Experimental study of seeding in genetic algorithms with non-binary genetic representation

Sadegh Mirshekarian[1], Gürsel A. Süer[2]

[1] Department of Industrial and Systems Engineering,
Ohio University, Stocker Center 285, Athens OH 45701, USA.
Email: sm774113@ohio.edu
(corresponding author)

[2] Department of Industrial and Systems Engineering,
Ohio University, Athens OH, USA
Email: suer@ohio.edu

Abstract Seeding is a technique used to leverage population diversity in genetic algorithms. This paper presents a quick survey of different seeding approaches, and evaluates one of the promising ones called the Seeding Genetic Algorithm. The Seeding GA does not include mutation, and it has been shown to work well on some GA-hard problems with binary representation, such as the Hierarchical If-and-Only-If or Deceptive Trap. This paper investigates the effectiveness of the Seeding GA on two problems with more complex non-binary representations: capacitated lot-sizing and single-machine scheduling. The results show, with statistical significance, that the new GA is consistently outperformed by the conventional GA, and that not including mutation is the main reason why. A detailed analysis of the results is presented and suggestions are made to enhance and improve the method.

Keywords Genetic Algorithm, Population Seeding, Single-machine Scheduling, Capacitated Lot-sizing, Genetic Operators

Introduction

Genetic algorithms (Holland, 1975) rely on crossover and mutation as their main operators—in conjunction with other operators like mating and selection—to improve population quality over time. Numerous attempts have been made to understand the underlying mechanism of these operators and how they are related to each other and to problem characteristics and GA performance. The building-block hypothesis (Forrest and Mitchell, 1993; Goldberg, 1989) is one such attempt. According to Goldberg, "Just as a child creates magnificent fortresses through the arrangement of simple blocks of wood, so does a genetic algorithm seek near optimal performance through the juxtaposition of short, low-order, high-performance schemata, or building-blocks" (p. 41). From this viewpoint, crossover can be seen as being responsible for *combination* of these building-blocks, with mutation as a way of *discovering* those building-blocks that are not already present in the population.

While the building-block hypothesis is widely accepted as the explanation of how GA works, the efficacy of crossover and mutation as operators for combination and discovery has been under scrutiny. Forrest and Mitchell (1993) discuss the undesirable tendency of low quality genetic information to be carried along with good building-blocks. Additionally, crossover seems to be effective only when high quality building-blocks are already present in the population (Wu et al., 1997). This requires a high level of building-block diversity in the population, which is difficult to achieve and difficult to maintain. Jones (1995) suggests a framework for assessing the usefulness of crossover, and concludes that in the absence of well-defined building-blocks, crossover may even have detrimental effects on GA performance. These results are in accordance with what Eshelman and Schaffer (1993)

call only a small "niche" of problems for which crossover gives GA a competitive advantage.

On the other hand, building-blocks that are destroyed or non-existing can be discovered by a discovery operator. Wu et al. (1997) studied the ability of both mutation and crossover for discovery and concluded that mutation is usually superior. However, any operator that performs discovery by randomly changing gene values, can simultaneously be disruptive to existing building-blocks. At an extreme, the mutation rate of 1.0 (equivalent to random search) is the best at discovery while it is the most disruptive. So disruptive in fact, that it tends to destroy 99% of the blocks it discovers (Skinner and Riddle, 2007; Skinner 2009).

Seeding is one of the techniques that researchers use to provide crossover with high-quality low-level building-blocks. It can be done in many different ways, depending on the stage of the algorithm at which it is performed, the method used to generate the seeds and whether the seeds are fixed or dynamically change. In term of timing, seeding can be performed either when generating the initial population, to increase initial population quality and improve convergence speed or to guide the algorithm towards the optimum and improve overall performance, or during execution, to maintain a constant supply of high-quality building-blocks. As for the seeds, they can be generated

a. totally randomly
b. by sampling a randomly generated pool
c. by sampling a pool of optimal (or best known) solutions of previously solved cases (also called case-based seeding)
d. by sampling a pool of suboptimal solutions generated by other heuristics and metaheuristics or by the same algorithm in execution
e. by sampling other populations being run in parallel, or
f. using heuristics specialized to promote some population characteristic like diversity.

Grefenstette (1987) was one of the first to incorporate seeding in GA. In his three experiments on the Travelling Salesman Problem (TSP), seeding was done when generating the initial population. In one of his experiments, the seeds were generated using a special heuristic that promotes diversity by disallowing similar individuals (method *f*), and in the other two by using constructive heuristics that create a pool of suboptimal high-quality solutions (method *d*). Bentley (1990) provides a detailed analysis of TSP, concluding that for this problem the initial population quality affects both the running time and final quality of an algorithm. Ahuja and Orlin (1997) discuss the truth of this conclusion using GA as the algorithm, and provide further evidence for the effectiveness of initial population seeding, at least on TSP.

