
Northcott Drive, Canberra, ACT 2600, Australia
<ruhul@cs.adfa.edu.au>
**Current Address: Dept. of Mechanical Engineering, University of Iceland, Iceland*

Abstract

We consider a class of constrained nonlinear integer programs, which arise in manufacturing batch sizing problems. We investigate the use of genetic algorithms (GAs) for solving these models. In this paper, a new penalty function based GA method is proposed. Both binary and real coded genetic algorithms with six other penalty functions are developed and compared with the proposed method. The real coded genetic algorithm works well for all penalty functions compared to binary coding and the new method shows the best performance. Numerical examples are provided and computational experiences are discussed.

1. Introduction

Consider a constrained nonlinear programming problem.

Problem P1	
Minimize	$f(\boldsymbol{x})$
Subject to	$\boldsymbol{g}(\boldsymbol{x}) \leq \boldsymbol{0}$
	$\boldsymbol{h}(\boldsymbol{x}) = \boldsymbol{0}$
	$\boldsymbol{x} \in X$

where $\boldsymbol{g}$ is a vector function with components $g_1, \ldots, g_m$ and $\boldsymbol{h}$ is a vector function with components $h_1, \ldots, h_l$. Here $f, g_1, \ldots, g_m, h_1, \ldots, h_l$ are functions on E_n and X is a nonempty set in E_n. The set X represents simple constraints that could be easily handled explicitly, such as lower and upper bounds on the variables.

In contrast to the unifying role of the simplex method in linear programming, there exists no unique approach to nonlinear optimization. Numerous algorithms have been proposed to solve the nonlinear programming problem such as method of steepest decent [1] for solving unconstrained optimization, method of Lagrange multipliers [2] for solving equality constrained optimization, and the gradient projection method [3]. The gradient projection method provides a good procedure for moving along the boundary of linearly constrained regions. In order to solve a general nonlinear constrained programming problem, many different techniques have been proposed. Courant [4] initially suggested the penalty function, which studies constrained problems via related unconstrained ones. In the penalty function method, a penalty term is added to the objective function for any violation of the constraints.

Similar to the penalty function, Carroll [5] proposed the barrier function approach in which a barrier term that prevents the points generated from leaving the feasible region is added to the objective function. This method can be used for inequality constrained problems. Abadie and Carpenter [6] generalized the reduced gradient methods to nonlinear constraints. The method is an extension of the Wolfe [7] algorithm to accommodate both nonlinear objective functions and nonlinear constraints. These methods have been studied in detail in the past and have been found to have weaknesses. In the penalty and barrier function methods, the unconstrained subproblem becomes extremely ill-conditioned for extreme values of the penalty/barrier parameters. Reduced gradient methods have difficulties following the boundary of high nonlinear constraints. These methods rely on local gradient information; the optimum is the best in the neighborhood of the current point. To enhance the reliability and robustness of the search technique in constrained optimization, a GA based search method incorporating several penalty functions is tested in this paper.

We consider a manufacturing batch sizing problem. A single raw material unconstrained manufacturing batch sizing model was developed by Sarker and Khan [8]. Later, this model was modified by Sarker and Newton [9] to incorporate the transportation module for finished products delivery. In this paper, we considered a problem similar to Sarker and Khan [8] but with multiple raw materials and several constraints to make the situation more realistic. The new problem forms a constrained nonlinear integer program. Considering the complexity of solving such a model, we investigated the use of genetic algorithms (GAs). We used both binary coding and real coded genetic algorithms with different crossovers and mutations. We also proposed a new penalty function based GA method. This method provides better solutions than the existing methods. It is found in the experimentation that the real coded genetic algorithm works well for this batch sizing problem. The detailed computational experiences are presented.

The organization of the paper is as follows: Following this introduction, this paper presents a brief introduction on penalty function methods and then on genetic algorithms. In section 4, the batch sizing problem is described, and its mathematical formulation is presented. In section 5, the different transformation methods are provided. The model solving approach using GAs is implemented in section 6. The computational experiences are presented in section 7. Finally, conclusions are provided in section 8.

2. Penalty Function Method

The penalty function method has been well known for solving constrained optimization problems for decades. The approach converts the problem into an equivalent unconstrained problem and then solves it using a suitable search algorithm. Two basic types of penalty functions exist: exterior penalty functions, which penalize infeasible solutions, and interior penalty functions, which penalize feasible solutions. We will discuss only the exterior penalty functions in this paper as the implementation of interior penalty functions is considerably more complex for multiple constraint cases [10].

