Comparing genetic algorithm crossover and mutation operators for the indexing problem

Diptesh Ghosh

W.P. No. 2016-03-29
March 2016

INDIAN INSTITUTE OF MANAGEMENT
AHMEDABAD – 380015
INDIA

Comparing genetic algorithm crossover and mutation operators for the indexing problem

Diptesh Ghosh

Abstract

The tool indexing problem is one of allocating tools to slots in a tool magazine so as to minimize the tool change time in automated machining. Genetic algorithms have been suggested in the literature to solve this problem, but the reasons behind the choice of operators for those algorithms are unclear. In this paper we compare the performances of four common crossover operators and four common mutation operators to find the one most suited for the problem. Our experiments show that the choice of operators for the genetic algorithms presented in the literature is suboptimal.

Keywords: Genetic algorithm, permutation problem, crossover, mutation

1 Introduction

The indexing problem is one of assigning tools to slots in a tool magazine in an optimal fashion with the objective of minimizing the tool changing time, and hence the processing time for jobs requiring multiple operations with multiple tools. Physically the process can be visualized as follows. A job to be processed needs a series of operations, requiring one of m tools. These tools are fetched by a tool changer that picks the appropriate tool from the index position of a tool magazine, and then returns the tool to the same position. The tool magazine is a circular disk with n equidistant slots $n \geq m$, in which each tool occupies a slot. Tools are brought to the index position by rotating the magazine, either clockwise or counter-clockwise. The indexing problem is one of allocating tools to slots so that the total amount of rotation of the tool magazine is minimized.

The total amount of rotation of the magazine in units of "operations". An operation is the amount of rotation required to move a slot adjacent to the one currently at the index position to the index position. So if two tools p and q occupy slots i and j in the magazine, then the number of operations required to interchange them is $d_{ij} = \min\{|j-i|, |i+n-j|\}$. Let the frequency of interchange of tools p and q in the process be f_{pq}. Denoting the assignment of tool r to slot k by a binary variable y_{rk}, the total amount of rotation for a process is given by

$$\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \sum_{p=1}^{m} \sum_{\substack{q=1 \\ q \neq p}}^{m} y_{pi} y_{qj} f_{pq} d_{ij}$$

This expression is minimized subject to the conditions that each tool can occupy exactly one slot, and that each slot can house at most one tool. Hence the indexing problem can be modeled as a special form of the quadratic assignment problem.

The literature on solving this problem is one primarily using metaheuristic techniques to solve the problem. For that purpose, a solution to the problem is seen as a permutation of 1 through n, the number of slots available. The number at a particular position in the solution codes the tool that occupies that position. Dereli et al. (1998) and Dereli and Filiz (2000) use a genetic algorithm (GA) to solve the problem in addition to a simulated annealing algorithm. Ghosh (2016c) presents a GA that supercedes the GA presented in Dereli et al. (1998) and Dereli and Filiz (2000). Velmurugan and Victor Raj (2013) use a particle swarm optimization algorithm. Ghosh (2016a) presents a tabu search algorithm using an exchange neighborhood for the problem. Ghosh (2016b) presents a neighborhood search and a tabu search algorithm using a Lin-Kernighan neighborhood structure.

In this paper we carry out a thorough analysis of the operators used in genetic algorithms to solve the indexing problem. The choice of crossover and mutation operators in Dereli et al. (1998) and Dereli and Filiz (2000) do not appear to be the result of a thorough examination of different operators. Ghosh (2016c) chose the same operators as the previous studies in order to make a fair comparison between their algorithm and the ones reported earlier. Several studies on genetic algorithms for the traveling salesman problem (see e.g., Grefenstette et al. 1985, Starkweather et al. 1991, Larranaga et al. 1999) report that operators different from the ones used for the indexing problem perform better. Therefore, in this paper we carry out a comparison between four commonly used crossover operators and four mutation operators to arrive at a competitive genetic algorithm for the indexing problem.

We describe the crossover operators and mutation operators that we use in our study in Section 2. We then describe the results of our computational experiments with GAs using these operators in Section 3. Finally we summarize our findings in Section 4.

2 Operators for GAs for the Indexing Problem

In a genetic algorithm (GA) we start with a collection of solutions which we call a generation, and iteratively improve the generation using reproduction, crossover, and mutation operators. The generation with which we start an iteration is called the current generation, and the generation we form out of it is called the next generation. At the end of the iteration, the current generation is replaced by the next generation. Iterations continue until a stopping condition is reached. Stopping conditions are typically based on the number of iterations performed, or the execution times required by the GA. After the iterations are over, the GA outputs the best solution it has encountered in all the generations it has processed as an approximation to an optimal solution.

