Neurocomputing 314 (2018) 409–428

Contents lists available at ScienceDirect

Neurocomputing

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

Efficient genetic algorithms for optimal assignment of tasks to teams of agents☆

Irfan Younas[a,*], Farzad Kamrani[b], Maryam Bashir[a], Johan Schubert[b,c]

[a] Department of Computer Science, National University of Computer and Emerging Sciences, Lahore Pakistan
[b] Swedish Defence Research Agency, Stockholm Sweden
[c] School of Electrical Engineering and Computer Science, KTH Royal Institute of Technology, Stockholm Sweden

ARTICLE INFO

Article history:
Received 6 July 2016
Revised 25 May 2018
Accepted 3 July 2018
Available online 10 July 2018

Communicated by Prof. Huaguang Zhang

Keywords:
Genetic algorithms
Combinatorial optimization
Shuffled list crossover
Team-based crossover
Large scale optimization
Team assignment problem

ABSTRACT

The problem of optimally assigning agents (resources) to a given set of tasks is known as the assignment problem (AP). The classical AP and many of its variations have been extensively discussed in the literature. In this paper, we examine a specific class of the problem, in which each task is assigned to a group of collaborating agents. APs in this class cannot be solved using the Hungarian or other known polynomial time algorithms. We employ the genetic algorithm (GA) to solve the problem. However, we show that if the size of the problem is large, then standard crossover operators cannot efficiently find near-optimal solutions within a reasonable time. In general, the efficiency of the GA depends on the choice of genetic operators (selection, crossover, and mutation) and the associated parameters.

In order to design an efficient GA for determining the near-optimal assignment of tasks to collaborative agents, we focus on the construction of crossover operators. We analyze why a naive implementation with standard crossover operators is not capable of sufficiently solving the problem. Furthermore, we suggest modifications to these operators by adding a shuffled list and introduce two new operators (team-based and team-based shuffled list). We demonstrate that the modified and new operators with shuffled lists perform significantly better than all operators without shuffled lists and solve the presented AP more efficiently. The performance of the GA can be further enhanced by using chaotic sequences. Moreover, the GA is also compared with the particle swarm optimization (PSO) and differential evolution (DE) algorithms, demonstrating the superiority of the GA over these search algorithms.

1. Introduction

The problem of assigning tasks to agents where a cost (or gain) is associated to each assignment represents a combinatorial optimization problem generally known as the assignment problem (AP). In the original AP, each task is assigned to a different agent, and each agent performs exactly one task, which implies that there are an equal number of tasks and agents. However, one can readily relax this constraint by adding "dummy" tasks (or agents) to the problem.

In a seminal paper, Kuhn [27] provided a method for finding the optimal solution of the AP with polynomial time complexity. The algorithm, also known as the *Hungarian method*, has had a fundamental influence on combinatorial optimization, and has become the prototype of a considerable number of algorithms in the field [15].

Pentico [37] presented a comprehensive survey on the most common variations of the AP that have appeared in the literature over the past 50 years. In this survey, three main categories of AP are recognized, and several variations in each category are discussed:

1. Models with at most one task per agent,
2. Models with multiple tasks per agent,
3. Multi-dimensional assignment problems, which study the matching of members of three (or more) sets, e.g., the problem of matching jobs with workers and machines or assigning students and teachers to classes and time slots.

Although minor modifications of the problem structure can be handled within the standard solution, there are many variations of the AP that demand completely different approaches. For instance, the generalized assignment problem (GAP), in which agents may perform multiple tasks but the sum of the costs of the tasks

☆ Some parts of the first 5 sections of this paper were presented at the 2011 IEEE Symposium on Computational Intelligence in Scheduling, see [44]

* Corresponding author.
E-mail addresses: irfan.younas@nu.edu.pk, irfany@kth.se (I. Younas), kamrani@kth.se (F. Kamrani), maryam.bashir@nu.edu.pk (M. Bashir), schubert@foi.se (J. Schubert).

https://doi.org/10.1016/j.neucom.2018.07.008
0925-2312/

assigned to each agent cannot exceed the agent's budget constraint, is an NP-hard combinatorial optimization problem [14]. Chu and Beasley [8] presented a genetic algorithm (GA)-based heuristic for solving the GAP, where the objective is to assign n jobs to m agents such that the cost is minimized and each job is assigned to exactly one agent subject to the capacity of that agent.

The problem of assigning tasks to teams of agents with the objective of maximizing the total value added by agents to tasks is distinguished from the APs discussed in the literature. Tasks may have different sizes and require different numbers of agents, while each agent is assigned to at most one task. Therefore, from the classification viewpoint this problem belongs to the first category described by Pentico [37]. Nevertheless, this problem is distinct from other instances, because agents performing a task constitute a team whose performance is a possibly non-linear function of the performances of the members. This characteristic makes the problem intractable for all but small input Sizes, and one must rely on heuristic methods to find near-optimal solutions.