Three degrees of exterior penalty functions exist: (i) barrier methods in which no infeasible solution is considered, (ii) partial penalty functions in which a penalty is applied near the feasibility boundary, and (iii) global penalty functions that are applied throughout the infeasible region [11]. In general a penalty function approach places the constraints into the objective function via a penalty parameter in such a way that it penalizes any violation of the constraints. A general form of the unconstrained problem is as follows:

Problem P2
Minimize $f_p(\mathbf{x}) = f(\mathbf{x}) + \mu\, \alpha(\mathbf{x})$

Subject to $\boldsymbol{x} \in X$

where $\mu > 0$ is a large number, and $\alpha(\boldsymbol{x})$ is the penalty function.

We define the problem *P2* as a penalty problem and the objective function of *P2* as the penalized objective function $(f_p(\boldsymbol{x}))$ which is the simple sum of the unpenalized objective function and a penalty (for a minimization problem). A suitable penalty function incurs a positive penalty for infeasible points and no penalty for feasible points. The penalty function α is usually of the form:

$$\alpha(\boldsymbol{x}) = \sum_{i=1}^{m} [maximum\{0, g_i(\boldsymbol{x})\}]^p + \sum_{i=1}^{l} |h_i(\boldsymbol{x})|^p \qquad \ldots\ldots \quad \ldots\ldots \quad \ldots\ldots\ldots \quad (1)$$

where p is a positive integer.

If the penalty parameter $\mu = 0$, the problem *P2* forms an unconstrained optimization problem. Normally an independent penalty parameter is imposed for the violation of each constraint i, instead of having a constant μ for all constraints.

The solution to the penalty problem can be made arbitraily close to the optimal solution of the original problem by choosing μ sufficiently large. However, if we choose a very large μ and attempt to solve the penalty problem, we may get into some computational difficulties of ill-conditioning [12]. With a large μ, more emphasis is placed on feasibility, and most procedures will move quickly toward a feasible point. Even though this point may be far from optimal, premature termination could occur.

As a result of the above difficulties associated with large penalty parameters, most algorithms use penalty functions that employ a sequence of increasing penalty parameters $[\mu_{k+1} = \beta\mu_k]$. With each new value of the penalty parameter, an optimization technique is employed, starting with the optimal solution corresponding to the previously chosen parameter value.

It is very difficult to choose an appropriate β [13]. A lower value of β means a higher number of iterations is required to solve the penalty problem. A number of methods for selecting β are discussed in a later section.

3. Introduction to GA

During the last two decades there has been a growing interest in algorithms which are based on a principle of evolution- survival of the fittest. A common term, accepted recently, refers to such techniques as *evolutionary computation* (EC) methods. The best known algorithms in this class include genetic algorithms, evolutionary programming, evolution strategies, and genetic programming. There are also many hybrid systems which incorporate various features of the above paradigms, and consequently are hard to classify; they are just referred to as EC methods [14].

The methods of EC are stochastic algorithms whose search methods model some natural phenomena: genetic inheritance and darwinian strife for survival. GAs follow a step-by-step procedure that mimic the process of natural evolution, following the principles of natural selection and "survival of the fittest". In these algorithms a population of individuals (potential solutions) undergoes a sequence of unary (mutation type) and higher order (crossover type) transformations. These individuals strive for survival. A selection scheme, biased towards fitter individuals, selects the next generation. This new generation contains a higher proportion of the characteristics possessed by the "good" members of the previous generation, in this way good characteristics are spread over the population and mixed with

other good characteristics. After some number of generations, the program converges and the best individuals represent a near-optimum solution. The GA procedure is shown on the next page.

```
begin
  t ← 0
  initialize Population (t)
  evaluate Population (t)
  while (not terminate-condition) do
  begin
    t ← t + 1
    select Population (t) from Population (t-1)
    alter Population (t)
    evaluate Population (t)
  end
end
```

4. Batch Sizing Problem (BSP)

We consider a batch manufacturing environment that processes raw materials procured from outside suppliers to convert them into finished products for retailers. The manufacturing batch size is dependent upon the retailer's sales volume (/market demand), unit product cost, set-up cost, and inventory holding cost. The raw material purchasing lot size is dependent upon the raw material requirement in the manufacturing system, unit raw material cost, ordering cost and inventory holding cost. Therefore, the optimal raw material purchasing quantity may not be equal to the raw material requirement for an optimal manufacturing batch size. To operate the manufacturing system optimally, it is necessary to optimise the activities of both raw material purchasing and production batch sizing simultaneously, taking all operating parameters into consideration. The detail problem description can be found in [8 & 9].

