BULETINUL Universităţii Petrol – Gaze din Ploieşti	Vol. LXI No. 2/2009	67 - 74	Seria Matematică - Informatică - Fizică

Efficient Implementations of Some Genetic Mutation Operators for the Permutation Encoding in Scheduling

Simona Nicoară

Petroleum –Gas University of Ploieşti, Informatics Department, Bd. Bucureşti, 39, Ploieşti
e-mail: snicoara@upg-ploiesti.ro

Abstract

A genetic algorithm is a time consuming technique, especially for the big complex problems. Therefore, any run-time optimization regarding applying the genetic operators is very useful, taking into account that these operators apply many times in every generation. In this paper we propose run-time efficient implementations for three well-known mutation operators, specific to the permutation encoding needed in the vast area of scheduling problems. These operators are: frame-shift, translocation and inversion.

Key words: *genetic algorithm , scheduling, frame-shift, translocation operator, inversion operator*

Introduction

A genetic algorithm is a simulation of the natural process of evolution, where a population of abstract representations, named chromosomes, of the candidate solutions named individuals, for an optimization problem, evolves towards better solutions.

The algorithm starts with an initial pseudo-random population of candidate solutions, which improves in many generations (hundreds or thousands) by applying to the candidate solutions, in every generation, three genetic operators: selection of parents for crossover, crossover for every pair of parents to obtain offspring and mutation of some few individuals to preserve the population diversity (see figure 1).

The first genetic algorithm, depicted in the figure 1, was designed by Holland in 1975 [3] and constitutes the kernel of the many future genetic algorithms that contain supplementary mechanisms to treat various types of difficult search spaces and various types of problems.

The goal of such an algorithm is to find the optimal solution(s) in the search space, generally hard to explore by other techniques. To make functional a genetic algorithm, the main aspects to manage are: the genetic encoding for the candidate solutions, the implementation of the genetic operators and the evaluation function for the individuals.

Every genetic algorithm needs for a good performance efficient implementations for the genetic operators, so that the overall run-time of the algorithm to be minimal. In the theory, the designed genetic operators models are defined either independent of the genetic encoding, either specific to some genetic encodings but regardless to the practical implementation and its influence upon the efficiency level of the algorithm, mainly from the run-time perspective.

```
1. $t$ <- 0 (first generation)
2. pseudo-random initialization of the initial population $P_t$
3. evaluate($P_t$)
4. while evolution is not ended
    4.1. $t$ <- $t$ + 1
    4.2. selection in $P_t$
    4.3. crossover of parents selected
    4.4. insert the descendents in the new population $P'_t$
    4.5. mutation for $P'_t$
    4.6. evaluate $P'_t$
    4.7. $P_t$ <- $P'_t$
5. return the best solutions in $P_t$
```

Fig. 1. The genetic algorithm framework

In this paper we propose some run-time efficient implementations for three well-known mutation operators specific to the permutation encoding needed in the vast area of scheduling. These operators are: frame-shift, translocation and inversion.

Genetic Encoding for Scheduling Problems

One major class of optimization problems is formed by the scheduling problems, where the goal is to optimally allocate limited resources over time to perform a collection of tasks. The scope of this paper is Minimum Job Shop Scheduling Problem (JSSP), as it is the most complex framework for scheduling problems.

In JSSP, a finite set of different structurally heterogeneous jobs must be processed on a finite set of machines with a minimum makespan, while satisfying non-preemption, temporal and resource constraints [4]. In other words, the operations must not be interrupted once allocated to the machines, the operations in the jobs must be scheduled satisfying a fixed order and every machine processes one operation at a time. For each operation is specified the machine which processes it and the corresponding processing time. The solution is an optimal schedule (a sequence of the operations and start processing times for each operation).

When using genetic algorithms to solve JSSP, we need a genetic encoding for the candidate solution (the schedule). An adequate genetic encoding is a sequencing of all the operations to be processed (the genes of the chromosome), consisting in a permutation of the operations set. The table of start processing times for the operations is determined by an external algorithm having as input this permutation. The algorithm returns the start processing times by decoding the permutation in semi-active schedule or active schedule or non-delay schedule [4].

