Breaking the Billion-Variable Barrier in Real-World Optimization Using a Customized Evolutionary Algorithm

Kalyanmoy Deb
Computational Optimization and Innovation Laboratory
Dept. of Electrical and Computer Eng.
Michigan State University
East Lansing, MI 48824, USA
kdeb@egr.msu.edu

Christie Myburgh
Principal R&D Engineer
Maptek Pty Ltd
Northbridge, WA 6003, Australia
christie.myburgh@maptek.com.au

ABSTRACT

Despite three decades of intense studies of evolutionary computation (EC), researchers outside the EC community still have a general impression that EC methods are expensive and are not efficient in solving large-scale problems. In this paper, we consider a specific integer linear programming (ILP) problem which, although comes from a specific industry, is similar to many other practical resource allocation and assignment problems. Based on a population based evolutionary optimization framework, we develop a computationally fast method to arrive at a near-optimal solution repeatedly. Two popular softwares (`glpk` and CPLEX) are not able to handle around 300 and 2,000 integer variable version of the problem, respectively, even after running for several hours. 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 that our method scales in a sub-quadratic 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 has ever been handled using any optimization algorithm. The study clearly demonstrates the reasons for such a fast and scale-up application of the proposed method. The work should remain as a successful case study of EC methods for years to come.

Keywords

Evolutionary Computing; Large-scale optimization, Billion-variable study

GECCO '16, July 20-24, 2016, Denver, CO, USA

DOI: http://dx.doi.org/10.1145/2908812.2908952

1. INTRODUCTION

The performance of an optimization algorithm is sensitive to the number of variables [2, 7]. As the number of variables increase, the number of interactions among them increase and an optimization algorithm must have the ability to detect all such interactions to construct a near-optimal solution. Another difficulty arises from the discreteness of the search space, as the continuity and gradient information become unavailable in such problems. Most point-based optimization algorithms [8, 16] handle the latter issue 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 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 still does not show any sign of convergence.

The above-mentioned difficulty with discrete search space is also true for linear programming (LP) problems. Although standard LP problems can solve tens of thousands or even a million-variable problem, but they show vulnerabilities in handling discrete variable problems. 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 teeth 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.

In this paper, we address a particular integer linear programming problem and algorithm for finding a near-optimal solution. Theoretically, the problem falls into the category of a knapsack problem [14], which is known to be weakly NPhard 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 [9, 11, 3] in which more than one solution is used in each iteration to collectively create a new population. Despite a few earlier studies on billion-variable Boolean and unconstrained (onemax) problems [10, 12] and on a real-parameter unconstrained embedded problem [17] in which all but two variables have no effect on the objective function, this study is remarkable from two 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 weakly NP-hard problems, (ii) the study handles, for the first time, a billion variable optimization problem originated from 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 pseudocodes. 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 20,000 variables in practice, they are not able to solve even a 2,000- 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 presents a scale-up study, in which our proposed PILP method successfully handles more than one billion variables and finds a near-optimal solution in a reasonable computational time. The sheer number of variables and the fast computational time recorded 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 of this study are discussed in Section 6.

2. CASTING SCHEDULING PROBLEM

The problem comes from a foundry in which objects of various sizes and numbers are cast by melting metal on a crucible of certain size (say W). Each such melt is called a *heat*. 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* for the heat.

To formulate the casting scheduling optimization problem, we assume that there are a total of N objects are to be made with a demand of exactly r_j (>0, an integer) copies for j-th object. Each object has a fixed weight w_i in kg, thereby requiring a total of $M = \sum_{j=1}^{N} r_j w_j$ kg of metal to make all copies of j-th object. Without loss of generality, let us also assume that W_i kg of metal is melted at i-th 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 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 $H = \left\lceil \frac{M}{\eta W} \right\rceil$.

