View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Ghent University Academic Bibliography

Crossover and Mutation Operators for Genetic Algorithm with Permutation Representation of Solution Domain

Dmitriy BORODIN[1], Wim DE BRUYN, Bert VAN VRECKEM, Viktor GORELIK

University College Ghent, Belgium

Dorodnycin Computing Center of Russian Academy of Science, Moscow, Russia

This paper presents the overview of crossover and mutation operators for permutation representation of genetic algorithm solution chromosome with examples.

Introduction

Genetic algorithms (GA) are based on natural evolution and use the “survival of the fittest” approach, where the best solutions survive and are varied until a good result is found.

This class of heuristic algorithms proved reasonably good efficiency for different combinatorial and optimization problems [5,6,7,8].

The classical GA consists of the following steps: 1) Generation of Initial Population (usually pseudo-randomly); 2) Selection of solutions for crossover; 3) Crossover of two or more solutions to produce new solutions; 4) Mutation of some solutions; 5) Selection for the new population; 5) If the stopping criteria is not satisfied, go to step 2; otherwise return the quasi-optimize solution.

A typical GA requires a genetic representation of the solution domain and a fitness function to evaluate the solution domain, which in many cases appears to be an objective function for the problem under study.

The whole genetic process is driven by a hope that new solutions generated by crossover operator will be better in terms of fitness function (FF) value than their “parent” solutions. Trapping in the local optima – a common problem of search techniques – is reduced by mutation operator – a minor change of solution components in order to change its position on the FF curve.

Classical GA was proposed by Holland in the 1970s [9] and was based on the binary representation of the solution domain. Many GA implementations include solution encoding and decoding procedures into binary string. This explains by a relatively easy work with such binary structure due to only two possible values of the solution components: 0 and 1. Effective crossover and mutation operators have been developed for the binary representation but nowadays more and more problems are encoded in a problem-specific way. There are three major interpretations of a permutation [3]. For example, in TSP, permutations represent tours and the relevant information is the *adjacency relation* among the elements of a permutation.

[1] Correspondence: Dmitry Borodin, Faculty of Business Information and ICT, Schoonmeersstraat 52, 9000 Ghent, Belgium. Tel: +32486335470. email: dmitriy.borodin@hogent.be

In resource scheduling problems, permutations represent priority lists and the relevant information is the *relative order* of the elements of a permutation. In other problems, the important characteristic is the *absolute position* of the elements in the permutation. We consider the permutation solution representation as a sequence of unique integer numbers, as shown on figure 1.

Location	0	1	2	3	4	5	6	7	…	N	Unique
Genes	**3**	**1**	**7**	**9**	**5**	**6**	**4**	**N**	**…**	**2**	

Fig. 1. Chromosome with Permutation Solution Representation

A solution is called a chromosome, solution items (elements) – genes. The rule for any genetic operator involving chromosome genes change is to keep the chromosome genes unique. In particular, this is a demand for crossover and mutation operators.

Crossovers

Normally, crossover is performed with two chromosomes, called parent chromosomes, or parents, to create new chromosomes, child chromosomes or children. The idea is to take information from both parents and transfer it to the children hoping that children would be better in terms of fitness function.

In most cases, from two parents it is possible to get one child or two children.

There are many various crossover operators developed for permutation chromosomes [1,6,7,8,10]. We consider the most popular and suitable for various problems.

1. *Sorting crossover:* this operator sorts one parent permutation toward the order of the other parent permutation.

The following pseudo-code[thesis] is a generic algorithm for sorting crossover, the details on sorting crossovers are discussed in [7].

```
1: Input: parent1, parent2 Output: offspring
2: normalised_parent1 = compose(parent1, inverse(parent2))
3: moves_sequence = sort(normalised_parent1)
4: distance = length(moves_sequence)
5: crossover-point = uniform_random(integer_range[0, distance])
6: offspring = parent1
7: while crossover-point > 0 do
8:    offspring = compose(offspring, moves_sequence[crossover-point])
9:    crossover-point = crossover-point - 1
10: end while
11: return offspring
```

2. *Partially-matched crossover (PMX).*

Step 1. Randomly select a segment of genes from parent 1 and copy them directly to child 1. Note the indexes of the segment. For the random selection of a segment two random numbers are generated (the condition is 0 <= Rand_N1 < Rand_N2 < Parent1_Length) and used as the first and last indexes of the segment respectively.

Step 2. In the same segment positions of parent 2, select each value that has not already been copied to child 1.

Step 3. For each of the selected values:

Step 3.1. Note the index of the value in parent 2 and locate the value from parent 1 in the same position.