As for the other methods, Louis et al. (1993) were among the first to investigate the combination of case-based reasoning and Genetic Algorithms, while Ramsy and Grefenstette (1993) were among the first to use a knowledge of previously solved cases for seeding the initial population and seeding during execution (method *c*). The method has since been shown to be effective on problems like Single-Machine Scheduling (Chang et al., 2006), and TSP (Oman and Cunningham, 2001), even though Oman and Cunningham did not achieve good results on Job Shop Scheduling Problems (JSSP). They concluded that whenever there is a direct relationship between offspring and parent fitness, case-based seeding is likely to be effective.

Skinner (2009) investigated the possibility of using seeding during execution, not only to promote population diversity, but also to help crossover in building-block discovery. He argues that since mutation can be disruptive to currently available building-blocks, it is no longer needed and replacing it with a seeding operator is a better option. In his variation of GA, which is called the Seeding Genetic Algorithm (SGA), crossover and seeding are the only operators. To provide crossover with high-quality building-blocks, a pool of highly fit seeds is generated before running the algorithm, and some of them are routinely injected into the population during execution, with some predefined probability and pattern (corresponding to method *b* in the list above). The seed pool is made up of top individuals of a randomly generated pre-sample population. These individuals are relatively more fit than others and so it is assumed that they contain most of the desirable building-blocks within them. As a result, their injection into the population would help maintain a constant supply of useful building-blocks, while creating enough diversity to avoid settling in local optima. Meadows et al. (2013) tested SGA on a range of well-known problems, including the Deceptive Trap (Goldberg, 1987) and the Hierarchical If-And-Only-If (Watson and Pollack, 2000), and concluded that it outperforms conventional GA (CGA) in all of the studied cases.

The problems explored by Meadows et al. (2013) all use binary genetic representation, in which each gene can only take two values, 0 or 1. However, a large portion of

optimization research is on problems that have more complex genes. For example, in permutation representation, each chromosome consists of a sequence of nonrepeating integers (which can represent jobs in a job scheduling problem, or cities in a TSP), resulting in a much higher number of possible chromosome variations and a larger solution space compared to a binary representation of the same length. Since a seeding operator that is based on random search can be hugely affected by this difference in solution space dimensionality, further research was required to assess the effectiveness of SGA when applied on such problems. The research is particularly valuable, because SGA has the potential to be a universal replacement for conventional GA.

This paper applies SGA to two important yet difficult optimization problems in the context of production planning: the Capacitated Lot Sizing Problem (CLSP) and the Single-Machine Scheduling Problem (SMSP). The GAs used for both of these problems have more complex genes than binary and therefore can be good new benchmarks for SGA. In the next section, first a brief review of CLSP is given and then both the utilized conventional and Seeding GA configurations are described. The methods are subsequently applied to eleven different problem cases, and the results are reported and discussed.

SGA and the Capacitated Lot Sizing Problem (CLSP)

Genetic algorithms have been used with success for a variety of lot-sizing problems (Goren et al. 2010), including Capacitated Lot-Sizing (Gicquel et al., 2008). This paper uses the single-resource, single-level CLSP variation with sequence-independent setup times, as formulated by Süer et al. (2008). Given the required amount of each of the *n* products (demand), to manufacture/order at each time period *t*, we want to plan the timing and amount of orders such that those requirements are met in full (without having any shortages at any time period), and the total cost, consisting of labor costs, inventory carrying cost and order cost is minimized.

A sample CLSP case is given in Tables 1 to 3, along with some of the calculations. Refer to Süer et al. (2008) for details of calculating the costs associated with labor[1]. The mathematical formulation is also the same as the one used by Süer et al. (2008), which is given below for convenience. Note that the main difference between the CLSP variation used in this paper and the conventional CLSP is that we allow over-ordering, but apply overtime penalties through overtime and subcontracting costs.

Objective function to minimize:

$$\sum_{i=1}^{n}\sum_{t=1}^{T}\left(A_i X_{i,t} + h_i I_{i,t}\right) + \sum_{t=1}^{T}\left(R_t.rr + OT_t.or + S_t.sr\right)$$

Subject to:

$$I_{i,t} = I_{i,t-1} + q_{i,t} - d_{i,t} \qquad i = 1 \dots n,\ t = 2 \dots T$$

$$I_{i,1} = q_{i,1} - d_{i,1} \qquad i = 1 \dots n$$

$$MX_{i,t} \geq q_{i,t} \qquad i = 1 \dots n,\ t = 1 \dots T$$

$$\sum_{i=1}^{n}(q_{i,t}.p_i + X_{i,t}.s_i) = R_t + OT_t + S_t \qquad t = 1 \dots T$$

$$R_t \leq ULR \qquad t = 1 \dots T$$

$$OT_t \leq ULO \qquad t = 1 \dots T$$

$$S_t \leq ULS \qquad t = 1 \dots T$$

$$X_{i,t} \in (0,1) \qquad i = 1 \dots n,\ t = 1 \dots T$$

$$q_{i,1},\ I_{i,t},\ R_t,\ OT_t,\ S_t \geq 0 \qquad i = 1 \dots n,\ t = 1 \dots T$$

Decision variables:

$X_{i,t}$: binary variable for whether an order is placed for product *i* at time period *t*
$q_{i,t}$: quantity of product *i* to be ordered at time period *t*
$I_{i,t}$: inventory carried of product *i* at end of time period *t*
R_t: regular labor time needed in time period *t*
S_t: sub-contracting time needed in time period *t*
OT_t: overtime labor needed in time period *t*

Parameters:

n: number of products
T: number of time periods
M: a number greater than the largest possible order
A_i: fixed order cost of product *i*
h_i: inventory carrying cost per order unit per time period of product *i*
p_i: processing time of product *i*
s_i: setup time of product *i*
$d_{i,t}$: demand of product *i* to be ordered at time period *t*
ULR: Regular labor time capacity

[1] The "labor cost" here entails regular and overtime labor costs, plus sub-contracting costs.

rr: regular labor cost per unit processing time
ULO: Overtime labor capacity
or: overtime labor cost per unit processing time
ULS: Sub-contracting time capacity
sr: sub-contracting cost per unit processing time

Indices:

i: product
t: time period

To understand the problem structure, it is important to observe the three major constraining drives at play: The drive to place as few orders as possible to reduce order costs and setup times, the drive to keep the amount of orders as close as possible to demand, to reduce inventory carrying costs, and the drive to evenly spread the orders for different products, in a way that the total order placed for all products is not significantly different in different time periods, in order to minimize labor costs. These constraints effectively entwine the decisions corresponding to orders of different products at different time periods, resulting in a high level of decision interdependence. Consequently, this makes CLSP a generally difficult optimization problem, making it suitable to solve with GA.

Table 1 A sample CLS problem and one of its solutions

	t	1	2	3	4	5	6
Product 1	Demand	25	21	21	28	26	24
	Order	**25**	**42**	**0**	**78**	**0**	**0**
Product 2	Demand	24	26	28	24	25	20
	Order	**50**	**0**	**28**	**24**	**45**	**0**

Table 2 Complementary problem data for the sample problem

	A_i	h_i	p_i (min/item)	s_i (hr/order)
Product 1	500	1.5	20	5
Product 2	300	2.0	19	3
ULR: 20 hr ***rr***: 10/hour		***ULO***: 10 hr ***or***: 15/hour		***ULS***: ∞ ***sr***: 20/hour

Table 3 Calculations for the period 1 of the sample problem

Labor for product 1	5 + 25 (20/60) = 13.33
Labor for product 2	3 + 50 (19/60) = 18.83
Total labor time (h)	32.17
Regular time (h)	20
Overtime (h)	10
Sub-contr. time (h)	2.17
Labor cost of Period 1	**20 (10) + 10 (15) + 2.17 (20) = 393.4**
Total order cost	3×500 + 4×300 = **2700**
Total inventory cost	(21 + 26 + 2(24))(1.5) + (26 + 20)(2.0) = **234.5**
Total labor cost	393.4 + 19(10) + 11.87(10) + (20(10) + 10(15) + 11.6(20)) + 17.25(10) = **1456.6**

Many different approaches can be taken to genetically represent CLSP, one of which being the multi-segment chromosome used by Süer et al. (2008), in which a single string of binary genes is divided into segments, each segment representing the order schedule of a single product. This paper uses a different multi-parameter binary genetic representation as illustrated in Figure 1. In this representation, every gene contains the order information of all the products for a given time period. As a result, we will have T genes per chromosome, while encoding the total $T.n$ binary decision variables. A value of 1 for a decision variable $X_{i,t}$ means that an order will be placed at t for the corresponding product i, in the amount equal to the sum of demand at t, and the demand of all of the time periods t' that follow t and for which $X_{i,t'} = 0$. For the example in Figure 1, if the demand for product 1 and time periods 1, 2 and 3 was 10, 20 and 30 respectively, since an order is placed only on $t = 1$, it must cover the demand of subsequent two time periods 2 and 3 as well. This means that the total order placed for product 1 at time period 1 is $q_{i,t} = 60$. Inventory carrying costs will therefore be imposed, because of the extra units carrying over to time periods 2 and 3. Note that an order must always be placed at the first time period of every product.

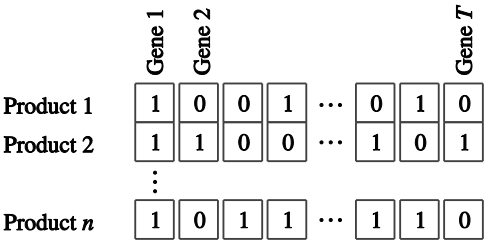

Fig. 1 Each gene contains the order information of every product

We implemented this representation for the conventional GA with crossover and mutation, the Seeding GA with crossover and its seeding operator, and a Hybrid SGA which combined both seeding and mutation (HSGA).

Eleven problem instances were generated randomly[1], each having exactly 120 decision variables, but with varying values of n/T, number of products per number of periods. Since each gene can have 2^n different values, having different values of n helps clarify the effect of gene complexity on SGA performance. The optimal solution of all eleven cases were calculated using CPLEX[2] and the results are also reported for comparison.

