Volume 2, No. 4, July-August 2011

ISSN No. 0976-5697

International Journal of Advanced Research in Computer Science

RESEARCH PAPER

Available Online at www.ijarcs.info

Operation Sequencing Using Genetic Algorithm with Greedy Crossover

Gurpreet Singh
Assistant Professor
Mechanical Engineering Department,
University College of Engineering,
Punjabi University Patiala, India

Amrinder Singh*
M-tech Student
Mechanical Engineering Department,
University College of Engineering,
Punjabi University Patiala, India
er.amrinderchahal@yahoo.com

Abstract: Operation Sequencing Problem (OSP) is a typical NP complete problem, of which the search space increases with the number of operations. Genetic Algorithm (GA) is an efficient optimization algorithm characterized with explicit parallelism and robustness, applicable to OSP. In this paper, we will solve a problem of operation sequencing using GA. But, we will not use simple GA; will use Greedy Crossover instead of simple crossover. Finally experimental results show that the new Greedy Crossover algorithms perform much better than the other techniques.

Keywords: Operation; Sequencing; Problem; Genetic Algorithm

I. INTRODUCTION

Genetic Algorithm (GA) is a kind of optimization algorithm which is based on the natural group evolution genetic mechanism; it depends on search algorithms [1][2]. It is very robust for problems of different types, with desirable characteristics such self-organization, self-learning and self adaption in optimization. Also, GAs have the advantages that do not need to describe all the characteristics of the problem in advance and thus can solve complex and unstructured problems [3]. Due to the advantages above, GA has attracted the interests of experts in various fields. Widely used for optimization problems in automatic, social and economic areas, GA has also infiltrated into many other disciplines, such as engineering design, aerospace, electronics and power systems. For the combinatorial optimization problems which are difficult to adopt traditional methods, such as the nonlinear, multi-model, multi-objective function problems, GA method has become a significant alternative. It's popular to solve travelling Salesman Problem (TSP) with GA [4][5]. Given a set of N cities and each of the distance between two cities, we need to find a close journey to travel every city once and make the total distance shortest for a TSP. Usually the satisfactory solution for a combinatorial optimization problem may not be unique, which means, there might be a number of solutions meeting the conditions which make the objective function to achieve the optimal value larger than the predetermined threshold, but the largest (or smallest) value of the objective function is always unique[6][7]. The searching space of a TSP increases with the number of cities, N. And the combination number of all the road trips is (N-1)!. We will convert our problem into travel salesman problem. Different cities will be the different operations and the path of travel salesman will be the sequencing of the operations. In travel salesman problem we find an optimum path to reduce the cost of travel in operation sequencing will find an optimum sequence to reduce the cost of production.

II. RELATED WORK

A permutation of all operations is represented as a chromosome when solving OSP. Assuming there are n operations, a possible sequence can be encoded as an integer vector (1, 2, 3, 4, 5…..n,) with the length of n. Each integer in the vector just appears once in a path. For a TSP with n operations, this paper uses 0 ~ n-1 different integer encoding to express sequence of these n operations, a permutation of which is a possible solution. The data structure is as follows:

```
// the definition of Gene (a operation)
struct Gene
{
int ID□ // the number of the opeartions
map<Gene*,float> linkCost //the overhead of cost of one operation to another
};
// the definition of Chrom (a permutation of all the operations)
struct Chrom
{
Vector <Gene*> chrom_gene□ // the definition of
chrom_gene (a sequence of all the operations)
float varible□ // the total cost overhead
float fitness□ // individual fitness
};
```

This is one of the simplest expression methods logically corresponding to the operation sequencing, which meets restrictive conditions of OSP. It can not only guarantee that every operation is added into the sequence once and only once, but also ensure that any subset of these operations would not form a loop [8].

