Optimal Scheduling of Casting Sequence Using Genetic Algorithms

Kalyanmoy Deb, Abbadi Raji Reddy, and Gulshan Singh
Kanpur Genetic Algorithms Laboratory (KanGAL)
Department of Mechanical Engineering
Indian Institute of Technology Kanpur
Kanpur, PIN 208 016, INDIA
{deb,arreddy,gulshan}@iitk.ac.in

Abstract

Scheduling a casting sequence involving a number of orders with different casting weights and satisfying due dates of production is an important optimization problem often encountered in foundries. In this paper, we attempt to solve this complex, multi-variable, and multi-constraint optimization problem using different implementations of genetic algorithms (GAs). In comparison to a mixed-integer linear programming solver, GAs with problem-specific operators are found to provide faster (with a sub-quadratic computational time complexity) and more reliable solutions to very large-sized (over one million integer variables) casting sequence optimization problem. In addition to solving the particular problem, the study demonstrates how problem-specific information can be introduced in a GA for solving complex real-world problems.

1 Introduction

Optimal scheduling problems often arise in different real-world problem solving activities and are routinely solved using classical search and optimization algorithms including linear programming methods. The difficulties often faced in solving such problems are (i) dimensionality of the search space and (ii) integer restriction of the decision variables. If the resulting problem is linear (that is, the objective function and constraints are all linear functions of the decision variables), the linear programming (LP) approaches are ideal candidates to solve such problems (Taha, 1989). Although the first difficulty is not a matter for solving such problems using an LP, the second difficulty requires an LP approach to be used with an integer programming approach, such as the branch-and-bound method. Since the branch-and-bound approach requires branching every non-integer variable into two different LPs, the presence of a large number of integer decision variables demands a large number of function evaluations to solve the problem to optimality.

For past few decades, optimal scheduling problems have also been solved using various non-traditional methods, such as simulated annealing, genetic algorithms, tabu search etc. (Kirkpatrick, Gelatt and Vecchi, 1983, Goldberg, 1989; Glover, 1997). In some of these methods, although the second difficulty of handling integer variables is not a matter, the first difficulty of handling a large number of decision variables is not well researched.

In this paper, we tackle one such scheduling problem in which the purpose is to find an optimal casting sequence often encountered in an automated foundry. Metal is melted in a large crucible and the molten metal is used to cast a number of molds. One such melting operation is called a heat. With a restriction of a finite sized crucible used in a heat, it is desired to find a schedule for casting different molds from a number of heats. It turns out that the resulting problem is a mixed-integer linear program (MILP) with decision variables x_{ki} representing the number of castings of the k-th order to be molded from the i-th heat. It is observed that a popular MILP

solver (LINGO) can solve the problem successfully having only a couple of hundred variables, but it requires a large number of function evaluations to find the optimum solution. Two commonly-used GA implementations to solve the same problem are also found to be unsatisfactory in terms of the number of function evaluations needed for finding the optimum solution.

In the remainder of the paper, we have designed new GA recombination and mutation operators using problem-specific information. Since the overall objective function is an average utilization of molten metal in individual heats, the best partial sequences from two existing solutions are combined together to form a new solution. The mutation operator attempts to make the new solution obtained in the recombination operator feasible. An innovative method of generating the initial population is also suggested. Using this modified GA, the MILP problem having the number of variables as large as 1,006,750 (more than a million!) is solved with 99.84% molten metal utilization. The resulting GA is able to find the optimum solution with a very high utilization of molten metal over a wide range of problem sizes in a sub-linear order of function evaluations and sub-quadratic computational time complexity, which is much better than that obtained using the LINGO or standard GA implementations.

2 Scheduling of Casting Sequences

In a typical foundry, casting of various sizes are made from a *heat* by melting metal in a large crucible. For convenience, more than one crucible are usually used. In this study, we shall assume a foundry using two crucibles of different sizes (W_I and W_{II}), so that each crucible is used on alternate days. Depending on the crucible size used on the j-th day, the number of heats (H_j) allowed per day vary. We assume that the total number of castings (R) to be made is so large that a multi-day schedule requiring a total of H heats is necessary. Let us also assume that r_k castings of order k having a weight w_k are to be made, such that the total number of castings is $R = \sum_{k=1}^{K} r_k$ (where K is the total number of orders) and the total amount of metal required to make all castings is $\sum_{k=1}^{K} r_k w_k$. Although each order may have a due date for completion, first we do not make the problem more complicated than the above.

With the above parameters, one can design a casting sequence for multiple days by introducing decision variables x_{ki} denoting the number of castings of order k made from the i-th heat. Using this variable, we immediately form a number of constraints which a feasible schedule (or solution) must satisfy:

$$\sum_{i=1}^{H} x_{ki} = r_k, \quad \text{for } k = 1, 2, \ldots, K. \tag{1}$$

The above constraint ensures that all desired castings of each order k are made from a combination of H heats. Since there is a limit of molten metal that can be used in a particular heat dictated by the size of the corresponding crucible W_i (can take either W_I or W_{II} depending on the day), the following sets of inequality constraints are also needed:

$$\sum_{k=1}^{K} w_k x_{ki} \leq W_i, \quad \text{for } i = 1, 2, \ldots, H. \tag{2}$$

To make the matters simple, we assume that the day 1 uses crucible 1, day 2 uses crucible 2, day 3 uses crucible 1, and so on, and that crucible 1 allows H_1 heats per day and crucible 2 allows H_2 heats per day. The parameter W_i for the i-th heat can be calculated by first finding on which day the i-th heat belongs and then by choosing the corresponding weight (W_I or W_{II}) of the crucible.

A sequence of operation can be evaluated based on the utilization of the molten metal in each heat. Since $\sum_{k=1}^{K} w_k x_{ki}$ kg of molten metal is utilized during the i-th heat, the percentage utilization of molten metal of the i-th heat is $100 \sum_{k=1}^{K} w_k x_{ki}/W_i$. Thus, we formulate the optimization

problem as follows:

$$\left.\begin{array}{rll} \text{Maximize} & \frac{1}{H}\sum_{i=1}^{H}\frac{100\sum_{k=1}^{K}w_k x_{ki}}{W_i}, & \\ \text{Subject to} & \sum_{i=1}^{H} x_{ki} = r_k, & \text{for } k = 1,2,\ldots,K, \\ & \sum_{k=1}^{K} w_k x_{ki} \leq W_i, & \text{for } i = 1,2,\ldots,H, \\ & x_{ki} \geq 0, & \\ & x_{ki} \text{ is an integer.} & \end{array}\right\} \quad (3)$$

As the percentage utilization speaks of the efficacy of a solution, we also call the objective function of a solution as *efficiency* of the solution. A little thought will reveal that for a fixed crucible of size (say W_I used for each day), the optimum efficiency of the above problem is the ratio of the total weight of the material needed to cast all molds to the total metal melted for the purpose, or $f^* = 100\sum_{k=1}^{K} w_k r_k/(H^* W_I)$, where H^* is the minimum number of heats required to have a feasible solution. However, when more than one crucible is used, the exact computation of the optimal efficiency is not so simple.

It is obvious from the above formulation that the problem is a linear programming (LP) problem. There can be two difficulties that an LP solver can face to solve the above problem:

1. The number of decision variables and constraints are large, and
2. All decision variables are integer-valued.

The total number of decision variables are $n = HK$ and total number of constraints are $(H + K)$. However, since an LP solver will first convert the inequality constraints into equality constraints using slack variables, the total number of decision variables for an LP solver become $n = (K+1)H$. For example, for $K = 10$ orders requiring $H = 100$ heats to complete all castings, the LP problem involves a total of $1,000$ integer-valued decision variables, 100 slack variables and a total of 110 equality constraints. Although these numbers may not be that daunting particularly for an LP solver, the difficulty may arise due to the discreteness of the decision variables. The commonly-used technique to handle such decision variables is the branch-and-bound (BB) method (Deb, 1995; Reklaitis, Ravindran and Ragsdell, 1983). The BB method first relaxes the integer restrictions and finds the real-parameter optimal solution. Thereafter, it chooses one particular variable (for which an non-integer value is obtained) and divides (branches) the original problem into two new problems (nodes) with an additional constraint to each problem. These procedure of branching into subproblems is terminated (bounded) when feasible integer-valued optimal solutions are found on a node or when a violation of some other optimality criteria cannot justify the continuation of the node. It is clear that as the number of decision variables increases, exponentially more such branching into new LPs are required, thereby making the overall approach computationally expensive.