GA Configuration:

A simple version of CGA was coded in MATLAB R2013a. GA parameters were chosen based on performance on a few test runs, but we did not run an exhaustive grid search for their optimal values. The configuration used for CGA is described below:

- Population size P=200 and number of generations G=1000 were fixed
- Fully random initialization, with equal probability for *1*'s and *0*'s
- Parents chosen randomly for mating
- Dual-cut crossover with random cut points, with crossover probability $P_c = 0.9$
- Random flip mutation, each gene with an equal probability P_m of being flipped
- Mutation probability P_m linearly changes from 0.05 to 0.0 according to generation number (a simple adaptive twist)
- Top P members of the population, including the offspring, form the next generation

Note that using a simple adaptive mutation rate does not disqualify this genetic algorithm from being a conventional version per se. One of the main features of SGA is the fact that mutation is replaced with seeding. If a simple tweak in mutation can help it outperform the new operator, it will still be the superior operator.

For the Seeding GA, we used the same configuration as above, except that mutation was replaced with a seeding operator:

- A seed pool is formed by selecting the best 200 individuals from a pre-sampling population of 100,000. Different values were tested for both the pre-sampling size and the seed pool size, but these were found to be around optimal. Fixing the seed pool selection was also important in studying the effect of problem complexity on performance.
- At each generation, there is a probability s=0.2 for r=80 individuals from the seed pool to be injected into the population (all at once), by randomly replacing r of the current individuals.

Results:

Table 4 summarizes the best results achieved by each of the algorithms out of a total of 20 replications that were run for each instance and each algorithm. One-way ANOVA was also performed on pairs of (CGA, SGA) and (CGA, HSGA) to test the significance of algorithm effect on performance, and the corresponding P-values are provided in Table 4. A P-value of 0.000 means that the two algorithms have a statistically significant difference in performance, and therefore one of them has significant advantage over the other. In contrast, a P-value greater than 0.05 means that with 95% confidence, the algorithms are performing almost equally.

Figure 2 shows the 95% confidence interval for the means of the 20 outputs generated by each algorithm on instance 1. It is clear that CGA and HSGA do not have a significant difference, while they both differ with SGA. Figure 3 shows the percentage gap between the best solution of each of the algorithms against different problem instances. The horizontal axis of this diagram can be interpreted also as the n/T ratio, because this ratio increases with problem instance.

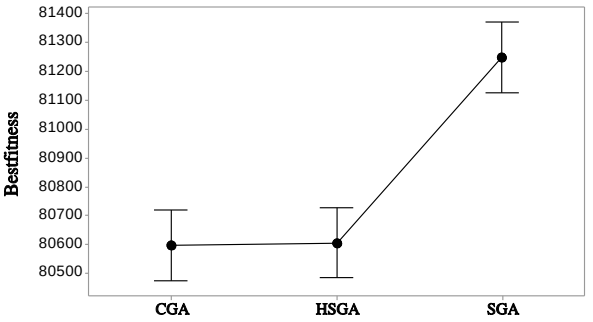

Fig. 2 95% confidence intervals for the mean performance of each algorithm on instance 1

[1] The data sets are available online and can also be provided upon request.

[2] IBM ILOG Optimization Studio Version 12.6 was used, and unless otherwise stated, all runs were on default setting.

Table 4 Best results, their gaps with the optimal and ANOVA P-values for CLSP

	Optimal	CGA	SGA	HSGA	ANOVA P-values
Instance 1 n = 1, T = 120	**80243.7**	80585.8 (0.43%)	80689.1 (0.55%)	80585.8 (0.43%)	CGA, SGA: 0.000 CGA, HSGA: 0.275
Instance 2 n = 2, T = 60	**94545.4**	94560.1 (0.02%)	95007.2 (0.49%)	**94545.4** (0.00%)	CGA, SGA: 0.000 CGA, HSGA: 0.051
Instance 3 n = 3, T = 40	**87107.1**	87229.2 (0.14%)	87946.9 (0.96%)	87351.5 (0.28%)	CGA, SGA: 0.000 CGA, HSGA: 0.008
Instance 4 n = 4, T = 30	**96804.8**	96897.7 (0.10%)	97818.7 (1.05%)	96958.9 (0.16%)	CGA, SGA: 0.000 CGA, HSGA: 0.555
Instance 5 n = 5, T =24	**102509.6**	102583.8 (0.07%)	103599.8 (1.06%)	102599.2 (0.09%)	CGA, SGA: 0.000 CGA, HSGA: 0.208
Instance 6 n = 6, T = 20	**106486.6**	106557.5 (0.09%)	107586.3 (1.03%)	106548.3 (0.06%)	CGA, SGA: 0.000 CGA, HSGA: 0.604
Instance 7 n = 8, T = 15	**97784.2**	**97784.2** (0.00%)	99413.6 (1.67%)	87817.7 (0.03%)	CGA, SGA: 0.000 CGA, HSGA: 0.053
Instance 8 n = 10, T = 12	**104320.7**	104335.0 (0.03%)	106141.7 (1.75%)	104359.1 (0.04%)	CGA, SGA: 0.000 CGA, HSGA: 0.664
Instance 9 n = 12, T = 10	**105885.5**	105914.0 (0.03%)	108125.6 (2.12%)	105914.0 (0.03%)	CGA, SGA: 0.000 CGA, HSGA: 0.268
Instance 10 n = 15, T = 8	**112227.3**	112235.4 (0.01%)	114422.3 (1.96%)	112241.9 (0.01%)	CGA, SGA: 0.000 CGA, HSGA: 0.590
Instance 11 n = 20, T = 6	**104237.6**	**104237.6** (0.00%)	106679.5 (2.34%)	**104237.6** (0.00%)	CGA, SGA: 0.000 CGA, HSGA: 0.957