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 $i = 1,2,\dots,H$ and $j = 1,2,\dots,H$, which represents the number of copies of j-th object to be made from the i-th 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 the i-th 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 $i = 1,2,\dots,H$. Second, exactly r_j copies of j-th object is to be made, not one more or not one less, thereby creating N equality constraints of the type: $\sum_{i=1}^{H} x_{ij} = r_j$ for all $j = 1,2,\dots,N$. We now present the resulting integer linear programming problem, as follows:

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

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

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

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

A little thought will also reveal that the above problem is a multiply constrained bounded knapsack problem [13], which is known to be a weakly 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. Due to the combination of equality and inequality constraints, the problem is non-separable. Dynamic programming (DP) methods [1] with linear complexity is available for solving 0-1 knapsack problem, but no DP methods are available for solving the above non-separable ILP problem.

3. A CUSTOMIZED COMPUTATIONALLY FAST ALGORITHM

Our proposed approach is motivated by an earlier preliminary study [6] in which a customized genetic algorithm with a problem-specific initialization and genetic operators were used. In this paper, we make those implementations computationally more efficient, evaluate the modified algorithm thoroughly, and extend its application to an unprecedented billion-variable problem.

3.1 Customized Initialization

To start a run, n randomly generated solutions are created, but every solution 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. For this purpose, first, a random set of integers within $y_j = [0, a]$ are created and then all N variables are repaired as follows: $y_j \leftarrow y_j \frac{a}{\sum_{i=1}^{N} y_i}$. 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 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 mutation operators which we describe in Section 3.4. The repaired solution is then evaluated.

3.2 Evaluation of Fitness Value

Every population member x is evaluated by adding the objective function value $f(x)$ described in Equation 1 and a penalty value from constraint violation, if any, computed as follows [4]:

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

Since the maximum expected objective value is η, the maximum possible feasible fitness value is also η. A penalty parameter value of $R = 10^3$ is used throughout this study. This value is chosen so that a violation of 1 kg from vessel capacity from maximum possible attainable objective value (one) corresponds to a fitness smaller than chosen target η.

3.3 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 [9]. We consider two parent solutions heat-wise. For each heat (index i), the heat utilization for the heat U_i is compared among all p solutions and then all variables x_{ij} for $j = 1, 2 \ldots, N$ from the largest U_i solution is copied to the offspring solution y. This is repeated for all H heats to construct the new offspring solution. The linearity of problem makes such a recombination operator to create better (than parents) solutions.

3.4 Customized Mutation Operators

Above recombination operator may not satisfy the equality constraints (Equation 3) automatically. We use the first mutation 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 metal utilization of each heat.

Next, the modified solution is sent to the second mutation 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. The heat having the maximum violation in inequality constraint is chosen for fixing. A randomly chosen object (say, objID) with non-zero assignment 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. After all infeasible heats are repaired as above, the feasibility of the overall solution is checked. If the solution is still infeasible, a constraint violation equal to the sum of the normalized violations of all infeasible heats is assigned to the fitness function, as given in Equation 6.

3.5 Overall Time Complexity

The initialization routine requires $O(nNH)$ operations. The recombination operator requires $O(nNH)$ copying operations. The first mutation operator requires $O(NH)$ operations for each new solution, thereby requiring a total of $O(nNH)$ operations. The while loop in the second mutation 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, even for the one billion variable version of the problem.

4. RESULTS USING INTEGER LINEAR PROGRAMMING (MILP)

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 largescale 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 [15]. 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 GB RAM and Windows 8.1 Pro operating system.

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

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

First, we choose the casting scheduling problem having a reasonably small number of variables presented in Table 1. With a single 650 kg crucible used for each heat and an expected metal utilization of $\eta = 99.7\%$, at least $\lceil 20,000/(0.997 \times 650) \rceil$ or 31 heats will be required to melt the requisite amount of metal to make all 200 objects. With 31 heats, a maximum heat utilization of $20,000/(31 \times 650)$ or 99.256% is expected. First, we apply the `glpk` routine of Octave software with bounds on each variable as $x_{ij} \in [0, 15]$. 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 may not be possible to achieve in 31 heats.

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 in only 0.05 sec 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.

Our proposed PILP algorithm is applied next. Interestingly, the PILP algorithm is also able to solve the problem, but in only 0.04 sec and requiring only 150 solution evaluations (compared to 249 solution evaluations needed by CPLEX) to achieve an identical metal utilization.

