European Journal of Operational Research 261 (2017) 460–474

Contents lists available at ScienceDirect

European Journal of Operational Research

journal homepage: www.elsevier.com/locate/ejor

Discrete optimization

A population-based fast algorithm for a billion-dimensional resource allocation problem with integer variables

Kalyanmoy Deb[a,*], Christie Myburgh[b]

[a] Department of Electrical and Computer Engineering, Michigan State University, East Lansing, MI 48824, USA
[b] Principal Research & Development Engineer, Maptek Pty Ltd, Level 2, 190 Aberdeen Street, Northbridge, WA 6003, Australia

ARTICLE INFO

Article history:
Received 3 June 2016
Accepted 10 February 2017
Available online 15 February 2017

Keywords:
Evolutionary computations
Population-based optimization algorithm
Large-scale optimization
Resource allocation problem
Billion variables

ABSTRACT

Among various complexities affecting the performance of an optimization algorithm, the search space dimension is a major factor. Another key complexity is the discreteness of the search space. When these two complexities are present in a single problem, optimization algorithms have been demonstrated to be inefficient, even in linear programming (LP) problems. In this paper, we consider a specific resource allocation problem constituting to an integer linear programming (ILP) problem which, although comes from a specific industry, is similar to other practical resource allocation and assignment problems. Based on a population based optimization approach, we present a computationally fast method to arrive at a near-optimal solution. Compared to two popular softwares (`glpk`, Makhorin, 2012 and `CPLEX`, Gay, 2015), which are not able to handle around 300 and 2000 integer variables while continuing to run for hours, respectively, our proposed method is able to find a near-optimal solution in less than second on the same computer. Moreover, the main highlight of this study is to propose a customized population based optimization algorithm that scales in almost a linear computational complexity in handling 50,000 to one billion (10^9) variables. We believe that this is the *first* time such a large-sized real-world constrained problem is ever handled using any optimization algorithm. We perform sensitivity studies of our method for different problem parameters and these studies clearly demonstrate the reasons for such a fast and scale-up application of the proposed method.

1. Introduction

Optimization algorithms are known to be vulnerable in handling a large number of variables (Bellman, 1954; Ermon, Gomes, Sabharwal, & Selman, 2013). This is particularly true in non-linear problems having highly interacting variables. As the number of variables increase, the number of interactions increase and an optimization algorithm must have the ability to detect all such interactions to construct a near-optimal solution. Researchers have developed various tractable methods to solve such large-scale problems involving 1000 or more variables: (i) decompose and approximate into multiple tractable problems (Li, Zhou, Ouyang, & Chen, 2014; Li & Yao, 2012; Omidvar, Li, Mei, & Yao, 2014; Yang & Xiao, 2009; Yang, Tang, & Yao, 2008), (ii) embed problem heuristics to speed up the search process (Kong & Tian, 2006; Michalewicz, 1995; Zanakis & Evans, 1981), (iii) use multi-fidelity methods (Huang, Gao, & Zhang, 2013; Molina-Cristbal, Palmer, Skinner, & Parks, 2010; Xu et al., 2014), and others. Another difficulty arises from the discreteness of the search space, as the continuity and gradient information are absent in such problems. Most point-based optimization algorithms (Fletcher & Reeves, 1964; Reklaitis, Ravindran, & Ragsdell, 1983) handle the latter problems by first assuming that the search space is continuous and then branching the problem into two non-overlapping problems based on one of the discrete variables. For a discrete search space problem having only a few variables, the branch-and-bound concept works reasonably well, but when the problem has a large number of variables, the number of branched problems becomes exponentially large. This process slows the overall algorithm and often an algorithm keeps on running for hours and days and still does not show any sign of convergence.

Interestingly, linear programming (LP) problems having linear objective and constraint functions and continuous variables can be solved to optimality for tens of thousands to even a million variables using smart and efficient methods (Dantzig, 1988; Karmarkar, 1984), but if there exists a large number of discrete variables (taking only integer values, for example), even the linear programming

* Corresponding author.
E-mail addresses: kdeb@egr.msu.edu (K. Deb), christie.myburgh@maptek.com.au (C. Myburgh).
URL: http://www.egr.msu.edu/~kdeb (K. Deb)

http://dx.doi.org/10.1016/j.ejor.2017.02.015
0377-2217/© 2017 Elsevier B.V. All rights reserved.

problems are difficult to solve by the branch-and-bound or branch-and-cut fix-ups mentioned above. Ironically, many real-world optimization problems are idealized and formulated as a linear or quadratic programming problems, but the continuous assumption of originally discrete variables may not always be assumed. For example, a variable indicating the number of tooth in a gear design problem, the number of floors in a high-raised building construction problem, the number of stocks to invest in portfolio management problem cannot be assumed as a continuous variable. Although the objective and constraint functions can be evaluated for a non-integer solution, the resulting values may not be meaningful. It is also important to note that such problems are common in practice, but optimization methods to handle such discrete programming problems are not efficient, even if the objective and constraint functions are linear.

In this paper, we address a particular resource allocation problem which is an integer linear program and propose a population-based algorithm for finding a near-optimal solution. Theoretically, the problem falls into the category of a knapsack problem (Martello & Toth, 1990), which is known to be NP-hard to solve to optimality. The main crux of our proposed algorithm is that it is able to find a near-optimal solution for an extremely large range of variables (from 50 thousand variables to one *billion* variables) in a polynomial computational time. Our approach uses a population-based approach (Deb, 1995; Goldberg, 1989; Holland, 1975) in which more than one solution is used in each iteration to collectively create a new population. Although two earlier EC studies (Goldberg, Sastry, & Llora, 2007; Wang, Zoghi, Hutter, Matheson, & de Freitas, 2013) have considered up to a billion variables, the first study and its follow-up study (Iturriaga & Nesmachnow, 2012) used a Boolean, unconstrained, noisy one-max problem in which the target was to find at least 51% correct variable values in a single string, where a string with 100% 1s is the optimal solution. The second study (Wang et al., 2013) embedded a two-variable real-parameter problem in one billion variable vector thereby making billion minus two variables unimportant for solving the problem. Our current study is remarkable from the following aspects: (i) the study clearly portrays the fact that if a near-optimal solution is desired, it is possible to develop a polynomial time algorithm for addressing NP-hard problems, (ii) the study handles, for the first time, a billion variable optimization problem motivated by a real-world constrained optimization problem, and (iii) the study directly compares with state-of-the-art popular commercial and public-domain softwares in establishing superiority of population-based methods in solving large-scale problems.

The remainder of the paper formulates the casting scheduling problem and reveals the integer linear programming nature of the problem in Section 2. The next section describes our proposed population-based integer linear programming (PILP) algorithm by detailing its operators and pseudo-codes. Thereafter, Section 4 evaluates the performance of two commonly-used optimization softwares – one publicly available `glpk` and other commercially available `CPLEX` – on small version of the casting scheduling problem. Although these methods have reportedly performed well on generic LP problems having millions of continuous variables, their vulnerability in handling discrete variables becomes clear from the study. Despite the need for handling about 30,000 variables in practice, they are not able to solve even a 2000-variable version of the casting scheduling problem. In Section 5, the same problems are solved using our proposed PILP algorithm in less than a second of computational time repeatedly. The section then makes a number of sensitivity analysis studies of evaluating its performance under different parameter settings of the problem and clearly brings out the working principle of the PILP algorithm. One of the sections presents a scale-up study, in which PILP successfully handles more than one billion variables. The sheer number of variables and the fast computational time for finding a near-optimal solution for billion-variable extension of a real-world problem probably make this study the first-ever optimization study to achieve this feat. Conclusions and future extensions of this study are discussed in Section 7.

2. Casting scheduling problem

A foundry casts objects of various sizes and numbers by melting metal on a crucible of certain size (say W). Each such melt is called a *heat*. The molten metal from each heat is then used to cast a number of objects. The amount of metal used in a heat may not add up to W exactly and some metal may remain in the crucible as unused from the heat. This introduces an inefficiency in the scheduling process and must be reduced to a minimum by using an appropriate scheduling process. The ratio of molten metal utilized to make objects to the crucible size (or metal melted) is called the *metal utilization ratio* for the heat. When a number of heats are to be used to make all required copies of the objects over a scheduling period, an average of metal utilization ratio for all heats can be used as a performance measure of the overall scheduling process.

To formulate the casting scheduling optimization problem, we assume that there are a total of N objects to be made with a demand of exactly r_j (>0, an integer) copies for jth object. Each object has a fixed weight w_i in kilograms, thereby requiring a total of $M=\sum_{j=1}^{N} r_j w_j$ kilogram of metal to make all copies of jth object. Without loss of generality, let us also assume that W_i kilogram of metal is melted at ith heat, thereby allowing us to consider a different crucible size at each heat. Then, the total number of heats (H) required to melt the above metal with an expected average efficiency of η metal utilization ratio from every heat can be computed by finding the minimum H to solve the following equation: $\sum_{i=1}^{H} \eta W_i \geq M$. If all heats use an identical crucible of capacity W, then, the above condition becomes $H=\lceil \frac{M}{\eta W} \rceil$.

To find an optimal assignment of objects from each heat, we need to solve an optimization problem with a two-dimensional $H \times N$-matrix of *variables* x_{ij} (with $i=1,2,\ldots,H$ and $j=1,2,\ldots,N$), which represents the number of copies of jth object to be made from the ith heat. Since none, one, or multiple complete copies can be made from each heat, the variable x_{ij} can only take an *integer* value between zero and r_j. This restriction of the problem makes the optimization problem a discrete programming problem. Moreover, there are two sets of constraints in the problem that an optimal assignment must satisfy. First, the total amount of metal used in ith heat must be at most the size of the crucible (W_i), that is, $\sum_{j=1}^{N} w_j x_{ij} \leq W_i$ for all $i=1,2,\ldots,H$. For a total of H heats, there are a total of H such inequality constraints. Second, exactly r_j copies of jth object is to be made, not one more and not one less, thereby creating N equality constraints of the type: $\sum_{i=1}^{H} x_{ij} = r_j$, for $j=1,2,\ldots,N$. We now present the resulting integer linear programming problem, as follows:

$$\text{Maximize} \quad f(\mathbf{x}) = \frac{1}{H}\sum_{i=1}^{H}\frac{1}{W_i}\sum_{j=1}^{N} w_j x_{ij}, \tag{1}$$

$$\text{Subject to} \quad \sum_{j=1}^{N} w_j x_{ij} \leq W_i, \quad \text{for } i=1,2,\ldots,H, \tag{2}$$

$$\sum_{i=1}^{H} x_{ij} = r_j, \quad \text{for } j=1,2,\ldots,N, \tag{3}$$

$$x_{ij} \geq 0 \text{ and is an integer.} \tag{4}$$

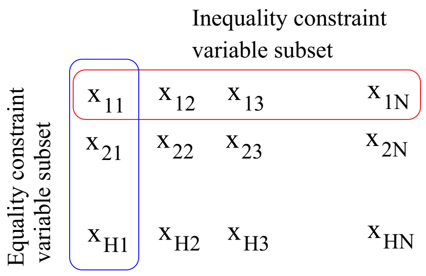

Inequality constraint variable subset
Equality constraint variable subset
x_{11} x_{12} x_{13} x_{1N}
x_{21} x_{22} x_{23} x_{2N}
x_{H1} x_{H2} x_{H3} x_{HN}

Fig. 1. Inequality and equality constraints involve different variable subsets, thereby making the problem inseparable.

The total number of integer variables in the above problem is $n = N \times H$ and the total number of constraints are $(H + N)$.