In the permutation string we choose to designate the operations as form (job_i, $operation_j$), meaning the j-th operation of the job i. A possible candidate solution for a problem with minimum 54 jobs is this:

$$(7,1) \ldots (22,3) \ldots (53,1) \ldots (31,5) \ldots (54,8) \ldots (12,7).$$

The feasibility of a candidate solution is judged only from the precedence constraints point of view, because the decoding of the candidate solution is made so that the other constraints to be satisfied.

Run-time Efficient Implementations for Some Mutation Operators

The genetic encoding enforces, for every problem, adequate crossover and mutation operators, to ensure that the returned results keep the same genetic encoding.

For the permutation genetic encoding, the most used crossover operators are: UX, PPX, PMX, SXX, OX, GOX, GPX, THX, taskCrossover, MSXF [1, 5, 6]. The mutation operators specific to the permutation encoding are: frame-shift, translocation, inversion, PBM, OBM and SBM.

Very few of these operators, as they are described in theory, ensure the feasibility of the result (the offspring or, respectively, the mutated schedule) and, therefore, every time when apply crossover and mutation we must check the result and, if unfeasible, we must execute one of the following steps:

- apply a validation (legalization) algorithm for the unfeasible result;
- repeat to apply the operator for the same or different input data until obtain a feasible result;
- discard the current applying.

This is a time-consuming task even for a single generation. For the entire evolution sustained by the genetic algorithm, all these trial and error tasks lead to a huge overall run-time.

In the following we submit some implementations for three mutation operators - frame-shift, translocation and inversion - so that the run-time efficiency of the genetic algorithm increases. This effect emerges because disappears the steps needed to check the mutated chromosome feasibility and, consequently, the steps needed to treat the unfeasible ones. The run-time efficiency increase is obvious because a mutation operator applies many times in every generation during many generations (hundreds, thousands or even billions). We implement the frame-shift operator to guarantee the feasibility of the result, and the translocation and the inversion operators to extensive reduce the run-time by removing the many repetitions frequently needed for random generation of the involved variables so that they be valid.

Implementation for the Frame-shift Operator

The frame-shift operator moves a randomly selected gene of the chromosome (operation in the schedule), over *k* positions, back or forth. Hence we deal with forward method and, respectively, backward method. The forward method acts like this [2]:

if the chromosome before applying the operator is:

$$c = c_1c_2\ldots c_{p-1}\underline{c_p}(c_{p+1}\ldots c_{p+k})c_{p+k+1}\ldots c_{\dim}\,, \tag{1}$$

then, the chromosome after applying will be:

$$c' = c_1c_2\ldots c_{p-1}(c_{p+1}\ldots c_{p+k})\underline{c_p}c_{p+k+1}\ldots c_{\dim}\,. \tag{2}$$

In other words, the operator shifts the gene c_p over the segment $(c_{p+1}\ldots\ c_{p+k})$. Here, to simplify the understanding, we used c_i as the *i*-th gene in the chromosome.
This operator is able to increase the efficiency of the genetic algorithm at the run-time level if its implementation ensures that the operator result (the mutated chromosome) is always feasible.

The input chromosome for every mutation operator is always feasible, otherwise it would not be part of the population. The implementation proposed in the following for the frame-shift operator guarantees to obtain a feasible result from a feasible input. A feasible result is one that satisfies the precedence constraint, i.e. the operations of every job maintain the mandatory order; in other words, operations 1, 2, …, n_i of the job *i* occur in the chromosome in the order $(i,1)$, $(i,2)$, …, (i,n_i), for every job *i*. This constraint may be violated in the frame-shift only by shifting the selected gene to a wrong position, breaching the order of operations in the corresponding job. The other jobs are not affected by the shifting.

As an example, if the chromosome is

(1,1) (5,1) … (5,3)(24,1)(4,3)(12,7)(65,6)(5,4) ... (77,10) and the randomly selected operation is (5,3), then the only way the mutated chromosome becomes unfeasible is by shifting the operation (5,3) beyond the operation (5,4). This will violate the imposed order of the operations in the job 5.

The position of the selected gene, *p*, is randomly generated in {1, …, *dim*-1}, because all the genes placed on the positions 1, …, *dim*-1 can shift to the right so to alter the input chromosome (step 1 of the procedure depicted in the figure 2). The form (1) of the input chromosome leads to the fact that *k*, the number of positions to shift over, is in {1, …, *dim-p*} because it is mandatory the relation $p + k \leq dim$: a gene can shift over maximum number of the positions in its right side. At extreme, if $p = 1$ (the gene to shift is the first in the chromosome) it can shift over 1, 2, …, *dim*-1 positions. If $p = dim$-1 (the gene to shift is the prior to the last in the chromosome), it can shift over 1 position. In the step 1 of the procedure we repeat the generation for *p* while the next operation is exactly the next operation of the job; for example, if we have … (5,3)(5,4) …, it is obvious that the operation (5,3) can not shift to the right so that the result remains feasible. In this case, $k = 0$ and therefore the generated *p* is not useful – it not produces any alteration of the input chromosome.