To investigate the efficacy of such a mixed-integer LP technique, we have applied LINGO (a commonly-used integer programming software) to the above problem. We have tried to solve a problem with $K = 10$ orders (with $r = \{7, 7, 6, 9, 9, 5, 2, 2, 7, 6\}$ castings of 10 orders, respectively) and with a total of 60 castings. The weight of each order varies as follows: $w_k = \{175, 150, 85, 75, 95, 76, 94, 200, 85, 50\}$ for 10 orders, respectively. We have used only one crucible having a capacity of $W_I = 650$ kg of molten metal. We have allowed 10 heats in total to cast all orders. These parameters cause the LP to have 10 equality constraints and 10 inequality constraints. Each inequality constraint is handled by using an additional slack variable, thereby making a total of $(10 \times 10 + 10)$ or 110 decision variables, of which 100 variables must take integer values only. The LINGO takes 592 function evaluations to arrive at a solution having an overall efficiency of 95.05%. Table 1 shows the obtained solution. Since only one crucible is used here, the optimum efficiency achievable with 10 heats is $100\sum_{k=1}^{10} r_k w_k/(10 \times 650)$ or $617,800/6,500$ or 95.05%, which is what the LINGO has found. It is important to highlight that with the standard

Table 1: The solution obtained by LINGO.

Heat	Order Number										Utilization/	Efficiency
No.	1	2	3	4	5	6	7	8	9	10	Cruc. Size	(%)
1	0	1	1	0	0	0	2	1	0	0	623/650	95.85
2	2	0	0	0	1	0	0	0	2	0	615/650	94.62
3	1	0	0	1	3	1	0	0	0	0	611/650	94.00
4	2	0	0	0	1	0	0	1	0	0	645/650	99.23
5	0	0	0	1	0	2	0	0	1	6	612/650	94.15
6	1	1	0	0	2	1	0	0	0	0	591/650	90.92
7	0	0	2	2	1	0	0	0	2	0	585/650	90.00
8	0	3	0	0	0	1	0	0	1	0	611/650	94.00
9	0	2	3	0	1	0	0	0	0	0	650/650	100.00
10	1	0	0	5	0	0	0	0	1	0	635/650	97.69
	7	7	6	9	9	5	2	2	7	6	Average	95.05

input-output procedure of using the LINGO software, we were unable to find a feasible solution (integer values on all variables) for a problem beyond 200 decision variables in any reasonable number of function evaluations.

2.1 Handling Due Date of Completion

The above problem can be made more realistic, by restricting each order to be completed within a pre-specified due date. Here, we impose the due date constraint by restricting all castings of order k to be completed within the first h_k heats. Only the following constraints must be added to the problem stated in (3):

$$\sum_{i=h_k+1}^{H} x_{ki} = 0, \quad \text{for } k = 1, 2, \ldots, K. \tag{4}$$

This increases the number of constraints to $(H + 2K)$. However, instead of this simple fix-up to the original problem, the problem with restricted due dates can also be solved by using a reduced number of decision variables. For the k-th order, the number of decision variables can be restricted to only h_k, instead of H, as considered earlier.

3 Genetic Algorithm Approaches

Genetic algorithms (GAs) are search and optimization strategies which mimic the working principles of natural evolution and genetics (Holland, 1975). The following properties of GAs have made them useful and popular in many real-world search and optimization problems:

1. GAs do not require any gradient information, thereby allowing them to be used in a wide variety of problems,
2. GAs use a population of solutions in one iteration, thereby allow them to search multiple regions simultaneously,
3. GAs use operators which are flexible and easy to tune according to the need of a problem,
4. GAs use stochastic operators, giving GAs a global perspective to problem solving.

For a detail description of GA operators, readers are encouraged to refer to standard texts (Goldberg, 1989; Holland, 1975; Mitchell, 1996; Michalewicz, 1992). In the following, we first evaluate standard GAs to the casting sequence optimization problem, and then suggest better GAs for the same task.

3.1 Using Standard GA Operators

In order to investigate how a standard GA would perform to solve the mixed-integer linear program given in (3), we first use a binary-coded GA, where each decision variable is coded in a 4-bit binary substring. This allows 2^4 or 16 different values for each variable. The decoded value of the 4-bit substring is used as the corresponding value of x_{ki}. Thus, each x_{ki} takes an integer in the range $[0, 15]$. For n variables, we require a total of $\ell = 4n$ bits. Each equality constraint is used to eliminate one decision variable. Only inequality constraints are handled by using the penalty function approach (Deb, 1995; Reklaitis, Ravindran, and Ragsdell, 1983). Each constraint is normalized and total normalized constraint violation is multiplied by a factor 1,000 and subtracted from the objective function value. (Note that the objective function is maximized here.)

Standard GA operators such as the tournament selection operator, single-point recombination (with a probability 0.8) and bit-wise mutation (with a probability of $1/\ell$, where ℓ is the string length) are used. To investigate the performance of GAs, we have applied GAs to the above casting sequencing problem with different number of variables. We always keep a fixed number of orders ($K = 10$), but vary the total number of heats H (thereby allowing to vary the number of castings desired for each order), so that the total number of decision variables (n) vary between 100 to 300. Binary-coded GAs with an increasing population size (from 100 to 1,000) for larger sized problems are used and GAs are run till a feasible solution is found. Table 2 shows the population size and the number of function evaluations (average of 5 runs is tabulated) needed in each case. It is clear from the table that GAs using generic operators such as single-point

Table 2: Average function evaluations needed to find feasible solution using standard binary-coded and real-coded GAs.

	Binary-coded GAs			Real-coded GAs		
Number of Variables	Population Size	Efficiency	Function Eval.	Population Size	Efficiency	Function Eval.
100	100	96.15	13,600	100	95.94	23,740
200	300	95.01	1,42,200	200	92.81	1,21,760
300	1,000	90.11	14,12,400	700	95.14	5,84,220

recombination and bit-wise mutation and using an initial random population cannot solve the MILP problem as competitively as LINGO (a 100-variable problem required only 592 function evaluations, although problems beyond 200 variables could not be solved using the LINGO). In fact, the fourth column shows that exponentially more number of function evaluations are needed to solve a problem with an increasing number of variables. Moreover, the efficiency in the obtained solutions is also reduced with the problem size. It is also important to highlight that the above function evaluations are needed to find the first feasible solution by the GA.

Next, we investigate the worthiness of a real-parameter GA (where no binary strings are used, but decision variables are used directly) involving special recombination and mutation operators ensuring that variables always take integer values. The discretized SBX operator and the discretized polynomial mutation operator (Deb and Goyal, 1998) are used for this purpose. The corresponding distribution indices of $\eta_c = 0.3$ and $\eta_m = 0.1$ are used to ensure that in significantly large number of cases the same solution is not created by either recombination or mutation

operator. Table 2 shows the function evaluations (average of 5 runs) needed to solve identical problems as attempted in the binary-coded GA study. Here too, the real-parameter GAs are not able to find competitive solutions using their generic operators.

These experiments demonstrate amply that the usual GAs (binary-coded or real-coded) with generic search operators are inadequate to solve large-sized linear programming problems. These results are not surprising. The LINGO software is designed particularly to solve linear programs and the casting sequence problem is a linear program. However, the generic binary and real-coded GAs are not particularly designed to solve linear problems. To make the GAs useful in solving these problems and make GAs competitive to a classical optimization method, their operators must be redesigned with problem-specific information. In the following, we suggest a set of GA operators for this purpose.