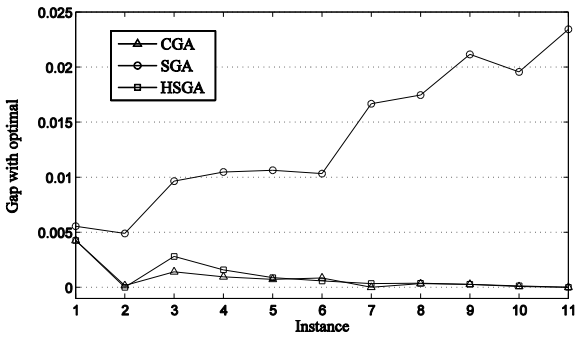

Fig. 3 Percentage gap with the optimal against instance number for each algorithm

Discussion:

CGA results: The conventional GA used in this paper proves to be capable of tackling the problem well, regardless of the different dimensionalities that instances 1 to 11 have. It finds the best solution among the three algorithms in 9 cases, and in two cases it finds the optimal. This is a good indication that the simple combination of mutation and crossover is versatile enough for this problem.

SGA results: As evident from Figure 3, the performance of SGA seems to increasingly suffer as the number of products increases from instance 1 to instance 11. The SGA performs at its best when applied to instance 1, where there is only one product and the chromosome is effectively a single string of binary genes. This is in accordance with the results obtained by Meadows et al. (2013), indicating that SGA can indeed be effective when there is a single-string binary representation involved. Nevertheless, as the genes become more complex, the SGA performance drops significantly. To understand why this happens, we start by taking a closer look at the seed pools generated for each instance. The average fitness of these pools is reported in Table 5.

The seed pool average fitness seems to follow a different pattern than SGA performance, as its gap with optimality (i.e. its quality) does not consistently drop with increasing number of products. Therefore, seed pool quality cannot be the reason behind increasingly poor SGA performance. Next, we look at diversity which is the other important characteristic of the seed pool. We need to have as many

Table 5 Average fitness of the seed pool generated for each problem

	n	T	Seed Pool Ave. Fitness	Optimal	Gap
Inst. 1	1	120	**85593.6**	**80243.7**	6.67%
Inst. 2	2	60	**107586.2**	**94545.4**	13.79%
Inst. 3	3	40	**97680.2**	**87107.0**	12.14%
Inst. 4	4	30	**106697.2**	**96804.9**	10.22%
Inst. 5	5	24	**111168.2**	**102510.0**	8.45%
Inst. 6	6	20	**115413.2**	**106490.0**	8.38%
Inst. 7	8	15	**105880.5**	**97784.2**	8.28%
Inst. 8	10	12	**113524.9**	**104320.7**	8.82%
Inst. 9	12	10	**114739.0**	**105885.5**	8.36%
Inst. 10	15	8	**121123.9**	**112232.9**	7.92%
Inst. 11	20	6	**113647.2**	**104237.6**	9.03%

low-level and medium-level building-blocks as possible, to help crossover construct the high-level ones. To analyze seed pool diversity, we first looked at diversity at the lowest level, which is the gene level. We tried to see at each gene position (each time period), how many of the total number of possible gene values are present in the seed pool and how many are missing. The motivation behind this analysis is the observation that if a certain gene value does not exist in the population, crossover is unable to discover it; therefore crossover can discover none of the building-blocks containing that gene.

Figure 4 shows the average number of missing gene values across all genes for every problem instance. It is clear that as the number of products increases, since the genes become more complex and the seed pool size is limited, an increasingly higher number of gene values will be left out. The change starts at around instance 6, the same as what happens to SGA performance in Figure 3. Therefore this lack of low-level gene diversity in the seed pool can be a reason behind the declining trend in SGA performance.