Once we have generated a gene that can usefully shift to the right ($k \geq 1$), we determine *kmax* (step 3). By the proposed procedure we narrow the range for *k* from {1, …, *dim* - *p*}, which can lead to unfeasible results, to the range {1,…, *kmax*}, where *kmax* is the maximal number of positions over that the gene can shift. This right bound remains *dim-p* if the operation in the position *p* is the last operation in its job – it can shift over maximum the last operation in the chromosome. But the range {1, …, *kmax*} is reduced, generally in a large proportion, if the operation is not the last operation of the job, and this happens the most times. The *kmax* in this case is the number of positions between *p* and the position of the next operation of the same job. In the previous example, the operation (5,3) can shift over maximum 4 positions so that the precedence constraint is not violated. Hence, *kmax* is 4 and *k* will be randomly generated in {1,…,4}.

```
1. repeat
     random generate p in{1,...,dim-1}
     i <- job_associated_to($c_p$)
   while i = job_associated_to($c_{p+1}$)
2. j <- operation_associated_to(g)
3. if j is the last operation of the job
     kmax <- dim-p
   else
     nextp <- the position of the next operation of the job i
     kmax <- nextp-p-1
4. random generate k in {1,...,kmax}
5. shift the gene from the position p over k position to the right
6. return the new chromosome
```

Fig. 2. The proposed implementation procedure for the frame-shift operator

Having the value of *kmax*, we randomly generate *k* in {1,…, *kmax*} (step 4) and apply the shift to the right of the operation in position *p* (step 5).

To give a full example, in the figure 3 we describe the major stages in applying the implementation procedure for the frame-shift operator. In the final stage, we note that the depicted precedence relation was preserved in the result.

We designed this procedure for the forward method of the frame-shift. The backward method can be easily tailored to the first one.

The efficiency of the proposed implementation emerges from the total time saved in every frame-shift application in every generation. The result being always feasible, is not needed anymore a procedure to check the feasibility and none supplementary procedure to deal with this unfeasibility. Therefore, the overall run-time of the genetic algorithm decreases, leading to an optimization of the algorithm in the perspective of the run-time.