The unconstrained models under various situations are presented in Sarker and Khan [8] and Sarker and Newton [9], and the single raw material constrained model was recently developed by Sarker and Newton [15]. The mathematical formulation for the multiple raw materials constrained model is presented in this section. The notations used in developing the model are as follows:

D_p = demand rate of a product p, units per year
P_p = production rate, units per year (here, $P_p > D_p$)
Q_p = production lot size
H_p = annual inventory holding cost, \$/unit/year
A_p = setup cost for a product p (\$/setup)
r_i = amount/quantity of raw material i required in producing one unit of a product
D_i = demand of raw material i for the product p in a year, $D_i = r_i D_p$
Q_i = ordering quantity of raw material i
A_i = ordering cost of a raw material i
H_i = annual inventory holding cost for raw material i
PR_i = price of raw material i
Q^*_i = optimum ordering quantity of raw material i

x_p = shipment quantity to customer at a regular interval (units/shipment)
L = time between successive shipments = x/D_p
T = cycle time measured in years = Q_p/D_p
m_p = number of full shipments during the cycle time = T/L
n_i = number of production cycles which consume one lot of raw material $i = Q_i/Q_p$
s_i = space required by one unit of raw material i
TC = total cost of the system

The mathematical model is presented below:

$$Minimize\ \mathrm{TC} = \frac{D_p}{m_p x_p}(A_p + \sum_i \frac{A_i}{n_i}) + \frac{m_p x_p}{2}((\frac{D_p}{P_p} + 1)H_p + \sum_i r_i H_i(\frac{D_p}{P_p} + n_i - 1)) - \frac{x_p}{2} H_p$$

Subject to:

$$\sum_i n_i r_i m_p x_p s_i \le Raw\_s\_Cap$$

$$\sum_i n_i r_i m_p x_p s_i \ge Min\_Truck\_Load$$

$$m_p x_p s_p \le Finish\_S - Cap$$

m_p and n_i are integers and greater than zero.

This is clearly a nonlinear-integer program. The first constraint indicates that the storage space required by the raw materials must be less than the space specified for raw materials. The second constraint represents the lower limit truck load for transportation and the third constraint is for the finished products storage capacity. If we assign $i = 1$, then the model represents a single raw material batch sizing problem.

5. Transformation Methods

The transformation method, from a constrained to an unconstrained problem, in evolutionary algorithms is almost the same as the classical approach but with a slight difference. The value of μ_k is changed as a function of generation k, but not is equal to $\beta\mu_{k-1}$.

Some selected transformation techniques used in the evolutionary computation literature for solving constrained optimization problems are presented below. Most, if not all, are of the exterior kind which will allow the initial population of solutions to be partially or completely infeasible. In some techniques, the controlling parameters are updated in a predetermined manner. The controlling (or penalty) parameters μ_k may be updated each generation or every n generations. We present our new approach in this section.

5.1 Static Penalties

The method of static penalties [16] assumes that for every constraint we establish a family of intervals that determine the appropriate penalty coefficient.

- For each constraint, create several (n) levels of violation.

- For each level of violation and for each constraint, create a penalty coefficient μ_{ij} (i = 1, 2, …, $m+l$; j = 1, 2, .., n); higher levels of violation require larger values of these coefficients.
- Start with a random population of individuals (feasible or infeasible).
- Evolve the population; evaluate individuals.

It is clear that the results are parameter dependent. It is quite likely that for a given problem there exists one optimal set of parameters for which the system returns a feasible near-optimum solution; however, it might be hard to find. A limited set of experiments reported by Michalewicz [13] indicate that the method can provide good results if violation levels and penalty coefficients are tuned to the problem.

We use three levels of fixed penalty coefficients for each constraints: 0.5, 0.5×10^3 and 0.5×10^6.

5.2 Dynamic Penalties

Joines and Houck [17] proposed dynamic penalties. The authors assumed that $\mu_k = (Ck)^\alpha$, where C and α are constants. The constant p in equation 1 has a value that is greater than one. A reasonable choice for these parameters is $C = 0.5$, $\alpha = p = 2$. This method requires a much smaller number (independent of the number of constraints) of parameters than the first method. Also, instead of defining several levels of violation, the pressure on infeasible solutions is increased due to the $(Ck)^\alpha$ component of the penalty term: towards the end of the process (for high values of the generation number k), this component assumes large values.

5.3 Annealing Penalties

The method of annealing penalties, called Genocop II (for Genetic algorithms for Numerical Optimization of Constrained Problems) is also based on dynamic penalties and was described by Michalewicz and Attia [18] and Michalewicz [19]. In this system, a fixed μ_k is used for all constraints of a given generation k, where $\mu_k = 1/2\tau$. An initial temperature τ is set and is used to evaluate the individuals. After a certain number of generations, the temperature τ is decreased and the best solution found so far serves as a starting point for the next iteration. This process continues until the temperature reaches freezing point.