2.1. Challenges in solving the problem

It is worth noting that the above problem is not an additively separable problem, as the first set of constraints involve a different set of variables than the second set of constraints. For example, the first inequality constraint involves variables x_{1j} (for $j = 1, 2, \ldots, N$), whereas the first equality constraint involves variables x_{i1} (for $i = 1, 2, \ldots, H$), as shown Fig. 1. Thus, there is no way the problem can be partitioned into separable sub-problems. For a large-sized problem having a large H or N or both, such problems are challenging to any optimization algorithm.

The commonly-used methods to handle discrete variables is the branch-and-bound method or branch-and-cut methods (Land & Doig, 1960; Mitchell, 2002; Reklaitis et al., 1983) which require exponentially more computational time with a linear increase in variables. These methods relax the integer restrictions and treat the problem as a real-parameter optimization problem to first find the real-parameter optimal solution. Thereafter, based on the closeness of the solution to an integer solution, a new problem is formulated and solved. For example, the branch-and-bound method (Nau, Kumar, & Kanal, 1984) chooses a 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 for each problem. This 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 increase, exponentially more such branching into new LPs are required, thereby making the overall approach computationally expensive.

A little thought will also reveal that the above problem is a multiply constrained bounded knapsack problem (MKP) Kellerer, Pferschy, and Pisinger (2004), which is known to be a NP-complete problem and is difficult to solve. However, pseudo-polynomial time complexity algorithms are possible to be developed for these problems, if a near-optimal solution is desired. A number of past studies using point-based and population-based methods have attempted to solve the MKP problem (Aminbaksh & Sonmez, 2016; Cheng et al., 2010; Kong & Tian, 2006) having Boolean variables. Two studies (Cheng et al., 2010; Kong & Tian, 2006) used repair methods and also combined particle swarm optimization (PSO) with classical linear programming (LP) methods, but restricted the applications to a maximum of $N = 100$ and $H = 10$, thereby having a total of 1000 variables. Another study (Aminbaksh & Sonmez, 2016) used 630 × 5 or 3150 Boolean variables in a discrete time-cost MKP problem. These are large-sized problems by any standard, but here, we propose and demonstrate a polynomial-time algorithm for solving the above MKP problem with integer variables over a large range of variables, spanning from 50,000 to one billion variables.

2.2. Upper bound on optimal metal utilization ratio

From each of N objects, the number of copies of the jth object (r_j) and its weight (w_j) are supplied. So, we can compute the total metal required to cast all objects: $M = \sum_{j=1}^{N} r_j w_j$. For a fixed crucible of size W for every heat, this means that at least $H = \lceil M/W \rceil$ heats are required. Thus, choosing H heats to cast all objects, the upper bound of the optimal metal utilization ratio is $\hat{U}^* = 100M/(H \times W)$, in percentage. The exact optimal metal utilization ratio depends on the combinations of different object weights (w_j) with the crucible size W. If $\hat{U}^*$ is achievable, the respective allocation is the optimal solution. However, if p extra heats are required to accommodate different weight combinations, the exact optimal metal utilization ratio reduces by a factor $p/(H + p)$.

3. A computationally fast PILP algorithm

The above-described casting scheduling problem involves a linear objective function, H linear inequality and N linear equality constraints. Our proposed approach is motivated by an earlier study (Deb, Reddy, & Singh, 2003) in which a customized genetic algorithm with a problem-specific initialization and genetic operators was used. Implementations used in this paper are computationally more efficient. They are achieved by evaluating the modified algorithm thoroughly, extending its application to a billion-variable problem, making additional but important parametric studies, and presenting it as a population-based integer linear programming (PILP) algorithm, so the algorithm is understandable from a traditional optimization framework as well. First, we provide a pseudo-code of the complete PILP procedure in Algorithm 1. ‘Random’ operator is applied with replacement.

3.1. Estimating number of heats

Algorithm 2 computes the required number of heats to melt the necessary amount of metal to make copies of all objects. At every heat, a metal utilization ratio factor of η is assumed. In most simulations here, to achieve a near-optimal solution, we have used $\eta = 0.997$.

3.2. Customized initialization

PILP works with a population $\mathbf{P}$ of n solutions in each iteration, instead of a single solution as in the case of point-based optimization algorithms. To start a run, n randomly generated solutions are created, but every solution (say, $\mathbf{y} \in R^N$) is guaranteed to satisfy the linear equality constraints of the following type:

$$\sum_{j=1}^{N} y_j = a, \tag{5}$$

where a is a predefined integer representing the demand of an object. For this purpose, first, a random set of integers within $y_j \in [0, A]$ (where A can be computed as the maximum number of copies allowed to each casting, or $\lceil \max(\mathbf{W})/\min(\mathbf{w}) \rceil$) are created and then all N variables are repaired as follows: $y_j \leftarrow y_j \frac{a}{\sum_{i=1}^{N} y_i}$. While an ad-hoc number was used for A in the previous study (Deb et al., 2003), here we use a number based on problem-specific information. This repair approach may not produce an integer value for y_j. In such a case, the real number y_j is rounded to its nearest integer value. Thereafter, if the sum of adjusted integer values is not

Algorithm 1 Population-based integer linear programming (PILP) algorithm for problems 1–4.

```
Input: Problem parameters $N$; $(w_j, r_j)$, for $j = 1, 2, \dots, N$; $W_i$ for $i = 1, 2, \dots$; objective target $\eta$; population size $n$; parent size $\mu$
Output: Number of rows ($H$); desired solution matrix $x_{ij}$ for $i = 1, 2, \dots, H$ and $j = 1, 2, \dots, N$; obtained objective $f^T$
1: $H \leftarrow$ Estimate_Heats($\mathbf{w}, \mathbf{r}, \mathbf{W}, \eta$)   {Calculate number of heats needed}
2: $\mathbf{P} \leftarrow$ Custom_Initialization($n$, $\mathbf{w}$, $\mathbf{W}$)   {Initialize n solutions}
3: $B \leftarrow$ Update_Best($\mathbf{P}$)~{Finds best feasible solution~in~$\mathbf{P}$}
4: $t \leftarrow 0$   {Iteration counter}
5: while ($F(B) < \eta$) do
6:   $\mathbf{Q} \leftarrow \emptyset$
7:   repeat
8:     [ID1, ID2, ID3, ID4] $\leftarrow$Random($\mathbf{P}$,4)   {Four solutions with ID1 toID4 are chosen at random from $\mathbf{P}$}
9:     parent1 $\leftarrow$ tournament_Selection($\mathbf{x}^{ID1}$, $\mathbf{x}^{ID2}$){Better of ID1 and ID2 is chosen}
10:    parent2 $\leftarrow$ tournament_Selection($\mathbf{x}^{ID3}$, $\mathbf{x}^{ID4}$){Better of ID3 and ID4 is chosen}
11:    offspring $\leftarrow$ Custom_Recombination(parent1, parent2)   {A new solution is created}
12:    repair1 $\leftarrow$Custom_Repair1(offspring);   {Repaired for equality constraint satisfaction}
13:    repair2 $\leftarrow$Custom_Repair2(repair1)   {Repaired for inequality constraint satisfaction}
14:    $\mathbf{Q} \leftarrow \mathbf{Q} \cup$ {repair2}   {Repaired solution is added in population $\mathbf{Q}$}
15:  until $|\mathbf{Q}| = n$   {Until n new solutions are created}
16:  $B \leftarrow$ Update_Best($\mathbf{Q}$)   {Bestsolution is updated}
17:  $\mathbf{P}$= $\mathbf{Q}$
18:  $t \leftarrow t + 1$   {Increment iteration counter}
19: end while
20: $f^T = f(B)$
```

Algorithm 2 Estimate_heats procedure.

```
Input: Problem parameters: $\mathbf{w}$, $\mathbf{r}$, $\mathbf{W}$, and $\eta$
Output: Number of heats, $H$
1: $M = \sum_{j=1}^{N} r_j w_j$   {Total metal needed}
2: metal $\leftarrow$ 0
3: $i \leftarrow 0$
4: repeat
5:   metal $\leftarrow$ metal + $\eta W_i${$\eta$ factor ofcrucible size is added}
6:   $i \leftarrow i + 1$
7: until metal $\geq M$
8: $H = i$   {Minimum number of heats needed to melt M}
```

a, reduction or increase in one of more adjusted integer values are made at random to make sure the equality constraint is satisfied. The above repair is achieved by using two repair operators which we describe in Section 3.6. The repaired solution is then evaluated. A pseudo-code for the custom initialization procedure is presented in Algorithm 3.

3.3. Evaluation of fitness value

Every population member $\mathbf{x}$ is evaluated by adding the objective function value $f(\mathbf{x})$ described in Eq. (1) and penalty value, if any, from constraint violation (Goldberg, 1989; Reklaitis et al., 1983), computed as follows:

Algorithm 3 Custom_initialization procedure.

```
Input: Population size $n$, $\mathbf{w}$ and $\mathbf{W}$
Output: Created and repaired population $\mathbf{P}$
1: $A = \lceil \max(\mathbf{W}) / \min(\mathbf{w}) \rceil$
2: for $k \leftarrow 1, n$ do
3:   for $j \leftarrow 1, N$ do
4:     sum $\leftarrow$ 0
5:     for $i \leftarrow 1, H$ do
6:       $x_{ij}^{(k)}$ = rnd(0, $A$){Create a random integer between 0 and $A$}
7:       sum $\leftarrow$ sum + $x_{ij}^{(k)}$
8:     end for
9:     for $i \leftarrow 1, H$ do
10:      $x_{ij}^{(k)}$ = round$\left(\frac{r_j}{\text{sum}} x_{ij}^{(k)}\right)$
11:    end for
12:  end for
13:  $\mathbf{x}^{(k)} \leftarrow$ Custom_Repair1($\mathbf{x}^{(k)}$)   {Described in Section 3.6}
14:  $P(k) \leftarrow$ Custom_Repair2($\mathbf{x}^{(k)}$)   {Described in Section 3.6}
15:  Evaluate fitness $F(P(k))$   {Described in Section 3.3}
16: end for
```

$$F(\mathbf{x}) = f(\mathbf{x}) - R\left[\sum_{j=1}^{N}\left(\sum_{i=1}^{H} x_{ij} - r_j\right)^2 + \sum_{i=1}^{H}\left\langle \frac{1}{W_i}\sum_{j=1}^{N} w_j x_{ij} - 1\right\rangle^2\right]. \tag{6}$$

The bracket operator $\langle\cdot\rangle$ is defined as the value of the operand, if it is positive, otherwise the value is zero. While the previous study (Deb et al., 2003) used an ad-hoc penalty based approach, here we provide a justification for the choice of an appropriate penalty parameter. A little thought will reveal that for a feasible solution $\mathbf{x}$ (satisfying all equality and inequality constraints), $F(\mathbf{x}) = f(\mathbf{x}) = \eta$. Since, penalty term is subtracted from $f(\mathbf{x}_{\text{infeas}})$, for any infeasible solution $\mathbf{x}_{\text{infeas}}$, we would like to set the penalty parameter R in a way so that $F(\mathbf{x}_{\text{infeas}}) < \eta$. Here we use problem knowledge to set this parameter. Satisfaction of demand equality constraint (first term within the bracket) is easier through our first repair operator, hence we set R in way so that a single unit violation will make $F(\mathbf{x}_{\text{infeas}}) < \eta$. This can be accomplished by setting $R \geq 1$, as the objective value $f(\mathbf{x}_{\text{infeas}})$ is expected to lie close to one (the maximum being $\sum_{j=1}^{N} r_j w_j / \left(H \min_{i=1}^{H} W_i\right)$). For the capacity inequality constraints, we would like to set R in a way so that even one kilogram capacity violation would make $F(\mathbf{x})$ worse than target metal utilization ratio η. Assuming $f(\mathbf{x}_{\text{infeas}}) \approx 1$, we have $1 - R/W_i^2 < \eta$ for all i, leading to $R > (1-\eta)W_i^2$. For our study here, we have chosen $\eta = 0.997$ and two vessel sizes of $W_i = 500$ and 650 kilograms, yielding R around 750 and 1268, respectively. We use an average value of $R = 1000$ throughout this study. This ensures that if a fitness value of $F(\mathbf{x}) = \eta$ is obtained by our method, it cannot be an infeasible solution. This suggests that we can eliminate $f(\mathbf{x})$ from the fitness expression and simply minimize the penalty term to obtain the desired targeted feasible solution. Our initial experimentation on smaller problem sizes finds identical results. A hierarchical approach, as suggested elsewhere (Deb, 2000), could also be used instead.

