Expert Systems With Applications 151 (2020) 113381

Contents lists available at ScienceDirect

Expert Systems With Applications

journal homepage: www.elsevier.com/locate/eswa

A crossover operator for improving the efficiency of permutation-based genetic algorithms

Behrooz Koohestani

Faculty of Electrical and Computer Engineering, University of Tabriz, Tabriz, Iran

ARTICLE INFO

Article history:
Received 22 August 2019
Revised 10 March 2020
Accepted 12 March 2020
Available online 13 March 2020

Keywords:
Genetic algorithms
Chromosome representation
Crossover operators
Combinatorial optimisation problems
Traveling salesman problem

ABSTRACT

Crossover is one of the most important operators in a genetic algorithm by which offspring production for the next generation is performed. There are a number of crossover operators for each type of chromosome representation of solutions that are closely related to different types of optimisation problems. Crossover operation in genetic algorithms, aimed at solving permutation-based combinatorial optimisation problems, is more computationally expensive compared to other cases. This is mainly caused by the fact that no duplicate numbers are allowed in a chromosome and therefore offspring legalisation is needed after each substring exchange. Under these conditions, the time required for performing crossover operation increases significantly with increasing chromosome size, which may deeply affect the efficiency of these genetic algorithms. In this paper, a genetic algorithm that uses path representation for chromosomes and benefits from an alternative form of the well-known partially mapped crossover is proposed. The results of numerical experiments performed on a set of benchmark problems clearly show that the use of this crossover operator can significantly increase the efficiency of permutation-based genetic algorithms and also help in producing good quality solutions.

1. Introduction

Genetic Algorithms (GAs) (Holland, 1992) are a special class of bio-inspired methods simulating Darwin's theory of evolution and natural selection. According to this theory, the more an organism is fit, the more it can survive in the nature and vice versa (i.e., the survival of the fittest). The GAs are, in fact, informed search methods capable of exploring a search space of solutions in order to find an optimal or near optimal solution for a particular problem. These search methods have considerable power and potential for dealing with a wide variety of hard problems in various fields.

In a GA, the search space is formed from individuals and each individual represents a possible solution to a particular problem. These individuals are encoded as chromosomes. The main genetic operators including crossover and mutation are then applied probabilistically to the chromosomes chosen by a selection method with the aim of creating new and potentially better solutions (Eiben & Smith, 2015).

Crossover is responsible for producing offspring by recombining parents' genetic material. The main aim of crossover operation is to enable GAs to more effectively explore the search space and produce better individuals in the succeeding generation. Mutation is also often required to explore new states, and it helps GAs to avoid local optima. These operations usually result in finding a globally optimal or near-optimal solution for a given problem (Eiben & Smith, 2015).

There are different types of chromosome representations with their associated operators, which are closely related to different types of optimisation problems (Umbarkar & Sheth, 2015). In this study, the focus is particularly on an important class of combinatorial optimisation problems whose solutions can be represented with a permutation. In such problems, the task is to arrange some objects, with no duplicates, in a certain order such that an objective function is optimised. The Travelling Salesman Problem (TSP), Bandwidth-Reduction Problem (BRP) and Linear Ordering Problem (LOP) are examples of permutation-based combinatorial optimisation problems (Papadimitriou & Steiglitz, 2013).

Permutation-based GAs are GAs that use path representation for chromosomes. In path representation, a solution is encoded as a permutation of a set of integers. This is the most common representation of chromosomes for permutation problems (Du & Swamy, 2016). A number of key studies on the application of GAs to permutation problems have used the path representation and proposed crossover and mutation operators specific to this representation, some of which are reviewed in the next section.

In this study, we propose a permutation-based GA for addressing combinatorial optimisation problems. The key feature of this

E-mail address: b.koohestani@tabrizu.ac.ir

https://doi.org/10.1016/j.eswa.2020.113381
0957-4174/

GA is that it incorporates a new version of the well-known Partially Mapped Crossover (PMX). In order to evaluate the efficiency and effectiveness of our crossover operator, two sets of experiments are carried out on standard benchmark problems. The results obtained are very satisfactory and clearly show the capability of the approach.

The remainder of this paper is organised as follows: Section 2 gives a brief overview of GAs for permutation-based combinatorial optimisation problems. In Section 3, we describe in detail our proposed GA. Section 4 presents a step-by-step implementation of the new version of the PMX. In Section 5, we present the experiments carried out to evaluate the performance of our approach. Finally, conclusions are given in Section 6.

2. Background

2.1. Combinatorial optimisation

Combinatorial optimisation is a discipline aiming at finding an object (e.g., a graph) that minimises or maximises a specific function from a finite set of mathematical objects. In combinatorial optimisation problems, variables are typically discrete. Permutation-based combinatorial optimisation problems are an important class of combinatorial optimisation problems whose solutions are represented as permutations. The TSP is one of the most well-known problems in this class and consists of discovering a tour (a cyclic permutation) visiting a set of cities (each city exactly once) that minimises the total travel distance. Such problems are generally very hard to solve because their space of possible solutions is typically very large. In terms of the TSP, there are $(N-1)!/2$ distinct solutions for N cities, because the starting city is defined and each route has an equal route in reverse with the same length (Kirkpatrick & Toulouse, 1985). For example, for only 10 cities, there are 181440 distinct routes. Therefore, no exact algorithm may be feasible to cope with these kinds of problems, and search algorithms must be employed (Onwubolu & Davendra, 2009).

2.2. Permutation-based GAs

GAs are highly parallel search algorithms, inspired by Darwin's theory of evolution and natural selection that transform a population of encoded candidate solutions (also called chromosomes), each with an associated fitness value, into a new population using genetic operations (i.e., selection, crossover, and mutation).

Chromosome representation methods used for combinatorial optimisation problems can generally be divided into five major types including path representation, adjacency representation, binary representation, matrix representation and ordinal representation (Larranaga et al., 1999; Pavai & Geetha, 2016). Based on these representations, a number of crossover and mutation operators have been designed to suit different needs.

In terms of permutation-based combinatorial optimisation problems, the path representation is considered to be the most important and widely used chromosome representation method in which a solution is encoded as a permutation of a set of integers (Pavai & Geetha, 2016). For example, in a 7-city TSP, the tour 4-7-5-1-6-3-2 is represented by a chromosome in the form of order (4 7 5 1 6 3 2). Permutation-based GAs are GAs that use path representation for chromosomes. A high-level description of the main operations in permutation-based GAs is given in Algorithm 1.

Several key studies on the application of GAs in solving permutation problems have employed the path representation and defined suitable crossover and mutation operators for this representation, some of which are as follows.

Algorithm 1 The main operations in permutation-based GAs.