Genocop proved its usefulness for linearly constrained optimization problems; it gave a surprisingly good performance for many functions [19 & 20]. Michalewicz and Attia [18] proposed the control parameters to be $\tau_{k+1} = 10.\tau_k$, where $\tau_0 = 1$ and τ_k remains constant once it reaches 1×10^6.

5.4 Adaptive Penalties

Adaptive transformation attempts to use the information from the search to adjust the control parameters. This is usually done by examining the fitness of feasible and infeasible members in the current population.

Bean and Hadi-Alouane proposed an adaptive penalty method in 1992 which uses the feedback from the search process [see 21]. This method allows either an increase or a decrease of the imposed penalty during evolution as shown below. This involves the selection of two constants, β_1 and β_2 ($\beta_1 > \beta_2 > 1$), to adaptively update the penalty function multiplier, and the evaluation of the feasibility of the best solution over successive intervals

of N_f generations. As the search progresses, the penalty function multiplier is updated every N_f generations based on whether or not the best solution was feasible during that interval. Specifically, the penalty function is as follows;

$$\mu_{k+1} = \begin{cases} \mu_k \beta_1 & \text{if previous } N_f \text{ generations have only } \textbf{infeasible} \text{ best solutions} \\ \mu_k / \beta_2 & \text{if previous } N_f \text{ generations have only feasible best solutions} \\ \mu_k & \text{Otherwise} \end{cases}$$

The parameters were chosen to be similar to the above methods with $N_f = 3$, $\beta_1 = 5$, $\beta_2 = 10$ and $\mu_0 = 1$. In our experimentation, the μ values were increased by 10.

5.5 Death Penalty

The death penalty method just rejects infeasible individuals. It can be interpreted as a truncation selection based on $\alpha(\boldsymbol{x})$ and then followed by a selection procedure based on $f(\boldsymbol{x})$. In this method, the initial population must be feasible. In the experiments reported by Michalewicz [13] it was shown that the method generally gives a poor performance. Also, the method is not as stable as others; the standard deviation of solutions returned is relatively high.

5.6 Superiority of Feasible Points

The method of superiority of feasible points was developed by Powell and Skolnick [22] and is based on a classical penalty approach, with one notable exception. Each individual is evaluated by the formula:

$$f_p(\boldsymbol{x}) = f(\boldsymbol{x}) + \mu\,[\alpha(\boldsymbol{x}) + \theta(t, \boldsymbol{x})]$$

where μ is a constant; however, the component $\theta(t, \boldsymbol{x})$ is an additional iteration-dependent function that influences the evaluation of infeasible solutions. The point is that the method distinguishes between feasible and infeasible individuals by adopting an additional heuristic rule: For any feasible individual $\boldsymbol{x}$ and any infeasible individual $\boldsymbol{y}$, $f_p(\boldsymbol{x}) < f_p(\boldsymbol{y})$ (i.e., any feasible solution is better than any infeasible one). The penalties are increased for infeasible individuals. The method performs reasonably well. However, for some case problems, the method may have some difficulty in locating a feasible solution [13].

5.7 A New Method

At this point it seems reasonable to re-examine our goals and what transformations mean for the evolutionary algorithm. In principle these transformations will influence selection only.

In this paper the commonly used binary tournament selection is applied. In binary tournament selection two individuals are chosen randomly from the population and the better of them is allowed to survive to the next generation. Consider this comparison between two individuals 1 and 2 transformed to take the general form from problem *P2*:

$$f_1(\boldsymbol{x}) + \mu\,\alpha_1(\boldsymbol{x}) \geq f_2(\boldsymbol{x}) + \mu\,\alpha_2(\boldsymbol{x}), \qquad \text{..... .... (2)}$$

for a given penalty parameter $\mu > 0$. From equation (2) we can write $\mu \leq [f_1(\boldsymbol{x}) - f_2(\boldsymbol{x})]/(\alpha_2(\boldsymbol{x}) - \alpha_1(\boldsymbol{x}))$. Now suppose $\mu_c = [f_1(\boldsymbol{x}) - f_2(\boldsymbol{x})]/(\alpha_2(\boldsymbol{x}) - \alpha_1(\boldsymbol{x}))$. This relation holds the following conditions:

1. If $\alpha_1 = \alpha_2$, common for feasible solutions, then μ_c is infinity. The competition is decided by objective function.
2. If $f_2 = f_1$, the value of μ_c is zero. The competition is decided by penalty function.
3. If $f_2 < f_1$ and $\alpha_2 < \alpha_1$. μ_c is negative in this case. Comparison is independent of μ.
4. If $f_2 < f_1$ and $\alpha_1 < \alpha_2$, μ_c is positive with objective function domination.
5. If $f_2 > f_1$ and $\alpha_1 > \alpha_2$, μ_c is positive with penalty function domination.