3.4. Update of best population member

The best population member of the population $\mathbf{P}$ is computed in this procedure. The solution having the largest fitness in $\mathbf{P}$ is the result of this operation, as shown in Algorithm 4.

Algorithm 4 Update_best procedure.

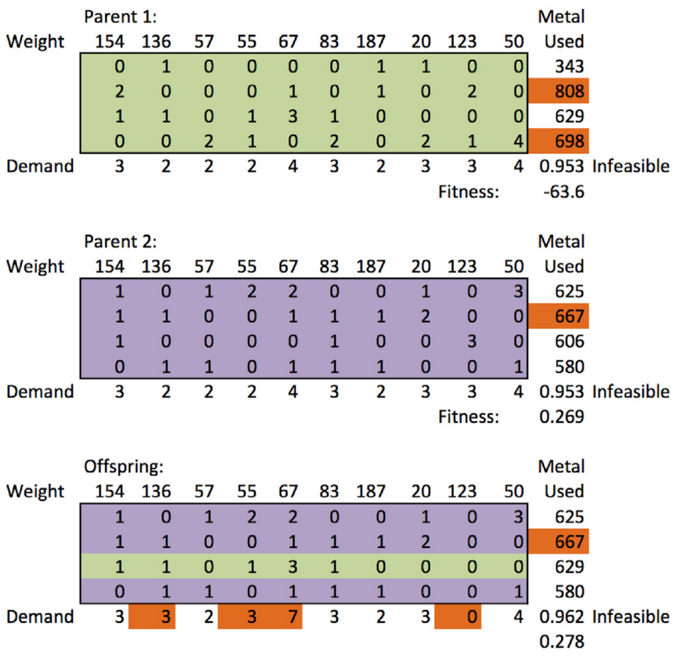
```
Input: Population $\mathbf{P}$
Output: Best solution $\mathbf{x}^{\text{bestID}}$
1: bestID $\leftarrow$ $\mathbf{argmax}_{k=1}^{n} F(P(k))$
```

3.5. Customized recombination operator

The purpose of a recombination operator is to mix partial information from two or more parent solutions and create one or more new offspring solutions (Goldberg, 1989). Population based evolutionary optimization algorithms are arguably unique and different from other optimization algorithms in their ability to recombine good contents of different evolving population members into a single offspring solution. By definition, the *superposition* principle of linear problems enables such a recombination operator to lead to a much better offspring solution than the individual parent solutions. It is in this spirit, we design the following customized recombination operator for the linear casting scheduling problem.

For a generic p-parent recombination operation (a generalization of our previous two-parent approach (Deb et al., 2003)), we consider all chosen p parent solutions heat-wise. For each heat (index i), the remaining crucible space U_i for the heat is compared among all p solutions. Then, all variables x_{ij} for $j = 1, 2 \ldots, N$ from the best U_i solution are copied to the offspring solution. This is repeated for all H heats to construct the new offspring solution. A two-parent version of the recombination operator is shown in Fig. 2 on two infeasible parent solutions. For a detailed explanation of the recombination operator, we provide a pseudo-code for $p = 2$ in Algorithm 5.

If the parent solutions satisfy all inequality constraints, the created solution will automatically satisfy all inequality constraints,



Parent 1:

											Metal	
Weight	154	136	57	55	67	83	187	20	123	50	Used	
	0	1	0	0	0	0	1	1	0	0	343	
	2	0	0	0	1	0	1	0	2	0	808	
	1	1	0	1	3	1	0	0	0	0	629	
	0	0	2	1	0	2	0	2	1	4	698	
Demand	3	2	2	2	4	3	2	3	3	4	0.953	Infeasible
									Fitness:		-63.6	

Parent 2:

											Metal	
Weight	154	136	57	55	67	83	187	20	123	50	Used	
	1	0	1	2	2	0	0	1	0	3	625	
	1	1	0	0	1	1	1	2	0	0	667	
	1	0	0	0	0	1	0	0	3	0	606	
	0	1	1	0	1	1	1	0	0	1	580	
Demand	3	2	2	2	4	3	2	3	3	4	0.953	Infeasible
									Fitness:		0.269	

Offspring:

											Metal	
Weight	154	136	57	55	67	83	187	20	123	50	Used	
	1	0	1	2	2	0	0	1	0	3	625	
	1	1	0	0	1	1	1	2	0	0	667	
	1	1	0	1	3	1	0	0	0	0	629	
	0	1	1	0	1	1	1	0	0	1	580	
Demand	3	3	2	3	7	3	2	3	0	4	0.962	Infeasible
											0.278	

Fig. 2. Recombination operator is illustrated for a small problem with $N = 10$, $H = 4$, and having $W = 650$ kilograms vessel. Demand (r_j) values of each object are shown in the final row. Columns indicate objects ($j = 1$ to 10) and rows indicate heats ($i = 1$ to 4). The last column prints $\sum_{j=1}^{N} w_j x_{ij}$ values for each heat. Offspring is constructed using best metal-utilized rows. For example, parent 2 has a better metal utilization for the first heat (first row) than parent 1 (625 kilograms is closer to $W = 650$ kilograms than 343 kilograms); hence first heat of offspring entirely comes from first heat of parent 2. The resulting offspring does not satisfy one inequality ($i = 2$) and four equality ($j = 2$,4,5, and 9) constraints.

Algorithm 5 Custom_recombination procedure for two parents.

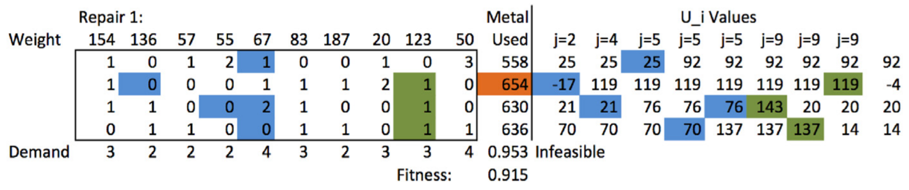
```
Input: Parent solutions $\mathbf{x}^{(p1)}$ and $\mathbf{x}^{(p2)}$ to be recombined
Output: Created offspring solution $\mathbf{x}^{(c)}$
1: for $i \leftarrow 1, H$ do
2:   $U_i = W_i - \sum_{j=1}^{N} w_j x_{ij}$ {Remaining crucible space in each heat}
3:   if $\left(U_i^{(p1)} \geq 0,\ U_i^{(p2)} \leq 0\right)$ or$(U_i^{(p1)} \geq 0,\ U_i^{(p2)} > 0$ and $U_i^{(p1)} < U_i^{(p2)})$ or $\left(U_i^{(p1)} < 0,\ U_i^{(p2)} < 0 \text{ and } U_i^{(p1)} > U_i^{(p2)}\right)$ then
4:     {If the first parent has a better metal utilization}
5:     for $j \leftarrow 1, N$ do
6:       $x_{ij}^{(c)} = x_{ij}^{(p1)}$ {Child solution inherits assignment from first parent}
7:     end for
8:   else
9:     for $j \leftarrow 1, N$ do
10:      $x_{ij}^{(c)} = x_{ij}^{(p2)}$ {Else child solution inherits assignment from second parent}
11:    end for
12:  end if
13: end for
```

however the above recombination process may not satisfy the equality constraints. Therefore, we attempt to fix the equality constraints first using a customized repair operator, followed by a second repair operator which attempts to fix any violated inequality constraint that may have resulted from the first repair operator, but ensures satisfaction of equality constraints. We describe these two repair operators next.

3.6. Customized repair operators

As mentioned above, the multi-parent recombination operation described in the previous section may not satisfy the equality constraints (Eq. (3)). We use the first repair operator to try to satisfy all *equality* constraints. For a solution, all x_{ij} values for jth object are added and compared with the desired number of copies r_j. If the added quantity is the same as r_j, the corresponding equality constraint is already satisfied and no further modification is needed. If the added quantity is larger than r_j, we identify the heat that requires molten metal closest to or more than the crucible size. We then decrease one assignment from this heat and repeat the process by recalculating the remaining crucible space of each heat. The procedure is presented in Algorithm 6.

There is one caution with this procedure, which is not detailed in the above algorithm for clarity. If $x_{\text{minID},j}$ is already equal to zero, the variable cannot be reduced any further. In this case, we follow the above procedure with the heat having the next-highest assignment of metal compared to the crucible size, and so on. On the other hand, if the total assignment for an object is smaller than the desired number of copies r_j, then the reverse of the above procedure is followed. First, the heat with number maxID having maximum available space is chosen and $x_{\text{maxID},j}$ is increased by one. Since an assignment is added, no further checks are necessary here. This repair operator is applied on the recombined solution obtained in Fig. 3. Next, the modified solution is sent to the second repair operator which attempts to alter the decision variables further without violating the equality constraints but attempting to satisfy the inequality constraints as much as possible, as presented in Algorithm 7. The heat (with number minID) having the maximum violation in inequality constraint is chosen for fixing. A randomly chosen object (objID) with non-zero assignment

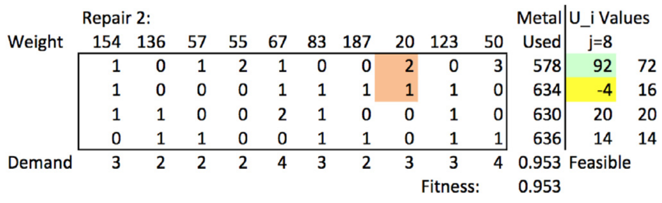


	Repair 1:										Metal				U_i Values					
Weight	154	136	57	55	67	83	187	20	123	50	Used	j=2	j=4	j=5	j=5	j=5	j=9	j=9	j=9	
	1	0	1	2	1	0	0	1	0	3	558	25	25	25	92	92	92	92	92	92
	1	0	0	0	1	1	1	2	1	0	654	-17	119	119	119	119	119	119	119	-4
	1	1	0	0	2	1	0	0	1	0	630	21	21	76	76	76	143	20	20	20
	0	1	1	0	0	1	1	0	1	1	636	70	70	70	70	137	137	137	14	14
Demand	3	2	2	2	4	3	2	3	3	4	0.953	Infeasible								
								Fitness:			0.915									

Fig. 3. Repair 1 operator is applied on offspring solution obtained in Fig. 2. Based on under or over-achievement of demand, the respective variables are increased (green) or decreased (blue) according to procedure outlined in Algorithm 6. For example, second ($j = 2$) casting has an extra allocation, which was reduced from second heat ($i = 2$), as $U_2 = -17$ is minimum at this row. Fourth column ($j = 4$) requires a reduction of one assignment from third ($i = 3$ heat with a minimum $U_3 = 21$ value) and fifth column ($j = 5$) required an adjustment three times (with $i = 1$, 4, and 3, respectively). Ninth column ($j = 9$) casting had no allocation and had to be adjusted three times ($i = 3$, 4 and 2, respectively) with maximum U_i values. The process results in a better infeasible solution (all equality constraints are satisfied and only second ($i = 2$) inequality constraint is not satisfied) than the offspring solution.