1: Randomly generate an initial population of chromosomes (i.e., candidate solutions encoded as permutations of a set of integers).
2: **repeat**
3: Calculate the fitness value of each chromosome in the current population.
4: Use a selection method (e.g., tournament selection) to choose chromosome(s) from the population based on fitness value.
5: Create a new population of chromosomes by applying genetic operators such as crossover and mutation (i.e., designed for path representation) to the selected chromosomes with specified probabilities.
6: Replace the current population with the new population created in the previous step.
7: **until** the termination criterion (e.g., the maximum number of generations) is satisfied.
8: **return** the chromosome with the best fitness value in the last generation.

Davis (1985) proposed the Order Crossover. This operator generates an offspring by selecting a sub-string of the first parent and maintaining the relative order of the components of the second parent. Goldberg and Lingle (1985) presented Partially Mapped Crossover (PMX) which is the most widely used permutation-based crossover operator. In the PMX, a sub-string of the first parent is mapped onto a sub-string of the second parent and the remaining information is swapped. The Cycle Crossover was suggested by Oliver et al. (1987). It produces an offspring from the parents such that each of the positions is filled with a corresponding component from one of the parents. Contrary to the previous crossover operators, the Heuristic Crossover (Grefenstette, 1987) emphasises edges. In other words, it exploits the length of the edges. The Position Crossover (Syswerda, 1991) is another permutation-based crossover, closely related to the order crossover and designed for scheduling problems.

The Genetic Edge Recombination Crossover was introduced by Whitley et al. (1991). This operator which is more appropriate for the symmetrical TSP, assumes that only the weights of the edges are of importance, not their directions. The Voting Recombination Crossover (Mühlenbein, 1991) is a p-sexual crossover operator, where p is a positive integer and $p \geq 2$. Eiben et al. (1994) introduced two multi-parent crossover operators in which more than two parents are employed in the recombination operation. The Alternating-position Crossover was developed by Larranaga et al. (1997). It produces an offspring by choosing alternately the next component of the first parent and the next component of the second parent, and removing the components already present in the offspring. The Complete Subtour Exchange Crossover (Katayama & Narihisa, 2001) is specifically designed for permutation problems and operates with path representation.

Misevičius and Kilda (2005) discussed the features and characteristics of different crossover operators in the context of the quadratic assignment problem and showed that the multi-parent crossover could achieve better solutions than other operators tested. Mumford (2006) introduced some new order based crossover operators for the graph coloring problem. Ting et al. (2010) proposed multi-parent extension of the PMX (MPPMX) for combinatorial optimization problems and demonstrated the notable advantage of the MPPMX over PMX. Abdoun and Abouchabaka (2012) implemented six different crossover operators and their variations and investigated the effect of these crossover operators on genetic algorithms when applied to the TSP.

Novel crossover operators for solving discrete optimization problems using genetic algorithms were suggested by Agarwal and Singh (2013) and Kumar et al. (2013). Lo et al. (2018) developed a genetic algorithm for addressing Multiple Traveling Salesman Problem. This algorithm incorporates a modified version of the edge recombination crossover and new local operators, which are used to optimise the paths at different scales.

Andrade et al. (2019) proposed a Biased Random-Key Genetic Algorithm (BRKGA) for solving the Permutation Flowshop Scheduling Problem. In BRKGA, each gene in a chromosome is a random real number uniformly generated from the interval [0,1]. In crossover operation, a parameterised uniform crossover scheme is first applied to a chromosome. A decoder is then used to map the altered chromosome into a feasible solution (i.e., a permutation).

Fogel (1988) suggested the Insertion Mutation operator for the TSP. This operator first randomly selects a city in a tour, and then remove that city from the tour and inserts it in a randomly chosen location. In Michalewicz (1992), the Displacement Mutation operator (also called cut mutation) was introduced. This operator first randomly selects a subtour, and then remove that subtour from the tour and inserts it in a randomly chosen location. The Inversion Mutation operator (Fogel, 1990; 1993) is similar to the displacement mutation operator. The only difference is that the selected subtour is inserted in reverse order. Banzhaf (1990) proposed the Exchange Mutation operator (also called swap mutation). It randomly chooses two cities in a tour of the TSP and exchanges them. The Scramble Mutation operator (Syswerda, 1991) was designed for schedule optimization. This operator randomly selects a substring of a chromosome and scrambles its genes. Albayrak and Allahverdi (2011) developed the Greedy Subtour Mutation in order to increase the performance of GAs when applied to the TSP. It employs classical and greedy techniques together with the aid of six different parameters. In Wang and Liu (2013), a new mutation operator is presented in which the inversion and migration operations are added to the mutation process.

3. Proposed GA

Our proposed GA is a permutation-based GA as described in the previous section. It uses path representation (see Section 2.2) for chromosomes and tournament selection (Miller & Goldberg, 1995) for choosing chromosomes. It is worth mentioning that tournament selection is a robust selection mechanism, commonly used by GAs, for selecting individuals from the population and inserting them into a mating pool. In tournament selection, a number of individuals are randomly chosen from the population. Theses individuals are then compared with each other and the best of them is selected to be a parent. The fitness of a chromosome is simply defined as the total travel distance (i.e., since the test problem in this study is the TSP). The exchange mutation (see Section 2.2) is used for mutation operation, and the termination criterion is based on the maximum number of generations. The distinct feature of the proposed GA is its crossover operator, which is specifically designed to increase the efficiency of permutation-based GAs. This crossover operator is an improved version of the well-known PMX, and we name it IPMX (improved PMX).

It is important to note that crossover operation in permutation-based GAs is more computationally expensive in comparison with other cases. The reason is that after each substring exchange, a legalisation process is required for offspring that have duplicate numbers. In such a situation, the time needed for performing crossover operation grows rapidly with increasing chromosome size, and this can have an adverse effect on the efficiency of permutation-based GAs. The motivation for introducing the IPMX is to address the above mentioned issue without affecting solution quality. In the next section, we provide the pseudocode and an effective implementation of the IPMX.

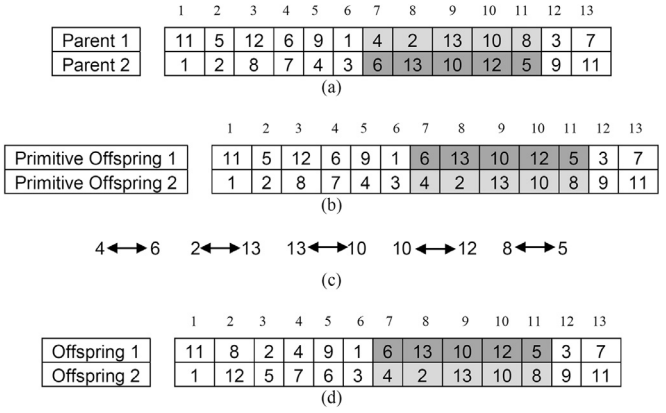

Fig. 1. (a) Substring selection. (b) Substring exchange. (c) Mapping list determination. (d) Offspring legalisation.

