A new genetic algorithm for the tool indexing problem

Diptesh Ghosh

W.P. No. 2016-03-17
March 2016

INDIAN INSTITUTE OF MANAGEMENT
AHMEDABAD – 380015
INDIA

A new genetic algorithm for the tool 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. This problem has been widely studied in the literature. A genetic algorithm has been suggested in the literature to solve this problem, but its implementation is non-standard. In this paper we describe a permutation based genetic algorithm for the tool indexing problem and compare its performance with an existing genetic algorithm.
Keywords: CNC tool indexing, genetic algorithm, permutation problem

1 Introduction

Consider a tool magazine capable of holding several tools to be used in an operation by a computer numerically controlled (CNC) machine. This can be visualized as a disk (see Figure 1) with tools in slots arranged at equal intervals on the disk. There may be more slots than tools and consequently some slots may be empty. This disk is pivoted at its center and can rotate in either direction to bring a tool to a position called an index position. The tool changer picks up tools from the index position before commencing an operation and returns the tool to that position after the operation.

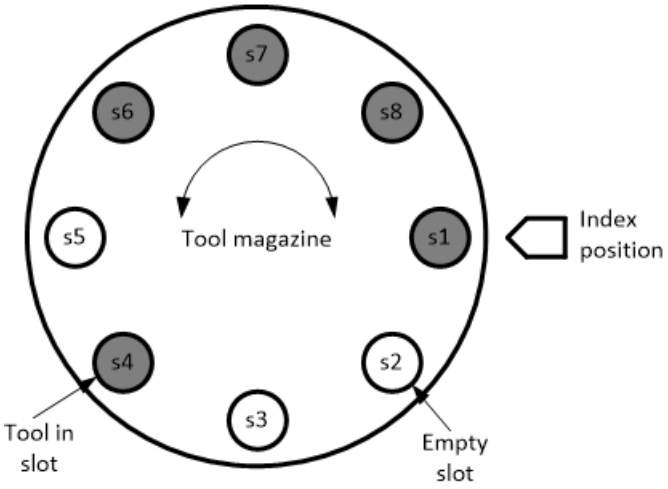

Figure 1: A conceptual representation of a tool magazine

The total processing time of a job requiring multiple operations on a CNC machine depends on the efficiency with which the machine changes tools. Some studies, (e.g., Gray et al. 1993) claim that tool changing accounts for about a quarter of the total cost in automated manufacturing. This efficiency can be improved by assigning tools to slots in a tool magazine in the most efficient way possible. This assignment problem is called indexing.

Formally stated, the indexing problem is the following. Given a tool magazine with n slots, and a manufacturing process requiring $m \leq n$ tools, and the frequency with which each pair of tools are required in consecutive operations, we are required to find the assignment of tools to slots such that the total amount of rotation of the tool magazine to bring all tools to the index position when required is the minimum.

In this paper we are concerned about the variant of the indexing problem in which only one copy of the tool can be present in the magazine. A more complicated version of the problem in which multiple copies of a tool are allowed have also been studied (see e.g., Baykasoğlu and Dereli 2004, Baykasoğlu and Ozsoydan 2015). Since the problem is hard, metaheuristic techniques for the version of the problem that we study are available in the literature. Dereli et al. (1998) and Dereli and Filiz (2000) use a genetic algorithm to solve the problem in addition to a simulated annealing algorithm. 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. Interestingly, the papers presenting genetic algorithms and particle swarm optimization algorithms describe the algorithms and provide an example to illustrate the working of the algoriths, but do not report results of computational experiments. Our focus in this paper is the genetic algorithm proposed in both Dereli et al. (1998) and Dereli and Filiz (2000), especially since the algorithm proposed is related to, but quite different from the usual implementation of genetic algorithms. We propose to test the performance of this algorithm along with a genetic algorithm that we develop for reasonably sized problem instances.

The remainder of the paper is organized as follows. In the next section, we describe the genetic algorithm presented in Dereli et al. (1998) and Dereli and Filiz (2000) along with the genetic algorithm that we propose in this paper. In Section 3 we provide configuration details of our algorithm and the the results of our computational experiments with the two genetic algorithms on the indexing problem. We summarize our findings in Section 4.