Algorithm 6 Custom_repair1 procedure.

```
Input: Solution x to be repaired
Output: Repaired new solution x
1:  for $j \leftarrow 1, N$ do
2:    while $\sum_{i=1}^{H} x_{ij} \neq r_j$ do
3:      {As long as copies made do not tally ordered copies}
4:      for $i \leftarrow 1, H$ do
5:        $U_i = W_i - \sum_{j=1}^{N} w_j x_{ij}$   {Remaining crucible space in each heat}
6:      end for
7:      if $\sum_{i=1}^{H} x_{ij} > r_j$ then
8:        {If more copies are assigned}
9:        minID ← $\textbf{argmin}_{i=1}^{H} U_i$  {Identify~heat~that occupies minimum (or negative) space in crucible}
10:       $x_{\text{minID},j} \leftarrow x_{\text{minID},j} - 1$   {Decrease one assignment}
11:     else if $\sum_{i=1}^{H} x_{ij} < r_j$ then
12:       {If less copies are assigned}
13:       maxID ← $\textbf{argmax}_{i=1}^{H} U_i$   {Identify heat having maximum crucible space}
14:       $x_{\text{maxID},j} \leftarrow x_{\text{maxID},j} + 1$   {Increase one assignment}
15:     end if
16:   end while
17: end for
```



	Repair 2:										Metal	U_i Values	
Weight	154	136	57	55	67	83	187	20	123	50	Used	j=8	
	1	0	1	2	1	0	0	2	0	3	578	92	72
	1	0	0	0	1	1	1	1	1	0	634	-4	16
	1	1	0	0	2	1	0	0	1	0	630	20	20
	0	1	1	0	0	1	1	0	1	1	636	14	14
Demand	3	2	2	2	4	3	2	3	3	4	0.953	Feasible	
								Fitness:			0.953		

Fig. 4. Repair 2 operator is applied to the solution obtained after repair 1 operator. The process is able to produce a feasible solution. Only one heat ($i = 2$) does not satisfy the respective inequality constraint with $U_2 = -4$. One ($j = 8$) of the non-zero allocations ($j = 1,\ 5,\ldots,9$) of second heat is chosen at random and one allocation is exchanged with the maximum U_i (having $U_1 = 92$ for $i = 1$). The altered variable values are marked in orange color. Since revised U_i values (last column) are all non-negative, the process is terminated, indicating that a feasible solution is obtained.

is selected and one assignment is reduced in an attempt to satisfy the inequality constraint. To satisfy the respective demand equality constraint, the heat with the maximum available crucible space is chosen and one assignment for object objID is increased. This process ensures that the equality constraint is always satisfied and a repetitive application of the above process is expected to satisfy inequality constraints as well. This repair operator is illustrated on the repaired solution obtained in Fig. 3. Each of the operations in recombination and repair operators involve re-computation of quantities over H and N loops over and over again. Compared to

Algorithm 7 Custom_repair2 procedure.

```
Input: Solution x to be repaired
Output: Repaired new solution x
1:  for $i \leftarrow 1, H$ do
2:    $U_i = W_i - \sum_{j=1}^{N} w_j x_{ij}$   {Remaining crucible space in each heat}
3:  end for
4:  minID ← $\textbf{argmin}_{i=1}^{H} U_i$ {Identify heat that leaves minimum (or negative) space in crucible}
5:  while $U_{\text{minID}} < 0$ do
6:    {As long as required metal weight exceeds crucible capacity}
7:    objID ← $\textbf{rnd}(j | x_{\text{minID},j} > 0)$   {Identify a random object with non-zero assigned copies}
8:    $x_{\text{minID,objID}} \leftarrow x_{\text{minID,objID}} - 1$   {Remove one assignment from exceeded heat}
9:    maxID ← $\textbf{argmax}_{i=1}^{H} U_i$  {Identify heat having maximum remaining crucible space}
10:   $x_{\text{maxID,objID}} \leftarrow x_{\text{maxID,objID}} + 1$   {Add one assignment to most available heat}
11:   for $i \leftarrow 1, H$ do
12:     $U_i = W_i - \sum_{j=1}^{N} w_j x_{ij}${Remaining crucible space in each heat}
13:   end for
14:   minID ← $\textbf{argmin}_{i=1}^{H} U_i$   {Most violated heat for next round in while loop}
15: end while
```

our previous implementation (Deb et al., 2003), we have used efficient data structures and efficient coding procedures so that the implementation is lean and the whole procedure can be extended to handle very large-sized problems. For instance, in finding maximum and minimum of an array of size n, we have used custom sorting using priority-queues, enabling a reduction in time complexity from n to $\log n$. For brevity, the pseudo-codes shown here represent only the concepts, but do not reflect the intricacies in the actual codes.

After all *possible* infeasible heats are rectified as above without violating the equality constraints, the resulting solution may still be infeasible. In such a case, a constraint violation equal to the sum of the normalized violations of all infeasible heats is assigned to the fitness function, as given in Eq. (6).

3.7. Overall time complexity

The initialization routine requires $O(nNH)$ operations. The recombination operator requires $O(NH)$ copying operations. The first repair operator requires $O(NH)$ operations for each new solution, thereby requiring a total of $O(nNH)$ operations. The while loop in

Table 1
Casting scheduling problem parameters used for initial comparative study.

No.	1	2	3	4	5	6	7	8	9	10	Total
Wt. (kilograms)	175	145	65	55	95	75	195	20	125	50	
# Copies	20	20	20	20	20	20	20	20	20	20	200
Total (kilograms)	3500	2900	1300	1100	1900	1500	3900	400	2500	1000	20,000

the second repair operator may take a few iterations, which is unknown for any problem, but the operations inside the loop is only $O(H)$. Thus, assuming a constant number of iterations inside the while loop, the overall the complexity of the proposed PILP per iteration is $O(nNH)$. The number of iterations required to achieve the desired target solution is not known beforehand, but in all our simulations, a maximum iteration of 20 to 30 was enough in problems up to one billion variables. It is interesting that the overall complexity of our approach is linear to the number of objects, number of heats, and population size.

4. Results using integer linear programming (ILP)

To investigate the ease (or difficulty) of solving the casting scheduling problem, first, we use two commonly-used mixed-integer linear programming softwares which use a different point-based optimization approach. One of them is the freely available Octave's `glpk` software which uses GNU Linear Programming Kit (GLPK) package for solving large-scale integer linear programming problems. The integer restriction of the variables is achieved with the branch-and-cut method which is a combination of branch-and-bound and cutting plane method (Mitchell, 2002). The second is a commercially available popular software CPLEX, which also uses the branch-and-cut method as a core optimization method. All simulations of this section are run on a Dell Precision M6800 Intel Core I7 4940MX CPU with 3.10 GHZ and having 32 Gigabytes RAM and Windows 8.1 Pro operating system.

First, we choose the casting scheduling problem having a reasonably small number of variables. The problem parameters for this first study are presented in Table 1. There are 10 different objects, each with a different individual weight. Only 20 copies of each object are to made and the total 20,000 kilograms of molten metal is required to make all 200 castings. With a single 650 kilograms crucible used for each heat, Section 2.2 suggests $\lceil 20,000/650 \rceil$ or 31 heats to complete all casting. This corresponds to an upper bound on metal utilization ratio of $(100 \times 20,000)/(31 \times 650)$ or 99.256%. If this is achievable for a solution, it is an optimal solution. For the above object parameters, no higher metal utilization ratio is possible, as one less number of heats (that is, 30 heats) will melt only 30×650 or 19,500 kilograms of metal – 500 kilograms short of the required amount of metal needed to cast all 200 objects. However, it is not clear whether the above upper bound is achievable or not, as it depends on the different combinations of object weights to meet the crucible size as close as possible. This can be verified after the optimization problem is solved, thereby requiring the need for applying an optimization algorithm. With $H = 31$ heats and $N = 10$ objects, there are 310 integer variables to the underlying integer linear programming problem. Thus, the variable matrix has 31 rows and 10 columns.

First, we apply the `glpk` routine of Octave software with bounds on each variable as $x_{ij} \in [0, 15]$ (we keep this bound for making all three methods comparable to each other). After running for one hour on the above-mentioned computer for each run starting from a different initial guess solution, the `glpk` routine could not come up with any feasible solution in all 10 runs. We extended one of the runs up to 15 hours on the same computer and still no feasible solution was found. This may indicate that the upper bound (99.256%) on metal utilization ratio may not be possible to achieve in 31 heats.

To test the level of difficulty of the casting scheduling problem with 310 variables, we relax the number of heats by one, that is, we now allow one more heat with the availability of additional 650 kilograms of metal to investigate if the `glpk` routine would now find the target solution. This increases the number of variables to 320 (32 rows and 10 columns), but now a metal utilization ratio of only $20,000/(32 \times 650)$ or 96.154% is possible. This time, the `glpk` routine is able to find a target solution in all 10 runs, requiring only 1.24 seconds computational time and with 44,675 solution evaluations, on an average, per run. This is interesting that by adding one more heat (implying an addition of 10 new variables), the algorithm gets enough flexibility with weight combinations to find a feasible schedule. Although this relaxation in problem variables has resulted in a solution with a reduced metal utilization ratio, it still does not portray whether there exists a solution with the maximum possible metal utilization ratio of 99.256% for the 310-variable version of the problem.

Next, we use IBM's CPLEX software to solve the 310-variable first. Again, variables are restricted to vary within [0, 15]. Interestingly, the CPLEX software is able to find a feasible solution with 99.256% metal utilization ratio in only 0.05 seconds on the same computer, on an average, on 10 different runs. This clearly states that the CPLEX software is more efficient in solving this integer LP problem than the `glpk` routine of Octave. Also, it confirms that the object parameters used for this problem allow a feasible solution having the upper bound of metal utilization ratio of 99.256%, hence the obtained solution is an *optimal* solution. For completeness, we also employ the CPLEX software to solve the relaxed 320-variable problem and it requires only 0.03 seconds on an average to find a solution with 96.154% metal utilization ratio – identical to that found by the `glpk` routine. CPLEX requires only 184 solution evaluations compared to 44,675 solution evaluations needed by the `glpk` routine.

Our proposed PILP algorithm is applied next to first solve the 320-variable version of the problem. It is observed that with a population of size 12, PILP takes only 0.02 seconds and 26 solution evaluations per run (requiring two or three generations in all 10 runs), on an average, to find the target solution in 10 different runs. With one extra heat than the minimal number of required heats and having a small number of variables, the problem is easier to solve by PILP. As can be seen from the table, compared to the CPLEX software, PILP algorithm is slightly faster and requires seven times less solution evaluations to achieve the same solution. Table 2 presents a summary of these results. Interestingly, our PILP algorithm is also able to solve the more difficult version of the problem having 310 variables, but in only 0.04 seconds (compared to 0.05 seconds required by CPLEX) and requiring 150 solution evaluations (compared to 249 solution evaluations needed by CPLEX). Recall that this version was not possible to be solved by the `glpk` procedure.

Despite the slight edge of our proposed PILP approach over the popular CPLEX software, we would like to highlight that a problem with 310 integer variables is a challenging problem to any optimization algorithm and the above comparison has already demonstrated the superiority of integer restriction handling of the CPLEX software compared to `glpk` implementation of Octave. To

Table 2
Comparison of `glpk`, `CPLEX`, and PILP algorithm algorithms. ‘SD’ represents standard deviation.