4. IPMX

As mentioned in the previous section, IPMX is an improved version of the well-known PMX, specifically designed to be more efficient. Before studying the structure of the IPMX, it is therefore useful to first consider the PMX procedure as described in Algorithm 2.

Algorithm 2 Pseudocode of the PMX.

1: Randomly select two cut points on parent chromosomes.
2: Exchange the substrings between cut points to produce primitive-offspring.
3: Determine the mapping relationship with respect to the chosen substrings.
4: Legalise primitive-offspring using the mapping relationship.

Fig. 1 also illustrates how PMX works in practice. In Step 1, two substrings [4 2 13 10 8] and [6 13 10 12 5] are formed based on randomly chosen two cut points on Parent 1 and Parent 2 shown in Fig. 1(a). In Step 2, these two substrings are exchanged to produce Primitive-offspring 1 and Primitive-offspring 2, which are typically illegal. In step 3, the mapping relationship is determined based on the chosen substrings (e.g., '6' to '4', '13' to '2', '10' to '13' etc). In step 4, the Primitive-offspring 1 and Primitive-offspring 2 are repaired by replacing duplicate genes with the corresponding genes in the mapping relationship, resulting in the creation of legal offspring (i.e., tours).

Here, we also provide the computational complexity of the PMX as follows: Let P_1 and P_2 be two parent chromosomes with length n, and S_1 and S_2 be two substrings between cut points with length m. Further, let $A_1 = S_2 - (S_1 \bigcap S_2)$, $A_2 = S_1 - (S_1 \bigcap S_2)$ and $t = |A_1| = |A_2|$. In fact, A_1 and A_2 are subsets of S_2 and S_1 that make Offspring 1 and Offspring 2 invalid, respectively. Now, it is possible to divide the elements of set $P_1 - S_1$ into two parts. The first one is a set that its intersection with S_2 is $\emptyset$, and the second one is the set A_1. These two parts need $m(n-m-t)$ and $t(m-t) + \frac{t(t+1)}{2}$ comparisons for permanent insertion of $(n-m-t)$ elements into Offspring 1 and identification of t elements, respectively. For permanent insertion of t elements that belong to A_1 into Offspring 1, (αmt) comparisons are required, where α is a real number. Overall, the total number of comparisons and insertions for producing both offspring in a worst case scenario, where $\alpha = 1$, $t = m$ and $m = \beta n$ $(\frac{1}{n} \leq \beta \leq \frac{n-1}{n})$ can be calculated as

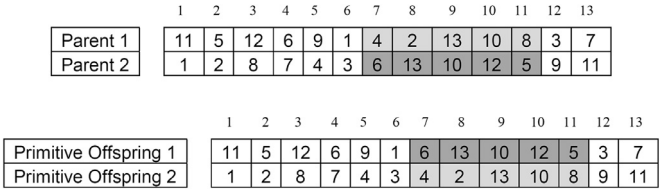

Fig. 2. Two parent chromosomes and their associated primitive-offspring.

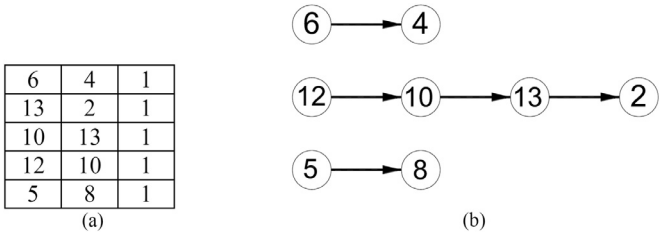

Fig. 3. (a) Exchange list. (b) Directed graph of the exchange list.

$(2\beta - \beta^2)n^2 + \beta n$ that accounts for computational complexity of $O(n^2)$. Thus, the PMX takes quadratic time.

After considering the PMX and its structure, it is now appropriate to introduce the IPMX. A high-level description of the operations of the IPMX is given in Algorithm 3. In the next subsections, a step-by-step implementation of the IPMX procedure is also presented, and a simple illustrative example is used for this purpose.

Algorithm 3 Pseudocode of the IPMX.

1: Randomly select two cut points, cp1 and cp2, on parent chromosomes, parent1 and parent2.
2: Produce primitive-offspring1 and primitive-offspring2 by swapping the substrings between cp1 and cp2.
3: Define an exchange list with respect to the chosen substrings.
4: Generate a directed graph of the exchange list.
5: Find all distinct paths between nodes in the graph.
6: For each path that contains more than two nodes, add an edge between two endpoints and then remove all mid nodes.
7: Update the exchange list, considering the refined paths.
8: Apply the updated exchange list to the primitive-offspring1 to produce offspring1.
9: Generate F, a list of the same length as the parent chromosomes, initialised to zero.
10: Produce offspring2 by performing the following operations:
(a) F[offspring1[i]] =parent1[i]. (b) offspring2[i] = F[parent2[i]].
($i = 1, 2, ..., n$ and n is the length of a parent chromosome)

4.1. Steps 1 and 2

In Fig. 2, two parent chromosomes are shown as examples. The crossover points, cp1 and cp2, are selected to be between elements 6 and 7, and elements 11 and 12 respectively. Primitive-offspring1 and primitive-offspring2 are then produced by swapping the substrings between cp1 and cp2. As it is clear, both primitive-offspring generated are illegal at the moment, because they have duplicate numbers. This is dealt with in the next steps.

4.2. Steps 3 and 4

An exchange list is defined with respect to the chosen substrings, and a directed graph of the exchange list is then generated as illustrated in Fig. 3. Note that the third column added to the exchange list is only for implementation purposes, and a 1 in each row indicates the existence of a directed path from the first element (or node) to the second element (or node). For more information about graph representation methods, data structures and algorithms, see Cormen et al. (2009).