In GAs, the reproduction operation is usually straightforward; a pre-specified number of the best solutions in the current generation are copied directly into the next generation. A crossover operator takes more than one solutions (usually two solutions) and combines them to form new solutions. A mutation operator takes a solution and modifies it to form a new solution. A variety of crossover and mutation operators are available though, and the performance of a genetic algorithm depends critically on the choice of these operators.

A solution for the indexing problem with n slots is a permutation of the numbers from 1 through n. If the k-th position in the permutation is occupied by the number p, and $p \leq m$, the number of tools used, then the k-th slot in the magazine is occupied by tool p. If $p > m$ then the k-th slot is empty. Thus the indexing problem is a permutation problem, in that a solution to an indexing problem instance is a permutation of the slots available in the instance. There are specific crossover and mutation operators available for permutation problems, which make sure that the solutions obtained as a result of the operation are also permutations. Crossover operator alternatives are the Order 1 crossover operator, the Alternating Edge crossover operator, the Edge Recombination crossover operator, and the Partially Mapped crossover operator. Mutation operator alternatives are the insert operator, the invert operator, the scramble operator, and the swap operator. A description of these operators can be obtained from the literature, (see e.g., Larranaga et al. 1999). We will describe them in the remainder of the section for the sake of completeness.

2.1 Crossover operators

The crossover operators that we study here accepts two solutions as arguments and combines them to form two more solutions as output. For permutation problem like the one we are addressing, the solutions output must be permutations. The following are the four crossover operators that we compare in this paper.

Order 1 crossover operator (O1X) The O1X crossover creates two solutions by copying parts of one solution into the other, and completing the solutions by adding elements in a particular order. The operator takes two solutions as argument. Suppose these are $(\pi_1, \ldots, \pi_i, \ldots, \pi_n)$ and $(\psi_1, \ldots, \psi_i, \ldots, \psi_n)$. It randomly chooses two indices l and r, $1 \leq l < r \leq n$. It then creates two empty solutions to output. The positions from l to r of the first newly created solution are populated by ψ_l through ψ_r maintaining the order. The same positions in the second one are populated by π_l through π_r maintaining the order. The remaining positions in the first solution to be output are filled starting from position $r + 1$ to position $l - 1$ (wrapping around after position n) by elements $\pi_{r+1}, \ldots, \pi_n, \pi_1, \ldots, \pi_r$ in that order, making sure that no element is duplicated in the solution. The second solution to be output is completed in the same manner, using elements $\psi_{r+1}, \ldots, \psi_n, \psi_1, \ldots, \psi_r$ in that order.

Consider for example that the two solutions that the O1X operator accepts as argument are (3, 9, 5, 2, 6, 4, 1, 7, 8) and (7, 8, 1, 9, 4, 3, 5, 6, 2). Let us also suppose that l and r are chosen as 3 and 6 respectively. So the two solutions that will be output as a result of the crossover are initialized as (*, *, 1, 9, 4, 3, *, *, *) and (*, *, 5, 2, 6, 4, *, *, *) respectively. The first solution will be filled starting from the seventh position by elements in the order 1, 7, 8, 3, 9, 5, 2, 6, 4 in that order without duplication. Hence it will be (2, 6, 1, 9, 4, 3, 7, 8, 5). Following a similar logic the other solution to be output will be (9, 3, 5, 2, 6, 4, 7, 8, 1).