Method	Time (s)		# Evals.		Metal	Time (s)		# Evals.		Metal
	Avg.	SD	Avg.	SD	Util	Avg.	SD	Avg.	SD	Util
	320-Variable					310-Variable				
`glpk`	1.24	0.22	44,675	0.00	96.154%	–	–	–	–	–
`CPLEX`	0.03	0.00	184	0.00	96.154%	0.05	0.00	249	0.00	99.256%
PILP	0.02	0.00	26	5.06	96.154%	0.04	0.01	150	48.08	99.256%
	1000-Variable					2000-Variable				
`CPLEX`	0.13	0.00	947	0.00	99.462%	–	–	–	–	–
PILP	0.05	0.004	220	37.71	99.462%	0.19	0.010	224	18.38	99.596%

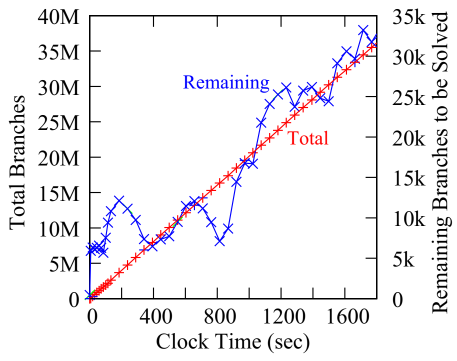


Fig. 5. For 2000-variable version of the cast scheduling problem, the `CPLEX` software produces linearly more branches with computational time, but the number of unsolved branches also increase, thereby demonstrating its inability to converge.

evaluate further, we scale up the number of orders of each object to 65 copies (except the first object to be made 63 copies), so that required number of heats is around 100. This requires a total of 650 objects to be made requiring 64,650 kilograms of molten metal. This needs at least ⌈64, 650/650⌉ or 100 heats. This problem introduces 100 × 10 or 1000 variables to the corresponding optimization problem. Thus, the maximum possible metal utilization ratio is (100 × 64, 650)/(100 × 650) or 99.462%. Interestingly, the `CPLEX` software is able to find a feasible solution with 99.462% metal utilization ratio in only 0.13 seconds and using only 947 solution evaluations, on an average over 10 runs, which is the optimal solution to the problem. Our proposed PILP algorithm, with 20 population members, is also able to repeatedly find the same metal utilization ratio (99.462%) solution with a much smaller computational time (0.05 seconds) and in less than one quarter of solution evaluations than `CPLEX`, on an average over 10 independent runs.

What is more interesting is when we scale up the problem further (the first object requiring 127 copies and all other 130 copies) to require a total of 129,475 kilograms of molten metal. With a crucible size of 650 kilograms per heat, this requires at least ⌈129, 475/650⌉ or 200 heats. Thus, the number of variables to the resulting optimization problem increases to 2000 and the upper bound of the metal utilization ratio is (100 × 129, 475)/(200 × 650) or 99.596%. This time, the `CPLEX` software is not able to find a feasible solution in any of the 10 runs (each ran for an hour), even when one of the runs was extended up to 15 hours on the above-mentioned computer. With 2000 integer variables, the branch-and-cut methods of the `CPLEX` software spans into too many branches to solve the problem in a reasonable amount of time. To have a better idea of convergence behavior of a typical `CPLEX` run for 2000 integer variables, Fig. 5 indicates that `CPLEX` steadily produced 36,576,895 branches after 1800 seconds (30 minutes) of running time, but still had 33,416 unsolved branches to be solved to find a feasible solution. The run is continued for another 14.5 hours without showing any sign of convergence. The figure indicates that with an increased number of branches produced, the number of unsolved branches also increases. This non-convergence behavior gets worse with computational time. On the contrary, our PILP algorithm, with only 20 population members, is able to solve the same problem in 0.19 seconds requiring only 224 solution evaluations, on an average, in 10 runs. Since a solution with 99.596% metal utilization ratio is obtained, it corresponds to the optimal solution. A typical foundry may plan for a month-long scheduling, requiring about $H = 3200$ heats with $N = 10$ objects. This makes a total number of variables to be as high as 32,000. Clearly, the current state-of-the-art softwares are not capable of addressing these practical problems.

The above results clearly demonstrate one aspect: when the problem size is large, the point-based optimization algorithms for handling an integer LP problem using the branch-and-cut fix-up method is not efficient – there are simply too many branches for an algorithm to negotiate in a reasonable amount of time or iterations. The use of an efficient population-based optimization approach with customized operators have more potential in solving such problems.

5. Results using PILP

Having demonstrated the usefulness of the population-based approach for handling the specific integer-valued casting scheduling problem, we now evaluate our proposed approach more rigorously for various parameter settings and scenarios. All simulations of this section are run on a 2 × Intel 8-Core Xeon-2640V3 2.66 Gigahertz, 20 Megabytes Cache, 8 Gigatesla/second, 16 threads, LGA 2011 computer having ASUS Z10PE-D16/4L Server Motherboard with 16 × 16 Gigabyte DDR4 ECC DIMM 2133 Megahertz RAM having a 1 × Galaxy GTX 980 SOC Nvidia graphics card with Windows 8.1 Pro 64Bit OEM. Due to the excessive run-time memory requirement for storing large number of integers, this computer was specially procured for the purpose.

5.1. Exploring extent of feasible solutions

First, we investigate the extent of the feasible region of the entire integer search space, we consider a million-variable version of the problem. Table 3 shows the problem parameters used for this purpose. Total number of objects is 550,666 and total metal required to cast them is M =56,352,140 kilograms. To bring the problem close to practice, two crucibles of sizes 650 kilograms and 500 kilograms are used on alternate days with 10 and 13 heats on respective days. Thus, the same amount of metal (650 × 10 = 500 × 13 or 6,500 kilograms) is melted each day. This is done to use crucible one day and maintain the other crucible the next day. All results from here-on are obtained for this two-crucible case used in tandem. The upper bound on metal utilization ratio for

Table 3
Casting scheduling problem parameters used as default in most of this study.

No.	1	2	3	4	5	6	7	8	9	10
Wt. (kilograms)	175	145	65	55	95	75	195	20	125	50
# Copies	59,227	58,329	53,327	53,229	53,429	53,526	57,022	52,322	58,229	52,026
Total (10^6 kilograms)	10.364	8.458	3.466	2.928	5.076	4.014	11.119	1.046	7.279	2.601

this problem is 99.9994% (with a melt of 56,352,500 kilograms of metal). If this is achievable, the respective solution is optimal, but it is not possible to know unless an optimization task is performed to exact optimality. It is a difficult task, particularly working on a very large-dimensional search space. Here, we attempt to find a near-optimal solution by fixing a target metal utilization ratio of $\eta = 99.7\%$ and estimate that M/η=56,521,705.12 kilograms molten metal is needed to find a satisfactory solution. This requires $H = 100,000$ heats (of which 43,480 heats of 650 kilograms crucible and 56,520 heats of 500 kilograms crucible). Since there are $N = 10$ objects, the total number of variables in the optimization problem is 1,000,000 (a million).

To investigate the ease of creating a feasible solution at random for the above one million variable problem, we create 10,000 solutions at random. It is observed that none of them is feasible. With a proportionate penalty function to indicate the extent of infeasibility for violating inequality constraints (note that equality constraints are always satisfied for every initial solution using the custom initialization process), the best, median and worst fitness values of these random solutions are found to be as follows: $-7,029,705$, $-7,219,503$, and $-7,380,404$, respectively. Since these values are negative, they indicate that the respective solutions are all infeasible. Note that for a feasible solution, the fitness value (given by Eq. (6)) is always positive and has a maximum value of $\eta = 0.997$.

To investigate the effect of two repair mechanisms (repair operators) suggested in this study, we apply two repair operators in sequence to try to repair each solution. However, even after repair, none of the 10,000 solutions could be made feasible by the repair operators alone, but the extent of constraint violation is reduced after the repair operators are applied. The best, median and worst fitness values of the repaired solutions are $-1,385,578$, $-1,411,499$, and $-1,447,657$, respectively.

Summary of the study: These values indicate that a randomly created solution or the proposed repair mechanisms alone are not enough to create a feasible solution for the million-variable problem. In such a large-scale MKP problem, feasible or near-feasible solutions occupy a tiny part of the search space. An efficient optimization algorithm is needed to locate such a high-performing search region quickly.

5.2. A Typical simulation

We now apply our proposed population-based integer linear programming (PILP) algorithm to solve the above one million version of the casting scheduling problem. To investigate its performance, we use a population of size 40 and run PILP 11 times from different random populations. The population-best and population-average fitness values are recorded for each run and the best, median and worst fitness values over 11 runs are noted. Fig. 6 plots one minus the fitness value on the y-axis in log scale versus the iteration counter on the x-axis. This is done so as to expect the targeted solution to reach a value of $(1 - 0.997)$ or 0.003. Note that all infeasible solutions that violate any demand equality constraint will cause a penalty of at least R (=1000). Thus, an infeasible solution having a y-axis value smaller than almost 1000 violates the inequality constraints alone. An almost linear drop in the

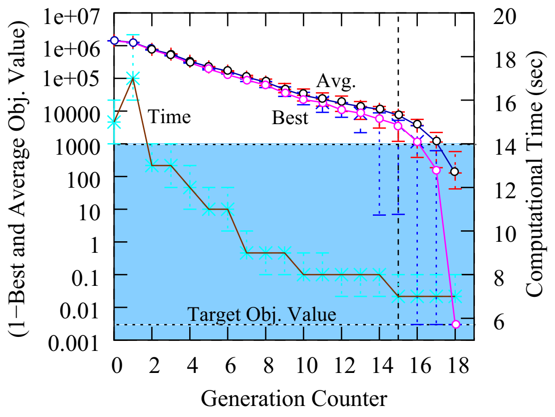


Fig. 6. Iteration-wise variation of population-average, population-best fitness value and computational time for one million version of the problem.

best and average y-axis values on a semi-logarithm plot with iteration counter indicates that while fixing the demand constraints the fitness values reduce in an *exponentially* fast manner with iteration counter. Thereafter, the convergence rate gets even faster in satisfying inequality constraints and the algorithm takes a minimum of 16 iterations to find a feasible target solution in some of the 11 runs. Different runs find the target solution (with a transformed fitness value of 0.003) within 16 to 20 iterations.

In Fig. 6, the right y-axis marks the computational time for each iteration in seconds. The plot with its run-wise variation, indicate that the maximum computational time is spent in completing the first iteration and thereafter the time has a monotonic non-increasing trend with iterations. This is because, with iterations, more inequality constraints get satisfied, thereby reducing the time needed to execute the second repair operator.

Summary of the study: The combined effect of (i) tournament selection operator for selecting better parent solutions, (ii) recombination operator to combine good parts of two parents into one offspring, and (iii) repair operators to repair offspring solutions is able to find increasingly better and better solutions in an exponential fast manner with iterations and locate the desired target solution in only 16 to 18 iterations for a million-variable version problem. This shows a typical working of our proposed PILP procedure.

5.3. Estimating an appropriate population size

An important parameter in a population-based optimization method is the size of the population. In the previous section, we have successfully used a population size of 40. To study the effect of population size, we use a wide range of population sizes (6 to 80), but limit a maximum of 10,000 solution evaluations. Thus, if a run is unable to find the desired metal utilization ratio of 99.7% earlier than 10,000 solution evaluations, the run is considered a failure, otherwise it is considered a success.

Table 4 tabulates the best, median and worst values of the best fitness value obtained in 10 runs. The next main column presents the elapsed number of solution evaluations to find the best

Table 4
Fitness value and the number of solution evaluations are tabulated against different population sizes for one million version of the casting scheduling problem. Best performing results are marked in bold.