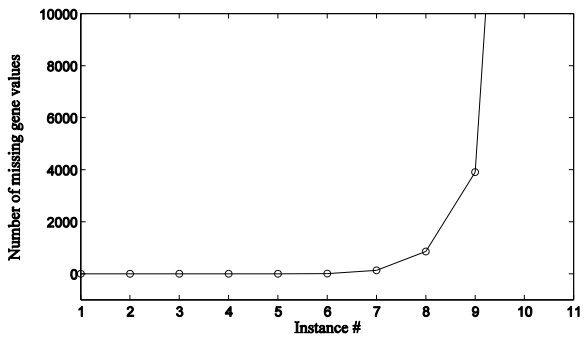

Fig. 4 Average number of missing gene values for each instance

It is important to see that mutation has the ability to overcome the abovementioned limitations of crossover by changing the population on the gene level. With mutation, all possible gene values need not be already present in the population or the seed pool. SGA however, replaces mutation with seeding, and while seeding provides extra building-block material to crossover, it becomes increasingly less effective for the seed pool diversity reasons we observed. If a certain gene has the same value across the population, AND none of the seeds in the seed pool are any different, crossover will be unable to discover any building-blocks other than those containing that gene. Similarly, if a certain gene value does not exist in the population or in the seed pool, crossover will be unable to discover building-blocks that contain it. What this all means is that without mutation, the SGA performance ultimately depends on seed pool quality and diversity, which ultimately depend on gene complexity and extent of the solution space. For a relatively simple problem like CLSP, such diversity proved to be difficult to achieve.

SGA and the Single-Machine Scheduling Problem (SMSP)

Single-machine scheduling is the problem of finding the optimal sequence of jobs to be processed on a single machine. In the version of machine scheduling problem studied here, *n* jobs with known processing times, due dates and ready times are to be assigned to one machine for processing. A job can only be assigned after its ready time, and it will be tardy if it is completed after its due date. No preemption is allowed, and the machine is assumed to work continuously, without any service or setup times required. The objective in SMSP is usually to find a sequence of jobs that minimizes some outcome, such as the total number of tardy jobs (nT), maximum tardiness among jobs (T_{max}), or the total flow time (TF)[1]. It has been shown that in the presence of non-zero ready times, the problem becomes NP-Complete (Lenstra et al. 1977). On the other hand, the problem has many applications, both in manufacturing and in other fields (Baker 1974), and therefore has been studied extensively. Heuristics and metaheuristics are usually suggested and refined for different variations of the problem (e.g. Liu and McCarthy, 1991), with genetic algorithm being one of the common methods (e.g. Süer et al. 2003). The mathematical model

[1] The flow time of a job is the amount of time it spends in the system after its readiness until its completion.

we used for SMSP is given below. Note that minimizing total flow time (TF) was chosen as the sole objective for the purposes of this study.

Objective function to minimize (total flow time):

$$\sum_{i=1}^{n}(C_i - RR_i)$$

Subject to:

$$PP_i = \sum_{j=1}^{n}(P_j * S_{i,j}) \qquad i = 1 \dots n$$

$$RR_i = \sum_{j=1}^{n}(R_j * S_{i,j}) \qquad i = 1 \dots n$$

$$DD_i = \sum_{j=1}^{n}(D_j * S_{i,j}) \qquad i = 1 \dots n$$

$$C_i = \max(C_{i-1}, RR_i) + PP_i \qquad i = 2 \dots n$$

$$C_1 = RR_1 + PP_1$$

$$\sum_{j=1}^{n} S_{i,j} = 1 \qquad i = 1 \dots n$$

$$\sum_{i=1}^{n} S_{i,j} = 1 \qquad j = 1 \dots n$$

$$S_{i,j} \in (0,1) \qquad i,\ j = 1 \dots n$$

$$PP_i,\ RR_i,\ DD_i,\ C_i \geq 0 \qquad i = 1 \dots n$$

Decision variables:

$S_{i,j}$: binary variable for whether job *j* is allocated to slot *i*
C_i: completion time of the job at slot *i*
PP_i: processing time of the job at slot *i*
RR_i: ready time of the job at slot *i*
DD_i: due data of the job at slot *i*

Parameters:

n: number of jobs
P_j: processing time of job *j*
R_j: ready time of job *j*
D_j: due date of job *j*

Indices:

i: slot in schedule
j: job

Figure 5 shows the genetic representation usually used for SMSP. It is easy to see how large the solution space can become, as the number of jobs increases. For *n* jobs, the solution space will be of size *n!*, a much bigger value compared to 2^n. This results in more complex building-blocks, and at the lowest level, more complex genes compared to CLSP. It will be interesting to see how well the Seeding GA will perform on this problem.

5	1	10	2	3	4	9	8	6	7

Fig. 5 Sample chromosome for 10 jobs

GA Configuration:

Similar to the previous problem, MATLAB was used to implement both of the algorithms. We tried to keep the same parameterization as well, with minimal fine-tuning for the GA parameters. Below is a description of the configuration used for CGA:

- Population size *P*=1000 and number of generations *G*=1500 were fixed
- Fully random initialization (random permutations using function `randperm`)
- Parents ordered based on fitness, with adjacent parents forming mates
- Dual-cut order crossover (OX) with random cut points and crossover probability $P_c = 0.8$
- Random swap mutation, with each chromosome having an equal probability $P_m = 0.9$ of exactly one swap
- Swap range in mutation linearly shrinking from *n* to a minimum of 4 according to generation number (a simple adaptive twist)
- Fittest members of the population, including the offspring, form the next generation

As for SGA, we implemented a configuration similar to what was used for CLSP (see Section 2.1) but with different parameters. The seed pool size was chosen to be 1000 sampled from a million randomly generated solutions, while a good seed injection probability was found to be *s*=0.2 with the number of injected seeds *r*=30.