3.2 Using Problem-Specific GA Operators

It is amply clear from the above study that standard GA operators and the use of an initial random population are far too luxurious for GAs to solve large-scale linear programming problems in a reasonable number of function evaluations. However, because of the flexibilities and modularities allowed in the GA operators, they can be redesigned with problem-specific information. In this subsection, we propose new methods for generating an initial population, and new recombination and mutation operators for the casting sequence optimization problem. Since the selection operator uses function value information, no change in this operator may be required and the standard tournament selection operator (with a tournament size of 2) can be used.

3.2.1 Initial Population Generation

The optimization problem described in (3) involves many constraints, all of which must be satisfied by a feasible solution. However, since the constraints are linear, a simple methodology can be adopted to ensure that the first set constraints are satisfied in all initial solutions.

First, every variable x_{ki} is initialized within $[0, a]$ (a being an upper limit to the number of castings of an order k that can be made from one heat). Thereafter, we normalize each entry x_{ki} such that the total number of castings allocated for an order k is the same as that desired (r_k), thereby satisfying the equality constraints given in equation 1. Here is the procedure:

```
initialize()
//  initializes a population of solutions satisfying all equality constraints
{
  for each population member
    for each order k
      sum = 0
      for each heat i
        x_ki = random_int(0, a)
        sum = sum + x_ki
      x_ki = (r_k / sum) x_ki
}
```

3.2.2 Recombination Operator

The main purpose of a recombination operator is to recombine partial good information of two or more solutions and create an offspring solution. In the current context, a solution can be called good, if the overall utilization of molten metal is close to 100%, so that the corresponding casting strategy causes minimal waste of energy. However, a solution in an intermediate generation may not contain close to 100% utilization in all its heats, but may have close to 100% utilization in some

of its heats. These partial solutions, in which a good utilization has been already achieved, are *building blocks* of the problem and should be propagated from parent solutions to their offsprings. Emphasizing and propagating such partial good sub-solutions through generations will enable GAs to allow more and more heats to reach better and better utilization levels. Such a procedure may lead to find the optimal or a near-optimal solution quickly.

Thus, instead of the generic single-point recombination or the discretized SBX operator used in standard GA implementations, we have designed the following recombination procedure involving two parent solutions:

```
offspring = recombination(parent1, parent2)
//   parent1 and  parent2 are recombined to create an  offspring
{
  for each heat i
    parent1.util = (∑_{k=1}^{K} w_k x_ki^(parent1)  −  W_i)/W_i
    parent2.util = (∑_{k=1}^{K} w_k x_ki^(parent2)  −  W_i)/W_i
    if parent1.util ≤ 0 & parent2.util ≤ 0  //  Both parents are feasible
      if parent1.util > parent2.util   // Parent1 has better utilization
        x_ki^(offspring)  =  x_ki^(parent1)  for all order k
      else                               // Parent2 has better utilization
        x_ki^(offspring)  =  x_ki^(parent2)  for all order k
    else if parent1.util ≤ 0 & parent2.util > 0 // Parent1 is feasible, parent2 isn't
      x_ki^(offspring)  =  x_ki^(parent1)  for all order k
    else if parent1.util > 0 & parent2.util ≤ 0 // Parent2 is feasible, parent1 isn't
      x_ki^(offspring)  =  x_ki^(parent2)  for all order k
    else                                 //  Both parents are not feasible
      if parent1.util < parent2.util  // Parent1 has smaller constraint violation
        x_ki^(offspring)  =  x_ki^(parent1)  for all order k
      else                               // Parent2 has smaller constraint violation
        x_ki^(offspring)  =  x_ki^(parent2)  for all order k
  return (offspring)
}
```

The above procedure simply chooses the sub-solution (x_{ki} for all orders used in the heat) from that parent which produces a better heat-wise utilization of molten metal. Although this is a simple recombination procedure, the offspring need not necessarily be a feasible solution satisfying all the given constraints, despite the two parent being feasible. We fix such a solution in the mutation operator described below.

3.2.3 Mutation Operators

The main purpose of a mutation operator is to perturb a solution obtained by the recombination operator in the hope of creating an even better solution. In this spirit, we develop two mutation operators to make an infeasible solution created by the recombination operator feasible.

In the first mutation operator, the decision variables of a solution are altered so as to satisfy the equality constraints. For a solution, all x_{ki} values for each order k are added and compared with the desired number of castings r_k. If the added quantity is the same as r_k, the corresponding equality constraint is satisfied and no modification is made. If the added quantity is larger than r_k, then the individual normalized utilization of molten metal ($U_i = (W_i - \sum_{k=1}^{K} w_k x_{ki})/W_i$ for the i-th heat) is compared against zero. The heats having an U_i value smaller than zero violates the corresponding inequality constraint. Thus, we have the freedom to reduce the number of castings to be made from those heats. When there exist more than one heat violating their

inequality constraints, any one of the heats can be chosen to reduce the corresponding x_{ki}. Here, we choose to reduce one copy of the k-th casting from the heat ($i_{\min}$) which violates the inequality constraint maximally. Mathematically, $x_{ki_{\min}}$ is reduced by one. The utilization $U_{i_{\min}}$ with the modified number of castings for the $i_{\min}$-th heat is recalculated and the same procedure is repeated till the equality constraint for the k-th order is satisfied. However, there is one difficulty with this procedure. If $x_{ki_{\min}}$ is equal to zero to start with, the variable $x_{ki_{\min}}$ cannot be reduced any further. In this case, we follow the above procedure with the heat having the second-highest constraint violation, and so on. If $U_i \geq 0$ for all heats, the heat ($i_{\max}$) having maximum U_i is chosen and the corresponding $x_{ki_{\max}}$ is reduced by one, provided $x_{ki_{\max}} > 0$; else the second-highest U_i is chosen and so on. This procedure is continued till the k-th equality constraint is satisfied.

On the other hand, if the added quantity is smaller than r_k then the reverse of the above procedure is followed. First, the heat $i_{\max}$ having maximum U_i (but not violating the corresponding inequality constraint) is chosen and $x_{ki_{\max}}$ is increased by one. After performing these operations for all orders, if the solution is still infeasible, we declare this solution as an infeasible solution by associating with it a constraint violation of $\left| r_i - \sum_{i=1}^{H} x_{ki} \right| / r_i$.

Next, the modified solution is sent to the second mutation operator which attempts to alter the decision variables further without violating the equality constraints in order to satisfy the inequality constraints.

The second mutation operator is started using the first heat and continued serially to the final heat. Each heat is checked for its inequality constraint violation. If $U_i \geq 0$, the constraint is satisfied and there is no need to modify the heat. On the other hand, if $U_i < 0$, we look for the heat ($i_{\min}$) which has the minimum feasible utilization. To make U_i feasible, we now have to transfer one copy of an order k from the i-th heat to the $i_{\min}$-th heat, irrespective of whether $U_{i_{\min}}$ is greater than or lesser than one. Here, we choose a random order k for which $x_{ki} > 0$. After all infeasible heats are rectified as above, the feasibility of the overall solution is checked. If the solution is found infeasible, a constraint violation equal to the sum of the normalized violations of all infeasible heats is assigned. Pseudo-codes for both of these mutation operators are given in the Appendix.

3.2.4 Heat-Updates

Unlike other problems, here, the objective function is a summation of independent heat utilization values (or efficiencies), divided by the total number of heats. Among all GA operators, the individual heat utilization values are computed only in the mutation operators. We have marked the places (in bold letters) where such heat-updates are used in the pseudo-codes for mutation attached in the Appendix. Once the individual heat updates are made and the revised utilization values are computed, the overall objective function is easy to calculate. Since one solution may require many such heat-updates in its mutation operators and one solution may not require updates of all its heats to make the solution feasible, the total number of heat-updates can be used as a measure of the computational complexity of the proposed approach. Since the initial population is operated by both mutation operators before a GA generation is started, the heat-update measure will include the heat utilization computations needed for the initial population members as well. But it should be made clear that it is difficult to compare the total number of heat-updates needed to find an optimum in the proposed approach with the total number of function evaluations needed for the purpose usually reported using other optimization methods.