A. Selection

The selection of individuals to produce successive generations plays an extremely important role in a GA. A probabilistic selection is performed based upon the individual's fitness such that the better individuals have an increased chance of being selected. An individual in the

© 2010, IJARCS All Rights Reserved

population can be selected more than once with all individuals in the population having a chance of being selected to reproduce into the next generation. Lee et al. (2001) [9] developed the roulette wheel, which was the first selection method. In addition, there are se-lection methods, such as roulette wheel's extensions, scaling techniques, tournaments, elitist models, and ranking methods. These selection operators were presented for numerical optimization and the main objective is to reduce the sampling error and improve calculation precision. When using GAs for operations sequencing, the natural number is used for coding. The fitness value of individual is only a relative concept (only used for value comparing and the value itself is not concerned). So the problem of sampling error does not exist. Compared with other selection operators, the ''tournament selection'' is more suitable for the problem of operations sequencing. In order to guarantee the astringency of GAs, the optimal individual in one generation must be kept to the next generation. Other individuals in population are selected using the ''tournament selection'' operator. Suppose there are W individuals to be selected, selecting two individuals randomly from the population and keeping the better one for the next generation. Repeating this process W-1 times and then at last all individuals in the next generation are obtained [10].

B. Simple Crossover

In this work, a partially mapped crossover (PMX) operator (Gorges-Schleuter, 1985) [11] is modified to produce offspring. After two parents are selected from the population, based on the string length (i.e., number of elements in the string), a crossover point is randomly generated and a segment of the string from that point to end of the string is selected. The offspring, child 1, is generated by arranging the elements of the selected segment in this parent according to the order in which they appear in the other parent with the order of the remaining elements the same as in the first parent [12]. The role of these parents is to generate another off- spring, namely, child 2. The crossover operator can be illustrated as follows:
Select two strings from the current population and denote them as parent 1 and parent 2:
Parent 1 : (1,4,6,3,5,2) Parent 2 : (4,5,6,1,2,3)

Consider a random crossover site as X=2 and the segment from parent 1 from the crossover site till the end of the string (6, 3, 5, 2)

Arrange the selected elements in the order of parent 2 and obtain (5, 6, 2, 3).
Then the offspring, child 1 from parent 1 is generated as: (1,4,5,6,2,3).

a. Problems with Simple Crossover

A portion of one parent's string is mapped onto a portion of the other parents string and the remaining information is exchanged [13][14]. Consider for example the following two parent tours:

(12345678) and (37516824)

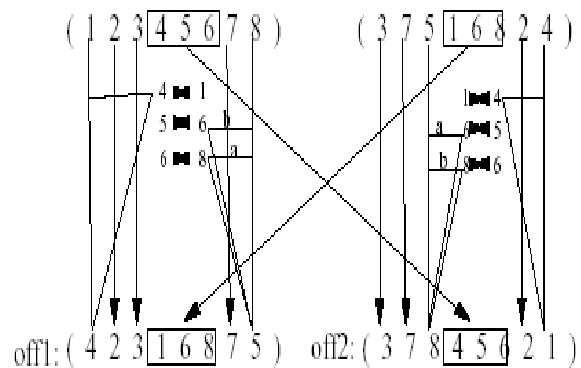

Figure 1. Simple crossover

First, it selects uniformly at random two cut points along the strings, which represent the parent's tours. The sub strings between the cut points are called the mapping sections. In the above example they define the mapping 4-1, 5-6, 6-8.Now the mapping section of the first parent is copied into the second offspring and the mapping section of the second parent is copied into the first offspring offspring1 (xxx168xx) and offspring2: (xxx456xx)

Then offspring 'i' ('i'=1, 2) is filled up by copying the elements of the I-th parents. In case an operation is already present in the offspring it is replaced according to the mapping. For example the first element of offspring 1 would be a 1 like the first element of the first parent. However there is already a 1 present in offspring1. Hence, because of the mapping 1-4 we choose the first element of the offspring 1 to be a 4.the second, third and seventh elements of offspring1 can be taken from the first parent. However, the last element of the offspring 1 would be an 8, which is already present. Because of the mapping 6-6 and 6-5, it is chosen to be a 5. Hence Offspring1=(42316875) similarly Offspring2=(37846521). Due to this problem we can't use this simple crossover. For the solution of this problem we had used Greedy crossover.