Results:

The specific problem instance studied in this paper is called $R_2P_1D_1$ with 100 jobs and is one of the 8 benchmark problems that the authors use for their SMS optimization research[1]. Table 6 summarizes the best results obtained for $R_2P_1D_1$ with 100 jobs, along with the P-value corresponding to a one-way ANOVA performed to test the

[1] The data sets are available upon request and they might be published in a future separate work

Table 6 Best results, their gaps with the optimal and ANOVA P-values for SMSP

	Best Known	CGA	SGA	HSGA	P-values
$R_2P_1D_1$ 100 jobs	**1575**	**1575** (0.00%)	13684 (768%)	1579 (0.25%)	CGA, SGA: 0.000 CGA, HSGA: 0.141

significance of algorithm effect on performance. Note that CPLEX did not converge for SMSP after 20 hours, so the reported result is labeled as the best known.

Discussion:

CGA results: The conventional GA managed to find the best known solution, which means that a well-implemented CGA, with some simple improvements like the adaptive swap mutation range used here, can perform sufficiently well for SMSP as well as CLSP. Without the adaptive mutation, the best result achieved was 1583 which is still very good.

SGA results: The Seeding GA was significantly outperformed by CGA for this problem. However, HSGA performed almost equally to CGA, which is a clear indication that the reason for the poor performance of SGA is nothing but the lack of mutation. We believe that here again, crossover's inferior ability for discovery compared to mutation makes it very difficult for SGA to explore the solution space properly. The algorithm therefore settles too quickly, and cannot get better beyond some initial improvements (Figure 6). To make matters worse for SGA, seed pool quality and diversity both suffer even more in SMSP. In the genetic representation of SMSP, each gene can have *n* values, and even though an experiment with the seed pool showed that with a pre-sampling size of one million and a seed pool size of 1000 the seed pool does contain almost all of the lowest level building-blocks (individual gene values), the pool content was still desperately poor in the next level building-blocks (gene pairs). Higher level building-blocks are particularly more important in SMSP compared to CLSP, because the objective function is less forgiving of wrong gene ordering. The seed pool quality is also low, since the average fitness was computed to be 46249, far off the best known fitness of 1575.

In addition to the complexity of low-level building-blocks in permutation representation, the results of SGA could be even worse if it wasn't for the specific characteristics pertaining to order crossover used for our experimentation. This is due to the fact that contrary to CLSP, crossover of gene blocks in SMSP can initially yield invalid schedules, and a repair procedure must be used to make the schedule valid again. During this procedure, a disruption in the originally inherited blocks can occur. This is shown in Figure 7.

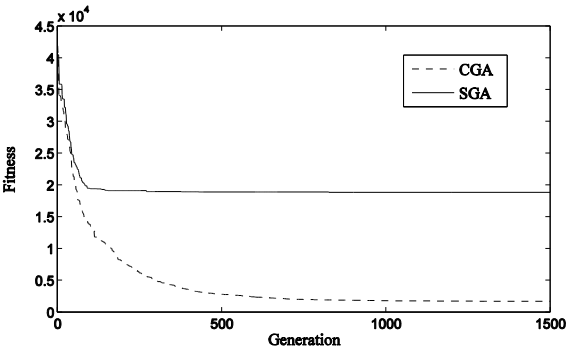

Fig. 6 Convergence behavior of CGA and SGA on SMSP

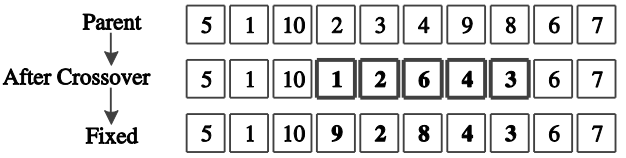

Fig. 7 Demonstration of the crossover procedure used for SMSP

This means that in effect, crossover is acting like mutation: it disrupts the order of jobs according to their initial order in each parent. Consequently, the seed pool need not contain every value at every gene position as a minimum requirement for convergence to global optima. Given enough time, the crossover in SGA can theoretically discover most of the building-blocks even without help from mutation. The problem is however, it is considerably faster and more effective with mutation. Results obtained for the SGA version with added mutation (HSGA) show that for this problem at least, mutation does the discovery job so effectively that the seeding operator is almost irrelevant.

Conclusions and Future Studies

A detailed computational analysis of the results obtained by the Seeding GA and the Conventional GA when applied to Capacitated Lot-Sizing and Single-Machine Scheduling problems was presented. The results for both of these problems show that SGA, not having mutation, cannot explore the solution space properly, and consistently falls

behind CGA in terms of performance. We believe that SGA does work well for the type of problems tested by Meadows et al. (2013), a trend that was confirmed by CLSP instance 1 as well, but the exclusion of mutation seems to be simply unjustified, because in almost all cases when mutation was added to SGA, performance improved significantly. We also believe that injecting seeds from a pool of highly fit individuals is likely to help crossover do its job of combination better, while simultaneously providing diversity to avoid immature convergence, but all of these benefits will be more prevalent if mutation is also present. In some cases however, when solution space is very large, acquiring a highly fit seed pool is not easy, and seeding can only provide diversity.