However, it is obvious that a large-sized problem involving a large number of heats will naturally require a large number of heat-updates to find the optimum. To compare the performance of the proposed approach with an iterative method which updates all H heats in one iteration, a *normalized heat-update* measure, which can be calculated by dividing the total heat-updates by

the total number of heats (H), can be used. This measure will give an estimate of the equivalent number of iterations needed to find the optimum solution by the proposed approach.

3.2.5 Handling Due Dates of Orders

When due dates h_k (or due heats) are prescribed, the initial population generation and mutation operators are modified slightly, but the recombination operator is not changed at all.

During the initial population creation, the random integer generation of all x_{ki} for $i = h_k + 1$ till H is set to be zero. This way, the initial population does not create any solution beyond the due date for each order.

For the first mutation operator, no change in the above described operator is needed for the case when $\sum_{i=1}^{h_k} x_{ki} > r_k$. However, for the other case (where more castings are to be added to satisfy the equality constraint), we restrict the search procedure within the first h_k heats for the k-th order.

For the second mutation operator, instead of choosing an order k at random, we choose the order having the maximum priority. If the corresponding x_{ki} is not positive, we choose the second-highest priority and so on. This way, the search is always restricted within the first h_k heats of the k-th order.

4 Simulation Results

In this section, we discuss the simulation results obtained using the proposed GA. First, we show a couple of proof-of-principle results on problems for which the exact optimal solution can be derived. Thereafter, we perform a number of GA parametric studies. We also show a scale-up study by varying the number of decision variables in the range $[100, 501340]$. Finally, we show the efficacy of the proposed approach in handling due dates of completion.

4.1 Proof-of-Principle Results

First, we apply the proposed GA to the 100-variable problem solved using LINGO earlier. With a population size of 10, the proposed GA finds the optimal solution (having an identical efficiency of 95.05%) in an average of only 32.68 normalized heat-updates (total heat-updates divided by the number of heats) from 10 independent runs, requiring only an average of 2.1 generations. The average of total number of heat-updates required by the proposed GA is 326.8, which itself is smaller than the 592 complete function evaluations (recall that each function evaluation will usually require more than one heat-update) needed by the LINGO. The corresponding solutions in all 10 runs are different from each other and are also different from that obtained by LINGO, but they all have 95.05% utilization. One of the solutions is shown in Table 3. Based on this result and similar outcomes obtained from other problems attempted using LINGO and the proposed GA (which are not shown here for brevity), we conclude that the proposed GA finds the optimal solution in a much smaller computational time than LINGO. Thus, we do not use LINGO any further to solve much larger sized problems.

As a proof-of-principle simulation run, next we apply the proposed GA to a problem, for which the optimum solution can be easily derived. We use castings having one of three different weights – 50, 100, and 150 kg – for each order. A total of 10 orders are considered. Two crucibles of sizes 650 and 500 kg are used. For the first crucible, 10 heats are allowed per day and for the second crucible 13 heats per day are allowed. Since these crucible sizes are linear combinations of the weights of the castings, we would expect that the overall optimal solution may have an efficiency of 100%. With the above parameters, the total number of variables become 920. With an average of 103.32 normalized heat-updates over 10 runs, we obtain the optimal solution, which is presented in Figure 1. It is seen from the figure that all 92 heats have 100% efficiency.

Heat No:	Orders 1	2	3	4	5	6	7	8	9	10	Ut(%)
					DAY:1						
1	2	1	1	1	0	0	0	2	0	1	100.00
2	1	1	1	0	1	0	0	1	1	1	100.00
3	1	1	1	0	1	2	0	0	0	0	100.00
4	2	0	1	0	1	1	0	2	0	1	100.00
5	0	0	1	0	0	1	0	1	2	0	100.00
6	1	0	1	1	0	1	0	1	1	1	100.00
7	0	1	0	1	0	1	0	0	1	3	100.00
8	1	0	1	1	1	1	1	1	0	0	100.00
9	1	1	1	1	0	1	0	0	1	0	100.00
10	0	1	0	0	0	1	0	2	0	3	100.00
					DAY:2						
11	0	0	0	1	1	1	1	1	0	0	100.00
12	1	1	0	0	0	0	0	1	1	1	100.00
13	0	0	0	0	0	1	0	2	1	0	100.00
14	1	0	0	1	0	0	0	2	1	1	100.00
15	2	0	1	0	0	1	1	0	0	0	100.00
16	1	0	1	0	0	1	0	1	0	2	100.00
17	1	0	0	1	0	1	0	1	1	0	100.00
18	1	0	1	0	1	2	0	0	0	0	100.00
19	1	0	1	0	1	2	0	0	0	0	100.00
20	1	0	1	0	0	1	0	0	1	1	100.00
21	0	0	0	1	1	1	0	0	1	2	100.00
22	1	0	1	1	0	1	0	0	1	0	100.00
23	0	0	0	1	0	1	0	1	1	1	100.00
					DAY:3						
24	2	1	0	1	0	1	0	2	0	0	100.00
25	2	0	1	1	1	1	0	2	0	0	100.00
26	1	0	1	0	1	1	0	3	0	0	100.00
27	0	0	1	1	0	1	0	2	1	0	100.00
28	0	0	1	1	0	0	0	1	2	2	100.00
29	0	1	0	0	0	1	0	0	2	1	100.00
30	1	0	1	1	1	1	0	1	1	0	100.00
31	2	0	1	1	0	0	2	1	0	0	100.00
32	1	1	1	0	1	0	0	0	2	0	100.00
33	0	0	0	2	0	2	0	1	1	0	100.00
					DAY:4						
34	1	0	1	0	0	0	0	2	1	0	100.00
35	0	1	0	1	0	0	0	2	0	2	100.00
36	0	0	0	0	0	1	1	0	1	1	100.00
37	0	0	1	1	0	0	1	2	0	0	100.00
38	1	0	1	1	1	1	0	1	0	0	100.00
39	1	0	1	1	0	1	1	0	0	0	100.00
40	0	1	0	0	0	0	0	0	2	1	100.00
41	1	0	1	0	1	1	0	1	0	1	100.00
42	2	1	2	1	0	0	0	0	0	0	100.00
43	2	1	0	1	1	0	0	1	0	1	100.00
44	1	0	0	1	0	1	1	1	0	0	100.00
45	1	0	0	1	1	0	0	2	1	0	100.00
46	1	0	1	0	0	1	0	2	0	0	100.00
					DAY:5						
47	1	0	0	1	1	1	0	2	1	0	100.00
48	2	0	0	0	0	1	1	1	1	0	100.00
49	0	0	1	1	1	1	1	0	1	0	100.00
50	1	0	0	0	0	1	1	1	1	1	100.00
51	1	0	0	0	0	2	0	1	1	1	100.00
52	1	1	1	0	1	0	0	1	1	1	100.00
53	2	1	1	0	0	1	0	1	0	1	100.00
54	0	0	1	1	0	1	1	2	0	0	100.00
55	1	0	0	0	0	3	0	1	0	1	100.00
56	2	0	1	1	0	1	0	1	1	0	100.00
					DAY:6						
57	1	0	1	1	1	0	0	1	1	0	100.00
58	2	1	2	1	0	0	0	0	0	0	100.00
59	1	0	1	0	2	0	0	0	1	2	100.00
60	2	0	0	0	0	1	1	1	0	0	100.00
61	0	0	1	1	0	1	0	2	0	0	100.00
62	1	0	1	0	1	1	0	0	1	0	100.00
63	1	0	1	1	1	0	0	1	1	0	100.00
64	1	0	1	1	0	1	1	0	0	0	100.00
65	1	0	0	1	0	1	1	1	0	0	100.00
66	1	1	1	1	0	0	1	0	0	0	100.00
67	0	0	1	0	0	1	1	1	0	0	100.00
68	1	0	1	1	1	0	0	2	0	1	100.00
69	0	1	1	1	0	1	0	0	0	1	100.00
					DAY:7						
70	1	0	1	1	1	1	0	1	1	0	100.00
71	1	0	1	0	0	2	1	0	0	1	100.00
72	1	1	0	0	0	0	1	3	0	0	100.00
73	1	0	1	1	1	1	1	1	0	0	100.00
74	1	0	1	0	0	1	1	0	1	1	100.00
75	2	1	0	0	1	0	0	0	2	1	100.00
76	1	0	1	1	0	1	1	0	1	0	100.00
77	1	0	1	0	1	2	0	1	0	1	100.00
78	1	0	1	2	0	1	0	2	0	1	100.00
79	1	0	1	1	1	1	0	1	1	0	100.00
					DAY:8						
80	2	0	1	1	1	1	0	0	0	1	100.00
81	0	0	1	1	0	1	0	0	1	1	100.00
82	2	0	1	1	0	1	0	1	0	0	100.00
83	1	0	1	1	1	0	0	1	1	0	100.00
84	2	0	0	1	0	2	0	0	0	1	100.00
85	0	1	1	0	0	1	0	0	0	2	100.00
86	1	0	1	1	1	0	0	1	1	0	100.00
87	0	0	1	0	0	1	0	1	1	0	100.00
88	1	0	1	1	1	1	0	1	0	0	100.00
89	1	0	1	0	1	1	1	0	0	0	100.00
90	1	0	1	1	1	0	1	1	0	0	100.00
91	1	1	0	0	1	0	1	1	0	0	100.00
92	0	1	1	1	1	0	0	0	1	0	100.00
Total	85	24	65	55	38	74	26	82	50	46	100.00