b. Greedy Crossover

In the design of the crossover operator, we first absorbed ideas from two previous studies. One was by Grefenstetts et al. (1985) [15] who used a heuristic crossover which constructs an offspring by choosing the better of two parental edges [16]. Another was by Starkweather et al. (1991) [17] who designed the edge recombination operator. One of their findings was that it is very important to preserve common edges between the two parents [18]. After studying the above work, we developed the following crossover operator, which is different from the previous two algorithms but combines the good ideas from both. Given two parent tours in the normalized path representation, **P1** and **P2,** the first offspring is constructed as follows: start with a random city ***c,*** then check whether either the edge leading to c or from c (i.e., the edge *<c, C,ight>* or *<cleft, c>)* is used in both **P1** and **P2.** If so, the common edge is chosen. Otherwise, we compare c's right side edge in each of **P1** and **P2.** The shorter one is chosen, unless it would introduce a cycle, in which case the longer one is chosen. If the longer one would also introduce a cycle, then we extend the tour by a carefully selected edge (details later). The second offspring will be constructed in a

© 2010, IJARCS All Rights Reserved

similar way, but we compare c's two left side edges instead of its right side edges.

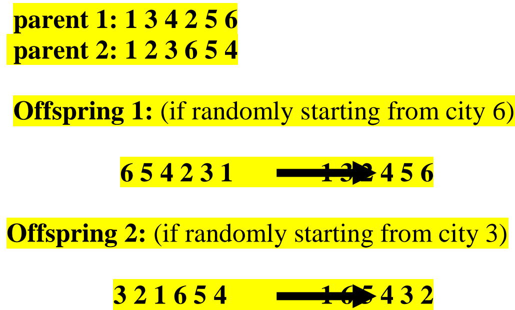

Figure. 2. An example of greedy crossover

Figure 1 shows a simple example of our crossover operation. Assume that we have two parents as shown in Figure 1. To generate the first offspring, we randomly start from city 6. The two right side edges are <6,1> in **P1** and <6,5> in **P2.** The distance table shows that <6,1> and <6,5> have exactly the same length. In this situation, the edge in **P1** would normally be selected first for offspring 1; the edge in **P2** for offspring 2. However, in our example, edge *<6,5>* is a common edge between the two parents, so <6,5> is chosen. Next, from city 5, we have two right side edges: *<5,6>* and <5,4>. This time, we choose <5,4> because the common edge *<5,6>* would create a cycle. From city 4, the edges are <4,2> and <4,1>: the shorter one <4,2> is chosen. Then from city 2, there are <2,5> and <2,3>: the shorter one <2,3> is chosen. Finally, we only have city 1 left, which has to be the last one in the tour. The second offspring is generated starting from a different random city, 3. This time, the left edges are compared. Luckily, all of the best edges in **P1** and **P2** have been inherited by offspring 2. The criterion of selecting a new edge to prevent a cycle is based on the same idea as used in our selective initialization. That is, edges not belonging to the k-nearest neighbor sub graph are most likely to be excluded from the optimal tour. Thus, we try to select from the k-nearest neighbor list first; (only if they are all used do we select randomly from the other, longer, edges. The actual selection rule implemented is a bit more sophisticated in how to choose from the k-nearest neighbour list. Assuming that c is the current city, we examine all unused cities in e's k-nearest neighbour list, then choose the one which has the fewest k-nearest neighbour cities available. The idea behind this is that the fewer k-nearest-neighbours ***c*** has, the more likely it is to become an isolated city in the knearest neighbour subgraph. So we should choose it, to avoid the danger of using a non-k-nearest-neighbour city [19].