Only for the last two conditions is the value for μ critical to the outcome of the comparison and this value is μ_c. Evolutionary algorithms are a population based search method. Population information can be used to estimate the most appropriate value for μ. This value would be the one that attempts to balances the number of competitions decided by the objective and penalty function. This value may be approximated by simulating the binary tournament selection and computing for each comparison the critical penalty coefficient μ_c. Then for the actual selection the μ used will be the average of these values, or more precisely

$$\mu = \frac{1}{M} \sum_{i=1,\mu_c>0}^{N} \mu_c(i), \quad \ldots\ldots \quad \ldots\ldots \quad \ldots\ldots \quad (3)$$

where N is the number of simulated competitions held and $M<N$ is the number of competitions for which $\mu_c > 0$. The number of simulated competitions held is at least twice the size of the population.

6. Solving BSP using Genetic Algorithms

It is generally accepted that any GA to solve a problem must have five basic components:

- problem representation,
- a way to create an initial population of solutions,
- an evaluation function rating solutions in terms of their "fitness",
- genetic operators that alter the genetic composition of parents during reproduction, and
- values for the parameters (population size, probabilities of applying genetic operators, etc.)

Two types of evolutionary algorithms were tested, for each of the methods discussed in the previous section: a Simple Genetic Algorithm (SGA) and a Real Coded Genetic Algorithm (RGA). In the light of the above five components, these two algorithms are discussed briefly below.

6.1 Problem Representation

The first hurdle to overcome in using GAs is *problem representation*- we must represent our problem in such a way that is suitable for handling by a GA, which works with strings of symbols that in structure resemble chromosomes. The representation often relies on binary coding. GAs work with a population of competing strings, each of which represents a potential solution for the problem under investigation. The individual strings within the population are gradually transformed using biological based operations. For

example, two strings might 'mate' and produce an offspring that combines the best features of its parents. In accordance with the law of the survival of the fittest, the best-performing individual in the population will eventually dominate the population.

6.2 Initialize the Population

There are no strict rules for determining the population size. Larger populations ensure greater diversity but require more computing resources. We use a population size of 50, a commonly used size (Back, 1996, page 123). Once the population size is chosen, then the initial population must be randomly generated. For binary coding, the random number generator generates random bits (0 or 1). Each individual in the population is a string of n bits. For real coding, the actual numbers are generated randomly.

6.3 Calculate fitness

Now that we have a population of potential problem solutions, we need to see how good they are. Therefore, we calculate a fitness, or performance, for each string. The fitness function is the penalty objective (f_p) function in our case. For binary coding, each string is decoded into its decimal equivalent. This gives us a candidate value for the solution. This candidate value is used to calculate the fitness value.

6.4 Selection and Genetic Operators

Genetic algorithms work with a population of competing problem solutions that gradually evolve over successive generations. Survival of the fittest means that only the best-performing members of the population will survive in the long run. In the short run we merely tip the odds in favour of the better performers; we do not eliminate all the poor performers. This can be done in a variety of ways. We use a binary tournament selection, where two individuals are selected at random from the population pool and the better one is allowed to survive to the next generation.

The chromosomes, which survive the selection step undergo genetic operations, crossover and mutation. *Crossover* is the step that really powers the GA. It allows the search to fan out in diverse directions looking for attractive solutions and permits two strings to 'mate'. This may result in offspring that are 'fitter' than their parents. Crossover is accomplished for binary coding in four small steps as follows.

- two potential parents are randomly chosen from the population;
- generate the crossover probability to answer 'yes, perform crossover' or 'no';
- if the answer is 'yes', randomly choose a cutting point; and
- cut each string at the cutting point and create two offspring by gluing parts of the parents.

Usually, the classical 1-point crossover is used. Sometimes more than one cutting point is used. For example, in case of two 2-point crossovers, the first segment of the first chromosome is followed by the second segment of the second chromosome.

We use a Heuristic crossover for real coding GAs. This is a unique crossover for the following reasons:

- it uses values of the penalty objective function in determining the direction of the search,

- it produces only one offspring, and
- it may produce no offspring at all. There is a 80% probability that a *variable* gets "crossed".

The operator generates a single offspring x_3 from two parents x_1 and x_2 according to the following rule:

$$x_3 = rounded[r.(x_2 - x_1) + x_2,]$$

where r is a random number between 0 and 1, and the parent x_2 is no worse than x_1. Actually, when the generated value x_3 is out of bounds (upper and lower bounds) it is set equal to the corresponding violated bound. Also, note that x_1 and x_2 are "*parent variables*" *not parents*. Also x_3 is rounded after crossing to integer. This is not required for binary coding. Rounding is also performed after the mutation operator.