Figure 1: Casting Schedule without priority obtained using the proposed GA.

Table 3: The solution obtained by the proposed GA approach.

Heat	Order Number										Utilization/	Efficiency
No.	1	2	3	4	5	6	7	8	9	10	Cruc. Size	(%)
1	1	0	1	3	1	0	0	0	0	0	580/650	89.23
2	1	1	0	1	1	1	0	0	0	1	621/650	95.54
3	1	0	1	0	1	1	0	1	0	0	631/650	97.08
4	1	0	0	1	1	0	0	0	3	1	650/650	100.00
5	0	2	1	1	1	0	0	0	0	0	555/650	85.38
6	1	1	0	0	1	0	1	0	1	1	649/650	99.85
7	0	1	0	1	1	1	0	0	2	1	616/650	94.77
8	1	1	1	0	0	0	1	0	1	1	639/650	98.31
9	1	0	1	1	1	0	0	1	0	0	630/650	96.92
10	0	1	1	1	1	2	0	0	0	1	607/650	93.38
	7	7	6	9	9	5	2	2	7	6	Average	95.05

These two experiments demonstrate the usefulness of the proposed algorithm. In the following subsections, we perform a number of parametric studies on the MILP problem involving very large number of variables.

4.2 Generation-wise Search Effort

In order to investigate the number of generation-wise heat-updates required in a GA run, we have considered a 2,000-variable problem having weight w_k of castings chosen randomly in the range $[50, 200]$ kg. The number of orders is 10 and two crucibles of weights 650 and 500 kg, as those used in the previous experiment are used here. A population of size 30 is used. A total of 200 heats are necessary to find a solution with 99.83% efficiency. The normalized heat-updates required in each generation are recorded and are plotted in Figure 2. It can be seen from the figure that more

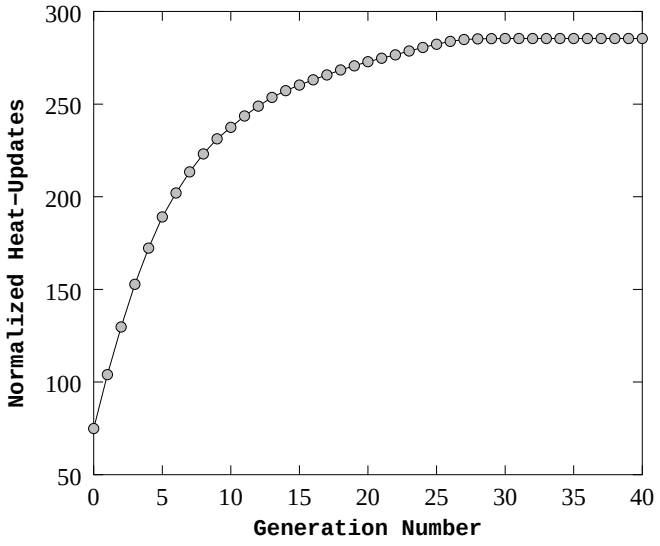


Figure 2: The number of normalized heat-updates versus the generation number for a 2,000-variable problem.

heat-updates are necessary during the early generations. Since the early solutions are usually far from being feasible, a considerable number of heat-updates are necessary to fix the equality and inequality constraints. When solutions approach the most efficient solution towards the latter generations, not much heat-updates are necessary. When all population members become feasible (happened at generation 32 in this case), no new heat-updates are necessary. However, in this particular simulation run, the first feasible solution was found at generation 16. Similar dynamics in the number of heat-updates versus generation number are observed in all experiments performed in this paper. We discuss the optimality of the above solution in the following paragraph.

In the above problem, the following weight of the castings are used:

	Order, k									
	1	2	3	4	5	6	7	8	9	10
Weight (w_k, kg)	175	150	85	75	95	75	95	200	85	50
Castings (r_k)	86	120	99	111	138	79	109	87	110	147

The total weight of all castings is $\sum_{k=1}^{10} w_k r_k$ = 113,280 kg. Here, 10 heats with a 650 kg crucible per day and 13 heats with a 500 kg crucible per day are used in alternate days. To complete all castings, a total of 17-day schedule with 90 heats of 650 kg crucible and 110 heats of 500 kg crucible are needed, thereby amounting to a total of 200 heats. This is because, with 199 heats the total metal melted would be ($90 \times 650 + 109 \times 500$) or 113,000 kg, which is 280 kg smaller than the overall required casting weight. Thus, the minimum number of heats required to find a feasible schedule is 200, which is also found to be the optimum by the proposed GA.

Now, with 200 heats (90 heats of 650 kg each and 110 heats of 500 kg each) the total metal melted is 113,500 kg. Thus, the *mean utilization* with 200 heats is 113,280/113,500 or 0.9981 (equivalent to 99.81%). Our GA has found a feasible schedule having a little better objective value (99.83%) than this mean utilization value.

4.3 Heat-Updates and Efficiency

To investigate the effect of the number of heat-updates on the obtained efficiency (the objective function value), we consider the same 2,000-variable problem as used in the previous study. The total number of heats (H) are adjusted so as to obtain different optimal efficiencies. If more heats than the optimal number of heats are allowed, the optimization task becomes easier. This is because the allocations can be made in many different ways in order to obtain a feasible solution. Thus, it is expected that an optimization algorithm would require less number of heat-updates to achieve a feasible solution. On the other hand, if less than optimal number of heats are chosen, the search space contains fewer feasible solutions, thereby making it difficult for an optimizer to find the corresponding optimum. Figure 3 shows different values of total number of normalized heat-updates required for finding different efficiencies in the obtained solutions. For each case, a population of size 30 is used and 10 runs are performed. The minimum, average and maximum heat-updates needed to get to the optimum are also recorded. The figure shows that, in general, more heat-updates are needed to obtain a solution with better efficiency. The following number of heats are used to obtain different efficiencies:

	Case Number										
	1	2	3	4	5	6	7	8	9	10	11
Efficiency (%)	91.56	92.24	93.02	93.91	94.85	95.77	96.67	97.50	98.28	99.06	99.83
Heats	220	218	216	214	212	210	208	206	204	202	200
Mean util.(%)	90.62	91.36	92.21	93.20	94.20	95.23	96.29	97.24	98.08	98.93	99.81

For each heat, we have also calculated and tabulated the mean utilization value as the ratio of the total casting weight to the total weight of metal melted. In all cases, the proposed GA has found an objective value better than the corresponding mean utilization value.