C. Mutation.

A mutation operator is used to investigate some of the unvisited points in the search space, and also to avoid premature convergence of the entire feasible space caused by sequences GAs, and only the main compulsive constraints are given in the formula [20]. Other constraints can be also added behind it. The explanation of each item in the formula is shown in the following section: some super chromosomes. This operator makes random changes in one or more elements of the string. Mutation is done with a small probability, called mutation probability or rate. This is done to protect loss of some potentially useful strings. Some individuals of the next generation, which are obtained through the two above-mentioned operators, are selected and then the positions of two codes in each individual are exchanged randomly to realize the mutation operation. For example, if the third and sixth positions are the selected mutation positions in individual O1, new individual O2 will be obtained as follows:

O1 : (1,4,5,6,2,3) ➞ O2 : (1,4,3,6,2,5)

The chromosomes resulting from these operators are often known as offspring or children and these form the next generation's population. This process is repeated for generating feasible solutions. A feasible solution is a sequence of operations that considers all of the compulsive constraints. The fitness value for a feasible solution is zero. Therefore, when the fitness value for some of solutions is zero, the genetic algorithm stops and selects these solutions as feasible solutions [21]. This process is repeated until the enough feasible solutions are generated.

D. List Of Operations with Codes

Table 1. List and coding of operations

Serial No.	Code for Operation	Name of Operation
1.	1	Rough Turning
2.	2	Finish Turning
3.	3	External Groove Cutting
4.	4	External Threading
5.	5	Internal Threading
6.	6	Drilling
7.	7	Boring

E. Fitness Function

The fitness function, which is a measure function used to express the adaptability of a string, is used to connect the problem and the algorithm. The adaptability is expressed by the fitness value. In this stage, additive constraints can be implemented. The optimization constraint is often considered as an additive constraint. This constraint means that some target functions should be met in the technological sequence decision, such as minimum processing times, minimum production cost and soon. In this research, the minimum production cost is employed to calculate the fitness of each operation sequence, and to measure the efficiency of a manufacturing system.

F. Cost Matrix

To make cost matrix, as shown below in the matrix, write the cost of operations in the matrix, if we perform operation 2 after operation 1 than its cost is 10 rupees, we will write this in box 1,2. The cost of operation 3 after 1 is rupees 7, so it is written into the box 1,3. All the costs will be written in corresponding boxes as shown below.

© 2010, IJARCS All Rights Reserved

Table 2. Cost Matrix

Operations	1	2	3	4	5	6	7
1	∞	10	7	9	5	4	3
2	2	∞	7	4	6	9	2
3	4	∞	∞	5	7	9	2
4	8	∞	∞	∞	6	4	7
5	9	4	6	5	∞	8	3
6	2	7	4	5	8	∞	6
7	4	3	5	5	∞	2	∞

a. Fitness Function in our Problem (α)

Where α is a positive number as large as it can be. The value of this fitness is calculated by adding the values of production cost of different operations (from cost matrix) according to operation sequence string generated by GA. Example:

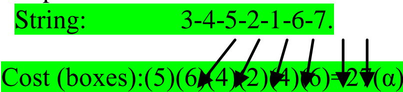

Figure.3. Calculation of Fitness function.

The chromosomes resulting from the three operators, namely selection, crossover and mutation, are often known as off spring or children and these form the next generation's population. This process is repeated for a desired number of generations, usually up to a point where the system converges to significant well-performing sequences and the value of fitness function is minimum.

G. *Methodology*

a. First randomly create 20 strings with the given operations.
b. Carry on top 10 sequences to the next generation leaving the other 10 sequences.
c. From the best 10 tours randomly select parents and producing 10 more children by performing the all three steps of GA.
d. Append these 10 children's to the previous best 10 parents to make them to total 20 strings.
e. Again sort the strings and take best 10 strings among them for next generation
f. Repeat the above operations to create 10 more children's and again sorting the combined 20 tours.
g. Repeat these steps until the value of fitness function get constant and it do not decreases further by more iteration.