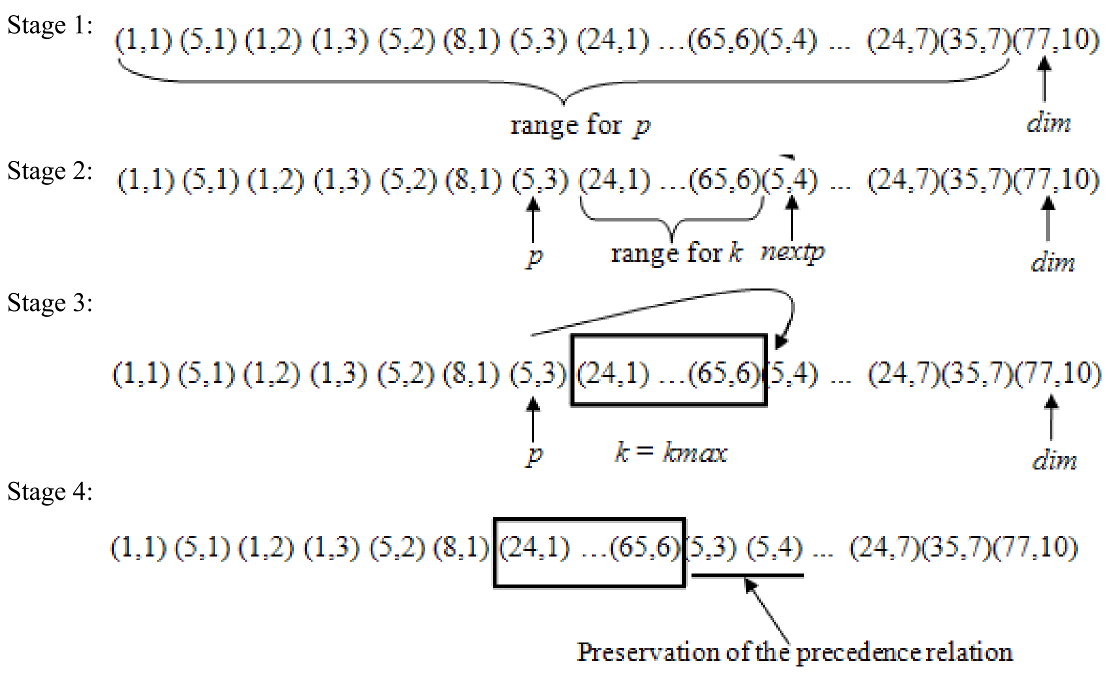

Fig. 3. A test for frame-shift operator implementation to guarantee the result feasibility

Implementation for the Translocation Operator

The translocation operator interchanges identical length chromosomial segments from a position to another [2]:

if the chromosome before applying the operator is:

$$c = c_1 ...(c_p ...c_{p+k-1})...(c_q ...c_{q+k-1})...c_{\dim} \tag{3}$$

then the chromosome after applying will be:

$$c' = c_1 ...(c_q ...c_{q+k-1})...(c_p ...c_{p+k-1})...c_{\dim} . \tag{4}$$

For this operator we restrict the positions of the interchanged segments to reduce the probability of unsuccess. The code sequence for interchange a genetic segment of length *k* from the position *p* to the position *q* is described in the code in figure 4.

Here, if the chromosome has one or two genes, the values for *k*, *p* and *q* are obvious. Otherwise, the length of the segment to interchange is forced not exceed *dim*/3, because the mutation has to induce a small alteration of the initial chromosome, similarly to the natural evolution. The maximum value for the start position of the left segment, *p*, must allow the right segment, with the same length *k*, to have space to its right side. So, this maximum value is dim-2*k+1.

Having *p* and *k*, we can hereinafter generate the start position of the right segment, *q*, accordingly to these values. The right segment must start somewhere after the position where the left one ends (so $p+k \leq q$), and so to ensure that after this position, in the chromosome there are at least *k* positions (so $q \leq dim\text{-}k+1$}. Afterwards, we interchange the two segments defined by *p*, *q* and *k*.

```
if dim <= 2
     k <- 1
     p <- 1
     q <- dim
else
     random generate k in {1,...,dim/3}
     random generate p in {1,...,dim-2*k+1}
     random generate q in {p+k,...,dim-k+1}
interchange the segments ($c_p$ ... $c_{p+k-1}$) and ($c_q$ ...$c_{q+k-1}$)
```

Fig. 4. The proposed implementation procedure for the translocation operator

Figure 5 presents an example on how works the translocation operator in our implementation.