Step 3.2. Locate the same value in parent 2.

Step 3.3. If the index of this value in parent 2 is a part of the original segment, go to **step 3.1** using this value.

Step 3.4. If the position isn't a part of the original segment, insert the value derived from **step 3** into child 1 in this position.

Step 4. Copy any remaining positions from parent 2 to child 1.

For child 2, swap the parents and perform **steps 1-4**.

Figure 2 gives a PMX example.

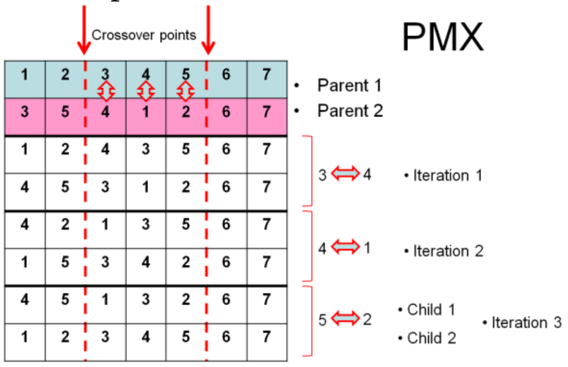

Fig 2. Partially-matched crossover example

3. *Cyclic crossover (CX):* identifies a number of so-called cycles between two parent chromosomes.

Step 1. Child 1 gets the first gene of parent 1.

Step 2. In Parent 1, find gene equal to gene 1 in parent 2; check if child 1 contains this gene. If not, copy this gene to child 1 at the same place, let it be place *x*. Otherwise – go to step 5.

Step 2.i. In parent 1, find gene equal to gene x in parent 2; check if child 1 contains gene x. If not, copy gene x to child 1 at the same place, let it be place *x1*; repeat **Step 2.i** with x=x1. Otherwise – go to **Step 3**.

Step 3. Final step: once **Step 2** is completed, fill in empty genes of child 1 with corresponding values of parent 1.

Figure 3 illustrates CX iterations.

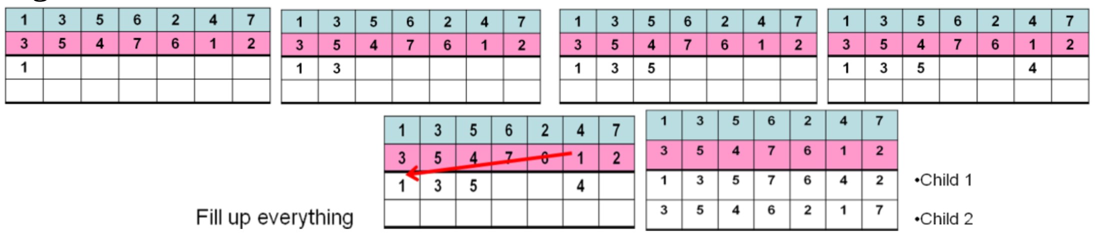

Fig 3. Cyclic crossover example, iterations from left to right

4. *Order crossover (OX):* segments of genes with same positions are copied from both parents to both children, the missing genes are copied to child 1 in the order these genes are located in parent 2.

Step 1. Select two random genes and copy the segment between them to both children from parent 1 and 2 respectively.

Step 2. Take a set of genes from parent 1, missing in child 1, and in order they are positioned in parent 2.

Step 3. Put genes from the set to empty places in child 1 consequently.

```
Parent 1: 8 4 7 | 3 6 2 5 | 1 9 0
Parent 2: 0 1 2 | 3 4 5 6 | 7 8 9
Step 1
Child 1:  _ _ _ | 3 6 2 5 | _ _ _
Step 2
Set of missing genes:
8 4 7 1 9 0
Reordering according to Parent 2
0 4 7 1 8 9
Step 3
Child 1:  0 4 7 | 3 6 2 5 | 1 8 9
```

For the child 2, just swap parents:

```
Parent 1: 0 1 2 | 3 4 5 6 | 7 8 9
Parent 2: 8 4 7 | 3 6 2 5 | 1 9 0
Step 1
Child 1:  _ _ _ | 3 4 5 6 | _ _ _
Step 2
Set of missing genes:
0 1 2 7 8 9
Reordering according to Parent 2
8 7 2 1 9 0
Step 3
Child 2:  8 7 2 | 3 6 2 5 | 1 9 0
```

```
Result
Parent 1: 8 4 7 3 6 2 5 1 9 0
Parent 2: 0 1 2 3 4 5 6 7 8 9
[order crossover]
Child 1:  0 4 7 3 6 2 5 1 8 9
Child 2:  8 7 2 3 4 5 6 1 9 0
```

Mutation

The most common mutation operators for permutation chromosomes are [1,2,3,7]:

1. *Inversion:* this operator selects two random genes along the chromosome and reverses the segment between these two genes. According to [7], “it is particularly well-suited for the TSP and for all the problems that naturally admit a permutation representation in which adjacency among elements plays an important role.”