Alternating Edge crossover operator (AEX) The AEX crossover creates solutions by considering each solution in its argument as a collection of directed arcs, and then choosing arcs alternately from the two solutions. Suppose that the solutions the operator takes as arguments are $(\pi_1, \ldots, \pi_i, \ldots, \pi_n)$ and $(\psi_1, \ldots, \psi_i, \ldots, \psi_n)$. From these two solutions, it creates two collections of "arcs": $S_1 = \{(\pi_1, \pi_2), (\pi_2, \pi_3), \ldots, (\pi_{n-1}, \pi_n), (\pi_n, \pi_1)\}$, and $S_2 = \{(\psi_1, \psi_2), (\psi_2, \psi_3), \ldots, (\psi_{n-1}, \psi_n), (\psi_n, \psi_1)\}$. We start building the first solution with π_1, and choose the arc (π_1, π_2) from S_1. The first two elements in the first solution are π_1 and π_2. For the third element in the solution, we choose that arc (ψ_k, ψ_{k+1}) from S_2 where $\psi_k = \pi_2$. The third element in the solution is ψ_{k+1}. For the fourth element, we look at S_1 and try to find arc $(\pi_l, \pi_{l+1}) \in S_1$ for which $\pi_l = \psi_{k+1}$ and π_{l+1} is not already in the solution. We continue in this way, alternating between S_1 and S_2 until the solution is complete, or until we cannot add any more elements to the solution. In the latter case, we choose an element at random from those which are not already in the solution, and start the process again. We continue this until the solution is complete. For the second solution, we start with ψ_1 in the first position, and alternate between S_1 and S_2, starting with S_2.

For example consider the situation in which the two solutions that the AEX operator accepts as argument are (3, 9, 5, 2, 6, 4, 1, 7, 8) and (7, 8, 1, 9, 4, 3, 5, 6, 2). So $S_1 = \{(3,9), (9,5), (5,2), \ldots, (7,8), (8,3)\}$ and $S_2 = \{(7,8), (8,1), (1,9), \ldots, (6,2), (2,7)\}$. The first solution starts with 3 and 9 in the first two positions. Then we look at S_2 and choose (9,4). So the third position is occupied by 4. Looking at S_1, we see that the arc with 4 as its tail is (4,1). The next arc to choose is (1,9) from S_2 but 9 already is in the solution. So we choose an element at random which is not part of the solution already, say, 2. Now looking at S_1, we add 2, then 6 and have to start off again. Finally the solution that we get is (3, 9, 4, 1, 2, 6, 5, 7, 8). Working in a similar manner but starting with arc (7,8) in S_2, we get the other solution to be output as (7, 8, 3, 5, 2, 1, 4, 6, 9).

Edge Recombination crossover operator (ERX) The ERX crossover operator creates solutions by considering each solution in its argument as a collection of undirected edges, and trying to keep as many of the edges that occur with higher frequencies as possible. Consider once again, solutions $(\pi_1, \ldots, \pi_i, \ldots, \pi_n)$ and $(\psi_1, \ldots, \psi_i, \ldots, \psi_n)$ as arguments to the ERX operator. For every element, the operator first creates a set of elements that are its neighbors. For example, if $\pi_j = \psi_k$, then the neighbor of π_j (and ψ_k will be $\{\pi_{j-1}, \pi_{j+1}, \psi_{k-1}, \psi_{k+1}\}$. Of course some of these neighbors can be identical, and the cardinality of the set of neighbors of an element can be smaller than 4. Now for the first solution, we start by choosing the element π_1 in the first position. We remove it from all the sets of neighbors of elements of which it is a part. We then look at the set of neighbors of the chosen element, and find the member of this set that has the lowest number of neighbors. Ties are broken arbitrarily. This element is added in the next position of the solution and is defined as the chosen neighbor. The chosen neighbor is also removed from all the sets of neighbors of the elements. This process of adding neighbors continue until the solution is complete, or we choose an element whose set of neighbors is empty. We then choose an element at random from among the elements not already part of the solution, and repeat the process. The second solution is obtained in the same way, but starting from ψ_1.

For example consider the situation in which the two solutions that the ERX operator accepts as argument are (3, 9, 5, 2, 6, 4, 1, 7, 8) and (7, 8, 1, 9, 4, 3, 5, 6, 2). The sets of neighbors for elements are

1:	{4, 7, 8, 9}	4:	{6, 1, 9, 3}	7:	{1, 8, 2}
2:	{5, 6, 7}	5:	{9, 2, 3, 6}	8:	{3, 7, 1}
3:	{9, 8, 4, 5}	6:	{2, 4, 5}	9:	{3, 5, 1, 4}

Choosing 3 as the first element, we first remove 3 from all the sets. Next we must consider 9, 8, 4, and 5 for the next position. Among these, 8 has the smallest number of neighbors, so 8 is the next element in the solution. After adding 3 and 8 as the first two elements, and removing 8 from the sets of neighbors, we have the sets of neighbors as

1:	{4, 7, 9}	4:	{6, 1, 9}	7:	{1, 2}
2:	{5, 6, 7}	5:	{9, 2, 6}	8:	{7, 1}
3:	{9, 4, 5}	6:	{2, 4, 5}	9:	{5, 1, 4}

Among the neighbors of 8, 7 has two neighbors and 1 has three, so the third element in the solution is 7. Carrying on in this manner, the first solution is obtained as (3, 8, 7, 1, 9, 4, 6, 2, 5). Starting with the first element of the second solution, i.e., 7, the other solution that is output is (7, 2, 6, 5, 3, 8, 1, 4, 9).