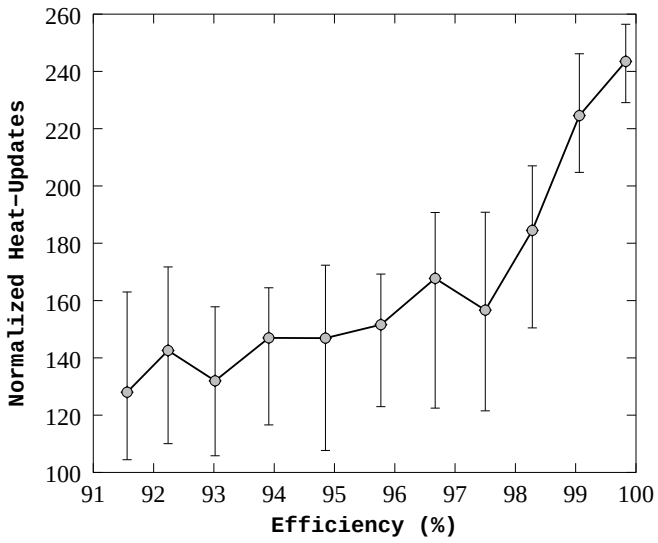


Figure 3: The number of normalized heat-updates needed to achieve different efficiencies for the 2000-variable problem.

4.4 Importance of Population Size

Next, we study the effect of population size on the efficiency obtained by the proposed GA. For the same 2,000 variable problem as discussed above, we have used different population sizes, but kept the total number of maximum allowed generations equal to $1,000/N$ (where N is the population size). For each population size, the GA is run to find the minimum number of heats needed to find a feasible solution. These values are shown below:

Population size	4	8	12	16	20	25	40	60	100
Min. Heats	211	204	201	201	200	200	200	200	200

As discussed earlier, for the 2000-variable problem, there exist no feasible solution with less than 200 heats. The above table shows that for small population sizes (up to 12), the proposed GA cannot find any feasible solution having 200 heats. But, GAs with a population size larger than or equal to 16 are able to find a feasible solution with the minimum possible number of heats (200, in this case). As discussed earlier, an interesting property of this problem is that with the number of heats larger than the optimal number of heats more feasible solutions exist in the search space, thereby making an optimization algorithm easier to solve the problem (but with a lesser efficiency). This is the reason why GAs with small population sizes cannot solve the problem with the optimal number of heats. However, when the population size is adequate, GAs consistently and reliably find feasible solutions with the optimal number of heats.

The corresponding efficiency obtained with each population size is shown in Figure 4. The efficiency obtained in the best solution increases with population size. Once again, the best, average, and the worst efficiencies obtained in each case are shown (but the difference is so small that in most cases they are not visible). These results agree with the past research studies (Deb and Agrawal, 1999; Goldberg, 1989) concluding that for a recombinative GA there exists a critical population size below which the recombination operator does not work well. With an adequate number of population members, the recombination operator has the flexibility to allow good solutions to exchange good partial information. As seen from the Figure 4 that beyond a critical population size there exists a large plateau in which the proposed GA works equally well.

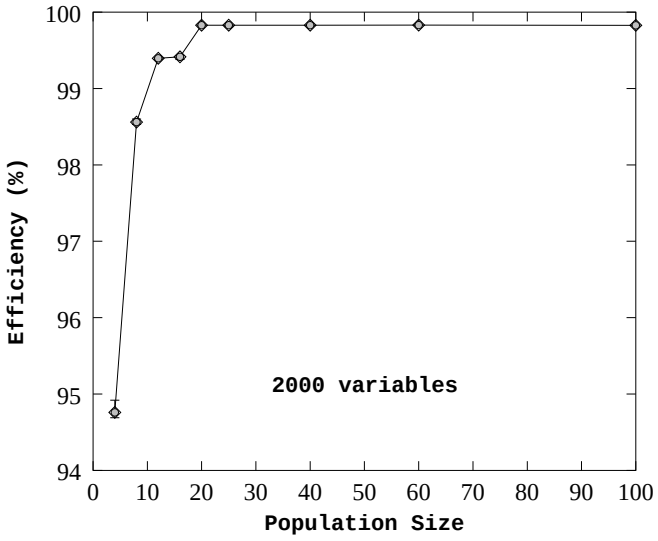


Figure 4: Efficiencies achieved with different population sizes.

4.5 Importance of Recombination Operator

To investigate the effect of the recombination operator, we have applied the proposed algorithm with and without the recombination operator on the problem having different number of variables. All GA parameters are kept the same for runs with and without the recombination operator. Both GAs are run for a fixed number of generations. Figure 5 shows that for a wide range of number of decision variables (between 100 to 10,000), the GA with only the mutation operator does not perform as good as that with recombination and mutation operators. These results show the

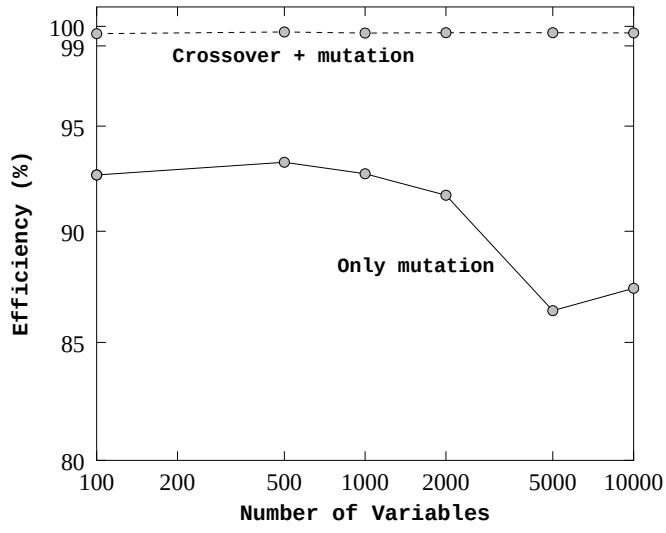


Figure 5: Efficiencies achieved with different number of variables for GAs with and without the recombination operator.

importance of the suggested recombination operator in finding solutions with very high efficiencies.

4.6 Scale-up Study

In order to investigate the computational complexity of the proposed approach on increasing number of decision variables, we have solved the problem with variables in the range $[100, 501690]$. For a population size of 30, we have taken 10 different runs for each case and run the proposed GA till a feasible solution with an efficiency above 99.65% is achieved. Figure 6 shows the average number of heat-updates over 10 runs for different problem sizes. The figure is plotted in logarithmic scales. A fitted straight line shows a polynomial complexity of heat-updates as $O(n^{0.946})$, where n is the number of integer variables. It is interesting to note that the proposed GA can solve a

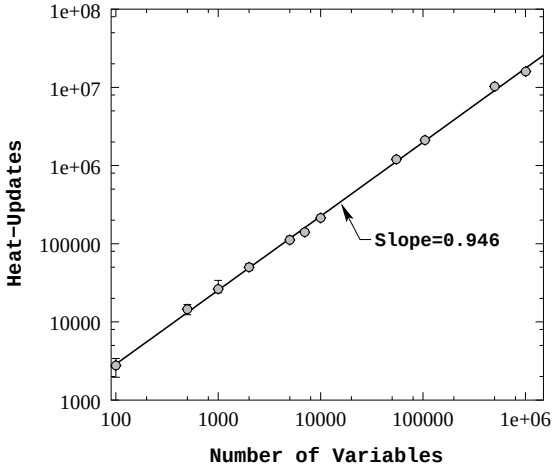


Figure 6: Average, minimum, and maximum heat-updates needed to find a solution having at least 99.65% utilization of molten metal for different problem sizes.