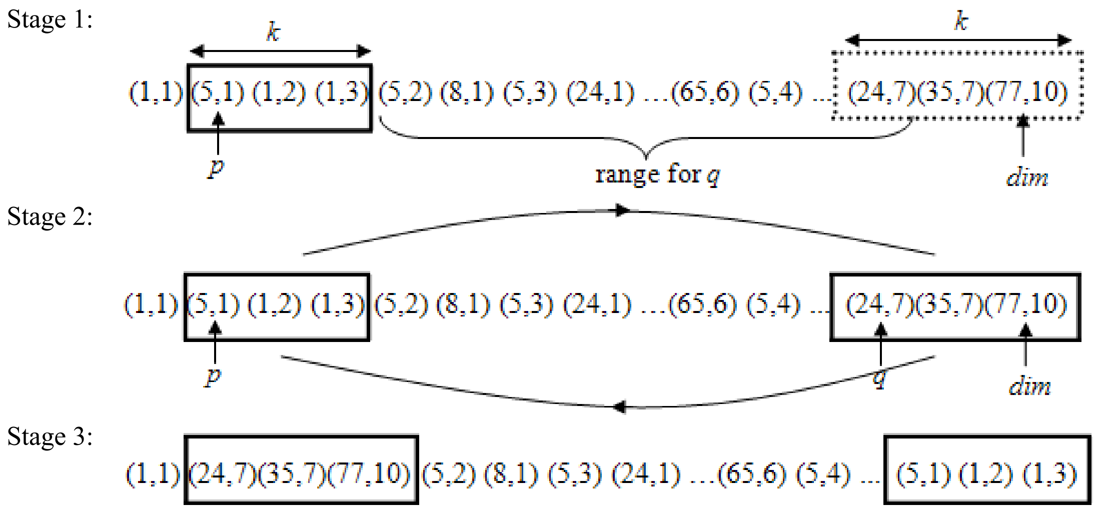

Fig. 5. A test for the translocation operator implementation, where $k = 3$

In the case of the translocation operator, the result is not mandatory feasible; in the example in figure 5, the operation (5,1) occurs in the chromosome after the operation (5,2). But reducing the variation ranges for *p* and *q* with respect to *k* and for *q* with respect to *p* also, we decrease the run-time of the algorithm. This is done by removing the many repetitions frequently needed for random generation of the tuple (*k*, *p*, *q*) so that they to be valid, i.e.:

$$1 \le k \le dim/3,\ 1 \le p \le dim\text{-}2*k\text{+}1,\ p\text{+}k \le q \le dim\text{-}k\text{+}1.$$

Implementation for the Inversion Operator

The inversion mutation was proposed by Holland, observing that the function of a gene is independent of its location on the chromosome. The operator reverses the order of genes in a segment of length *k*, between two random positions, *p* and *q*. Our implementation generates *p* and afterwards *k,* dependent to *p*, accordingly to the code sequence in the figure 6. The first position of the segment to be reversed, *p*, may be in {1, …, *dim*-1} to ensure that the segment has at least 2 genes; a segment formed by a single gene, when reversed, returns an identical chromosome. This condition imposed to *p* ensures therefore that $k \ge 1$.
Then, the proposed procedure determines the maximal value of *k* so that, for the operation placed on the position *p*, the next operation in the job is not contained in the segment to reverse. Hence, for this job, we ensure the precedence preservation. But this is not applicable also for the other jobs having operations in the selected segment. Consequently, the implementation does

not ensure the feasibility of the result, but decreases the run-time of the algorithm by reducing the variation range for *p* with respect to *k.* This is done by removing the many repetitions frequently needed for random generation of the tuple (*p*, *k*) so that they to be valid.

```
1. repeat
      random generate $p$ in{1,...,$dim$-1}
      $i$ <- job_associated_to($c_p$)
   while $i$ = job_associated_to($c_{p+1}$)
2. $j$ <- operation_associated_to($g$)
3. if $j$ is the last operation of the job
      $kmax$ <- $dim$-$p$
   else
      $nextp$ <- the position of the next operation of the job $i$
      $kmax$ <- $nextp$-$p$-1
4. random generate $k$ in {1,...,$kmax$}
5. if p < q
     reverse the genes in the range $p..q$
6. return the new chromosome
```

Fig. 6. The proposed implementation procedure for the inversion operator

In figure7 we have an example on how works the inversion operator in our implementation.