Before we proceed to describe the genetic algorithms, we shall define some notation and explain some calculations. Consider an indexing problem instance with n slots and m tools. The *size* of the instance is the value of n. A *solution* to the instance is an allocation of tools to slots. We represent it as a permutation of numbers from 1 through n. If the number at the j-th position of the permutation is k, then we imply that in the solution, tool k occupies the j-th slot if $k \leq m$ or that the j-th slot is empty (if $k > m$). Another way of looking at this is to assume "dummy" tools $m+1$, $m+2$, ..., n which are not used in any operation during processing. Our objective in the indexing problem is to generate a tool assignment that minimizes the rotation of the tool magazine. While exchanging tools, the magazine has to rotate by an integer multiple of the amount of rotation required to bring a slot adjacent to the index position to the index position. We call this unit of rotation an *operation*, and measure the total amount of rotation required in terms of the number of operations used. This total number of operations is called the *cost* of the solution.

Using these terms, we now describe the genetic algorithms compared in this paper.

2 Their Genetic Algorithm, Our Genetic Algorithm

Dereli et al. (1998) and Dereli and Filiz (2000) presented a genetic algorithm to solve the indexing problem. That genetic algorithm was not the conventional genetic algorithm that is known in the literature. This algorithm was described in the papers and was illustrated with an example, but its performance was not tested in detailed computer experiments. In this paper, we present a genetic algorithm closer to the conventional genetic algorithm and compare it with the algorithm presented in Dereli et al. (1998) and Dereli and Filiz (2000).

In order to compare the comparable, we use the same operations in or genetic algorithm as used in Dereli et al. (1998) and Dereli and Filiz (2000). These are the partially mapped crossover operation, the inversion operation, and the rotation mutation. We first describe these operators and then proceed to describe the genetic algorithms.

Partially mapped crossover Partially mapped crossover (PMX, see e.g., Larranaga et al. 1999) is one of the most common crossover operation in genetic algorithms in which each solution is a permutation of problem elements. Consider two solutions $P^1 = (p_1^1, p_2^1, \ldots, p_k^1, \ldots, p_n^1))$ and $P^2 = (p_1^2, p_2^2, \ldots, p_k^2, \ldots, p_n^2))$ to be crossed over. We call them the parent solutions. The result of the crossover would be two other solutions, say C^1 and C^2. These two solutions are called children.

In order to cross the parent solutions, we choose two crossover points i and j with $1 \leq i < j \leq n$. we then copy the portion of P^1 between positions i and j, both inclusive, to the same position in C^2, and the portion on P^2 between positions i and j to the same position in C^1.

We also create two partial mappings, map_1 from the elements in positions between i and j, both inclusive, in P^1 to the elements in positions between i and j in P^2, and map_2 from the elements in positions between i and j in P^2 to the elements in positions between i and j in P^1. To create map_1, for each of the positions k between i and j, both inclusive, we define $map_1(p_k^1) = p_k^2$. Then we check whether for all elements k in the domain of map_1, $map_1(k)$ equals $map_1(map_1(k))$. If not we replace $map_1(k)$ with $map_1(map_1(k))$. This continues until the condition $map_1(k) = map_1(map_1(k))$ holds for all elements k in positions between i and j in P^1. We create map_2 analogously.

Once the mappings are created, the solutions C^1 and C^2 are completed. We demonstrate the completion of C^2; the completion of C^1 is analogous. The elements of C^2 in positions between 1 and n excluding the positions between i and j, both inclusive, are the elements in the same position in P^2, with the condition that if any of these elements belong to the domain of map_1, then they are replaced with that element in P^2 to which the element maps. This ensures that the solution C^2 thus obtained is a permutation of elements from 1 through n.

Inversion operation The inversion operation is a common mutation operation in genetic algorithms for problems in which each solution is a permutation of the elements of the problem. Consider a solution $P = (p_1, p_2, \ldots, p_n)$. The mutation operation chooses two positions, i and j $(i < j)$ and inverts the sequence of elements between p_i and p_j. So after an inversion operation in which positions i and j are chosen with $i < j$, P changes to $(p_1, \ldots, p_{i-2}, p_{i-1}, p_j, p_{j-1}, \ldots, p_{i+1}, p_i, p_{j+1}, p_{j+2}, \ldots, p_n)$.

Rotation mutation operation The rotation mutation operation is a non-standard mutation operation that has been used by Dereli et al. (1998) and Dereli and Filiz (2000). Consider a solution $P = (p_1, p_2, \ldots, p_n)$. The rotation mutation randomly chooses an offset, say k $(1 \leq k \leq n)$ and makes p_k the first element of the permutation. After this operation, P mutates to $(p_k, p_{k+1}, \ldots, p_n, p_1, p_2, \ldots, p_{k-1})$.