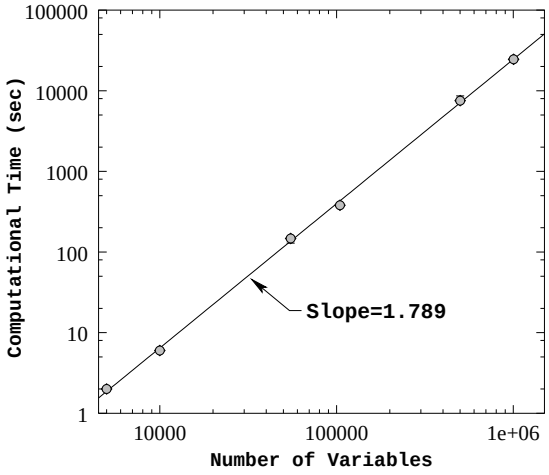


Figure 7: Average, minimum, and maximum computational time needed to find the optimum solution.

problem having as much as 1,006,750 integer decision variables in only about $1.6(10^7)$ normalized heat-updates in finding a solution with 99.84% efficiency, which is better than the corresponding mean utilization value of 99.82% for this problem. Figure 7 shows the computational time needed to find the same optimum solutions on a 1.7 GHz Pentium IV processor. The figure shows a sub-quadratic computational time complexity of the proposed algorithm. The log-log plot shows a complexity of $O(n^{1.789})$. Since problems having 2,000 or less number of variables take less than one second computational time, we do not show those results in Fugure 7.

It is worth mentioning here that to our knowledge this is by far one of the largest problems which a GA has solved successfully and reliably. Where a commonly-used classical mixed-integer programming method has only been able to solve the MILP problem with at most 200 integer variables, the proposed GA's success to solve the same MILP problem with more than one million integer variables in a reasonable computational effort marks a landmark achievement in the field of search and optimization.

4.7 Handling Priority

To investigate the efficacy of the priority handling strategy suggested earlier, we have taken the same problem as discussed as the second problem in the proof-of-principle study, but introduce a due date of completion for each order. For each order, the maximum number of heats allowed to complete all castings are given below:

	Order Number									
	1	2	3	4	5	6	7	8	9	10
Due Heat	10	25	45	–	70	80	40	90	–	50

Here a '–' means that there is no particular due date for this order. Figure 8 shows the optimum schedule with priority. An average of 1,197.28 normalized heat-updates of 10 runs are needed. The schedule satisfies all the above due heats and simultaneously maximizes the overall efficiency (by achieving a 100% efficient schedule).

5 Conclusions

Over the years, genetic algorithms (GAs) have been applied to numerous search and optimization problems. However, there exists very few studies where GAs have been tested on very large scale problems, such as problems having hundreds of thousands of variables. In this paper, we have shown that the standard GA implementations have difficulty in solving a real-world casting sequencing problem in a competitive manner. We have highlighted that GA operators designed with problem information must be used in order to solve them in any reasonable computational time. Based on one such set of problem-specific GA operators and an initial population generation procedure, we have been able to solve mixed-integer linear programming problems having as much as 1,006,750 variables in a sub-linear order of function evaluations and a sub-quadratic computational time complexity. Parametric studies with the population size and importance of the recombination operator have also been performed. Handling priority in completing individual orders has also been implemented. Here are the outcomes of our study:

1. The proposed GA finds the optimum solution in a computational complexity much smaller than a classical branch-and-bound based LP solver.

2. The computational complexity of the proposed GA is sub-linear in terms of required function evaluations and sub-quadratic in terms of computational time for a wide range $[100, 1,006,750]$ of decision variables.

3. The new recombination operator developed here is the key operator for the proposed GA's success, although mutation operators suggested here are helpful but are not adequate to solve the problem efficiently by itself.

4. There exists a critical population size beyond which the proposed recombination operator works well.

5. In all problems tried in this paper, very high quality solutions (near 100% utilization of molten metal) are found.

Besides the applicability of GAs to the casting sequence problem, to our knowledge, GAs have not been applied to such large-sized problems (with more than one million variables) with success and reliability too often in the past. Starting with an initial population of good solutions and the use of good problem-specific recombination operator have allowed good partial solutions (building blocks) to propagate from one iteration to another to finally build the optimum solution with a computational time complexity less than quadratic to the number of decision variables. Such careful design of GA operators are the key for using GAs to large-scale search and optimization problems. Hopefully, this paper demonstrates the power of a recombinative GA in solving large-sized problems to optimality and will motivate the design and application of GAs to similar other large-scale real-world optimization problems. This paper has also shown that with a careful design of GA operators, GAs can be made very efficient and competitive beyond the reach of specialized classical algorithms, even to solve linear programming problems.

Heat No:	Orders 1	2	3	4	5	6	7	8	9	10	Ut(%)
	DAY:1										
1	11	0	1	0	0	0	0	0	0	0	100.00
2	7	1	0	0	0	0	1	0	0	0	100.00
3	8	1	1	0	0	0	0	0	0	0	100.00
4	10	1	0	0	0	0	0	0	0	0	100.00
5	10	1	0	0	0	0	0	0	0	0	100.00
6	5	0	2	0	0	0	1	0	0	1	100.00
7	7	1	0	0	0	0	1	0	0	0	100.00
8	13	0	0	0	0	0	0	0	0	0	100.00
9	13	0	0	0	0	0	0	0	0	0	100.00
10	1	1	1	0	0	1	1	0	0	1	100.00
	DAY:2										
11	0	2	2	0	0	0	0	0	0	0	100.00
12	0	1	1	0	1	0	1	0	0	1	100.00
13	0	0	3	0	0	0	1	0	0	1	100.00
14	0	0	3	0	0	0	1	0	0	1	100.00
15	0	1	2	0	0	0	1	0	0	0	100.00
16	0	2	2	0	0	0	0	0	0	0	100.00
17	0	2	2	0	0	0	0	0	0	0	100.00
18	0	1	0	0	0	1	1	0	0	1	100.00
19	0	1	2	0	0	0	1	0	0	0	100.00
20	0	1	2	0	0	1	0	0	0	0	100.00
21	0	2	2	0	0	0	0	0	0	0	100.00
22	0	2	1	0	1	0	0	0	0	1	100.00
23	0	0	2	0	1	0	1	0	0	2	100.00
	DAY:3										
24	0	2	2	0	0	0	1	0	0	0	100.00
25	0	1	3	0	0	1	0	0	0	1	100.00
26	0	0	3	0	1	0	1	1	0	1	100.00
27	0	0	3	0	0	1	1	0	0	1	100.00
28	0	0	2	1	0	0	1	0	1	2	100.00
29	0	0	1	0	0	2	0	2	0	1	100.00
30	0	0	0	0	0	2	1	1	0	2	100.00
31	0	0	0	1	1	0	2	1	1	0	100.00
32	0	0	1	0	1	1	1	1	0	2	100.00
33	0	0	2	0	0	1	1	1	0	1	100.00
	DAY:4										
34	0	0	3	0	0	1	0	0	0	1	100.00
35	0	0	0	0	1	1	1	1	0	1	100.00
36	0	0	3	0	1	1	0	0	0	0	100.00
37	0	0	1	0	1	1	1	0	0	1	100.00
38	0	0	2	1	0	1	0	1	0	0	100.00
39	0	0	2	0	0	0	1	1	0	1	100.00
40	0	0	0	0	0	0	3	0	0	1	100.00
41	0	0	2	1	1	0	0	2	0	0	100.00
42	0	0	2	1	1	0	0	0	1	1	100.00
43	0	0	1	0	1	0	0	0	0	7	100.00
44	0	0	1	0	4	1	0	0	0	1	100.00
45	0	0	2	0	0	2	0	0	0	0	100.00
46	0	0	0	0	1	1	0	1	1	1	100.00