Mutation introduces random deviations into the population. For binary coding, mutation changes a 0 into a 1 and vice versa. Mutation is usually performed with low probability, otherwise it would defeat the order building being generated through selection and crossover. Mutation attempts to bump the population gently into a slightly better course. *Nonuniform mutation* is also used in real coding GA, where

$$x_k^{t+1} = \begin{cases} x_k^t + \Delta(t, u(k) - x_k) & \text{if a random binary digit is } 0 \\ x_k^t + \Delta(t, x_k - l(k)) & \text{if a random binary digit is } 1 \end{cases}$$

for k = 1, …, n. The *u(k)* and *l(k)* are upper and lower domain bounds of the variable x_k. The function $\Delta(t,y)$ returns a value in the range [0,*y*] such that the probability of $\Delta(t,y)$ being close to 0 increases as *t* increases (*t* is the generation number). This property causes this operator to search the space uniformly initially (when *t* is small), and very locally at a later stage. Michalewicz et al. [20] used the following function for experimentation:

$$\Delta(t, y) = y.r.(1 - \frac{t}{T})^b$$

where *r* is a random number from (0..1), *T* is the maximal generation number, and *b (b=6.r+1 was used)* is a system parameter determining the degree of nonuniformity.

6.5 Parameters of GA

All GA runs have the following standard characteristics:

- Probability of crossover: 1.0
- Probability of mutation: 1/(string length of the chromosome)
- Population size: 50
- Number of generations in each run: 200
- Number of independent runs: 300
- for SGA: binary coding, two point crossover, and bit-wise mutation.
- for RGA: heuristic crossover and non-uniform mutations

7. Computational Results

The mathematical model presented in Section 4 is solved for a test problem of a single product with three raw materials using seven different penalty functions based on binary and real coded GAs described in Section 5. The data used for the model are as follows:

Sample Data: $D_p = 4x10^6$; $P_p= 5x10^6$; $A_p=$ \$ 50.00 ; $H_p=$ \$ 1.20 , x_P = 1,000 units, S_p = 1.00. Other data are as follows:

	Single Raw Material	Multiple Raw Materials		
	For i = 1	Raw material 1	Raw material 2	Raw material 3
A_i	3,000	3,000	2,500	4,600
H_i	1.00	1.00	1.20	1.50
s_i	1.00	1.00	1.10	1.20
r_i	1.00	0.5	0.2	0.3

The right hand sides (RHS) for different test problems are given below:

RHS	Problem#1	Problem#2	Problem#3	Problem#4	Problem#5
Raw_S_Cap	400,000	300,000	200,000	150,000	130,000
Truck_Min_load	200,000	100,000	40,000	30,000	25,000
Fin_S_Cap	20,000	18,000	17,000	15,000	13,000

For single raw material with RHS similar to problem#1, the best solution found was: m = 13 and n = 10 with TC = \$1.8483E5. For multiple raw materials case, the best solution found for problem#1 was: m = 15, n1 = 15, n2 = 19, and n3 = 19 with TC = \$3.3471E5 and for problem#5 was: m = 12, n1 = 8, n2 = 11, and n3 = 12 with TC = \$4.2837E5.

A sample detail comparison of results, for single raw material, provided by seven different penalty functions are shown in Table 1 for real coded GAs and in Table 2 for binary coded GAs. The minimum, presented in column 2, means the best objective function value obtained in any of 300 runs and maximum, presented in column 6, indicates the worst value. The statistics (like mean, standard deviation and median) of the objective function values based on 300 runs are also produced to sense the variability of the target value. The right most columns represent the number of times we hit the optimal solution out of 300 runs, expressed in percentage.

The detail sample comparison of results, for multiple raw materials case problem#1, provided by seven different penalty functions are shown in Table 3 for real coded GAs and in Table 4 for binary coded GAs.