To evaluate further, we scale up the number of orders of each object to 65 copies (except the first object to be made 63 copies). This requires a total of 650 objects to be made requiring 64, 650 kg of molten metal. This needs a total of $\lceil 64,650/(0.997 \times 650) \rceil$ or 100 heats, instead of 31 heats required before. This problem introduces 100×10 or $1{,}000$ variables to the corresponding optimization problem. The maximum possible metal utilization is $64,650/(100 \times 650)$ or 99.462%. Interestingly, the CPLEX software is also able to find a feasible solution with 99.462% metal utilization in only 0.13 sec and using only 947 solution evaluations, on an average over 10 runs, which is a remarkable result. Our proposed PILP algorithm, with 20 population members, is also able to repeatedly find the same metal utilization (99.462%) solution with a much smaller computational time (0.05 sec) 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 kg of molten metal. With a crucible size of 650 kg per heat, this requires at least $\lceil 129,475/(0.997 \times 650) \rceil$ or 200 heats. Thus, the number of variables to the resulting optimization problem increases to 2,000 and the maximum possible metal utilization is $129,475/(200 \times 650)$ or 99.596%. This time, the CPLEX software is not able to find a feasible solution in any of the 10 runs (each simulation was run for an hour), even when one of the runs was extended up to 15 hours on the abovementioned computer. A particular CPLEX run formulated 88,882,899 branches having 10,382 unsolved branches after $1.65(10^8)$ iterations requiring 4,594 sec (1 hour 16 min) of computational time, and still it could not find any feasible solution. Our PILP algorithm, with only 20 population members, is able to solve the same problem in 0.19 sec requiring only 224 solution evaluations, on an average, in 10 runs. A typical foundry may plan for a casting scheduling for a month, requiring about $H = 3,200$ heats and for about $N = 10$ objects. This makes a total number of variables

to be as high as 32, 000. Clearly, the current state-of-the-art softwares (such as CPLEX) 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 for handling discrete variables is not efficient – there are simply too many branches for an algorithm to negotiate in a reasonable amount of time or iterations. As demonstrated here, population-based optimization algorithms have a greater 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. All simulations of this section are run on a 2×Intel 8-Core Xeon-2640V3 2.66 GHz, 20 MB Cache, 8GT/sec, 16 threads, LGA 2011 computer having ASUS Z10PE-D16/4L Server Motherboard with 16×16GB DDR4 ECC DIMM 2133 Mhz RAM having a 1×Galaxy GTX 980 SOC Nvidia graphics card with Windows 8.1 Pro 64Bit OEM. This computer was specially procured for performing very large-sized problems reported in this study.

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 2 shows the problem parameters used for this purpose. Total number of objects is 5,50,666 and total metal required to cast them is 56,352,140 kg. To bring the problem close to practice, two crucibles of sizes 650 kg and 500 kg are used on alternate days with 10 and 13 heats on respective days. A total of $H = 100,000$ is needed. Since there are $N = 10$ objects, the total number of variables in the optimization problem is 1,000,000 (one million).

Table 2: Casting scheduling problem parameters used as default in most of this study.

No.	1	2	3	4	5	6	7	8	9	10
Wt. (kg)	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 kg)	10.364	8.458	3.466	2.928	5.076	4.014	11.119	1.046	7.279	2.601

To investigate the ease of creating a feasible solution at random for the above one million variable problem, first, we create 10,000 solutions at random. It is observed that none of them is feasible. The best, median and worst fitness values of these random solutions are found to be $-7,029,705$, $-7,219,503$, 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 is always positive and has a maximum value of $\eta = 0.997$ (for a target of 99.7% target utilization).

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

5.2 A Typical Simulation

We now apply our proposed 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. Figure 1 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 optimized solution to reach a value of $(1 - 0.997)$ or 0.003. Since all infeasible solutions have negative values, a y-axis value larger than one indicates that the corresponding solution is infeasible. In other words, all solutions having a y-axis value equal to or less than one are feasible. Since the best fitness value can be 0.997 (expected target metal utilization), the best expected has a y-axis value of $(1 - 0.997)$ or 0.003. An almost linear drop in best and average y-axis values with iteration indicate the fitness values approach the desired target in an *exponentially* fast manner with iteration counter. When solutions come closer to the feasible region (at around iteration 15), the convergence is even faster. It can be seen from the figure that the algorithm takes a minimum of 16 iterations to find a feasible solution in any of the 11 runs. Different runs find the target solution (with a transformed fitness value of 0.003) within 16 to 20 iterations.