In a genetic algorithm, one starts with a collection of solutions to the problem, called a generation of solutions and applies genetic algorithm operations to create a new generation of solutions. This new generation of solutions then replace the original generation of solutions. This iterative replacement of a generation of solutions with a newly constructed generation continues until a stopping condition is satisfied. At that stage the algorithm terminates and reports the best solution that it has encountered during the execution of the algorithm. The initial generation of solutions is usually created by putting together a collection of randomly generated solutions. So a description of a genetic algorithm consists of a description of the process of creating a generation of solutions from another generation and a description of the stopping criterion.

In the genetic algorithm proposed in Dereli et al. (1998) and Dereli and Filiz (2000) a generation is defined as a collection of 200 solutions. It uses the following method to create one generation

from another. First it creates an intermediate collection of size 400. The first 200 solutions in this collection are all the 200 solutions in the starting generation. Then the 200 solutions are paired into 100 pairs and these pairs are crossed over using the partially mapped crossover operator to create 200 children. These 200 children are then added to the intermediate collection. The 400 solutions in the collection are then sorted in non-increasing order of their costs. The best 160 and the worst 40 of these solutions are then copied to the next generation. All the 200 solutions in the next generation thus produced are then subjected to the inversion operation and the rotation mutation operation to yield the final components of the next generation. This process of creating generations is iterated 100 times, and the genetic algorithm terminates after reporting the best solution that it has encountered during its execution. A pseudocode for the genetic algorithm in Dereli et al. (1998) and Dereli and Filiz (2000) is given below. We use the following sets and variables in the pseudocode.

CURRENTGEN	The current generation of solutions.
INTERMEDIATE	A set of solutions from which to create the next generation.
NEXTGEN	The next generation of solutions.
Best-Sol	The best (least cost) solution encountered by the algorithm.
GA-Iter	Iteration count for the genetic algorithm.

Algorithm DFGA

```
1.  begin
2.      set Best-Sol to ∞;
3.      set CURRENTGEN to a set of 200 randomly generated solutions;
4.      for (GA-Iter from 1 to 100) begin
5.          set INTERMEDIATE to ∅;
6.          if (least cost solution in CURRENTGEN is better than Best-Sol)
7.              set Best-Sol ⟵least cost solution in CURRENTGEN;
8.          copy all solutions in CURRENTGEN to INTERMEDIATE;
9.          while (CURRENTGEN ≠ ∅) do begin
10.             remove two solutions P1 and P2 from CURRENTGEN;
11.             obtain solutions C1 and C2 by applying PMX on P1 and P2;
12.             add C1 and C2 to INTERMEDIATE;
13.         end;
14.         create a list by sorting the solutions in INTERMEDIATE in non-decreasing order
            of costs;
15.         copy 160 solutions from the top of the list to NEXTGEN;
16.         copy 40 solutions from the bottom of the list to NEXTGEN;
17.         for (each solution P in NEXTGEN) begin
18.             apply inversion operation to P;
19.             apply rotation mutation to P;
20.         end;
21.         set CURRENTGEN to ∅;
22.         copy all solutions from NEXTGEN to CURRENTGEN;
23.     end;
24.     output Best-Sol;
25. end.
```

The genetic algorithm presented above is different from the conventional genetic algorithms seen in the literature. The one that we propose is closer to a conventional genetic algorithm. Our

algorithm starts with a generation of randomly generated solutions and proceeds to create the next generation using reproduction, crossover, and mutation operators. We describe these operators below.

Reproduction A pre-specified number of "good" solutions in a particular generation are directly copied to the next generation. This copying process is called reproduction, and is intended to ensure that good solutions in a generation are not lost but move to the next generation. We do this using a tournament selection operation. The tournament selection operation that we use works as follows. We choose four solutions from the generation at random. Let us assume that the costs of these four solutions are z_1, z_2, z_3, and z_4. In a first round, we choose one among the first two solutions at random, such that the probability of choosing the first solution is $z_2/(z_1 + z_2$. Note that in this process, the solution having the lower cost (i.e., the better solution) has a better chance of being selected. In a similar manner we also choose one among the third and fourth solutions. In the second round, we choose one among the two solutions chosen in the first round, using a similar method. The chosen solution is said to be the solution chosen by the tournament selection operation.