Overall, seeding is an interesting technique that has shown its effectiveness in many different applications (as mentioned in Introduction), but the combination of crossover and mutation is what gives GA its versatility, and seeding operator of SGA cannot effectively replace any of these two operators. Having said that, the results obtained by HSGA showed promise, and as a main avenue of further investigation, it would be interesting to see if there is a good combination of seeding and CGA that can outperform CGA on a large domain of problems. We believe only then a case can be made for a universal upgrade to the conventional GA methodology.

References

Ahuja, R.K., & Orlin, J.B. (1997). Developing fitter genetic algorithms. *INFORMS J Comput*, 9(3): 251–253.

Baker, K.R. (1974). *Introduction to Sequence and Scheduling*. Wiley, New York.

Bentley, J. L. (1990). Experiments on traveling salesman heuristics. *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms (SODA '90)*. 91-99.

Chang, P.C., Hsieh, J.C., & Liu, C.H. (2006). A case-injected genetic algorithm for single machine scheduling problems with release time, *International Journal of Production Economics*, 103(2), pp. 551–564

Eshelman, L., & Schaffer, J. J. D. (1993). Crossover's niche. *Proceedings of 5th International Conference on Genetic Algorithms (ICGA)*, 9 -14.

Forrest, S., & Mitchell, M. (1993). Relative building-block fitness and the building-block hypothesis. *Foundations of genetic algorithms*, 109-126.

Gicquel, C., Minoux, M., & Dallery, Y. (2008). Capacitated Lot Sizing Models: A Literature Review., Open Access Article hal-00255830.

Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison Wesley.

Goldberg, D.E. (1987). Simple genetic algorithms and the minimal deceptive problem. *Genetic algorithms and simulated annealing*, 74-88.

Goren, H. G., Tunali, S., & Jans, R. (2010). A review of applications of genetic algorithms in lot sizing. *Journal of Intelligent Manufacturing*, 21(4), 575-590.

Grefenstette, J. J. (1987). Incorporating Problem Specific Knowledge into Genetic Algorithms. In *Genetic Algorithms and Simulated Annealing* (pp. 42–60).

Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems*. University of Michigan Press.

Jones, T. (1995). Crossover, macromutation, and population based search. *Proc. of 6th International Conference on Genetic Algorithms (ICGA)*, 73 -80.

Lenstra, J. K., Rinnooy Kan, A. H. G., & Brucker, P. (1977). Complexity of Machine Scheduling Problems. *Annals of Discrete Mathematics*, *1*, 343–362.

Liu, J., & McCarthy, B. L. (1991). Effective heuristic for the single machine scheduling problem with ready times. *International J of Production Research*, *29*, 1521-1533.

Louis, S. J., McGraw, G., & Wyckoff, R. (1993). Case-based reasoning assisted explanation of genetic algorithm results. *Journal of Experimental and Theoretical Artificial Intelligence*, 5, 21-37.

Meadows, B., Riddle, P., & Skinner, C., M. Barley, M. (2013). Evaluating the Seeding Genetic Algorithm. *Advances in Artificial Intelligence*, Lecture Notes in Computer Science Volume 8272, 221-227

Oman, S., & Cunningham, P. (2001). Using case retrieval to seed genetic algorithms. *International Journal of Computational Intelligence and Applications*, 1, 71–82.

Ramsey, C., & Grefensttete, J. (1993). Case-based initialization of genetic algorithms, *Proceedings of the Fifth International Conference on Genetic Algorithms*.

Skinner, C., Riddle, P. J. (2007). Random search can outperform mutation. In *2007 IEEE Congress on Evolutionary Computation, CEC 2007*, 2584–2590.

Skinner, C. (2009). On the discovery, selection and combination of building-blocks in evolutionary algorithms. PhD thesis, Department of Computer Science, University of Auckland.

Süer, G. A., Badurdeen, F., & Dissanayake, N. (2008). Capacitated lot sizing by using multi-chromosome crossover strategy. *J of Intel Manuf*, *19*, 273–282.

Süer, G. A., Vazquez, R., & Santos, J. (2003). Evolutionary programming for minimizing the average flow time in the presence of non-zero ready times. In *Computers and Industrial Engineering*, *45*, 331–344.

Watson, R.A., Pollack, J.B. (2000). Recombination without respect: Schema combination and disruption in genetic algorithm crossover. *Proc. of Genetic and Evolutionary Computation Conference (GECCO)*, 112–119.

Wu, A.S., Lindsay, R.K., Riolo, R.L. (1997). Empirical observations on the roles of crossover and mutation. *Proc. 7th International Conference on Genetic Algorithms*, 362–369.