GAs are heuristic search algorithms that have been successfully applied to a variety of optimization problems, and are widely adopted in practice. GAs can generally solve large-scale problems, partly owing to the concurrent exploration of diverse parts of a search space, provided that "sufficient" time is available. However, in many problems, there may be constraints on the execution time, and it is required that near-optimal solutions be found within a relatively short timeframe. The efficiency of the GA depends to a large extent on the choices of the genetic operators (such as selection, crossover, and mutation) and the fine-tuning of the control parameters (such as mutation rate). A careful selection of the operators and parameters of the GA is necessary to maintain an appropriate balance between the exploitation and exploration properties of the search algorithm, without which it is difficult to find near-optimal solutions within a reasonable time. Exploration is concerned with investigating new and unknown areas in the search space, while exploitation exploits knowledge acquired at previously visited points to find better points.

The crossover operator employed for the baseline solution is a one-point crossover operator. Although the baseline GA provides near-optimal solutions for instances of a problem in which the number of required agents is relatively small, for larger problem sizes, the performance of the GA is degraded. The main contribution of this paper is to design different crossover operators that solve APs in this specific class efficiently. We design some new crossover operators, and demonstrate that they can solve large instances of our AP efficiently and effectively.

In this paper, we focus on solving APs of a specific class, where each task is assigned to a team of agents. Tasks require the collaboration of all team members, and cannot be performed by an individual agent. Thus, collaboration, i.e., the interactions of the team members and how they affect each other, is the main concern. The outcome of a team is not simply the algebraic sum of the capabilities of the members. Such an AP is substantially different from the cases where each agent may perform a task independently. For instance, the task of repairing faulty computers in a company could be assigned to a group of technicians. However, in this case each technician could perform the task independently, and no collaboration between them is required. There is a substantial body of literature providing solutions to different APs, but to the best of our knowledge, there are no solutions for our specific type of AP. We provide a mathematical formulation of the AP for teams of agents and discuss the implementation details and efficiency of the GA for solving this problem.

Although the AP as formulated here is a far more complex and constitutes a generalization of the original assignment problem, it does not incorporate the notion of time and time constraints. As a result, dynamic problems (e.g., where the number of agents required to accomplish one task changes over time) or problems that impose different time constraints on the completion of tasks remain outside the scope of this work.

The outline of the remainder of this paper is as follows. In Section 2, related work on task assignment problems from recent years is presented. In Section 3, a mathematical formulation of the problem is presented. In Sections 4 and 5, the GA heuristic, its implementation details, and its computational complexity are discussed. Section 6 presents some experimental results, and demonstrates that standard GA crossover operators are not capable of solving the problem efficiently. In Section 7, modifications to the standard crossover operators and some new crossover operators are proposed, and in Section 8 the performances of these operators are compared and analyzed. Section 9 compares GA with particle swarm optimization (PSO) [11,26], differential evolution (DE) [38], and a proposed approximation algorithm. Finally, Section 10 concludes the paper.

2. Related work

A considerable body of literature has focused on the task assignment problem and its applications in various domains. This section describes related work concerning the task assignment problem in the domains of software project scheduling (SPS) and crowdsourcing, and the use of genetic algorithms for solving the problem.

The task assignment problem has previously been employed for solving SPS problems. In SPS, each resource masters several skills with given levels, and each task in the project requires a certain skill to be executed at a standard level. Alba and Chicano [2] employed the GA to solve an SPS problem, in which employees that each have a certain set of skills and salary are assigned to a given set of tasks so as to reduce the cost and duration of the project. These employees have a maximum degree of dedication to the project. Each of the tasks requires certain skills, and is assigned to at least one employee. In the SPS problem discussed in [2,7], more than one employee can be assigned to a single task, and the employees work independently. No collaboration between the employees is considered. Instead of aggregating the cost and time into a single objective, Chicano et al. [7] considered the SPS problem as a bi-objective model, and compared the performances of different multi-objective evolutionary algorithms in solving the proposed SPS model.