According to Grefenstette et al. (1985), Starkweather et al. (1991), and Larranaga et al. (1999) the ERX crossover operator is the favored operator for GAs to solve the traveling salesman problem.

Partially Mapped crossover operator (PMX) The PMX operator creates solutions by copying parts of one solution into another, and then completing the solutions using a partial mapping generated from the parts of the solution copied into one another. Suppose it takes $(\pi_1, \ldots, \pi_i, \ldots, \pi_n)$ and $(\psi_1, \ldots, \psi_i, \ldots, \psi_n)$ as input. It randomly chooses two indices l and r, $1 \leq l < r \leq n$. It then creates two empty solutions to output. The positions from l to r of the first newly created solution are populated by ψ_l through ψ_r maintaining the order. The same positions in the second one are populated by π_l through π_r maintaining the order. The remaining positions in the solutions to be output are filled as follows. Two partial maps are created, $\text{map1} : (\pi_l, \ldots, \pi_r) \rightarrow (\psi_l, \ldots, \psi_r)$ with $\text{map1}(\pi_k) = \psi_k$ if $\psi_k \notin \{\pi_l, \ldots, \pi_r\}$ and $\text{map1}(\pi_k) = \text{map1}(\psi_k)$ otherwise, for $k = l, \ldots, r$; and $\text{map2} : (\psi_l, \ldots, \psi_r) \rightarrow (\pi_l, \ldots, \pi_r)$ with $\text{map1}(\psi_k) = \pi_k$ if $\pi_k \notin \{\psi_l, \ldots, \psi_r\}$ and $\text{map1}(\psi_k) = \text{map1}(\pi_k)$ otherwise, for $k = l, \ldots, r$. The elements in positions from 1 through $l - 1$ and $r + 1$

through n in the first solution are filled with π_1 through π_{l-1} and π_{r+1} through π_n respectively, if the element π is not a member of the domain of map2. Otherwise it is filled with the element map2(π). Similarly the elements in positions from 1 through $l-1$ and $r+1$ through n in the second solution are filled with ψ_1 through ψ_{l-1} and ψ_{r+1} through ψ_n respectively, if the element ψ is not a member of the domain of map1. Otherwise it is filled with the element map1(ψ).

Consider the situation in which the two solutions that the PMX operator accepts as argument are (3, 9, 5, 2, 6, 4, 1, 7, 8) and (7, 8, 1, 9, 4, 3, 5, 6, 2). Let $l = 4$ and $r = 8$. Then map1(2) = 9, map1(6) = map1(4) = 3, map1(1) = 5, and map1(7) - map1(6) = 3. Similarly, map2(9) = 2, map2(4) = map2(6) = 7, map2(3) = map2(4) = 7, map2(5) = 1. After copying the relevant part of the second solution into the first solution to be output, we have the first solution as (*, *, *, 9, 4, 3, 5, 6, *) which when filled by elements in the first input solution and map2($\cdot$) ends up as (4, 2, 1, 9, 4, 3, 5, 6, 8). In a similar way, using the relevant part of the first solution and map1($\cdot$), the second solution to be output is (6, 8, 5, 2, 6, 4, 1, 7, 9).

2.2 Mutation operators

A mutation operator takes a solution as an input and changes it to generate another solution. Since the problems that we are addressing are permutation problems, the mutation operators output permutations of the problem elements. The following are four mutation operators are compared in this paper.

Insert mutation operator This operator removes one element of the solution from its position in the solution and inserts it at some other place in the solution. For example, let the solution it takes as argument be $(\pi_1, \ldots, \pi_i, \ldots, \pi_n)$. It randomly chooses two indices j and k, $1 \leq j/neqk \leq n$, removes π_j and adds it back after π_k to yield the solution $(\pi_1, \ldots, \pi_{j-1}, \pi_{j+1}, \ldots, \pi_k, \pi_j, \pi_{k+1}, \ldots, \pi_n)$.