Scramble: this operator randomly reorders the genes between two randomly selected points. Another implementation is possible: randomly chosen genes are randomly reordered while keeping the rest chromosome genes in the absolute order.

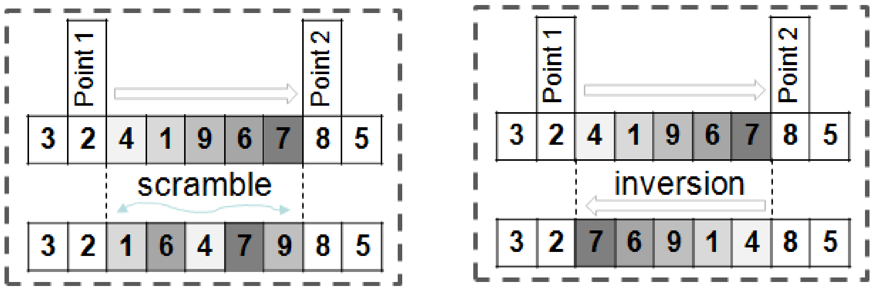

Fig. 4. Scramble and inversion mutation operators.

2. *Insert:* this operator randomly selects one gene and inserts it before or after a different randomly selected gene in the chromosome (see figure 5, point 1 and point 2 respectively). These operators are used for scheduling problems in which relative order of elements is important.

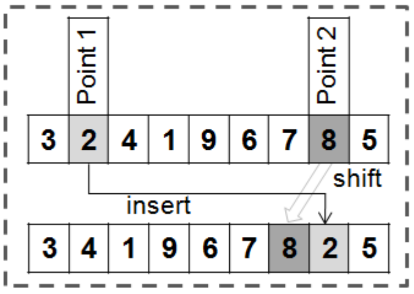

Fig. 5. Insert mutation operator.

3. *Swap and adjacent swap (two-element swap):* the swap operator selects two genes and swaps them. The adjacent swap swaps two neighbor genes.

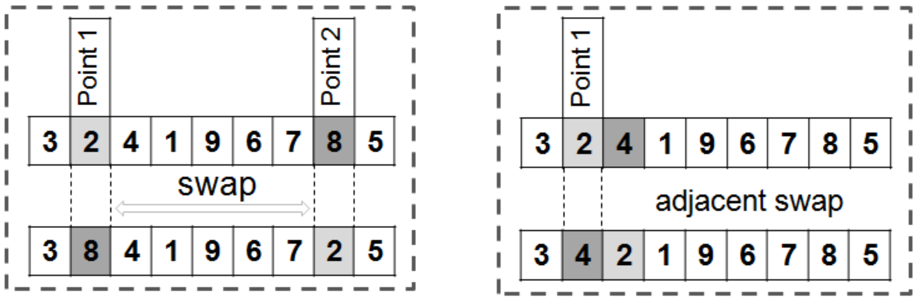

Fig. 6. Swap and adjacent swap mutation operators.

Mutation is a non-deterministic operator with a given probability distribution, and it can generate an offspring more than one mutation operator away from the parent.

Real Case Application

Figure 7 below illustrates the real-case problem solved by GA with chromosome permutation representation.