Pop.	Fitness value			Solution evaluations		
Size	Best	Median	Worst	Best	Median	Worst
6	−240, 191.10	−358, 443.74	−426, 859.38	42	210	301
10	−114, 310.68	−144, 787.23	−226, 824.27	374	528	968
20	−9, 978.56	−20, 500.75	−30, 271.75	924	2079	2100
28	0.997	0.997	0.997	551	928	2639
34	**0.997**	**0.997**	**0.997**	**630**	**665**	**700**
40	0.997	0.997	0.997	656	738	820
50	0.997	0.997	0.997	663	714	867
60	0.997	0.997	0.997	793	793	854
80	0.997	0.997	0.997	1053	1053	1134

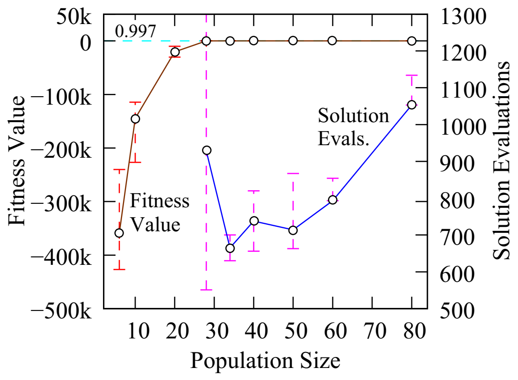


Fig. 7. Effect of population size on the performance on the casting scheduling problem.

fitness value in 10 runs. It is clear that a population of size 34 produces the best median performance over 10 runs. Fig. 7 shows the best, average and worst fitness value obtained for each population size. A negative value indicates that the PILP algorithm is unable to find any feasible solution in 10,000 solution evaluations. When the population size is small, there are not enough samples in the population to provide an initial or temporal diversity needed to find new and improved solutions for this large-scale problem. However, when an adequate population size (here, a population of 28 members) is used, our customized recombination operator is able to exploit the population diversity to create new and improved solutions. With an increase of population size from 28, additional diversity is maintained and the proposed algorithm is able to solve the problem every time. This is a typical performance demonstrated by a successful recombinative genetic algorithm in other studies (Deb & Agrawal, 1999).

In Fig. 7, the right y-axis plots the total number of solution evaluations needed to find the desired solution in each case. The median function evaluations are joined with a line. It is clear from the figure that a population of size 28, although finds the target solutions in all runs, does not produce reliable results, as there is a large variation in number of solution evaluations over 11 runs. But, a population size of 34 uses the smallest number of solution evaluations in all 11 runs and also does not produce a large variation in number of solution evaluations across runs. As the population size is increased from 34 to 80, the desired solution with 99.7% metal utilization ratio is always achieved, but the number of solution evaluations to find this solution also increases. Although this optimal population size may depend on the size of the problem, it is interesting to note that due to the customization in initialization and PILP operators, a population of size 34 is adequate for solving the one million variable integer LP problem.

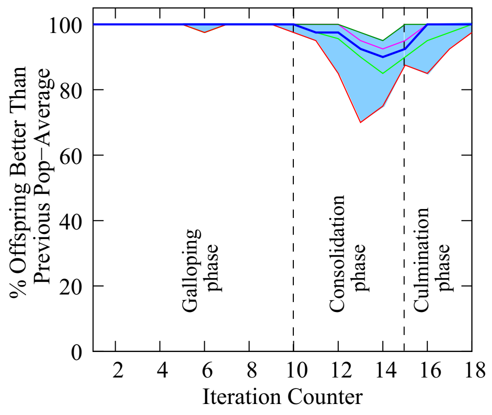


Fig. 8. Percentage of offspring population having better fitness value than previous iteration population-average solution.

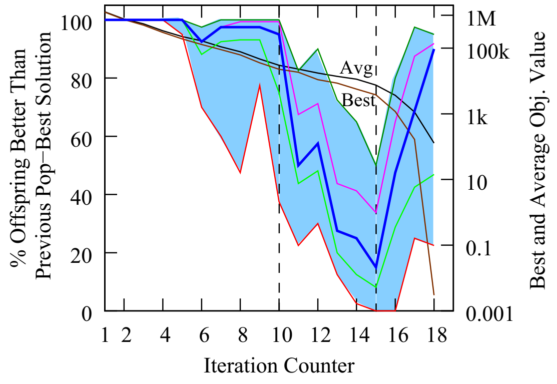


Fig. 9. Percentage of offspring population having better fitness value than previous iteration population-best solution.

Summary of the study: This study clearly indicates that a population-based optimization algorithm requires a *critical* population size, below which there is not enough room to store all necessary building blocks to eventually create the target solution, and beyond which there is unnecessarily more space requiring a large computational effort.

5.4. Revealing dynamics for creating successful solutions

It is clear from the above study that a critical population size is needed for the PILP algorithm to exploit diversity present in the population to consistently create useful solutions. In this section, we evaluate exactly how efficient these operators are for achieving this task.

For this purpose, we consider the same one million version of the problem and run PILP 11 times with a population size of 40. At any iteration t, after the new offspring solutions are created by recombination and repair operators, we count the number of offspring solutions that are better than the previous (at iteration $(t-1)$) population-best and population-average fitness values. Then, we plot the variation of the percentage of these new and improved offspring members from previous population-average and population-best fitness values with iteration (iteration) counter in Figs. 8 and 9, respectively. In both figures, the minimum, first quartile, median (in a thick line), third quartile, and maximum percentage of better offspring solutions are shown with five lines within which all 11 runs lie. Fig. 8 shows that for the entire duration of the runs, our PILP algorithm is able to continuously produce better solutions (near 100%). For the first 10 iterations, almost 100%

of offspring population is better than the previous iteration average solution. Although it is difficult to produce near 100% solutions in all iterations, it is astonishing that at least 70% offspring population members are better than before in any iteration of any run. Fig. 9 shows the proportion of offspring population members which are better than the previous-iteration best solution. The median performance of population-average and best fitness values are also plotted. The first four iterations produce 100% offspring better than the best solution found before. It is relatively easier to find better than random solutions by our proposed PILP operators initially. Until about 10 iterations (we call these initial iterations as the *galloping* phase), a large portion of the offspring population is better than previous population-best solutions for a median run. By this time, the penalty for infeasible solutions have reduced from $1.4(10^6)$ to about $3(10^4)$. From here on, the algorithm finds it difficult to continuously produce better offspring than the previous-best solutions. This is because during these critical intermediate iterations, the algorithm passes through a stage where there exist many solutions of similar objective value and while there are few offspring solutions better than previous-best solutions found, other offspring solutions are also not far behind. This consolidation of good solutions during the intermediate phase (we call this as the *consolidation* phase) allows our PILP to exploit the multitude of near target solutions to be discovered so that in the final phase (we call it the *culminating* phase), a rapid convergence to the target solution is achieved. We observe these three distinct phases in all our simulation runs. By comparing with Fig. 6, it is worth noticing that the transition in convergence rate takes place at the boundary of consolidation and culminating phases.

Summary of the study: The proposed PILP algorithm with all its customized operators is able to consistently find new and improved solutions with generations from start to finish. There are three phases of working of the PILP algorithm. In the *initial galloping phase*, infeasible solutions approach feasible region at a fast rate. In the *intermediate consolidation phase*, the algorithm accumulates adequate salient sub-parts (or building blocks) on various parts of the variable vector. Eventually, in the *final culminating phase*, the algorithm is able to recombine the building blocks from various parts of the problem length quickly to construct the target solution.

5.5. Effect of recombination operator

Demonstrating that the customized recombination and repair operators produce increasingly better solutions with iterations – a prime reason for having a computationally fast approach, we now investigate the effect of recombination operator alone, possibly in improving the performance further with multiple parents in the recombination process.

We use the same one million version of the problem and set up a number of experiments as follows. First, we switch off the recombination operator completely and re-apply on our PILP approach with the two repair operators alone. We call this the one-parent simulation, as with one parent no recombination is possible. Then, we use ρ ($\geq$2) parents to create a new solution by the recombination operator. To achieve this, we copy the best of ρ parents heat-wise and construct a new solution. The generated new solution is then repaired (with two repair operators) as usual in an effort of making it feasible. It is anticipated that as the parent size increases, better heat-wise schedules may be found and used to construct the offspring solution. But the flip side of increasing the parent size is the danger of losing diversity in the population, thereby sacrificing on the overall performance of the algorithm.

A population of size 60 is used in all simulations of this study. Other parameters are identical to those used in the previous sections.

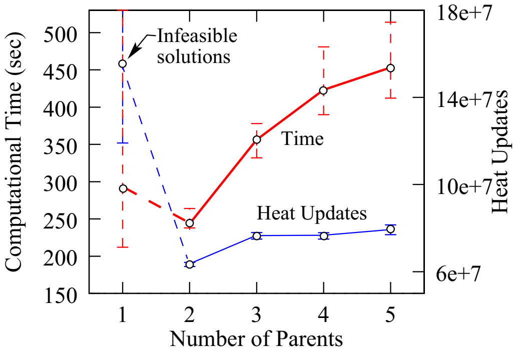


Fig. 10. Effect of parent size on the computational time and heat-updates for the casting scheduling problem.

Fig. 10 shows the results of our PILP with up to a five-parent recombination. As mentioned above, the parent-size of one signifies no recombination effect, meaning that only repair operators are solely responsible for creating offspring solutions. It is observed from the figure that the PILP algorithm without the recombination operator is unable to find any feasible solution in any of the 11 runs. PILP algorithm gets terminated when the maximum stipulated iterations are completed. The values marked for $\rho = 1$ is for the best, median and worst objective values (albeit infeasible solutions) of all 11 runs. The dashed line indicates that the solutions are infeasible. When the parent size is two, PILP algorithm finds the feasible and target solution in all 11 runs. Interestingly, when the parent is increased from two, the algorithm is still able to find the target solution, but at the expense of more computational time and heat-updates. For this particular problem and chosen population size, it is clear that PILP works the best with the two-parent recombination operator. A larger parent size reduces diversity of the population and ends up taking more computational time in finding the target solution.

Summary of the study: This study clearly portrays that the customized recombination operator is the *key* search operator for successfully solving the problem.

5.6. Effect of problem size: the scale-up study

This section provides the most intriguing results of this paper. We evaluate the performance of the PILP algorithm on a scale-up study on problem size, in which variables span from 50,000 to *one billion*. For every problem, the desired accuracy is fixed at 99.7%, meaning that as soon as a feasible solution with a metal utilization ratio of 99.7% is obtained, the algorithm is terminated and the overall CPU time and number of heat updates from the start to termination is recorded. If no such solution is found in a maximum of 200 iterations, the run is considered unsuccessful. Table 5 shows the parameters of the casting scheduling problem for different problem sizes. The population size is kept fixed to 60 for all problems. Table 6 shows the statistics of the obtained results using the PILP algorithm. The second and third column show the average and standard deviation of computational time for multiple runs of PILP from different initial populations. The next two columns show the same for total number of heat updates (HU), indicating the total number of variable manipulations performed by the PILP algorithm from start to completion. The next two columns indicate the average number of heat updates (heat updates divided by $N \times H$). It is interesting to note that although heat updates increase with the problem size, the average heat updates remain more or less the same at around 1000. The final two columns show the average and standard deviation of total number of solution evaluations. Since a

Table 5
Casting scheduling problem parameters for the scale-up study.