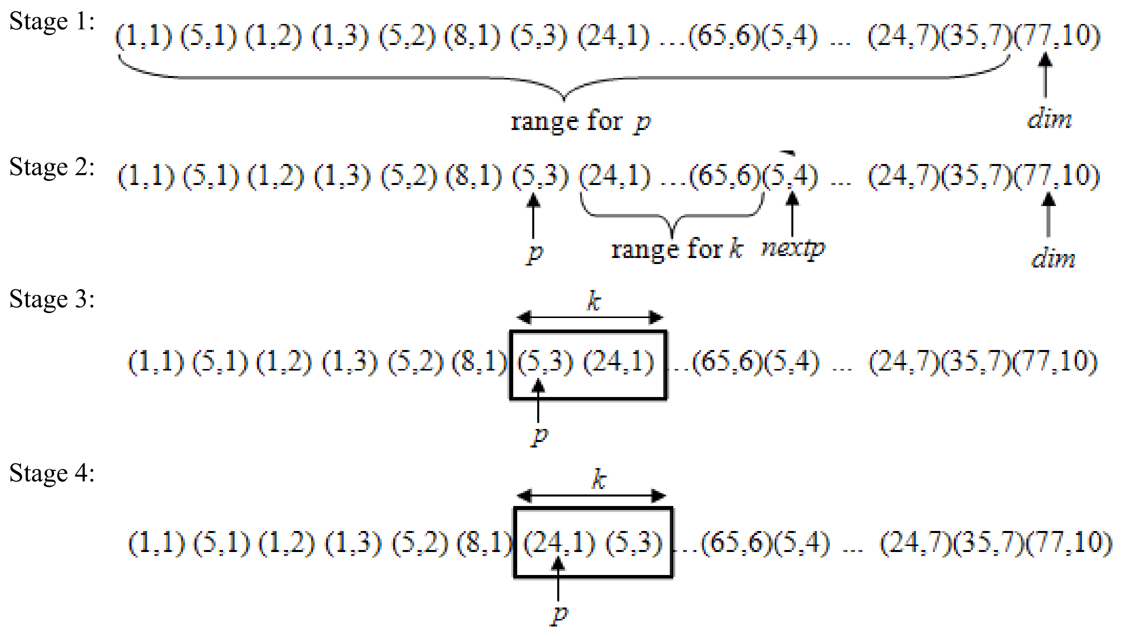

Fig. 7. A test for the inversion operator implementation, where $k = 2$

The result of the operator is not mandatory feasible, but, thanks to this implementation, the number of unsuccessful trials is much lower than if we do not impose no constraints on *p* and *k*.

Conclusions

The genetic algorithms search the optimal solution(s) of a problem by repeatedly applying the genetic operators to the population of candidate solutions. In every generation, the operators (selection, crossover and mutation) apply to many candidate solutions. And the number of generations is, ordinarily, of order of thousands. Moreover, very few operators ensure the feasibility of the result and, therefore, every time the operator applies, is necessary to check the feasibility of the result and, if unfeasible, to overtake some additional actions. Another aspect is that many genetic algorithms involve some supplementary mechanisms, also time-consuming. Consequently, for a good run-time performance of the algorithm, it is a great help to optimize the implementations of the operators and mechanisms.

In the paper we submitted three run-time efficient implementations of three mutation operators. For the frame-shift operator, the procedure certifies the feasibility of the result, and therefore it is not needed anymore a feasibility check stage and no procedure to either validate the unfeasible result, either repeat the applying until obtaining a feasible result. As a consequence, the overall run-time of the algorithm decreases. For the other two operators, the translocation and the inversion, the proposed implementations eliminate the frequent repetitions needed for the random generations of the involved variables so that they to be valid. This is done by reducing the variation ranges for these variables and leads to a shorter run-time of the algorithm.

The designed implementations are easily to adjust for any permutation encoding, different from that used for JSSP.

References

1. Chen, S., Smith, S.F. - Improving genetic algorithms by search space reductions (with applications to flow shop scheduling), *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)* 99, Morgan Kaufmann. pp. 135-140, 1999
2. De Falco, I., Della Cioppa, A., Tarantiono, E. - Mutation-based genetic algorithm: performance evaluation, *Applied Soft Computing* 1(4), pp. 285-299, 2002
3. Holland, J.H. - *Adaptation in natural and artificial systems*, MIT Press, 1975
4. Jensen, M.T. - *Robust and flexible scheduling with evolutionary computation*, doctoral thesis, Dissertation Series DS-01-10, Aarhus University, Denmark, 2001
5. Kobayashi, S., Ono, I., Yamamura, M. - An efficient genetic algorithm for Job Shop Scheduling Problems, *Proceedings of ICGA'95*, pp. 506-511, 1995
6. Lin, S.C. et al. - Investigating Parallel Genetic Algorithms on JSSP, *Proceedings of the sixth International Conference on Evolutionary Programming*, Springer Verlag, pp. 383–394, 1997
7. *** - A compendium of NP optimization problems, in *Complexity and approximation combinatorial optimization problems and their approximability properties*, by Ausiello, G., Crescenzi, P., Gambosi, G., Kann, V., Marchetti-Spaccamela, A., Protasi, M., Springer Verlag, 2004

Implementări eficiente ale unor operatori de mutație genetică pentru codificarea permutare în planificare

Rezumat

Un algoritm genetic este o tehnică aproximativă de optimizare mare consumatoare de timp, în special pentru problemele complexe de mari dimensiuni. În consecință, orice optimizare cu privire la aplicarea operatorilor genetici este binevenită, ținând seama că aceşti operatori se aplică repetat la fiecare generație a algoritmului. În lucrare s-au propus nişte implementări eficiente din punct de vedere al timpului de rulare pentru trei operatori de mutație specifici codificării permutare necesare în domeniul vast al planificării JSSP. Aceşti operatori sunt: frame-shift, translocare şi inversiune.