Crossover A pre-specified collection of solutions from a particular generation is chosen into a mating pool during crossover. This number is the difference between the size of each generation and the pre-specified number of solutions chosen through reproduction. The solutions in the mating pool will participate in crossover operations to generate solutions in the next generation. Each solution in this mating pool is chosen from the current generation using tournament selection. Once the mating pool is created, the solutions in the mating pool are paired off randomly, and each pair of solutions is crossed over using the partially mapped crossover operation to generate two solutions that enter the next generation.

Mutation Each solution in the next generation after the reproduction and crossover operations is subjected to a mutation operation with a pre-specified probability. If the solution is to undergo a mutation operation, then the inversion mutation operation is used on that solution. We did not consider the rotation mutation in our genetic algorithm since we did not see particular value in that operation.

Once after a pre-specified number of generations are generated, a local optimization process is applied to some of the best solutions in the current generation. A pre-specified number of the solutions in a generation are chosen, and each is subjected to local search using an exchange neighborhood (see Ghosh 2016a, for a description of local search using exchange neighborhoods). The number of solutions to which local search is applied should not be high, since these operations are computationally expensive.

Once the stopping condition is reached, each solution in the final generation is subjected to local search using an exchange neighborhood.

A pseudocode of the algorithm that we propose here is given below. In addition to the sets and variables used in DFGA we use the following.

Count	A general purpose counter.
Division	The number of iterations after which some solutions in NEXTGEN will be subjected to local search.
Gen-Size	Size of each generation.
Mutate-Prob	Probability of mutation
Nr-Reproduce	Number of solutions in CURRENTGEN that will be reproduced in NEXTGEN.
Some-Top	Number of solutions in NEXTGEN that will undergo local search.

Algorithm OURGA

```
1.  begin
2.      set Best-Sol to ∞;
3.      set CURRENTGEN to a set of 200 randomly generated solutions;
4.      for (GA-Iter from 1 to 100) begin
5.          if (least cost solution in CURRENTGEN is better than Best-Sol)
6.              set Best-Sol ⟵least cost solution in CURRENTGEN;
7.          copy the best Nr-Reproduce solutions in CURRENTGEN to NEWGEN;
8.          set MATEPOOL to ∅;
9.          for (Count from 0 to (Gen-Size−Nr-Reproduce)/2) begin
10.             obtain solution P1 using tournament selection from CURRENTGEN;
11.             obtain solution P2 using tournament selection from CURRENTGEN;
12.             add P1 and P2 to MATEPOOL;
13.         end;
14.         while (MATEPOOL ≠ ∅) do begin
15.             remove two solutions P1 and P2 from MATEPOOL;
16.             obtain solutions C1 and C2 by applying PMX on P1 and P2;
17.             add C1 and C2 to NEXTGEN;
18.         end;
19.         for (each solution P in NEXTGEN) apply inversion operation to P with proba
            bility Mutate-Prob;
20.         if (GA-Iter is a multiple of Division) begin
21.             create a list by sorting the solutions in NEXTGEN in non-decreasing order
                of costs;
22.             perform local search with exchange neighborhoods on Some-Top solutions
                from the top of the list;
23.         end;
24.         set CURRENTGEN to ∅;
25.         copy all solutions from NEXTGEN to CURRENTGEN;
26.     end;
27.     perform local search with exchange neighborhoods on all solutions in CURRENTGEN;
28.     if (least cost solution in CURRENTGEN is better than Best-Sol)
29.         set Best-Sol ⟵least cost solution in CURRENTGEN;
30.     output Best-Sol;
31. end.
```

3 Computational Experiments

We coded the genetic algorithms described in Section 2 in C. We ran our experiments on a machine with Intel i-5-2500 64-bit processor at 3.30 GHz with 4GB RAM. The DFGA algorithm did not require any parameter setting, since we used the parameters specified in Dereli et al. (1998) and Dereli and Filiz (2000). Both the tournament selection and local search operations in OURGA

are computationally expensive, and hence the parameters we chose for OURGA were decided upon mainly to keep the execution times in check. With that objective, we chose the values of Gen-size as 50, Nr-Reproduce as 10, Some-Top as 2, and Division as 25. In order to determine the value of Mutate-Prob, we carried out some initial experiments. We randomly generated five instances each of sizes 30, 45, 60, 75, and 90, and ran OURGA on these instances with the maximum number of genetic algorithm iterations varying between 50 and 250, and the mutation probability varying between 0.0 and 0.25. For each of the 125 runs we found the mutation probability that was best for that run. The average of the best mutation probabilities was found to be 0.22, and we chose this value for our experiments. Both DFGA and OURGA were implemented as a multi-start algorithms with 20 starts each and in our tables and we report the best solution obtained from among all starts.