Obj. No.	1	2	3	4	5
Wt.(kilograms)	79	66	31	26	44
#var	Number of copies				
50k	6240	6262	6217	6267	6262
100k	12,560	12,562	12,517	12,567	12,562
500k	62,000	62,545	62,517	62,567	62,562
1M	125,600	125,620	125,170	125,670	125,620
5M	620,025	625,450	625,170	625,670	625,620
10M	1,255,980	1,256,200	1,251,700	1,256,700	1,256,200
50M	6,200,270	6,254,500	6,251,700	6,256,700	6,256,200
100M	12,559,745	12,562,000	12,517,000	12,567,000	12,562,000
500M	61,649,750	61,706,480	61,609,500	61,752,675	61,654,900
1G	123,649,750	122,647,195	123,609,500	123,752,675	123,654,900
Obj. No.	6	7	8	9	10
Wt.(kilograms)	35	88	9	57	22
#var	Number of copies				
50k	6172	6076	6052	6017	6012
100k	12,172	12,076	12,052	12,017	12,012
500k	62,172	60,576	60,552	60,517	60,512
1M	121,720	120,760	120,520	120,170	120,120
5M	621,720	605,760	605,520	605,170	605,120
10M	1,217,200	1,207,600	1,205,200	1,201,700	1,201,200
50M	6,217,200	6,057,600	6,055,200	6,051,700	6,051,200
100M	12,172,000	12,076,000	12,052,000	12,017,000	12,012,000
500M	61,680,400	61,621,160	61,621,600	61,621,600	61,652,160
1G	123,680,400	122,621,160	123,621,600	123,621,600	123,652,160

Table 6
PILP results on scale-up study.

Problem	Time (seconds)		Heat update		Avg HU		Soln. Eval.	
Size	Avg.	SD	Avg.	SD	Avg.	SD	Avg.	SD
50,000	7	0.2	4,464,856	100,699	95	2	1020	52
100,000	26	0.6	8,807,564	167,494	94	2	1032	17
500,000	143	4	46,064,337	1,184,570	98	2	1128	35
1,000,000	308	9	91,345,801	2,048,330	97	2	1080	35
5,000,000	1749	53	459,919,887	12,615,715	98	3	1092	35
10,000,000	4207	124	976,903,439	19,038,115	100	2	1104	35
50,000,000	24,000	1697	4,561,785,823	87,843,193	97	2	1056	17
100,000,000	47,593	325	8,945,635,983	62,156,023	96	1	1040	17
500,000,000	261,951	8742	48,364,776,584	988,448,176	103	2	1280	52
1,000,000,000	535,503	11,073	96,229,010,019	1,410,837,470	102	1	1260	35

Table 7
Statistics of 25 runs with different desired accuracy. Only 21 runs are successful for 99.9% case.

Desired	Max.	Number of	Succ.	Comput. Time (seconds)			# Heat-Updts. ($\times$ 10^6)		
Accuracy	loss (kilograms)	Variables	Runs	Min.	Median	Max.	Min.	Median	Max.
95%	32.50	916,800	25	68	70	74	17.31	17.57	17.72
97%	19.50	897,900	25	88	89	93	27.40	27.62	28.10
99%	6.50	879,760	25	163	165	177	53.42	53.70	54.71
99.7%	1.95	873,580	25	346	408	447	79.49	80.11	80.68
99.9%	0.65	871,840	21	1,237	2070	4426	149.35	154.64	170.00

population of size 60 is used for all problems, this means that all problems require between 17 to 21 iterations to find the desired solution. This validates our complexity computation of $O(nNH)$ per iteration performed in Section 3.7. Since n is identical to all problems and number of iterations is more or less same, the average heat update $(\mathrm{HU}/(N \times H))$ is almost identical for all problems. We discuss further the complexity issue in Fig. 12.

Fig. 11 shows a remarkable plot. The x-axis marks the problem size, whereas the y-axis shows the computational time in seconds. The same computer is used stand-alone (without any time-sharing with any other tasks) for these runs. The best, average and worst computational time for 5 runs, each starting from a different initial population, are shown in the figure. For the 100-million, 500-million, and one-billion variable problems, three runs are performed, due to multiple day requirement for this astoundingly large dimensional search spaces. Both axes are shown in logarithmic scale. Since the resulting relationship is almost linear with a slope of 1.11, this means a polynomial time $(= 5.34(10^{-5})|\mathbf{x}|^{1.11})$ increase of CPU time with an increase in number of variables $(|\mathbf{x}|)$. A 10% increase in number of variables requires about 11.16% increase in computational time. This complexity is close to linear and is much smaller than quadratic. Moreover, the variation of computational time in 5 runs is small in all cases as seen by very close horizontal bounds around the circles, thereby indicating a reliable performance of PILP on variables spanning over more than four orders of magnitude. The right vertical axis marks the respective CPU time in actual seconds, minutes, hours and days, to have a better comprehension of the time used to solve the problems. The 50,000

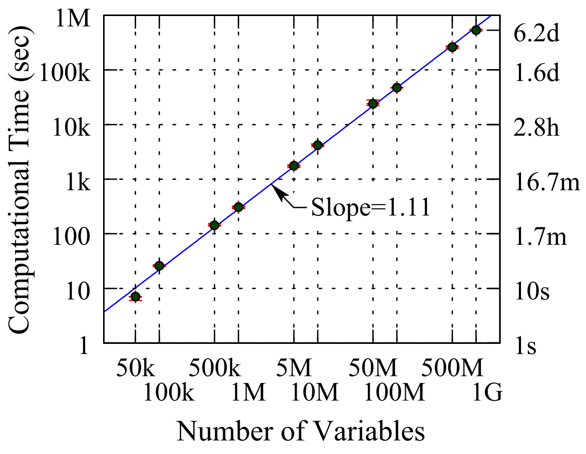


Fig. 11. Effect of problem size on the computational time for solving the casting scheduling problem.

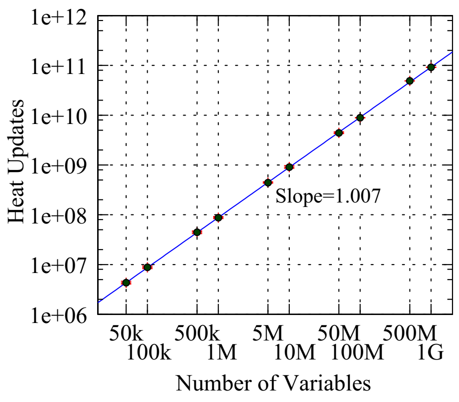


Fig. 12. Effect of problem size on the number of heat-updates for solving the casting scheduling problem.

variable problem takes about 7 seconds and one billion variable problem takes about 6.2 days, on an average. The previous study (Deb et al., 2003) used a different implementation to solve the same problem up to one million variables. The complexity of the previous algorithm was found to be $O(|\mathbf{x}|^{1.79})$. The new implementation with efficient data management methods and coding procedures produce a complexity of $O(|\mathbf{x}|^{1.11})$ which makes it more computationally efficient and allows us to extend the methodology to be applied to a billion variable problem.

We also record the number of heat-updates (the total number of variable manipulations) that a simulation run considered before finding the final desired solution from start to end of the runs. The best, average, and worst number of heat-updates are plotted with the problem size in Fig. 12.

A log-log plot shows an almost linear relation between the number of heat-updates and problem size, thereby indicating a polynomial ($= 83.2|\mathbf{x}|^{1.007}$) complexity. This is very close to a linear complexity with number of variables ($|\mathbf{x}| = N \times H$), meaning that with our proposed algorithm, an increase in one variable requires only about 83 additional variable manipulations to find the respective target solution. Since an identical population size (of 60) is used for all problem sizes, the overall heat update varies almost linear to $N \times H$, which agrees with our theoretical calculation executed in Section 3.7.

Summary of the study: The proposed PILP algorithm finds a near-optimal solution in a polynomial (sub-quadratic) time complexity for a huge range of variable space spanning five orders of magnitude and solving a billion-variable problem. The simulation results also agree with our theoretical estimate for the number of variable changes to be linearly proportional to the problem size.

5.6.1. *Implications of solving a billion variable problem*

Solving a billion variable problem requires a high-performing computer, the implications of which we discuss here. To store one solution having 1G integer variables requires 1 Gigabyte of memory itself (integer options from 0 to 255). With a population of size 60, this means a storage of 60 GB. Since the proposed PILP requires two populations (parent and offspring) to be stored at every iteration, this requires at least 120 Gigabytes of RAM in a computer to store the populations themselves. For this study, we have procured a desktop computer with 256 Gigabytes DDR4 RAM. We observe from a snap-shot of memory usage during a run that in most part of the simulation 129 Gigabytes of 256 Gigabytes are used by PILP code. This is consistent with our above rough calculation and indicates that the population approach of the PILP algorithm demands a higher memory capacity as a flip side of its operation, but the ability of PILP method to solve the specific ILP problems demonstrated in this study and the low-cost availability of RAM to date outweigh and justify the use of a population-based approach for solving the ILP problem efficiently.

Our PILP algorithm is able to remarkably and efficiently work on an astronomically large search space. Although 16 values (integer values from 0 to 15) are used for each variable, considering even 10 different integer options, there are 10^{10^9} possible solutions in the search space, of which a very tiny fraction (almost $1/10^{10^9}$) of solutions (with only 1260 solution evaluations requiring a total of 96 billion variable changes, on an average) were visited by our PILP algorithm to find the target solution for the billion-variable problem. This is a remarkable achievement by any account.

One can argue that such a large problem may not be of use in practice at this point, but recall that the PILP algorithm has also outperformed two popular softwares – `CPLEX` and `glpk` – on small-sized problems (310 to 2000-variable) and is able to solve consistently on very large-sized ILP problems. For the first time, we are able to break the billion-variable barrier on an optimization problem that is conceived from a real-world context with a population-based algorithm that is inherently parallel and computationally fast.

5.7. *Effect of desired accuracy*

Having shown the scalability of PILP algorithm to large-sized problems, let us now investigate the sensitivity of PILP on a few other parameters of the casting scheduling problems.

In this section, we consider the effect of the desired accuracy (η) for metal utilization on the computational time for solving the casting scheduling problem. In all the above problems, we have used 99.7% target on metal utilization. In this section, we consider five different target values: 95%, 97%, 99%, 99.7%, and 99.9%. To simplify the problem, we have used a single crucible of 650 kilograms capacity for all heats. For all cases, an identical number of objects, similar demand for each object, and the same weight of each object as in Table 3 are used here. Since the number of heats depends on the desired accuracy in the solution, the number of variables are different for different accuracy. Table 7 presents the required number of variables for each accuracy level. When a feasible solution having an objective value of η is obtained, a run is considered successful and the run is terminated. As evident from the table, the problems have roughly one million variables. The best, median and worst computational time and number of heat-updates are plotted versus desired target in Fig. 13. Since the y-axis is in logarithmic scale and x-axis is in linear scale, and the nature of variations of both quantities is worse than linear, the increase in

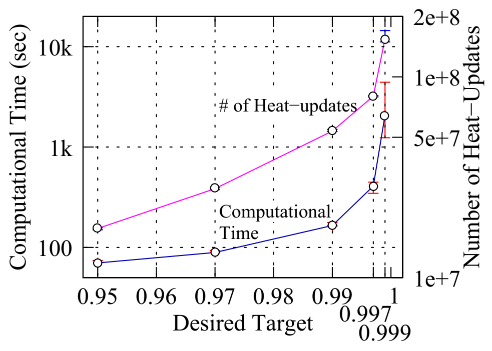


Fig. 13. Effect of desired accuracy on the computational time for solving the casting scheduling problem.

computational time with the desired accuracy is more than exponential, meaning that as the desired accuracy is chosen to be close to the true optimum, the problem gets more than exponentially difficult to solve. This is because with an increase in desired accuracy, there exists increasingly fewer number of better solutions in the search space. Hence, the algorithm spends increasingly more time to find the desired solution. When we set the desired accuracy to 99.9%, 21 out of 25 runs are able to find a feasible solution and also the variations of the time and heat-updates are large.