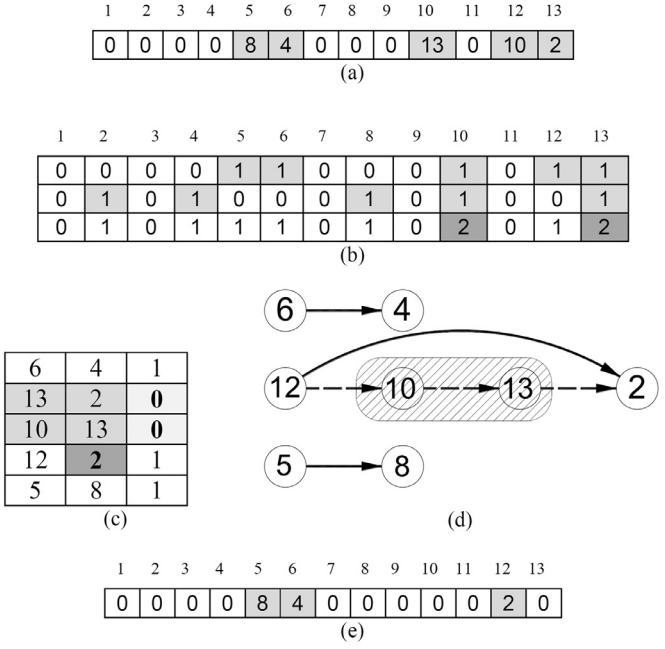

Fig. 4. (a) Guide list. (b) Three lists L1, L2, and L1+L2. (c) Updated exchange list. (d) Directed graph of the updated exchange list. (e) Updated guide list.

4.3. Steps 5, 6 and 7

First, a list that we call it 'Guide List' is generated, and all its entries are initialised to zero. The guide list is then filled by considering the first column of the exchange list as indexes, and the second column of the exchange list as their corresponding values.

Two lists, L1 and L2, are also generated, and all their entries are initialised to zero. Next, the first and second columns of the exchange list are considered as indexes of L1 and L2 respectively, and their corresponding values are set to 1. Finally, a list L1+L2 is generated by adding corresponding entries in L1 and L2. Where an entry in the list L1+L2 is equal to 2, this denotes that its corresponding node (i.e., indicated by the index of that entry) is a mid node. In that case, the exchange list is updated by assigning 0 to the third column of the related node (i.e., removing the mid node). Now, by using the exchange list and guide list, the two endpoints of the path can simply be obtained. These two are then connected by an edge, and a new path is generated. Therefore, the exchange list should be updated accordingly. Finally, based on the updated exchange list, the guide list is also updated. Fig. 4 shows the whole process.

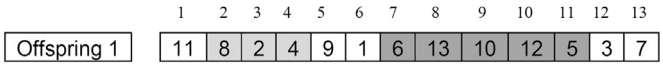

Fig. 5. Offspring1 produced by performing step 8.

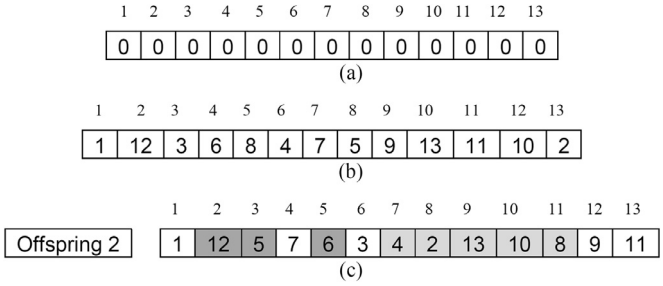

Fig. 6. (a) Initial list F. (b) Determining the elements of list F. (c) Producing offspring2.

4.4. Step 8

In this step, the updated guide list is applied to the primitive-offspring1, and offspring1 is produced as indicated in Fig. 5. By adopting this approach, the offspring1 is generated efficiently with a small number of operations.

4.5. Steps 9 and 10

For producing offspring2, our approach is to not repeat the steps performed for producing offspring1 in order to further speed up this process. First, a list of the same length as the parent chromosomes called 'F' is generated, and all its entries are initialised to zero.

Next, we perform F[offspring1[i]] = parent1[i], where $i = 1, 2, ..., n$ and n is the length of a parent chromosome. This operation is aimed at determining the elements of list F by considering the ith element of offspring1 as an index of F, and the ith element of parent1 as the corresponding value of the index of F.

Finally, we perform offspring2[i] = F[parent2[i]], where $i = 1, 2, ..., n$ and n is the length of a parent chromosome. The purpose of this operation is to produce offspring2 by considering the ith element of parent2 as an index of F, and the corresponding value of the index of F as the ith element of offspring2. Fig. 6 illustrates the steps described above.

Now, it would be interesting to explore the difference between the IPMX and PMX in terms of computational complexity. In the previous section, the computational complexity of the PMX was presented. Here, the computational complexity of the IPMX is provided as follows: Let P_1 and P_2 be two parent chromosomes with length n, and S_1 and S_2 be two substrings between cut points with length m. The number of operations for generating the exchange list and guide list is $3m + m$. The number of operations for generating the lists L1, L2, and L1+L2 is $m + m + n$. Finding entries in the list L1+L2, which are equal to 2 (i.e., indicating mid nodes or nodes with degree 2 in the directed graph of the exchange list), requires n operations. In order to update the guide list, $n + m$ operations are needed. The number of operations for producing Offspring 1 using the updated guide list is n. Finally, producing Offspring 2 by adopting the proposed approach needs $n + n$ operations. Overall, the total number of operations for producing both offspring in a worst case scenario, where $m = \beta n$ ($\frac{1}{n} \leq \beta \leq \frac{n-1}{n}$) can be calculated as $7\beta n + 6n$ or $n(6 + 7\beta)$ that accounts for computational complexity of $O(n)$. Thus, the IPMX takes linear time, which is an advantage over the PMX.

Table 1
Parameters of the runs for experiment set 1.

Parameter	Value
Population size	500
Tournament size	3
Crossover rate	100%
Mutation rate	0%
Number of generations	50
Number of runs	30

5. Numerical experiments

In this section, the experiments carried out to evaluate the performance of the IPMX are presented. The test problem in this study is the TSP. We employed TSPLIB, which is a very well-known library of TSP instances (available from http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/). We also employed VLSI collection containing TSP instances (available from http://www.math.uwaterloo.ca/tsp/vlsi/). The VLSI collection is different from the TSPLIB collection in terms of its discipline and the class of problems. All the algorithms were implemented in C#, and all the experiments were performed on an Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz. In this study, two sets of experiments were performed to assess the impact of using the IPMX as a crossover operator on the performance of permutation-based GAs, as follows.

5.1. Experiment set 1

In experiment set 1, the aim was to test the influence of IPMX on the speed of permutation-based GAs. To determine this, we first selected 20 TSP instances from TSPLIB with sizes ranging from 51 to 3038, and 16 TSP instances from VLSI collection with sizes ranging from 131 to 1083. Overall, the test data used include 36 TSP instances from two different collections (i.e., the TSPLIB and VLSI) that seem large and diversified enough to be used reliably for assessing the performance of the algorithms under test.