For example, if the solution it take in is (3, 1, 5, 2, 4, 9, 6, 8, 7) and the positions chosen are $j = 5$, and $k = 2$, then the result after the inversion operation is (3, 1, *4*, 5, 2, 9, 6, 8, 7), with the italics marking the element with changed position.

Invert mutation operator This operator reverses the sequence of some of the elements in the permutation denoting the solution it takes as argument. Suppose the solution it takes as argument is $(\pi_1, \ldots, \pi_i, \ldots, \pi_n)$. It randomly chooses two indices l and r, $1 \leq l < r \leq n$ and reverses the portion from π_l to π_r to output the permutation $(\pi_1, \ldots, \pi_{l-1}, \pi_r, \pi_{r-1} \ldots, \pi_{l+1}, \pi_l, \pi_{r+1}, \ldots, \pi_n)$.

For example, if the solution it take in is (3, 1, 5, 2, 4, 9, 6, 8, 7) and the positions chosen are $l = 2$, and $r = 5$, then the result after the inversion operation is (3, *4, 2, 5, 1*, 9, 6, 8, 7), with the italics marking the elements with changed positions.

Scramble mutation operator This operator jumbles the sequence of some of the elements in the permutation denoting the solution it takes as argument. Suppose the solution it takes as argument is $(\pi_1, \ldots, \pi_i, \ldots, \pi_n)$. It randomly chooses two indices l and r, $1 \leq l < r \leq n$ and jumbles the elements in the portion from π_l to π_r to generate its output.

For example, if the solution it take in is (3, 1, 5, 2, 4, 9, 6, 8, 7) and the positions chosen are $l = 2$, and $r = 5$, then the result after the inversion operation can be (3, *5, 1, 4, 2,* 9, 6, 8, 7), with the italics marking the elements with changed positions.

Swap mutation operator This operator picks two elements in the permutation denoting the solution it takes as argument and interchanges their positions. Suppose the solution it takes as argument is $(\pi_1, \ldots, \pi_i, \ldots, \pi_n)$. It randomly chooses two indices l and r, $1 \leq l < r \leq n$ and interchanges the elements π_l and π_r to output the permutation $(\pi_1, \ldots, \pi_{l-1}, \pi_r, \pi_{l+1} \ldots, \pi_{r-1}, \pi_l, \pi_{r+1}, \ldots, \pi_n)$.

For example, if the solution it take in is (3, 1, 5, 2, 4, 9, 6, 8, 7) and the positions chosen are $l = 2$, and $r = 5$, then the result after the inversion operation is (3, *4*, 5, 2, *1*, 9, 6, 8, 7), with the italics marking the elements with changed positions.

3 Computational Experiments

In the previous section, we described four crossover operators and four mutation operators. Using these, we coded 16 GAs for the indexing problem, one for each crossover-mutation pair. Each of these GAs employed a population size of 100, and copied the best 20 solutions of each generation directly to the next generation. Hence the size of the mating pool was set to 80. The probability of mutation for each solution was chosen as 0.22 based on the preliminary computational experiments described in Ghosh (2016c).

Each of the sixteen GAs was used to solve 25 randomly generated problem instances. These consisted of five sets of five instances each. The number of tools and the number of slots in the five sets were 30 and 45, 45 and 60, 60 and 75, 75 and 90, and 90 and 120 respectively. The GAs were implemented in a multi-start format with 20 starts for each instance and GA combination. The solution reported by a GA for an instance is the best among the solutions output by the 20 starts of the GA on the instance. The time reported was the total time required for the 20 starts.

The results obtained by the GAs on the problem instances are depicted in Tables 1 and 2. In these tables, instead of reporting the results obtained by the GAs on each of the five instances in a set, we have reported the average of the values. For example, the costs and times required by the GA using O1X crossover and insert mutation on the five instances of size 30 are 112708 and 1.328s, 108269 and 1.328s, 109141 and 1.296s, 109260 and 1.281s, and 107544 and 1.312s respectively.We report the average of these cost figures, i.e., 109384.4 at the appropriate place in Table 1 and the average of the time values, i.e., 1.309s at the appropriate place in Table 2.

From Table 1 we see that for each of the problem sizes for each of the mutation operators used, the AEX crossover function produces better (i.e., lower cost) solutions than the other three crossover functions. This is highlighted using italics in the table. From Table 2 we see that the times required by AEX, although not the fastest among the four crossover operators is not very slow. In fact, three of the operators require execution times that are quite close to each other, while the slowest crossover operator, ERX, requires significantly more execution times. Based on this, we feel that the AEX operator is the best crossover operator for the indexing problem.