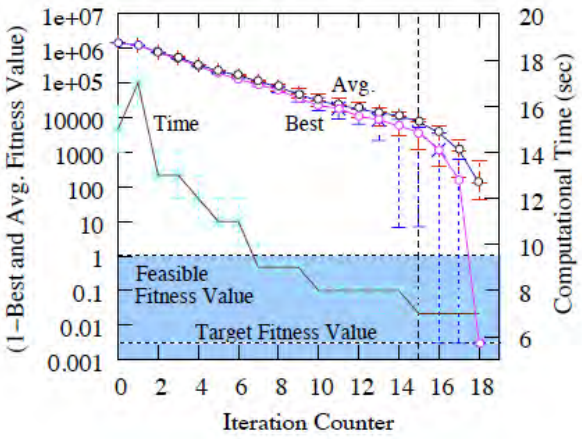

Figure 1: Iteration-wise variation of population-average, population-best fitness value and computational time for one million version of the problem.

The right-side y-axis marks the computational time for each iteration in seconds. The plot with its run-wise variations, 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 with volume requirement get satisfied, thereby reducing the time needed to execute the second mutation operator.

The combined effect of tournament selection operator to choose better parent solutions, recombination operators to combine good parts of two parents into one offspring, and mutation operators to repair offspring solutions is able to find increasingly better and better solutions with iterations and locate the desired target solution in only 16 to 18 iterations. This shows a typical working of our proposed PILP methodology on a million-variable version problem. Such a large-sized optimization problem has been rarely attempted to be solved in practice and with so few solution evaluations and with so less computational time.

5.3 Estimating an Appropriate Population Size

An important parameter in a population-based optimization method is the size of the population. 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. Table 3 tabulates the best, median and worst objective value and number of solution evaluations for different population sizes over 11 runs. It is clear that a population of size 34 produces the best median performance over 11 runs. Figure 2 shows the best, average and worst fitness value obtained for each population size. 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 [5].

Table 3: Fitness value and the number of solution evaluations are tabulated against different population sizes for 1M version. Results for the smallest median solution evaluations are marked in bold.

Pop. Size	Fitness Value			Solution Evaluations		
	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	2,079	2,100
28	0.997	0.997	0.997	551	928	2,639
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	1,053	1,053	1,134

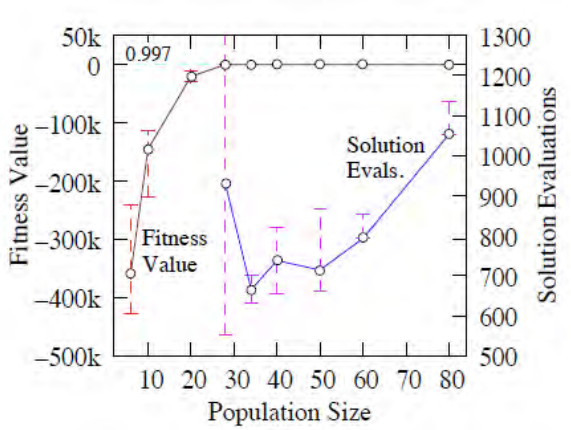

Figure 2: Effect of population size on the performance on the casting scheduling problem

5.4 Effect of Recombination and Mutation Operators

Next, we consider the same one million version of the problem and study the sensitivity of each PILP operator. At any iteration t, after the new offspring solutions are created by recombination and mutation 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 fitness values with iteration counter in Figure 3. In the figure, the minimum, first quartile, (in a thick line), third quartile, and maximum percentage of better offspring solutions are shown with five lines within which all 11 runs lie. The figure shows that from 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. 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, but for brevity we do not show other results.