We then compared our proposed GA that incorporates IPMX (see Section 4) against the same GA with standard PMX (Goldberg & Lingle, 1985), MX (Kumar et al., 2013) and BRKX (Andrade et al., 2019) on the selected set of TSP instances. See Section 2.2 for more details about these crossover operators.

In order to take the non-deterministic nature of GA into consideration, 30 independent runs were executed for each instance in the dataset, and the time spent for performing crossover operation was precisely measured. The parameters of these runs are presented in Table 1. It should be noted that for the purpose of this set of experiments, we concentrated only on crossover operation and the related parameters were chosen accordingly (i.e., crossover occurred at 100% with no mutation).

Tables 2 and 3 list the statistics (best, mean and standard deviation) of computing times obtained by the PMX, BRKX, MX and IPMX for a set of 36 TSP instances of different sizes, performing 30 runs per instance. The CPU times reported, clearly reveal that the IPMX outperforms all the methods under test by a significant margin. The sum of the CPU times of all instances also demonstrates that the IPMX is approximately 9 times faster than the PMX, which performs faster than MX and BRKX under the conditions of our experiments. This is in agreement with the computational complexity of the IPMX and PMX reported in the previous section. Tables 2 and 3 also present best objective function values associated with PMX, BRKX, MX and IPMX for each instance. A simple inspection of the results shows that the IPMX is not only very fast

Table 2
Computing times of the PMX, BRKX, MX and IPMX for a set of 36 TSP instances, shown with the following format: Best (Mean, Standard Deviation) and their associated best objective function values obtained, shown in italics.

TSP instance	Description	PMX	BRKX	MX	IPMX
eil51	51-city problem	0.1411 (0.1771, 0.0184) *600*	0.3144 (0.3312, 0.0100) *613*	0.1321 (0.1542, 0.0141) *631*	**0.0797** (0.0862, 0.0077) *593*
st70	70-city problem	0.2544 (0.3128, 0.0305) *1453*	0.5853 (0.6194, 0.0175) *1432*	0.2436 (0.2796, 0.0178) *1383*	**0.1139** (0.1226, 0.0098) ***1369***
pr76	76-city problem	0.3268 (0.3628, 0.0191) *233147*	0.6608 (0.7146, 0.0293) *227716*	0.2873 (0.3331, 0.0241) *237314*	**0.1265** (0.1309, 0.0032) ***222364***
lin105	105-city problem	0.5463 (0.6860, 0.0607) ***49488***	1.2467 (1.3484, 0.0493) *51253*	0.5678 (0.6444, 0.0454) *50727*	**0.1945** (0.2033, 0.0051) *51445*
d198	Drilling problem	2.2850 (2.6419, 0.1788) *85073*	4.4844 (4.7541, 0.1274) *81744*	2.0897 (2.3418, 0.1810) ***81269***	**0.4723** (0.5022, 0.0152) *81961*
pr226	226-city problem	2.9222 (3.3151, 0.2004) *909430*	6.0260 (6.1863, 0.0999) *919885*	2.6860 (3.1657, 0.2182) *919953*	**0.5769** (0.6261, 0.0232) ***900831***
pr264	264-city problem	3.8152 (4.6160, 0.3749) *565232*	8.3725 (8.6692, 0.1431) *562542*	3.8977 (4.3731, 0.2108) ***552146***	**0.7693** (0.8096, 0.0290) *563014*
pr299	299-city problem	4.9015 (5.7650, 0.4283) ***434147***	10.8229 (11.2586, 0.4082) *446306*	5.0920 (5.9693, 0.4941) *438905*	**0.9257** (0.9951, 0.0346) *436505*
lin318	318-city problem	5.7605 (6.7991, 0.4942) ***370552***	12.4209 (12.5161, 0.0784) *380286*	5.7462 (6.5982, 0.4933) *371027*	**1.0505** (1.1131, 0.0436) *372610*
pr439	439-city problem	10.6604 (13.1100, 1.0372) *1266361*	23.4947 (24.2322, 0.4230) ***1260566***	11.5645 (13.3609, 0.9854) *1273484*	**1.7406** (1.9024, 0.0728) *1264850*
d493	Drilling problem	14.7093 (18.6065, 2.1312) ***304539***	30.5771 (30.9398, 0.2718) *304956*	14.9678 (16.9331, 1.2598) *307721*	**2.1850** (2.3448, 0.0831) *305414*
rat575	Rattled grid problem	21.0062 (25.0886, 2.1291) *79061*	41.7143 (42.4207, 0.5770) *79420*	20.8135 (24.3785, 1.8185) *80459*	**2.8137** (3.0763, 0.1129) ***76815***
d657	Drilling problem	27.2169 (30.2025, 2.2969) ***622511***	55.0861 (55.5647, 0.6634) *622571*	28.8551 (33.2867, 2.7608) *639037*	**3.5992** (3.9104, 0.1471) *635306*
u724	Drilling problem	32.3592 (38.7305, 2.4248) *632404*	67.7509 (68.0499, 0.4041) *639199*	33.1371 (39.7687, 4.5737) *649106*	**4.4974** (4.8571, 0.2721) ***624561***
rat783	Rattled grid problem	38.8181 (45.8914, 3.3537) *131816*	77.9358 (79.0037, 0.8691) *130895*	40.9188 (47.9978, 4.4779) *133665*	**5.0821** (5.7679, 0.7627) ***130817***
pr1002	1002-city problem	62.9191 (75.5716, 5.2763) *5024038*	130.0221 (130.7902, 0.5706) *4970672*	68.4253 (80.8361, 8.4179) *5086670*	**7.9063** (8.6990, 0.3113) ***4964965***
d1291	Drilling problem	97.7342 (125.1184, 10.3539) *1404879*	220.1945 (220.4408, 0.24630) ***1401986***	125.0393 (139.9825, 11.8363) *1428906*	**12.6199** (13.8249, 0.6251) *1404833*
d1655	Drilling problem	190.7995 (215.8063, 14.2441) ***1785635***	365.0797 (365.8307, 0.75103) *1808874*	216.9726 (247.8239, 20.0794) *1824611*	**20.0846** (22.4201, 0.9105) *1807110*
u1817	Drilling problem	225.1480 (262.2252, 18.9452) *1739232*	439.6913 (443.5588, 3.8675) ***1718856***	261.9256 (297.5908, 23.7747) *1753633*	**23.3181** (26.9595, 1.2496) *1760303*
pcb3038	Drilling problem	655.5607 (769.6929, 58.5211) *4724293*	1244.5353 (1257.4957, 8.6771) *4723976*	770.7431 (920.7725, 73.4113) *4762984*	**65.5933** (73.8239, 3.1729) ***4666769***

but also effective, yielding the lowest objective function value on 15 benchmark instances out of 36 (the number of wins for PMX is 10, for BRKX is 6, and for MX is only 5).