Interestingly, this observation goes against observations on the performance of the crossover operators for the traveling salesman problem and the vehicle routing problem (see e.g., Grefenstette et al. 1985, Starkweather et al. 1991, Larranaga et al. 1999). For these problems, the ERX operator has been observed to output the best solutions.

We next find the best mutation operator for GAs using the AEX crossover operator. To do so, we re-arrange the cost and time data for GAs using AEX crossover various mutation operators. The rearranged data are shown in Tables 3 and 4. We see from Table 3 that the invert mutation operator combined with the AEX crossover operator outputs the best solutions for the test problems except for problem with 45 tools. The execution times for the four GAs with AEX crossover and various mutation operators are not very different from each other.

Hence, based on our experiments we feel that for the indexing problem, the best GA is one using the AEX crossover operator and the invert mutation operator.

Table 1: Average solution costs for the GAs on the randomly generated problem instances

		Mutation			
Size	Crossover	insert	invert	scramble	swap
30	O1X	109384.4	110288.6	110280.2	109762.8
	AEX	*106370.4*	*105746.8*	*106441.2*	*106433.8*
	ERX	110791.4	112082.2	111452.6	111277.4
	PMX	109357.0	111670.4	111730.4	110126.4
45	O1X	352806.2	354281.6	356572.8	354253.0
	AEX	*348447.8*	*347093.2*	*347380.8*	*345626.4*
	ERX	356603.2	355786.6	356892.8	354591.8
	PMX	353199.4	354797.8	356247.2	352747.2
60	O1X	808434.6	809841.4	811969.0	807744.0
	AEX	*797436.2*	*794315.0*	*797572.6*	*800118.0*
	ERX	811929.4	811957.8	810994.4	811102.8
	PMX	808155.8	807571.6	811590.0	805757.2
75	O1X	1523535.8	1530373.6	1528976.4	1524928.8
	AEX	*1515964.2*	*1509860.0*	*1515687.8*	*1517165.4*
	ERX	1531476.0	1531019.0	1530130.8	1528222.2
	PMX	1524429.0	1526049.8	1531321.8	1525623.6
90	O1X	2940200.4	2953611.4	2941261.6	2931455.0
	AEX	*2882414.0*	*2854692.0*	*2856393.6*	*2878378.0*
	ERX	2949736.4	2949742.0	2949014.0	2953132.8
	PMX	2938760.4	2950915.8	2944408.0	2935055.0

We experimented with our choice of operators on benchmark instances that we have used in other work. For this, we modified the OURGA algorithm described in Ghosh (2016c) which is currently the best GA in the literature. The OURGA algorithm is a GA using a PMX crossover operator and a invert mutation operator, which periodically applies an exchange based local search on some of the best solutions in a generation, and on completion of the GA iterations, on all the solutions in the final generation. We created two main variations of this algorithm, one in which we did not use the local search operation, and the other in which we used it. In both variations, we created two variants, OURGA-PMX with PMX crossover operator, and OURGA-AEX in which we replaced the PMX operator with the AEX operator. (Hence the OURGA algorithm in Ghosh (2016c) is the OURGA-PMX variant of the second variation, i.e., the one using local search.)

We tested the two algorithms on two sets of benchmark instances. The first set was the set of Anjos instances, adapted from instances used in Anjos et al. (2005) for the single row facility layout problem (SRFLP). There are four subsets of five instances in this set, in which the number of tools used were 60, 70, 75, and 80 respectively. The number of slots was fixed at 100 for instances in this set. The second set was a set of sko instances, used in Anjos and Yen (2009) for computational experiments for the SRFLP. There are seven instances in this set. The number of tools in the instances were 42, 49, 56, 64, 72, 81, and 100. We used a tool magazine with 60 slots for the first three instances, and a tool magazine with 100 slots for the others.

The results of our experiments are presented in Table 5.

In the variation not using local search, the costs of the solutions output by OURGA-AEX were lower than the costs output by OURGA-PMX for 26 of the 27 instances in out test sets. The execution times taken by OURGA-AEX were marginally higher than those by OURGA-PMX. This validates our earlier findings about the superiority of the AEX crossover operation.