Table 1: Results of single raw material case using real coded GAs

Method	Minimum	Mean	Std. Dev.	Median	Maximum	% times Opt. Sol.
New/ Proposed	1.8483	1.8483	0.0000	1.8483	1.8483	100
Death Penalty	1.8483	1.8483	0.0000	1.8483	1.8483	100
Superiority of Feasible points	1.8483	1.8492	0.0047	1.8483	1.8868	96
Dynamic Penalty	1.8483	1.8483	0.0000	1.8483	1.8483	100
Annealing Penalty	1.8483	1.8483	0.0000	1.8483	1.8483	100
Static Penalty	1.8483	1.8483	0.0000	1.8483	1.8483	100
Adaptive Penalty	1.8483	1.8483	0.0000	1.8483	1.8483	100

Table 2: Results of single raw material case using binary coded GAs

Method	Minimum	Mean	Std. Dev.	Median	Maximum	% times Opt. Sol.
New/ Proposed	1.8483	1.8484	0.0010	1.8483	1.8651	100
Death Penalty	1.8483	1.8558	0.0320	1.8483	2.2653	3
Superiority of Feasible points	1.8483	1.9266	0.1521	1.8868	3.5813	22
Dynamic Penalty	1.8483	1.8484	0.0014	1.8483	1.8651	99
Annealing Penalty	1.8483	1.8484	0.0014	1.8483	1.8651	99

Static Penalty	1.8483	1.8484	0.0010	1.8483	1.8651	100
Adaptive Penalty	1.8483	1.8484	0.0014	1.8483	1.8651	99

Table 3: Results of multiple raw materials case problem#1 using real coded GAs

Method	Minimum	Mean	Std. Dev.	Median	Maximum	% times Opt. Sol.
New/ Proposed	3.3471	3.3471	0.0000	3.3471	3.3471	100
Death Penalty	3.3471	3.3471	0.0000	3.3471	3.3471	100
Superiority of Feasible points	3.3471	3.3472	0.0002	3.3472	3.3482	35
Dynamic Penalty	3.3471	3.3471	0.0000	3.3471	3.3471	100
Annealing Penalty	3.3471	3.3471	0.0000	3.3471	3.3471	100
Static Penalty	3.3471	3.3471	0.0000	3.3471	3.3471	100
Adaptive Penalty	3.3471	3.3471	0.0000	3.3471	3.3471	100

Table 4: Results of multiple raw materials case problem#1 using binary coded GAs

Method	Minimum	Mean	Std. Dev.	Median	Maximum	% times Opt. Sol.
New/ Proposed	3.3472	3.3773	0.0210	3.3853	3.4344	0
Death Penalty	3.3471	3.4478	0.0881	3.4151	3.6287	1
Superiority of Feasible points	3.3561	3.4420	0.0535	3.4314	3.6462	0
Dynamic Penalty	3.3471	3.3773	0.0232	3.3795	3.5313	1
Annealing Penalty	3.3471	3.3802	0.0256	3.3933	3.6076	1
Static Penalty	3.3471	3.3789	0.0219	3.3850	3.4661	1
Adaptive Penalty	3.3471	3.3774	0.0223	3.3961	3.4270	4

The results, in terms of percentage of times the optimal solution was obtained, are always better with real coded GAs as compared to binary coded GAs for all seven penalty functions considered in this paper. All the penalty functions, except *Superiority of Feasible Points*, worked extremely well with real coded GAs for the above two test problems. The results are worse with tighter constrained problems which are demonstrated in Table 5. The *Death Penalt*y method along with *Superiority of Feasible Points* performed badly for tighter constraints. However, all seven penalty functions, with real coding GAs, provide unique optimal solutions for all the test problems. These solutions are acceptable irrespective of how many times the optimal solutions are generated in a single run. The binary coded GAs failed to produce any optimal solution for the tighter constrained test problems. The proposed method seems a bit superior for the tighter constrained problems.

Table 5: Percentage times optimum solution obtained by different methods

Method	Problem#1	Problem#2	Problem#3	Problem#4	Problem#5
New/ Proposed	100	100	80	64	49
Death Penalty	100	90	9	6	5
Superiority of Feasible points	36	30	7	3	2
Dynamic Penalty	100	100	76	52	42
Annealing Penalty	100	100	74	54	41
Static Penalty	100	100	78	58	37
Adaptive Penalty	100	100	73	58	38

8. Conclusions

We considered a single product- multiple raw materials joint batch sizing problem. The mathematical programming model of this problem forms a constrained nonlinear integer program. It is difficult to solve such a complex model using the existing algorithms. In this

paper, we investigate the use of genetic algorithms (GAs) for solving such a model with one product and single or three raw materials. We used both binary and real coded genetic algorithms with six different penalty functions reported in the literature. We also presented a new method in this paper. Several test problems were solved to demonstrate the use of GAs in solving BSP. The results were compared and analysed for all seven penalty functions with binary and real coded GAs. The real coded genetic algorithms worked well compared to binary coding. Among the penalty functions, the Death Penalty method and Superiority of Feasible Points were found to be worse. The new method showed its superiority for tight constrained problems.

9. Acknowledgement

This work is supported by UC special research grant, ADFA, University of New South Wales, Australia, awarded to Dr. Ruhul Sarker. We like to thank Prof. Xin Yao for many useful comments during conduct of this research.