We used three types of instances for our computational experiments. These were

B-D instances These are 30 instances from Baykasoğlu and Ozsoydan (2015) used for experiments on a version of the indexing problem in which duplication of tools in the tool magazine was allowed. There are 8 tools to be assigned to 12 slots in the first ten instances (8-20-01 through 8-20-10) for a process consisting of 20 operations. The next ten instances (8-30-01 through 8-30-10) require 8 tools to be assigned to 12 slots for a process consisting of 30 operations. The last ten instances (12-50-01 through 12-50-10) require 12 tools to be assigned to 16 slots for a process consisting of 50 operations.

Anjos instances These instances are adapted from instances used in Anjos et al. (2005) for the single row facility layout problem (SRFLP). Each of the instances in Anjos et al. (2005) had a frequency matrix denoting the frequency of interaction between a pair of facilities in the SRFLP instance. We use the frequency data to denote the number of times a pair of tools are used in consequent operations and a tool magazine with 100 slots for our experiments.

sko instances These instances have been used in Anjos and Yen (2009) for computational experiments for the single row facility layout problem. Each problem in the set had five variants, but all the variants of a problem had the same frequency matrix. Therefore we had a total of seven instances for our experiments. We used a tool magazine with 60 slots when the number of tools were less than 60. Otherwise, we used a tool magazine with 100 slots.

Results for B-D instances

The results from our experiments with the B-D instances are presented in Table 1. The first three columns of each row of the table provide the instance name, the number of tools and the number of slots in the tool magazine. The next two columns in each row present the costs, i.e., the number of operations required by the solutions output by the two genetic algorithms to complete the process described by the instance. The last two columns in each row present the execution times required by the two algorithms in CPU seconds to solve the instance.

From the results we see that OURGA outputs the same solutions as DFGA in the first two sets with 8 tools and 12 slots in the magazine. The times required by OURGA for these instances is on average 16.6% of the times required by DFGA. For instances with 12 tools and 16 slots we see that OURGA outputs better solutions than DFGA in 9 out of 10 instances, while requiring only 20.2% of the execution times required by DFGA. Hence for this set of instances, OURGA clearly outperforms DFGA. However, as problem sizes increase, the times required by OURGA increases much faster than those required by DFGA. So for the next two classes of instances, we expect OURGA to take more time than DFGA.

Table 1: Results of experiments on the B-D instances.

Instance	Tools	Slots	Solution cost		Solution time (sec)	
			DFGA	OURGA	DFGA	OURGA
8-20-01	8	12	19	19	0.968	0.156
8-20-02	8	12	31	31	0.968	0.171
8-20-03	8	12	25	25	0.968	0.171
8-20-04	8	12	29	29	0.953	0.171
8-20-05	8	12	38	38	0.968	0.156
8-20-06	8	12	28	28	0.984	0.171
8-20-07	8	12	24	24	0.984	0.171
8-20-08	8	12	31	31	0.968	0.171
8-20-09	8	12	31	31	0.953	0.171
8-20-10	8	12	25	25	0.984	0.171
8-30-01	8	12	48	48	0.984	0.171
8-30-02	8	12	48	48	0.968	0.156
8-30-03	8	12	50	50	0.984	0.156
8-30-04	8	12	50	50	0.984	0.171
8-30-05	8	12	48	48	0.984	0.156
8-30-06	8	12	51	51	0.968	0.171
8-30-07	8	12	54	54	0.968	0.140
8-30-08	8	12	55	55	0.968	0.171
8-30-09	8	12	46	46	0.968	0.171
8-30-10	8	12	54	54	0.984	0.171
12-50-01	12	50	121	117	1.234	0.328
12-50-02	12	50	114	114	1.234	0.328
12-50-03	12	50	121	118	1.250	0.328
12-50-04	12	50	101	95	1.250	0.343
12-50-05	12	50	126	121	1.265	0.328
12-50-06	12	50	119	112	1.250	0.343
12-50-07	12	50	114	107	1.296	0.328
12-50-08	12	50	140	136	1.250	0.328
12-50-09	12	50	138	136	1.234	0.312
12-50-10	12	50	118	117	1.266	0.343

Results for Anjos Instances

Table 2 presents our results for these instances. The structure of the table is the same as that of Table 1. From Table 2 we see that OURGA outputs significantly better solutions than DFGA for all 20 Anjos instances. The costs of the solutions output by OURGA were on average 76.7% of the costs of solutions output by DFGA. However, for these instances, DFGA was significantly faster than OURGA, taking on average 5.3% of the execution times required by OURGA.