Table 2: Average execution times (cpu seconds) for the GAs on the randomly generated problem instances

		Mutation			
Size	Crossover	insert	invert	scramble	swap
30	O1X	1.309	1.309	1.356	1.306
	AEX	1.768	1.787	1.787	1.753
	ERX	6.834	6.821	6.875	6.818
	PMX	1.284	1.287	1.290	1.272
45	O1X	2.134	2.147	2.237	2.109
	AEX	2.918	2.953	2.968	2.909
	ERX	10.774	10.800	10.834	10.709
	PMX	2.096	2.090	2.100	2.068
60	O1X	3.162	3.215	3.259	3.175
	AEX	4.369	4.450	4.453	4.340
	ERX	15.796	15.927	15.862	15.831
	PMX	3.144	3.122	3.131	3.103
75	O1X	4.372	4.425	4.503	4.375
	AEX	6.090	6.227	6.240	6.099
	ERX	21.425	21.528	21.506	21.525
	PMX	4.372	4.331	4.375	4.300
90	O1X	7.655	7.518	7.603	7.428
	AEX	10.512	10.659	10.728	10.428
	ERX	36.100	35.405	35.893	35.671
	PMX	7.468	7.400	7.453	7.350

Table 3: Average solution costs for the GAs using AEX crossover operator

	Mutation			
Size	insert	invert	scramble	swap
30	106370.4	*105746.8*	106441.2	106433.8
45	348447.8	347093.2	347380.8	*345626.4*
60	797436.2	*794315.0*	797572.6	800118.0
75	1515964.2	*1509860.0*	1515687.8	1517165.4
90	2882414.0	*2854692.0*	2856393.6	2878378.0

In the variation using local search, the dominance of the AEX operator is not clear. The local search operation in both variants nullify most of the advantages that the AEX operator offers over PMX. We see that among the 27 instances for which results are shown in the table, OURGA-AEX outperformed OURGA-PMX in 10 instances. The costs output by OURGA-AEX for these instances have been highlighted in the table using italics. However, for several other instances OURGA-PMX performed better. The execution times required by OURGA-AEX were marginally higher than those required by OURGA-PMX, although the differences were never significant.

Based on these observations we conclude that the Alternate Edge crossover operator (AEX) and insert mutation operator form a pair of GA operators that is preferred over other pairs of crossover and mutation operators. However, the benefits of AEX are often masked if the GA is hybridized with a local search algorithm.

Table 4: Average execution times (cpu seconds) for the GAs using AEX crossover operator

	Mutation			
Size	insert	invert	scramble	swap
30	1.768	1.787	1.787	1.753
45	2.918	2.953	2.968	2.909
60	4.369	4.450	4.453	4.340
75	6.090	6.227	6.240	6.099
90	10.512	10.659	10.728	10.428

4 Summary

In this paper, we have studied various crossover and mutation operators for genetic algorithms to solve the indexing problem. These operators are the order 1, alternating edge, edge recombination, and partial mapping crossover operators and the insert, invert, scramble, and swap mutation operators. They have been described in Section 2 of the paper. In Section 3 we have presented our results from two computational experiments. In the first experiment, we have compared the performance of the sixteen crossover-mutation operators pairs on randomly generated indexing problem instances with the number of tools varying between 30 and 90, and the number of slots varying between 45 and 120. Based on the results from this experiment, we come to the conclusion that the combination of the alternating edge crossover operator and the invert mutation operator output the best results for this problem. This is interesting, since it goes against the use of the popular partially mapped crossover operator used earlier for the indexing problem, and the literature on the traveling salesman problem which suggests the used of the edge recombination crossover. In the second experiment, we used the alternating edge crossover and invert mutation operator in genetic algorithms and compared the performance of this algorithm with the best genetic algorithm from the literature. We based our experiments on benchmark instances available in the literature. From this experiment we concluded that our choice of operators is indeed better than the ones reported in the literature for genetic algorithms. However if the genetic algorithm also used a neighborhood search within itself, then the improvement using our operators is not completely obvious, although we did find better solutions than the best known for 10 of the 27 benchmark instances that we studied.

In summary, the main contribution of this paper is to suggest the use of the alternating edge crossover and invert mutation for competitive genetic algorithms for the indexing problem.

References

M.F. Anjos, A. Kennings, and A. Vannelli. A Semidefinite Optimization Approach for the Single-Row Layout Problem with Unequal Dimensions. Discrete Optimization 2 (2005) pp. 113–122.