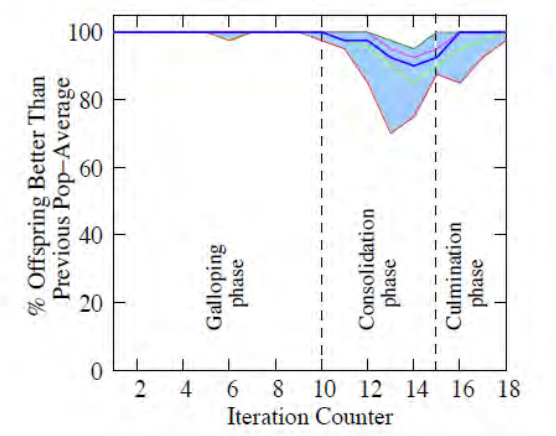

Figure 3: Percentage of offspring population having better fitness value than previous iteration population-average solution.

5.5 Effect of Problem Size: The Scale-up Study

This subsection 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 a staggering and unprecedented *one billion*. All problems are run 5 times, except 100M variables or more which are run 3 times, to complete simulations in a reasonable time. For every problem, the desired accuracy is fixed at 99.7%, meaning that as soon as a feasible solution with a metal utilization 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.

The population size is kept fixed to 60 for all problems. Table 4 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 1,000. The final two columns show the average and standard deviation of total number of solution evaluations. Since a 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 3.5. 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 about the complexity issue in Figure 5.

Table 4: PILP results on scale-up study.

Problem Size	Time (s)		Heat Update		Avg HU		Soln. Eval.	
	Avg.	SD	Avg.	SD	Avg.	SD	Avg.	SD
50,000	7	0.2	4,464,856	100,699	953	20	1,020	52
100,000	26	0.6	8,807,564	167,494	941	17	1,032	17
500,000	143	4	46,064,337	1,184,570	981	24	1,128	35
1,000,000	308	9	91,345,801	2,048,330	973	20	1,080	35
5,000,000	1,749	53	459,919,887	12,615,715	980	25	1,092	35
10,000,000	4,207	124	476,903,439	19,038,115	1,000	19	1,104	35
50,000,000	24,000	1,697	4,561,785,823	87,843,193	972	17	1,056	17
100,000,000	47,593	325	8,945,635,983	62,156,023	955	6	1,040	17
500,000,000	261,951	8,742	48,364,776,587	988,448,176	1,027	19	1,280	52
1,000,000,000	535,503	11,073	96,229,010,019	1,410,837,470	1,022	14	1,260	35

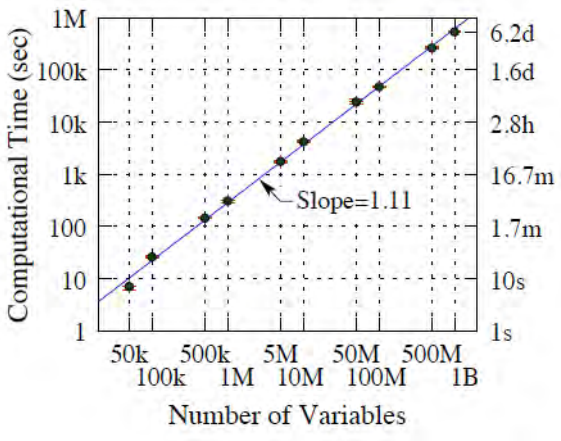

Figure 4: Effect of problem size on the computational time for solving the casting scheduling problem.

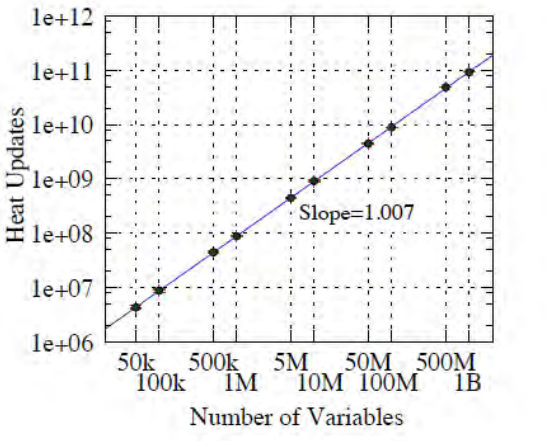

Figure 5: Effect of problem size on the number of heat-updates for solving the casting scheduling problem.