III. RESULTS AND DISCUSSIONS

For the described example, the operation information is shown in Table 1. The cost of Operations is given into Table 2. According to first step of the genetic algorithm randomly created 10 strings are shown in Table 3.

Table 3. (10 strings selected from the randomly selected 20 strings)

String No.	String	Fitness value
1	1-2-4-3-6-7-5	29
2	1-4-6-2-7-3-5	34
3	1-6-4-7-3-5-2	32
4	4-3-1-2-6-7-5	29
5	4-1-6-2-7-5-3	27
6	4-6-1-2-3-7-5	25
7	4-1-6-2-3-7-5	28
8	6-4-7-5-1-3-2	28
9	6-1-4-7-2-3-5	35
10	6-1-4-7-5-3-2	24

After performing the all three steps on these 10 strings or parents, we will get 10 new strings or children and got new value of fitness function according to these new children as shown in Table 4. In Table 3. ,the least value of fitness function is 24 for the string 6-1-4-7-5-3-2. In Table 4., least value of fitness function is 16 for string 6-4-3-1-7-5-2.

Table 4. String generated by GA.

String No.	String	Fitness value
1	6-4-3-1-7-5-2	16
2	6-4-3-1-7-2-5	21
3	1-6-4-7-5-3-2	22
4	4-6-1-2-7-5-3	24
5	6-1-4-7-5-3-2	24
6	4-6-1-2-3-7-5	25
7	1-6-4-7-3-2-5	27
8	4-1-6-2-7-5-3	27
9	1-4-6-2-7-5-3	28
10	6-4-7-5-1-3-2	28

Since the performance of GAs is not guaranteed and can never be assessed on the basis of a single run, in this case, the program will be repeatedly run for 10 times, and the production cost or fitness function of every optimal process plan is generated and it can be seen that the production cost varies from 35 to 11 and it can not be reduced more by further iterations, so final least cost is 11 and the most optimal sequence is 1-6-4-3-2-7-5. As shown in Table 5.

© 2010, IJARCS All Rights Reserved

Table 5. String generated by GA after 10 iterations.

String No	String	Fitness Value
1	1-6-4-3-2-7-5	11
2	1-6-4-3-7-5-2	15
3	4-6-1-7-5-3-2	15
4	6-4-3-1-7-5-2	16
5	4-6-3-1-7-5-2	19
6	1-6-4-3-7-2-5	20
7	6-4-3-17-2-5	21
8	6-4-3-1-2-7-5	21
9	4-6-3-1-2-7-5	24
10	4-6-3-1-7-2-5	24

In this work, we used a different genetic algorithm for the **OSP.** The algorithm applies a greedy crossover and mutation. A selective initialization operator is also proposed. The results show that combining the greedy crossover and genetic algorithms is a promising approach for solving the large OSP. Two concluding remarks are as follows.

From the point of view of genetic algorithms, by Combination of greedy crossover with GA it becomes a very effective tool to solve the problem of OSP. Moreover, it makes small population sizes sufficient to solve large problems.

From the point of view of this method, by incorporating the genetic algorithms technique, we can escape from local optima in many cases, so that much better results can be obtained than by using heuristic methods alone. We can also achieve very high stability.

IV. CONCLUSION & FUTURE SCOPES

There are two interesting directions for future work. One is to further improve the algorithm by introducing a simplified form of the Lin-Kernighan heuristic. Another direction is to parallelize the algorithm. We would like to investigate whether the algorithm can be factorized efficiently. We are also interested in modifying the algorithm to a coarse-grained parallel genetic algorithm.

V. REFERENCES

[1] D. E. Goldberg and J. R. Lingle, "Alleles, Loci, and the Traveling Salesman Problem,".1st International Conference on Genetic Algorithms (ICGA 85), 1985, pp. 154–159.

[2] J. H. Holland, Adaptation in Natural and Artificial Systems, MIT Press, 1975.