M.F. Anjos and G. Yen. Provably Near-Optimal Solutions for Very Large Single-Row Facility Layout Problems. Optimization Methods and Software 24 (2009) pp. 805–817

A. Baykasoğlu and T. Dereli. Heuristic Optimization System for the Determination of Index Positions on CNC Magazines with the Consideration of Cutting Tool Duplications. International Journal of Production Research 42 (2004) pp. 1281–1303.

A. Baykasoğlu and F.B. Ozsoydan. An improved approach for determination of index positions on CNC magazines with cutting tool duplications by integrating shortest path algorithm. International Journal of Production Research. DOI: 10.1080/00207543.2015.1055351

Table 5: Const of solutions output by OURGA-PMX and OURGA-AEX on benchmark instances

	Without local search		With local search	
Instance	OURGA-PMX	OURGA-AEX	OURGA-PMX	OURGA-AEX
		Anjos instances		
Anjos-60-01	83761	73699	54063	54063
Anjos-60-02	48158	43365	31281	31279
Anjos-60-03	38480	35264	23514	23512
Anjos-60-04	20425	18857	11592	11592
Anjos-60-05	26213	22449	15168	15175
Anjos-70-01	62900	56970	42312	42297
Anjos-70-02	73133	69730	51723	51723
Anjos-70-03	60338	56180	43794	43795
Anjos-70-04	39855	37482	27727	27704
Anjos-70-05	183262	177153	134263	134259
Anjos-75-01	86352	85266	66686	66673
Anjos-75-02	149764	143757	111814	111828
Anjos-75-03	53237	50719	38160	38153
Anjos-75-04	138861	133855	106341	106343
Anjos-75-05	62093	59917	47033	47037
Anjos-80-01	74008	71753	54463	54463
Anjos-80-02	68208	67034	52871	52852
Anjos-80-03	120280	118598	95093	95099
Anjos-80-04	127363	122484	100848	100857
Anjos-80-05	46897	46067	36233	36225
		sko instances		
sko-42	32440	31036	24410	24424
sko-49	43971	43896	36652	36683
sko-56	61174	60909	52927	52963
sko-64	129883	120524	95544	95370
sko-72	168741	162416	132871	133045
sko-81	217362	213985	184529	184869
sko-100	324490	327302	289448	290775

T. Dereli, A. Baykasoğlu, N.N.Z. Gindy, and İ.H. Filiz. Determination of Optimal Turret Index Positions by Genetic Algorithms. Proceedings of 2nd International Symposium on Intelligent Manufacturing Systems. (1998) pp. 743–750. Turkey.

T. Dereli and İ.H. Filiz. Allocating Optimal Index Positions on Tool Magazines Using Genetic Algorithms. Robotics and Autonomous Systems 33 (2000) pp. 155–167.

D. Ghosh. Allocating Tools to Index Positions in Tool Magazines using Tabu Search. Working Paper 2016-02-06. IIM Ahmedabad. (2016a)

D. Ghosh. Exploring Lin Kernighan Neighborhoods for the Indexing Problem. Working Paper 2016-02-13. IIM Ahmedabad. (2016b)

D. Ghosh. A New Genetic Algorithm for the Tool Indexing Problem. Working Paper 2016-03-17. IIM Ahmedabad. (2016c)

J.J. Grefenstette, R. Gopal, B.J. Rosmaita, D. van Gucht. Genetic Algorithms for the Traveling Salesman Problem. In: Proceedings of the 1st International Conference on Genetic Algorithms (ed. J.J. Grefenstette). Lawrence Erlbaum Associates, Mahwah NJ (1985) pp. 160–168.

P. Larranaga, C.M.H. Kuipers, R.H. Murga, I. Inza, and S. Dizdarevic. Genetic Algorithms for the Traveling Salesman Problem: A Review of Representations and Operators. Artificial Intelligence Review 13 (1999) pp. 129-170.

T. Starkweather, S. McDaniel, K.E. Mathias, L.D. Whitley, C. Whitley. A Comparison of Genetic Sequencing Operators. In: Proceedings of the 4th International Conference on Genetic Algorithms (eds. R.K. Belew, L.B. Booker). Morgan Kaufmann, San Francisco CA (1991) pp. 69–76.

M. Velmurugan and M. Victor Raj. Optimal Allocation of Index Positions on Tool Magazines Using Particle Swarm Optimization Algorithm. International Journal of Artificial Intelligence and Mechatronics 1 (2013) pp. 5–8.