Figure 4 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 $(= 5.34(10^{-5})|\boldsymbol{x}|^{1.11})$ increase of CPU time with an increase in number of variables $(|\boldsymbol{x}|)$. 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 variable problem takes about 7 seconds and one billion variable problem takes about 6.2 days, on an average.

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 Figure 5. A log-log plot shows an almost linear relation between the number of heat-updates and problem size, thereby indicating a polynomial $(= 80.1|\boldsymbol{x}|^{1.007})$ complexity. This is very close to a linear complexity with number of variables $(N \times H)$, meaning that with our proposed algorithm, an increase in one variable requires only about 80 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 was also arrived at in Section 3.5.

5.5.1 Implications of Handling Billion Variables

Solving a billion variable problem requires a high-performing computer, the implications of which we discuss here. To store one solution having 1B integer variables requires 1 GByte of memory itself (integer options from 0 to 255). With a population of size 60, this means a storage of 60 GBytes. Since the proposed PILP requires two populations (parent and offspring) to be stored at every iteration, this requires at least 120 GBytes of RAM in a computer to store both populations. For this study, we have procured a desktop computer with 256 GB DDR4 RAM. We observe from a snap-shot of memory usage during a run that in most part of the simulation 129 GB of 256 GB 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 1,260 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.

6. CONCLUSIONS

In this paper, we have considered a casting scheduling problem motivated from a real-world foundry and developed a customized optimization algorithm (PILP) for finding near-optimal solutions in a computationally fast manner. The difficulty of the problem is the sheer number of integer variables, leading to tens of thousands of variables. It has been observed that a public-domain `glpk` software was able to solve only up to about 300-variable version of the problem in any reasonable amount of computational time, and a commercial CPLEX software from IBM was not able to solve 2,000-variable version of the problem. Due to the use of branch-and-cut methods 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 small population of solutions in each iteration. 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 operator providing the computational speed of reaching near-optimal search region quickly. On very large-sized problems from 50,000 to record-setting one billion variables, our PILP algorithm has shown a polynomial time complexity with an almost linear order. This is remarkable considering the wide range of problem sizes being considered. 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.

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

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 applicable to many such problems with a small or no change in its structure. Our future attempts would be to identify other such problems and develop modified methodologies for solving large-scale version of them as computationally efficiently as demonstrated in this paper.

7. REFERENCES

[1] R. Andonov, V. Poirriez, and S. Rajopadhye. Unbounded knapsack problem: Dynamic programming revisited. *Euro. J. of Operational Research*, 123(2):168–181, 2000.

[2] R. E. Bellman. The theory of dynamic programming. *Bull. of Am. Mathematical Soc.*, 60(6):503–515, 1954.

[3] K. Deb. *Optimization for Engineering Design: Algorithms and Examples.* Delhi: Prentice-Hall, 1995.

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

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

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

[7] S. Ermon, C. Gomes, A. Sabharwal, and B. Selman. Taming the curse of dimensionality: Discrete integration by hashing and optimization. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, 2013.

[8] R. Fletcher and C. M. Reeves. Function minimization by conjugate gradients. *Computer Journal*, 7:149–154, 1964.

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

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

[11] J. H. Holland. *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI: MIT Press, 1975.

[12] S. Iturriaga and S. Nesmachnow. Solving very large optimization problems (up to one billion variables) with a parallel evolutionary algorithm in CPU and GPU. In *Proc. of P2P, Parallel, Grid, Cloud, and Internet Computing (3PGCIC-2012)*, pages 267–272. IEEE Press, 2012.

[13] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springer Verlag, 2004.

[14] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. Wiley, 1990.

[15] J. E. Mitchell. Branch-and-cut algorithms for combinatorial optimization problems. In *Handbook of Applied Optimization*, pages 65–77. 2002.

[16] G. V. Reklaitis, A. Ravindran, and K. M. Ragsdell. *Engn. Optimization Methods and Applications*. Wiley, 1983.

[17] Z. Wang, M. Zoghi, F. Hutter, D. Matheson, and N. de Freitas. Bayesian optimization in high dimensions via random embeddings. In *Proc. of 23rd Intl. Joint Conf. on Artificial Intelligence*, pages 1778–1784, 2013.