10. References

1. T. Back, *Evolutionary Algorithms in Theory and Practice*, Oxford University Press, New York, 1996.
2. A. Cauchy, Method General pour la Resolution des Systeme's d'Equations Simulanees, *Compt. Rend.,* **25** (1847) 536-538.
3. R. Courant and D. Hilbert, Methods of Mathematical Physic, *Interscience*, New York, **1** 1953.
4. J. B. Rosen, The Gradient Projection Method for Nonlinear Programming, Part-1: Linear Constraints, *J. Soc. Ind. Appl. Math.,* **9** (1960) 181-217.
5. R. Courant, Variational Methods for the Solution of Problems of Equilibrium and Variations, *Bull. Am. Math. Soc.*, **49** (1960) 1-23.
6. C. W. Carroll, The Created Response Surface Technique for Optimizing Nonlinear Restrained Systems, *Operations Research,* **9** (1961) 169-184.
7. J. Abadie and J. Carpenter, Generation of the Wolfe Reduced Gradient Method to the Case of Nonlinear Constraints, *In Optimization,* R. Fletcher(ed), 1969.
8. P. Wolfe, Methods of Nonlinear Programming, *Notices Am. Math. Soc.*, **9** (1962) 308.
9. R. A. Sarker and L. R. Khan, An Optimal Batch Size for a Manufacturing System Operating Under a Periodic Delivery Policy, *Conference of the Asia-Pacific Operational Research Societies (APORS)*, Melbourne, 1997.
10. R. A. Sarker and C. Newton, Determination of Optimal Batch Size for a Manufacturing System, *Progress in Optimization II*, Yang, Mees, Fisher & Jennings (editted), Kluwer Academic Publishers (in press), 1999.
11. A. E. Smith and D. W. Coit, Constraint-Handling Techniques: Penalty Functions, *Handbook of Evolutionary Computation,* Release 97/1, IOP Publishing Ltd and Oxford University Press, (1997) C5.2:1-2:6.
12. H-P Schwefel, *Evolution and Optimum Seeking*, Wiley, New York, (1995) 16.
13. M. S. Bazaraa and C. M. Shetty, *Nonlinear Programming: Theory and Algorithms*, John Wiley & Sons, New York, Chapter-9, 1979.
14. Z. Michalewicz, Genetic Algorithms Numerical Optimization and Constraints, *Proc. 6th Int. Conf. on Genetic Algorithms* (Pittsburgh, PA, July 1995) ed L J Eshelman (San Mateo, CA: Morgan Kaufmann), 151-158.
15. M. Khouja, Z. Michalewicz, and M. Wilmot, The Use of Genetic Algorithms to Solve the Economic Lot Size Scheduling Problem, *European Journal of Operational Research,* **110** (1998) 509-524.
16. R. A. Sarker and C. Newton, Genetic Algorithms for Solving Economic Lot Size Problems, Accepted for publication in *the Proc. of the 26th International conference on Computers and Industrial Engineering*, Melbourne, Australia, December 1999.
17. A. Homaifar, C. X. Qi, and S. H. Lai, Constrained Optimization Vis Genetic Algorithms, *Simulation*, April (1994) 242-253.

18. J. A. Joines and C. R. Houck, On the Use of Non-Stationary Penalty Functions to Solve Nonlinear Constrained Optimization Problems With GAs, *Proc. of the IEEE ICEC*, (1994) 579-584.
19. Z. Michalewicz and N. Attia, Evolutionary Optimization of Constrained Problems, In A. V. Sebald & L. J. Fogel (Eds.), *Proc. of the 3rd Annual Conf. on Evolutionary Programming*, River Edge, NJ: World Scientific, (1994) 98-108.
20. Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs*, 3rd ed., New York, Springer-Verlag, 1996.
21. Z. Michalewicz, T. Logan, and S. Swaminathan, Evolutionary Operators for Continuous Convex Parameter Spaces, In A. V. Sebald & L. J. Fogel (Eds.), *Proc. of the 3rd Annual Conf. on Evolutionary Programming*, River Edge, NJ: World Scientific, (1994) 84-97.
22. Z. Michalewicz, and M. Schoenauer, Evolutionary Algorithms for Constrained Parameter Optimization Problems, *Evolutionary Computation*, **4** (1996) 1-32.
23. D. Powell and M. M. Skolnick, Using Genetic Algorithms in Engineering Design Optimization with Nonlinear Constraints, In S. Forrest (Ed.), *Proc. of the 5th Int. Conf. on Genetic Algorithms*, (San Mateo, CA: Morgan Kaufmann), (1993) 424-430.