5.2. Experiment set 2

In experiment set 2, the aim was to test the influence of IPMX on the quality of solutions produced by permutation-based GAs. For this purpose, we used the study of Ting et al. (2010) in which a highly effective crossover operator called multi-parent partially mapped crossover (MPPMX) was introduced and compared with the standard PMX. In this study, five TSP instances from TSPLIB (i.e., eil51, st70, pr76, lin105, and d198) were selected for examining the performance of MPPMX. We employed our proposed GA incorporating IPMX with the same parameter setting as suggested in Ting et al. (2010) and applied it to the five TSP instances mentioned above. We also adopted the same approach for BRKX (Andrade et al., 2019) to be able to compare its results with other methods under test in this set of experiment. It should be noted that the structure of the proposed GA is identical to the GA used by Ting et al. (2010), and the only difference between the two is their crossover operators.

Table 4 summarises the results obtained in these experiments. Note that the results associated with the MPPMX and the standard PMX were taken from Ting et al. (2010). As it is clear from Table 4, MPPMX outperforms PMX in solution quality by 3.68% (eil51), 7.53% (st70), 5.82% (pr76), 13.95% (lin105), and 13.32% (d198). IPMX also outperforms PMX in solution quality by 2.97% (eil51), 2.89% (st70), 2.49% (pr76), 9.82% (lin105), and 3.22% (d198), which is very satisfactory.

In addition, IPMX performs better than MPPMX (n = 2, 3, 4, 6 and 9) on (eil51), MPPMX (n = 2 and 3) on (st70), MPPMX (n = 2 and 3) on (pr76), MPPMX (n = 2, 3 and 5) on (lin105) and MPPMX (n = 2) on (d198). However, considering the best result obtained for each test problem, a direct comparison between IPMX and MPPMX shows that MPPMX is a clear winner. MPPMX also outperforms BRKX on all instances in terms of solution quality, and IPMX yields the lowest tour length on 3 benchmark instances out of 5 (i.e., eil51, st70 and d198) in comparison with BRKX. These results indicate that the IPMX is able to produce good quality solutions apart from being extremely fast, which is its distinctive feature.

Note that in experiment set 1, we were unable to report the CPU times of MPPMX on the selected TSP instances, since we did not have access to the original code or a trustworthy software package. However, there is no doubt that increasing the number of parents complicates the PMX process, especially when the problem size grows. Therefore, PMX can be considerably faster than its multi-parent extension (i.e., MPPMX). This simply implies that IPMX is extremely faster than MPPMX.

Table 3
Computing times of the PMX, BRKX, MX and IPMX for a set of 36 TSP instances, shown with the following format: Best (Mean, Standard Deviation) and their associated best objective function values obtained, shown in italics (continued from Table 2).

TSP instance	Description	PMX	BRKX	MX	IPMX
xqf131	VLSI data	0.8063 (1.0699, 0.0892) *2141*	1.9110 (2.0625, 0.0657) *2164*	0.9047 (0.9866, 0.0467) ***2096***	**0.2604** (0.2739, 0.0089) *2150*
xqg237	VLSI data	3.2447 (3.6417, 0.2090) *6930*	6.5109 (6.9143, 0.3853) *7006*	3.1908 (3.5322, 0.1673) *6902*	**0.6338** (0.6788, 0.0203) ***6801***
pma343	VLSI data	7.0247 (7.9709, 0.4821) *19452*	14.1675 (14.5960, 0.2060) *19680*	6.8867 (7.9896, 0.5491) ***19360***	**1.1747** (1.2596, 0.0457) *19378*
pka379	VLSI data	8.2160 (9.9206, 0.6564) *20775*	17.3513 (17.9097, 0.3786) ***20118***	8.3437 (9.8103, 0.8805) *20971*	**1.3569** (1.4892, 0.0518) *20358*
bcl380	VLSI data	8.3012 (9.7415, 0.6000) ***16430***	18.1587 (18.5510, 0.2060) *17033*	8.6391 (9.8524, 0.7166) *16739*	**1.3244** (1.4812, 0.0554) *16627*
pbl395	VLSI data	9.0107 (10.9758, 0.6930) *13259*	18.6592 (19.3299, 0.2968) *13189*	9.4915 (10.6665, 0.7306) *13130*	**1.4753** (1.6146, 0.0618) ***12858***
pbk411	VLSI data	9.3230 (11.5521, 0.9618) *14917*	21.1865 (21.9697, 0.7879) ***14808***	10.2483 (11.5626, 0.9529) *14956*	**1.5979** (1.7165, 0.0685) *15006*
pbn423	VLSI data	9.7713 (12.0691, 0.8615) ***14873***	22.2393 (22.5774, 0.2980) *15456*	10.9169 (12.3578, 0.6733) *15566*	**1.7286** (1.8441, 0.0624) *15173*
pbm436	VLSI data	12.6450 (14.7793, 0.8284) *17805*	23.2794 (23.8501, 0.3161) *16996*	10.9962 (13.0268, 0.8707) *15946*	**1.7415** (1.9186, 0.0781) ***15858***
xql662	VLSI data	27.1992 (31.0611, 2.1761) *37999*	55.3743 (56.3075, 0.4907) *39634*	29.0334 (33.8008, 3.4469) *38348*	**3.7611** (4.0465, 0.1407) ***37944***
rbx711	VLSI data	31.5452 (36.2866, 2.5666) *46962*	63.7630 (65.0720, 0.8802) ***46167***	33.9258 (38.5114, 2.7170) *47815*	**4.2620** (4.5823, 0.1647) *46695*
rbu737	VLSI data	32.6247 (39.6029, 2.5864) ***57502***	69.6738 (70.1516, 0.2967) *60163*	35.6344 (41.7768, 3.4258) *59664*	**4.6702** (4.9675, 0.1537) *58246*
dkg813	VLSI data	42.3694 (47.9601, 3.5364) *65321*	85.0408 (85.8864, 0.4949) *65260*	43.8414 (50.8419, 3.5755) *66233*	**5.3711** (5.9485, 0.2663) ***64292***
lim963	VLSI data	60.2718 (69.1656, 4.9726) ***53676***	120.6553 (122.2418, 1.9533) *55808*	61.0873 (71.5991, 6.3354) *54411*	**7.1216** (8.0095, 0.3542) *54031*
pbd984	VLSI data	57.7021 (70.5514, 4.9019) *53452*	126.1607 (127.7723, 1.4055) *53904*	70.4891 (80.0405, 7.7038) *54192*	**7.9988** (8.4936, 0.2239) ***53032***
xit1083	VLSI data	74.6868 (88.2586, 5.4804) *82833*	151.1570 (154.1882, 1.4189) *83464*	86.8494 (100.8957, 10.5016) ***81435***	**9.4549** (10.1511, 0.3454) *81975*
Sum (computing times)		1792.6267	3556.3044	2044.5838	**207.6827**

PMX: Standard PMX operator. BRKX: Biased random-key crossover. MX: Mixed crossover. IPMX: Improved PMX operator (proposed crossover operator). All CPU times are in seconds. Numbers in bold face are the best results.