Results for sko Instances

Table 3 presents our results for these instances. The structure of the table is the same as that of Table 1. The results of our experiments on these instances closely match those for the Anjos instances. Here too, OURGA outputs significantly better solutions than DFGA, with the average of the solutions costs output by OURGA being 84.5% that of the average of solution costs output by

Table 2: Results of experiments on the Anjos instances.

Instance	Tools	Slots	Solution cost		Solution time (sec)	
			DFGA	OURGA	DFGA	OURGA
Anjos-60-01	60	100	76100	54063	21.062	288.172
Anjos-60-02	60	100	44465	31281	21.390	291.671
Anjos-60-03	60	100	33803	23514	21.155	280.624
Anjos-60-04	60	100	17647	11592	21.016	282.515
Anjos-60-05	60	100	22541	15168	21.125	289.671
Anjos-70-01	70	100	57263	42312	21.093	394.468
Anjos-70-02	70	100	68608	51723	21.046	405.015
Anjos-70-03	70	100	56301	43794	21.140	406.515
Anjos-70-04	70	100	36407	27727	21.312	405.546
Anjos-70-05	70	100	170012	134263	21.171	396.531
Anjos-75-01	75	100	82648	66686	21.141	457.406
Anjos-75-02	75	100	139273	111814	21.125	454.015
Anjos-75-03	75	100	48945	38160	21.249	453.671
Anjos-75-04	75	100	130124	106341	21.359	478.859
Anjos-75-05	75	100	58769	47033	21.234	463.593
Anjos-80-01	80	100	69448	54463	21.156	519.592
Anjos-80-02	80	100	64293	52871	21.156	522.313
Anjos-80-03	80	100	115204	95093	21.109	519.717
Anjos-80-04	80	100	120431	100848	21.140	520.140
Anjos-80-05	80	100	44967	36233	21.328	530.202

Table 3: Results of experiments on the sko instances.

Instance	Tools	Slots	Solution cost		Solution time (sec)	
			DFGA	OURGA	DFGA	OURGA
sko-42	42	60	30839	24410	8.483	52.312
sko-49	49	60	42363	36652	8.484	65.828
sko-56	56	60	58978	52927	8.453	81.265
sko-64	64	100	123559	95544	21.203	387.562
sko-72	72	100	162311	132871	21.171	497.015
sko-81	81	100	213422	184529	21.110	587.312
sko-100	100	100	321218	289448	21.280	835.077

DFGA. However, OURGA was much slower than DFGA, taking on average, 10.5 times the execution times of DFGA when the number of slots was 60 and on average, 30.2 times the execution times of DFGA when the number of slots was 100.

We end this section with a general observation about the performance of OURGA. We found that local search played a very important role in helping OURGA produce good quality solutions. If the local search component was removed, then the solutions output by OURGA were worse than those output by DFGA for large instances. However, the local search component was extremely time consuming. For example, if the local search component was removed from OURGA, then its execution time would reduce from 835.077 seconds to 2.656 seconds. This means that speeding up local search would be very useful for OURGA.

4 Summary and Contributions

In this paper we present a genetic algorithm called OURGA for the tool indexing problem, and compare the performance of OURGA with a ganetic algorithm presented for the same problem in Dereli et al. (1998) and Dereli and Filiz (2000), which we call DFGA. In our opinion, DFGA is non-conventional in the way they use the genetic algorithm operators while OURGA is more conventional. Our computational experiments show that for indexing problems of practical sizes (with up to 100 slots), OURGA outputs solutions that are of much lower cost than those output by DFGA, although the times required are higher.

In OURGA, we have used the same operators that were used in DFGA to make the comparison as fair as possible. It will be interesting to see whether the performance of OURGA can be improved further by using different crossover and mutation operators.

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

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)

A.E. Gray, A. Seidmann, and K.E. Stecke. A Synthesis of Decision Models for Tool Management in Automated Manufacturing. Management Science 39 (1993) pp. 549–567.

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.

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.