	Orders 1	2	3	4	5	6	7	8	9	10	Ut(%)
	DAY:5										
47	0	0	0	0	1	2	0	0	0	6	100.00
48	0	0	0	1	0	2	0	1	1	1	100.00
49	0	0	0	0	1	2	0	2	0	2	100.00
50	0	0	0	0	1	1	0	2	1	2	100.00
51	0	0	0	2	3	1	0	1	1	0	100.00
52	0	0	0	2	1	1	0	2	1	0	100.00
53	0	0	0	1	1	2	0	1	1	0	100.00
54	0	0	0	0	2	0	0	4	1	0	100.00
55	0	0	0	2	0	2	0	1	1	0	100.00
56	0	0	0	0	1	3	0	0	1	0	100.00
	DAY:6										
57	0	0	0	1	1	2	0	1	0	0	100.00
58	0	0	0	1	0	1	0	3	0	0	100.00
59	0	0	0	1	1	1	0	1	1	0	100.00
60	0	0	0	1	1	2	0	1	0	0	100.00
61	0	0	0	1	0	2	0	0	1	0	100.00
62	0	0	0	1	0	1	0	3	0	0	100.00
63	0	0	0	1	1	1	0	1	1	0	100.00
64	0	0	0	1	1	1	0	1	1	0	100.00
65	0	0	0	1	1	2	0	1	0	0	100.00
66	0	0	0	1	1	2	0	1	0	0	100.00
67	0	0	0	1	1	2	0	1	0	0	100.00
68	0	0	0	1	1	2	0	1	0	0	100.00
69	0	0	0	1	1	2	0	1	0	0	100.00
	DAY:7										
70	0	0	0	0	0	2	0	2	1	0	100.00
71	0	0	0	3	0	1	0	2	1	0	100.00
72	0	0	0	1	0	1	0	0	3	0	100.00
73	0	0	0	2	0	2	0	1	1	0	100.00
74	0	0	0	0	0	2	0	2	1	0	100.00
75	0	0	0	0	0	1	0	2	2	0	100.00
76	0	0	0	2	0	2	0	1	1	0	100.00
77	0	0	0	2	0	1	0	1	2	0	100.00
78	0	0	0	1	0	2	0	0	2	0	100.00
79	0	0	0	1	0	2	0	0	2	0	100.00
	DAY:8										
80	0	0	0	0	0	1	0	2	1	0	100.00
81	0	0	0	2	0	0	0	4	0	0	100.00
82	0	0	0	1	0	0	0	3	1	0	100.00
83	0	0	0	0	0	0	0	2	2	0	100.00
84	0	0	0	2	0	0	0	1	2	0	100.00
85	0	0	0	1	0	0	0	3	1	0	100.00
86	0	0	0	1	0	0	0	3	1	0	100.00
87	0	0	0	3	0	0	0	2	1	0	100.00
88	0	0	0	2	0	0	0	1	2	0	100.00
89	0	0	0	0	0	0	0	2	2	0	100.00
90	0	0	0	0	0	0	0	5	0	0	100.00
91	0	0	0	1	0	0	0	0	3	0	100.00
92	0	0	0	4	0	0	0	0	2	0	100.00
Total	85	24	65	55	38	74	26	82	50	46	100.00

Figure 8: Optimal casting schedule for the problem with priority.

Acknowledgments

The optimal casting sequence scheduling problem was introduced to the authors by Mr. Sachin Agarwal of PTC Industries Limited, Lucknow, India. Authors highly appreciate his time and efforts.

References

Deb, K. (1995). *Optimization for engineering design: Algorithms and examples.* New Delhi: Prentice-Hall.

Deb, K. and Agrawal, S. (1999). Understanding interactions among genetic algorithm parameters. In *Foundations of Genetic Algorithms 5 (FOGA-5)*, pp. 265–286.

Deb, K. and Goyal, M. (1998). A robust optimization procedure for mechanical component design based on genetic adaptive search. *Transactions of the ASME: Journal of Mechanical Design, 120*(2), 162–164.

Glover, F. and Laguna, M. (1997). *Tabu search.* Boston: Kluwer.

Goldberg, D. E. (1989). *Genetic Algorithms for Search, Optimization, and Machine Learning.* Reading, MA: Addison-Wesley.

Holland J. H. (1975). *Adaptation in Natural and Artificial Systems.* Ann Arbor, MI: MIT Press.

Kirkpatrick, S., Gelatt, C. D., Vecchi, M. P. (1983). Optimization by simulated annealing. *Science, 220*, 671–680.

Michalewicz, Z. (1992). *Genetic Algorithms + Data Structures = Evolution Programs.* Berlin: Springer-Verlag.

Mitchell, M. (1996). *Introduction to Genetic Algorithms.* Ann Arbor, MI: MIT Press.

Reklaitis, G. V., Ravindran, A. and Ragsdell, K. M. (1983). *Engineering Optimization Methods and Applications.* New York: Wiley.

Taha, H. A. (1989): *Operations Research.* New York: Macmillan.

A Pseudo-codes for Mutation Operators

Here, we outline pseudo-codes for the two mutation operators (`mutation1` and `mutation2`) used in the proposed GA. The accompanying subroutines are also presented. Simplified procedures are used in the codes below for clarity, but in practice they can be made more computationally efficient.

```
mut1 = mutation1(x)    //  First mutation operator
//  Attempts to fix the equality constraint violations
{
 1  x.constraint_violation = 0.0
 2  for each order k
 3    sum_k = sum_all_heat(x, k)
 4    while (sum_k  >  r_k)  //  if over-specified
 5      for each heat i
 6        util[i] = (utilization(x, i) - W_i)/W_i
```

```
 7      ivec = descending_order_sort(util)  //  ivec stores heat index of sorted util
 8      m = 0
 9      while (util[ivec[m]] > 0)
10        if (x_{k(ivec[m])} > 0)
11          x_{k(ivec[m])} = x_{k(ivec[m])} - 1        //  Heat-Update
12          Go to Line 22
13        else                  //  no violated constraints
14          m = m + 1
15      m = H
16      while (util[ivec[m]] ≤ 0)
17        if (x_{k(ivec[m])} > 0)
18          x_{k(ivec[m])} = x_{k(ivec[m])} - 1        //  Heat-Update
19          Go to Line 22
20        else
21          m = m - 1
22      sum_k = sum_k - 1
23
24    while (sum_k < r_k)   //  if under-specified
25      id = min_utilization(x)
26      if (id ≥ 0)                //  some heats are feasible
27        x_{k(id)} = x_{k(id)} + 1                //  Heat-Update
28      sum_k = sum_k + 1
29  x.constraint_violation = x.constraint_violation + (|r_k - sum_k|)/r_k
30  mut1 = x
}

mut2 = mutation2(x)   //  Second mutation operator
//  Attempts to fix the inequality constraint violations
{
  for each heat i
    util_i = (utilization(x, i) - W_i)/W_i
    if (util_i > 0)      //  infeasible heat
      repeat
        k = rnd(1, K)   //  A random order with x_{ki} > 0
      until (x_{ki} > 0)
      id = min_utilization(x)
      x_{ki} = x_{ki} - 1                        //  Heat-Update
      x_{k(id)} = x_{k(id)} + 1                  //  Heat-Update
  if x is infeasible         //  At least one heat is infeasible
    x.constraint_violation = x.constraint_violation
              + Σ_{i=1}^{H} ( utilization(x, i) - W_i)/W_i
  mut2 = x
}

int min_utilization(x)
//  Finds the heat index which corresponds to the minimum utilization in x
{
  id = -1
  umin = 0
  for each heat i
```

```
    u_i = utilization(x, i)
    if (umin > (u_i − W_i)/W_i)
      id = i
      umin = (u_i − W_i)/W_i
  return (id)
}

float utilization(x, i)
//  Calculates the metal utilization of i-th heat in x
{
  u_i = 0
  for each order k
    u_i = u_i + w_k x_ki
  return (u_i)
}

float sum_all_heat(x, k)
//  Calculates the allocated number of castings of order k in x
{
    sum_k = 0
    for each heat i
      sum_k = sum_k + x_ki
    return (sum_k)
}
```