Table 4
Mean, standard deviation (in parenthesis), and mutation rate (marked with an asterisk) related to the best tour length over 30 runs of the PMX, MPPMX, BRKX and IPMX.

	eil51	st70	pr76	lin105	d198
PMX	533.52 (31.56) 0.04*	1015.90 (68.79) 0.05*	162172.2 (9898.4) 0.03*	28409.2 (2527.5) 0.02*	37275.8 (4472.8) 0.005*
MPPMX (n = 2)	541.71 (30.74) 0.06*	1039.51 (72.28) 0.03*	163586.6 (10820.0) 0.03*	28759.3 (2652.8) 0.02*	37838.7 (4518.4) 0.006*
MPPMX (n = 3)	527.18 (28.93) 0.05*	1002.48 (73.47) 0.05*	161547.9 (11214.7) 0.03*	27049.8 (2545.3) 0.01*	36049.1 (2803.7) 0.01*
MPPMX (n = 4)	522.28 (28.93) 0.06*	973.89 (67.54) 0.04*	157436.3 (11099.9) 0.04*	25768.0 (2267.6) 0.02*	34812.9 (3669.4) 0.01*
MPPMX (n = 5)	**513.86** (28.31) 0.04*	962.79 (66.38) 0.04*	154115.7 (10660.5) 0.04*	25958.1 (1776.2) 0.005*	34220.0 (3371.5) 0.01*
MPPMX (n = 6)	520.19 (32.03) 0.05*	959.57 (63.46) 0.05*	156310.9 (11335.8) 0.04*	25111.6 (2636.9) 0.03*	34163.2 (3004.0) 0.01*
MPPMX (n = 7)	517.77 (26.58) 0.06*	956.28 (81.60) 0.05*	155252.7 (10066.9) 0.04*	25045.3 (2310.7) 0.02*	33398.7 (3253.1) 0.005*
MPPMX (n = 8)	516.41 (28.27) 0.06*	951.51 (66.56) 0.05*	155301.8 (10731.1) 0.04*	25176.9 (2045.2) 0.02*	33371.0 (3240.0) 0.01*
MPPMX (n = 9)	519.09 (35.58) 0.05*	**939.41** (63.71) 0.03*	**152737.9** (9697.3) 0.04*	24566.2 (1561.9) 0.02*	**32309.6** (3459.1) 0.004*
MPPMX (n = 10)	514.01 (30.48) 0.06*	947.07 (59.42) 0.04*	153161.7 (11058.6) 0.04*	**24444.9** (2013.7) 0.005*	33365.0 (3912.9) 0.004*
BRKX	532.6 (23.52) 0.04*	989.2 (49.85) 0.05*	154850.1 (9022.38) 0.03*	24596.7 (1312.22) 0.02*	37112.0 (4834.002) 0.005*
IPMX	518.13 (22.36) 0.04*	987.36 (71.35) 0.05*	158228.9 (10651.1) 0.03*	25867.86 (2200.38) 0.02*	36110.33 (3546.13) 0.005*

PMX: Standard PMX operator. MPPMX: Multi-parent PMX operator. BRKX: Biased random-key crossover. IPMX: Improved PMX operator (proposed crossover operator). Numbers in bold face are the best results.

6. Conclusions

In this paper, a GA for addressing permutation-based combinatorial optimisation problems has been proposed. The key feature of the proposed GA is that it incorporates an improved version of the well-known PMX, termed IPMX. IPMX was designed with the aim of increasing the efficiency of permutation-based GAs without affecting solution quality. In order to examine the capability of IPMX, two sets of experiments were conducted.

In experiment set 1, the influence of IPMX on the speed of permutation-based GAs was tested, in which IPMX showed remarkable performance on 36 benchmark problems, being approximately 9 times faster than the standard PMX (performing faster than MX and BRKX) with respect to the sum of the CPU times of all instances. This was in agreement with the computational complexity of the IPMX and PMX, which were also reported in the present study. The new crossover, IPMX, appears to owe much of its success to the ability to significantly decrease the number of operations required for the elimination of repeated elements and the generation of second offspring. The IPMX also yielded the lowest objective function value on 15 benchmark instances, which was much better than the results obtained by PMX, MX and BRKX.

In experiment set 2, the influence of IPMX on the quality of solutions produced by permutation-based GAs was assessed. The results obtained indicated that in terms of solution quality, IPMX improved the standard PMX by 2.49-9.82% on the five TSP instances selected for test. IPMX also produced better solutions than MPPMX (i.e., a highly effective crossover operator) in some cases, and BRKX in most cases. In addition, it can be inferred from the results that IPMX is extremely faster than MPPMX.

We believe that the results reported here are promising and show that the IPMX is able to produce good quality solutions apart from being extremely fast, which is its distinctive feature. Therefore, in future work, we will investigate the possibility of further developing the IPMX and applying it to GAs designed for solving other permutation-based combinatorial optimisation problems.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Credit authorship contribution statement

Behrooz Koohestani: Conceptualization, Methodology, Software, Data curation, Writing - original draft, Visualization, Investigation, Supervision, Validation, Writing - review & editing.

References

Abdoun, O., & Abouchabaka, J. (2012). A comparative study of adaptive crossover operators for genetic algorithms to resolve the traveling salesman problem. *CoRR, abs/1203.3097*.

Agarwal, T., & Singh, K. (2013). Using new variation crossover operator of genetic algorithm for solving the traveling salesmen problem. *MIT International Journal of Computer Science and Information Technology, 3*(1), 35–37.

Albayrak, M., & Allahverdi, N. (2011). Development a new mutation operator to solve the traveling salesman problem by aid of genetic algorithms. *Expert Systems with Applications, 38*(3), 1313–1320. doi:10.1016/j.eswa.2010.07.006.

Andrade, C. E., Silva, T., & Pessoa, L. S. (2019). Minimizing flowtime in a flowshop scheduling problem with a biased random-key genetic algorithm. *Expert Systems with Applications, 128*, 67–80. doi:10.1016/j.eswa.2019.03.007.

Banzhaf, W. (1990). The "molecular" traveling salesman. *Biological Cybernetics, 64*(1), 7–14. doi:10.1007/BF00203625.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms*. MIT press.