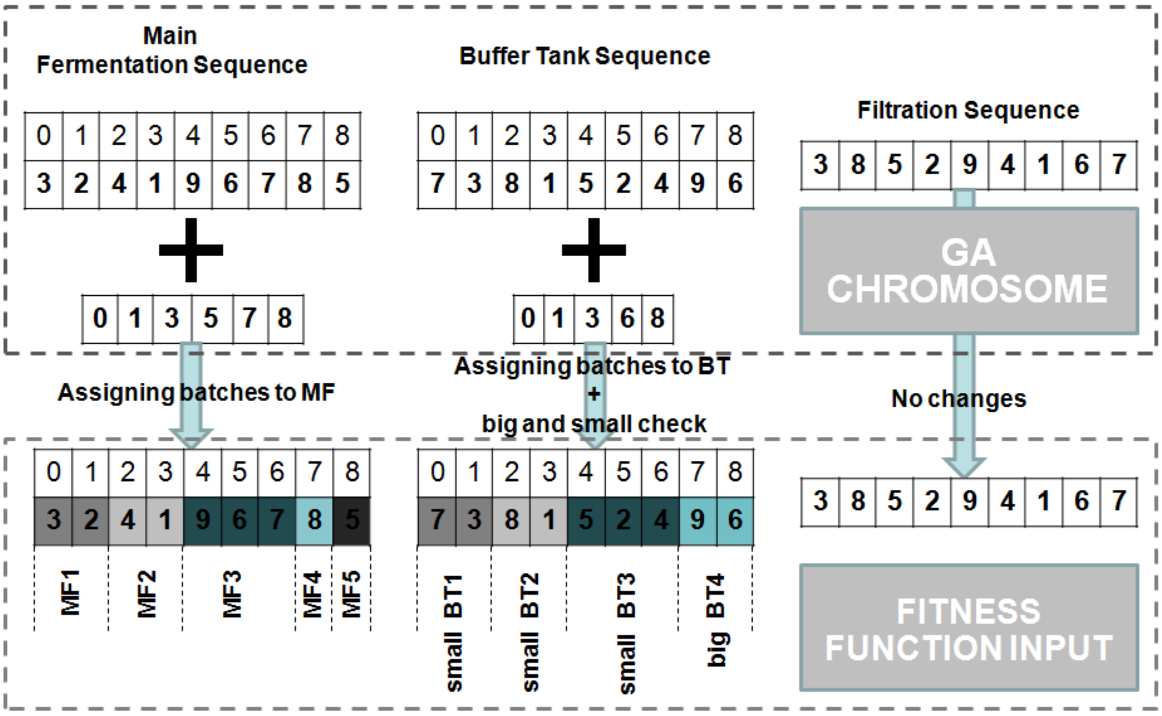

Fig. 7. Example of real-case permutation chromosome implementation.

Conclusion and Future Work

GA is a widely used and powerful optimization method. It has been very successful when applied to problems that can be coded naturally as binary strings. However, ordering problems are more naturally coded as ordered lists and there is no standard GA for manipulating such representations. One of the important problems is to design a suitable crossover operator together with an effective mutation. We suggest that it is possible to start with universal crossover and mutation operators such as the ones described in the current paper and then modify them to be problem-specific by taking into account genes relations. Problem specific information can not only make the solution quality better but also improve the performance of GA.

For the further work it is reasonable to make evaluation criteria for crossover and mutation operators in order to make them predictable, which should lead to a theory, such as described in [7].

References

(1) *Abido, M. A., Elazouni, Ashraf M. Precedence-Preserving GAs Operators for Scheduling Problems with Activities' Start Times Encoding // Journal of Computing in Civil Engineering, Vol 24, Issue 4, pp. 345-356*

(2) *T. Bäck, D. B. Fogel, T. Michalewicz (eds.) (2000) Evolutionary computation 1: Basic algorithms and operators // Institute of Physics Publishing.*

(3) *T. Back, D. B. Fogel, and Z. Michalewicz (eds.) (1997), Handbook of evolutionary computation, Oxford press.*

(4) *L. Davis. (1985) Job-shop Scheduling with Genetic Algorithms // Proceedings of an International Conference on Genetic Algorithms and Their Applications, pp. 136-140.*

(5) *Gorelik V., Fomina T. (2004) Fundamentals of the Operations Research // Textbook for university students. Moscow State Pedagogical University in association with Lipetsk State Pedagogical University, 248 p. (in Russian)*

(6) *C. Moon, J. Kim, G. Choi and Y. Seo. (2002) An efficient genetic algorithm for the traveling salesman problem with precedence constraints // European Journal of Operational Research 140, pp. 606-617.*

(7) *A. Moraglio (2007) Towards a Geometric Unification of Evolutionary Algorithms / PhD Thesis, Department of Computer Science, University of Essex, 392 p.*

(8) *I. Oliver, D. Smith, and J. Holland (1987) A study of permutation crossover operators on the traveling salesman problem // Proceedings of the Second International Conference on Genetic Algorithms, pp. 224–230.*

(9) *Pinedo Michael (2008) Scheduling: Theory, Algorithms and Systems. Springer: Third Edition. 671 p.*

(10) *P.W. Poon, J.N. Carter (1995) Genetic Algorithm Crossover Operators for Ordering Applications // Computers&Operations Research, Vol. 22, pp. 135-147.*

(11) *Zhao Wei (1987) Zhao Wei, Krithi Ramamritham. Simple and Integrated Heuristic Algorithms for Scheduling Tasks with Time and Resource Constraints // The Journal of Systems and Software, Vol. 7, pp. 195-205.*