The above phenomenon raises an important aspect about the accuracy and computational time. If an approximate solution is satisfactory, a considerable gain in computational time can be achieved in solving large-scale problems. The maximum loss in metal utilization (shown in the second column in the table) shows that if at most 1.95 kilograms deviation compared to 0.65 kilogram deviation in a 650 kilograms crucible from the true optimal metal utilization is acceptable, then a 80% saving in computational time is possible. For all practical purposes, such approximations are usually acceptable and our proposed PILP is a quick way to arrive at such a solution for a large-dimensional problem.

Summary of the study: The ILP problem of this study is exponentially harder to solve with an increased required accuracy in the target solution, thereby revealing the NP-hardness of the ILP problem. However, as demonstrated in the previous section, to achieve a fixed accuracy (such as $\eta = 0.997$, whereas the true optimum lies somewhere close to one), the PILP approach is polynomial to the number of variables.

6. Similarities between casting scheduling problem and other assignment problems

The casting scheduling problem formulated in Eqs. (1)–(4) is equivalent to the following generic integer programming problem arising from different practical problems:

$$\left.\begin{array}{ll}\text{Maximize} & f(\mathbf{x}) = \sum_{i=1}^{H}\sum_{j=1}^{N} a_{ij}x_{ij},\\ \text{Subject to} & \sum_{j=1}^{N} b_{ij}x_{ij} \le c_i, \quad \text{for } i = 1,2,\ldots,H,\\ & \sum_{i=1}^{H} d_{ij}x_{ij} = e_j, \quad \text{for } j = 1,2,\ldots,N,\\ & x_{ij} \ge 0 \text{ and is an integer.} \quad \text{for all } (i,j).\end{array}\right\} \qquad (7)$$

The PILP algorithm proposed in this paper is generic enough to solve above problem to near optimality. Many resource allocation, assignment, and combinatorial optimization problems have objective and constraint functions of the above form. If any or both types of constraints are absent, our PILP algorithm can still be applied. The generalized *multiple* or multidimensional knapsack problem and its relaxations (Kellerer et al., 2004) with N items and H knapsacks is identical to above problem with $d_{ij} = 1$ for every i and j combination. On this account, multiply constrained knapsack problem and multiple subset sum problem (Martello & Toth, 1990) also fall in the same category. Assignment problems (Akgül, Hamacher, & Tufecki, 1992) are also similar to the above formulation with binary variables and cover a large number of problems from practice. With a slight change in the initialization and repair operators, the PILP algorithm can be applied to binary variables as well.

7. Conclusions

In this paper, we have considered a specific resource allocation problem involving a casting scheduling process motivated from a real-world foundry and presented a customized population-based optimization algorithm (PILP) for finding near-optimal solutions in a computationally fast manner. The underlying problem has linear objective and constraint functions, but the difficulty arises from the integer restriction of each decision variable. The other difficulty of the problem is the sheer number of integer variables needed to satisfy industrial need, leading to tens of thousands of variables. The current approach, being computationally faster than our previous implementation (Deb et al., 2003), has also been investigated thoroughly to reveal its working principle, compared with popular existing methods, and applied to 1000 times larger problem sizes.

First, we have used two commonly-used available mixed-integer programming softwares – one commercial (IBM's CPLEX) and another public-domain (Octave's glpk) – to test the ease or difficulty of solving the low-dimensional version of the problem. It has been observed that glpk and CPLEX are not able to solve the problem having 2000 or more variables. Due to the use of branch-and-cut fix-up to handle integer restrictions, these point-based continuous-variable optimization algorithms are not scalable for handling a large number of integer variables.

The proposed PILP algorithm uses a population of solutions in each iteration. Moreover, the initialization and population update methods are customized to exploit the linearity aspect of the problem. The PILP method uses a recombination operator that is considered and found to be the main search power providing the computational speed for reaching near-target search region quickly. The following key conclusions can be made from this extensive simulation study:

1. On the low-dimensional problems involving up to 2000 integer variables, our proposed PILP method is able to find the desired solution in less than a second and with fewer solution evaluations than both glpk and CPLEX softwares repeatedly.
2. On very large-sized problems from 50,000 to one billion variables, our PILP algorithm has shown a polynomial time complexity with almost linear order. This is remarkable considering the large range of problem sizes being considered.
3. Although the casting scheduling problem is exponentially more time-consuming to solve for *optimality* with an increase in number of variables, the proposed PILP algorithm has demonstrated a polynomial time complexity for finding a near-optimal solution.
4. The power of a population-based optimization algorithm lies in its recombination operator, which has a unique ability to recombine partial and good information of two or more population members into one new solution. It has been clearly demonstrated that the PILP algorithm performs well mainly due to its recombination operator.
5. It has also been clearly shown that a critical population size is essential for a population-based optimization algorithm to work well. The diversity of an evolving population gives PILP algorithm's recombination operator its power which a point-based algorithm lacks. Thus, a critical population size and an efficient recombination operator make the overall PILP algorithm supe-

rior to point-based algorithms in handling the very large-scale ILP problem of this paper.

6. For the first time, in this paper, we have overcome the billion-variable barrier in a specific real-world optimization problem-solving tasks and demonstrated finding a near-optimal solution (with 99.7% close to the maximum possible solution) in a sub-quadratic computational complexity.

In addition, we have provided systematic studies in demonstrating the role of each operator of the algorithm, so as to have a clear understanding of the algorithm.

Many other resource allocation and assignment problems have a similar structure to the casting scheduling problem solved in this paper. Thus, we believe our proposed PILP algorithm is equally applicable to many such problems with a small or no change in its structure. Future attempts would be to identify some such problems and develop modified methodologies for solving large-scale version of them as computationally efficiently as demonstrated in this paper.

References

Akgül, M., Hamacher, H. W., & Tufecki, S. (1992). The linear assignment problem. In *Combinatorial optimization* (pp. 85–122). Berlin: Springer Verlag.

Aminbaksh, S., & Sonmez, R. (2016). Discrete particle swarm optimization method for the large scale discrete time-cost trade-off problem. *Expert Systems With Applications, 51*, 177–185.

Bellman, R. E. (1954). The theory of dynamic programming. *Bulletin of American Mathematical Society, 60*(6), 503–515.

Cheng, W. N., Zhang, J., Chung, H. S. H., Zhong, W. L., ad, W.-G. W., & Shi, Y. H. (2010). A novel set-based particle swarm optimization method for discrete optimization problems. *IEEE Transactions on Evolutionary Computation, 14*(3), 278–300.

Dantzig, G. B. (1988). *Linear programming and extensions*. Princeton University Press.

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

Deb, K. (2000). An efficient constraint handling method for genetic algorithms. *Computer Methods in Applied Mechanics and Engineering, 186*(2–4), 311–338.

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

Deb, K., Reddy, A. R., & Singh, G. (2003). Optimal scheduling of casting sequence using genetic algorithms. *Journal of Materials and Manufacturing Processes, 18*(3), 409–432.

Ermon, S., Gomes, C. P., Sabharwal, A., & Selman, B. (2013). Taming the curse of dimensionality: Discreteintegration by hashing and optimization. In *Proceedings of the 30th International Conference on Machine Learning: volume 28* (pp. 334–342). Atlanta, Georgia, USA.

Fletcher, R., & Reeves, C. M. (1964). Function minimization by conjugate gradients. *Computer Journal, 7*, 149–154.

Gay, D. M. (2015). IBM ILOG CPLEX optimization studio: Getting started with CPLEX (12th ed.) IBM.

Goldberg, D. E. (1989). *Genetic algorithms for search, optimization, and machine learning*. Reading, MA: Addison-Wesley.

Goldberg, D. E., Sastry, K., & Llora, X. (2007). Toward routine billion-variable optimization using genetic algorithms. *Complexity, 12*(3), 27–29.

Holland, J. H. (1975). *Adaptation in natural and artificial systems*. Ann Arbor MI: MIT Press.

Huang, L., Gao, Z., & Zhang, D. (2013). Research on multi-fidelity aerodynamic optimization methods. *Chinese Journal of Aeronautics, 26*(2), 279–286.

Iturriaga, S., & Nesmachnow, S. (2012). Solving very large optimization problems (up to one billion variables) with a parallel evolutionary algorithm in CPU and GPU. In *Proceedings of the P2P, parallel, grid, cloud, and internet computing* (pp. 267–272). Piscatway, NJ:: IEEE Press.

Karmarkar, N. (1984). A new polynomial time algorithm for linear programming. *Combinatorica, 4*(4), 373–395.

Kellerer, H., Pferschy, U., & Pisinger, D. (2004). *Knapsack problems*. Springer Verlag.

Kong, M., & Tian, P. (2006). Apply the particle swarm optimization to the multidimensional knapsack problem. In *Proceedings of the artificial intelligence and soft computing conference (ICAISC)* (pp. 1140–1149). Springer.

Land, A. H., & Doig, A. G. (1960). An automatic method of solving discrete programming problems. *Econometrica, 28*(3), 497–520.

Li, C., Zhou, J., Ouyang, S., & Chen, X. D. L. (2014). Improved decomposition coordination and discrete differential dynamic programming for optimization of large-scale hydropower system. *Energy Conversion and Management, 84*, 363–373.

Li, X., & Yao, X. (2012). Cooperatively coevolving particle swarms for large scale optimization. *IEEE Transactions on Evolutionary Computation, 16*(2), 210–224.

Makhorin, A. (2012). GNU linear programming kit. URL https://www.gnu.org/software/glpk.

Martello, S., & Toth, P. (1990). *Knapsack problems: Algorithms and computer implementations*. Wiley.

Michalewicz, Z. (1995). Heuristic methods for evolutionary computation techniques. *Journal of Heuristics, 1*, 177–206.

Mitchell, J. E. (2002). Branch-and-cut algorithms for combinatorial optimization problems. In *Handbook of applied optimization* (pp. 65–77). Oxford University Press.

Molina-Cristbal, A., Palmer, P. R., Skinner, B. A., & Parks, G. T. (2010). Multi-fidelity simulation modelling in optimization of a submarine propulsion system. In *Proceedings of the IEEE vehicle power and propulsion conference (VPPC)* (pp. 1–6). Piscatway:: IEEE Press.

Nau, D. S., Kumar, V., & Kanal, L. (1984). General branch and bound, and its relation to A^* and AO^*. *Artificial Intelligence, 23*(1), 29–58.

Omidvar, M. N., Li, X., Mei, Y., & Yao, X. (2014). Cooperative co-evolution with differential grouping for large scale optimization. *IEEE Transactions on Evolutionary Computation, 18*(3), 378–393.

Reklaitis, G. V., Ravindran, A., & Ragsdell, K. M. (1983). *Engineering optimization methods and applications*. New York: Wiley.

Wang, Z., Zoghi, M., Hutter, F., Matheson, D., & de Freitas, N. (2013). Bayesian optimization in high dimensions via random embeddings. In *Proceedings of the 23rd international joint conference on artificial intelligence* (pp. 1778–1784).

Xu, J., Zhang, S., Huang, E., Chen, C.-H., Lee, L. H., & Celik, N. (2014). Efficient multi-fidelity simulation optimization. In *Proceedings of the 2014 winter simulation conference* (pp. 3940–3951). Piscataway, NJ, USA: IEEE Press.

Yang, B., & Xiao, H. (2009). On large scale evolutionary optimization using simplex-based cooperative coevolution genetic algorithm. In *Proceedings of the International conference on computational intelligence and software engineering* (pp. 1–5). doi:10.1109/CISE.2009.5366863.

Yang, Z., Tang, K., & Yao, X. (2008). Large scale evolutionary optimization using cooperative coevolution. *Information Sciences, 178*(15), 2985–2999.

Zanakis, S. H., & Evans, J. R. (1981). Heuristic "optimization": Why, when and how to use it. *Interfaces, 11*(5), 84–91.