[3] P. Larranaga, C. M. H. Kuijpers, R. H. Murga, I. Inza. and S. Dizdarevic. "Genetic Algorithms for the Traveling Salesman Problem: A Review of Representations and Operators," Artificial Intelligence Review, Apr. 1999, pp. 129–170.

[4] D. E. Goldberg, Genetic Algorithms in Search, Optimization, and Machine Learning, Addison-Wesley, 1989.

[5] L. Davis, "Applying Adaptive Algorithms to Epistatic Domains," 9th International Joint Conference on Artificial Intelligence (IJCAI 85), 1985, pp.162–164.

[6] I. M. Oliver, D. J. Smith, and J. R. C. Holland, "A study of permutation crossover operators on the traveling salesman problem,"2nd International Conference on Genetic Algorithms(ICGA 87), 1987, pp. 224–230.

[7] B. A. Julstrom, "Very greedy crossover in a genetic algorithm for the traveling salesman problem," 10th Symposium on Applied Computing(SAC 95), 1995, pp. 324–328.

[8] Y. Mu. "The application of Genetic Algorithm solving TSP (In Chinese)," master's thesis, Dept. Computer Sciences, Tianjin Normal University, Tianjin, China, 2004

[9] Lee, D.-H., Kiritsis, D., Xirouchakis, P., "Branch and fathoming algorithms for operation sequencing in process planning", International Journal of Production Research (39),2001, pp. 1649–1669.

[10] Guo, Y.W., Mileham, A.R., Owen, G.W., Li, W.D., "Operation sequencing optimization using a particle swarm optimization approach", Proceedings of the Institution of Mechanical Engineers, Part B: Journal of Engineering Manufacture 220 (B12), 2006, pp. 1945–1958

[11] Gorges-Schleuter, M., "An asynchronous parallel genetic optimization strategy", First International Conference on Genetic Algorithms. Lawrence Erlbaum Associates, Hillsdale, NJ", 1985, pp. 422–427.

[12] Li, W.D., Ong, S.K., Nee, A.Y.C., "Optimization of process plans using a constraint-based tabu search approach", International Journal of Production Research 42 (10), 2004, pp. 1955–1985.

[13] Reddy, S.V.B., Shunmugam, M.S., Narendran, T.T., "Operation sequencing in CAPP using genetic algorithm", International Journal of Production Research 37, 1999, pp. 1063–1074.

[14] Lin, C.-J., Wang, H.-P., "Optimal operation planning and sequencing: minimization of tool changeovers", International Journal of Production Research 31, 1993, pp. 311–324.

[15] J. J. Grefenstetts, R. Gopal, B. Rosmaita, and D. Van Gucht ,"Genetic Algorithms for the Traveling Salesman problem", International Conference on Genetic Algorithms and Their Applications, 1985, pp. 160-168.

[16] C. R. Reeves, "Using Genetic Algorithms with Small Populations", Fifth intenational Conference on Genetic Algorithms, 1993, pp. 92- 99.

[17] T. Starkweather, S. McDaniel, K. Mathias, D. Whitley, and C. Whitley, "A Comparison of Genetic Sequencing Operators", Fourth Intemational Conference on Genetic Algorithms, 1991, pp. 69-76.

[18] D. Whitley, T. Starkweather, and D. Shaner, "The Traveling Salesman and Sequence Scheduling: Quality Solutions Using Genetic Edge Recombination", Lawrence Davis, editor, Handbookof Genetic Algorithms, 1990.

© 2010, IJARCS All Rights Reserved

[19] A. Homaifar, S. Guan, and G. E. Liepins, "A New Approach on the Traveling Salesman Problem by Genetic Algorithms", Fifth International Conference on Genetic Algorithms, 1993, pp. 460-466.

[20] Ding, L., Yue, Y., Ahmet, K., Jackson, M., Parkin, R., "Global optimization of a feature-based process sequence using GA and ANN techniques", International Journal of Production Research 43 (15), 2005, pp. 3247–3272.

© 2010, IJARCS All Rights Reserved