Davis, L. (1985). Applying adaptive algorithms to epistatic domains. In *Proceedings of the 9th international joint conference on artificial intelligence - volume 1*. In *IJCAI'85* (pp. 162–164). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Du, K., & Swamy, M. (2016). *Search and optimization by metaheuristics: techniques and algorithms inspired by nature*. Springer International Publishing.

Eiben, A., & Smith, J. (2015). Introduction to evolutionary computing. *Natural computing series*. Springer Berlin Heidelberg.

Eiben, A. E., Raué, P. E., & Ruttkay, Z. (1994). Genetic algorithms with multi-parent recombination. In Y. Davidor, H.-P. Schwefel, & R. Männer (Eds.), *Parallel problem solving from nature — ppsn iii* (pp. 78–87). Berlin, Heidelberg: Springer Berlin Heidelberg.

Fogel, D. B. (1988). An evolutionary approach to the traveling salesman problem. *Biological Cybernetics, 60*(2), 139–144. doi:10.1007/BF00202901.

Fogel, D. B. (1990). A parallel processing approach to a multiple traveling salesman problem using evolutionary programming. In *Proceedings of the fourth annual symposium on parallel processing* (pp. 318–326). Fullerton, CA.

Fogel, D. B. (1993). Applying evolutionary programming to selected traveling salesman problems. *Cybernetics and Systems, 24*(1), 27–36. doi:10.1080/01969729308961697.

Goldberg, D. E., & Lingle, R. Jr. (1985). Alleleslociand the traveling salesman problem. In *Proceedings of the 1st international conference on genetic algorithms* (pp. 154–159). Hillsdale, NJ, USA: L. Erlbaum Associates Inc.

Grefenstette, J. J. (1987). In L. Davis (Ed.), *Incorporating problem specific knowledge into genetic algorithms* (pp. 42–60)). Los Altos, CA: Morgan Kaufmann.

Holland, J. H. (1992). *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control and artificial intelligence*. Cambridge, MA, USA: MIT Press.

Katayama, K., & Narihisa, H. (2001). An efficient hybrid genetic algorithm for the traveling salesman problem. *Electronics and Communications in Japan (Part III: Fundamental Electronic Science), 84*(2), 76–83. doi:10.1002/1520-6440(200102)84:2<76::AID-ECJC9>3.0.CO;2-O.

Kirkpatrick, S., & Toulouse, G. (1985). Configuration space analysis of travelling salesman problems. *Journal de Physique, 46*(8), 1277–1292.

Kumar, R., Gopal, G., & Kumar, R. (2013). Novel crossover operator for genetic algorithm for permutation problems. *International Journal of Soft Computing and Engineering (IJSCE), 3*(2), 252–258.

Larrañaga, P., Kuijpers, C., Murga, R., Inza, I., & Dizdarevic, S. (1999). Genetic algorithms for the travelling salesman problem: A review of representations and operators. *Artificial Intelligence Review, 13*(2), 129–170. doi:10.1023/A:1006529012972.

Larrañaga, P., Kuijpers, C. M. H., Poza, M., & Murga, R. H. (1997). Decomposing bayesian networks: triangulation of the moral graph with genetic algorithms. *Statistics and Computing, 7*(1), 19–34. doi:10.1023/A:1018553211613.

Lo, K. M., Yi, W. Y., Wong, P.-K., Leung, K.-S., Leung, Y., & Mak, S.-T. (2018). A genetic algorithm with new local operators for multiple traveling salesman problems.. *Int. J. Comput. Intell. Syst., 11*(1), 692–705.

Michalewicz, Z. (1992). *Genetic algorithms+ data structures = evolution program*. Berlin Heidelberg: Springer Verlag.

Miller, B. L., & Goldberg, D. E. (1995). Genetic algorithms, tournament selection, and the effects of noise. *Complex systems, 9*(3), 193–212.

Misevičius, A., & Kilda, B. (2005). Comparison of crossover operators for the quadratic assignment problem. *Information Technology and Control, 34*(2).

Mühlenbein, H. (1991). Parallel genetic algorithms, population genetics and combinatorial optimization. In J. D. Becker, I. Eisele, & F. W. Mündemann (Eds.), *Parallelism, learning, evolution* (pp. 398–406). Berlin, Heidelberg: Springer Berlin Heidelberg.

Mumford, C. L. (2006). New order-based crossovers for the graph coloring problem. In T. P. Runarsson, H.-G. Beyer, E. Burke, J. J. Merelo-Guervós, L. D. Whitley, & X. Yao (Eds.), *Parallel problem solving from nature - ppsn ix* (pp. 880–889). Berlin, Heidelberg: Springer Berlin Heidelberg.

Oliver, I. M., Smith, D. J., & Holland, J. R. C. (1987). A study of permutation crossover operators on the traveling salesman problem. In *Proceedings of the second international conference on genetic algorithms on genetic algorithms and their application* (pp. 224–230). Hillsdale, NJ, USA: L. Erlbaum Associates Inc.

Onwubolu, G., & Davendra, D. (2009). Differential evolution: a handbook for global permutation-based combinatorial optimization. *Studies in Computational Intelligence*. Springer.

Papadimitriou, C., & Steiglitz, K. (2013). Combinatorial optimization: algorithms and complexity. *Dover Books on Computer Science*. Dover Publications.

Pavai, G., & Geetha, T. V. (2016). A survey on crossover operators. *ACM Comput. Surv., 49*(4), 72:1–72:43. doi:10.1145/3009966.

Syswerda, G. (1991). Schedule optimization using genetic algorithms. In L. Davis (Ed.), *Handbook of genetic algorithms* (pp. 332–349). New York: Van Nostrand Reinhold.

Ting, C.-K., Su, C.-H., & Lee, C.-N. (2010). Multi-parent extension of partially mapped crossover for combinatorial optimization problems. *Expert Systems with Applications, 37*(3), 1879–1886. doi:10.1016/j.eswa.2009.07.082.

Umbarkar, A., & Sheth, P. (2015). Crossover operators in genetic algorithms: a review.. *ICTACT journal on soft computing, 6*(1), 1083–1092.

Wang, Y. H., & Liu, L. F. (2013). A new genetic algorithm using order coding and a novel genetic operator. In *Advanced information and computer technology in engineering and manufacturing, environmental engineering*. In *Advanced Materials Research: 765* (pp. 662–666). Trans Tech Publications. 10.4028/ http://www.scientific.net/AMR.765-767.662.

Whitley, D., Starkweather, T., & Shaner, D. (1991). The traveling salesman and sequence scheduling: Quality solutions using genetic edge recombination. In L. Davis (Ed.), *Handbook of genetic algorithms* (pp. 350–372). New York: Van Nostrand Reinhold.