Al-Anzi et al. [1] modeled the SPS problem as a weighted-multi-skill project scheduling problem (WMSPSP), where one staff member can perform multiple tasks with different proficiency levels and the resources are constrained. A lower bound was proposed that employs a linear programming scheme to solve a resource-constrained project scheduling problem. Wang and Zheng [40] applied a guided multi-objective fruit fly optimization algorithm (MOFOA) to solve a resource-constrained project scheduling problem. Montoya et al. [33] presented an integrated column generation and Lagrangian relaxation approach for solving the SPS problem. A lower bound was obtained for minimizing the project completion time. Myszkowski et al. [35] proposed a hybrid approach that combines classical heuristic priority rules for project scheduling with ant colony optimization. Zheng et al. [45] employed a teaching-learning-based optimization algorithm for the resource constrained SPS problem, which minimizes the project completion time. Luna et al. [29] solved the SPS problem using a multi-objective approach, and analyzed the scalability of eight existing multi-objective algorithms. Other applications of task assignment can be found in social security organizations, consultative service companies, service centers, and research-based organizations [24].

Another application of the task assignment problem is for assigning heterogeneous tasks to workers with different and unknown skill sets in crowdsourcing markets, such as Amazon Mechanical Turk. In the offline task assignment problem, a requester has a fixed set of tasks and a budget, and the goal is to allocate workers to tasks in a manner that maximizes the total benefit. In an online setting, the task assignment problem can be formulated in a similar manner to the online adwords problem [13]. For these problems, approximation algorithms can be employed to achieve a competitive ratio of $(1-1/e)$ in a setting with adversarial arrivals [4] or $(1-\epsilon)$ for stochastic arrivals, where e is very small when the total budget is large [10]. Ho, and Vaughan [21] presented a two-phase exploration-exploitation assignment algorithm for solving the online task assignment problem in a crowdsource setting. Cheng et al. [6] modeled a crowdsourcing task using a multi-skill spatial crowdsourcing approach, which determines an optimal worker-and-task assignment strategy such that the skills of workers and tasks match each other and workers' benefits are maximized under the budget constraint. The authors tackle this problem by proposing three effective approximation approaches, including greedy, g-divide-and-conquer, and cost-model-based adaptive algorithms.

Most of the above-mentioned related approaches to task assignment assume a non-collaborative model of workers. Such approaches cannot be applied to our collaborative model. We have employed GA to solve our problem. However, the efficiency of GA depends largely on the choices of the genetic operators. A considerable body of literature on GAs addresses the selection of the genetic operators and associated parameters, and how these affect the efficiency of the algorithm [28,31]. There is no unique "best" answer to these questions, and the choice of operators depends on the problem domain and the structure of the search space (the properties of the search space such as the search space size and distribution of solutions within neighborhood). For instance, the partially mapped crossover operator achieves a good performance for the traveling salesman problem [18]. However, its performance for solving the one-machine total weighted tardiness problem is inferior to the order-based and position-based crossover operators [25]. The efficiencies of 11 genetic crossover operators are compared for the one-machine total weighted tardiness problem in [25]. The discussed operators are the position-based crossover operator, order-based crossover operator [39], one-point crossover, cycle crossover operator [36], order crossover [9], linear order crossover, partially mapped crossover [19], edge recombination operator [41], and three flavors of two-point crossover operators [34]. These operators have been widely employed to solve various types of assignment and scheduling problems.

3. Problem formulation

In this section, a mathematical formulation for the optimal assignment of tasks to teams of agents is introduced. Let $\mathcal{A}=\{a_1,\ldots,a_m\}$ be the set of m agents and $\mathcal{T}=\{t_1,\ldots,t_n\}$ the set of n tasks, where in general $m\neq n$. Assume that each task $t_j\in\mathcal{T}$ requires a fixed number d_j of agents, while each agent $a_i\in\mathcal{A}$ performs at most one task, implying that $\sum_{j=1}^{n} d_j \leq m$. The group of d_j agents performing a task t_j is denoted by $g_{\mathcal{S}_j}$, where the index $\mathcal{S}_j$ is a set

$$\mathcal{S}_j \subset \{1,2,\ldots m\}, \quad |\mathcal{S}_j| = d_j, \quad a_i \in g_{\mathcal{S}_j} \Leftrightarrow i \in \mathcal{S}_j.$$

The goal is to optimally (defined later) assign all tasks to groups of agents. That is, to determine subsets $\mathcal{S}_j$ for all t_j. Clearly, our assumption that each agent performs at most one task implies that these subsets are disjoint.

Moreover, we assume that agents in a group collaborate in a team Environment, and the value produced by a team is a possibly non-linear real function of its members and the task. That is, a team $g_{\mathcal{S}_j}$ that performs the task t_j produces the value $f(g_{\mathcal{S}_j}, t_j)$. Optimality is defined as maximizing the sum of the values produced by the teams of agents, i.e., maximizing the objective function

$$u(\mathcal{S}_1,\ldots\mathcal{S}_n) = \sum_{j=1}^{n} f(g_{\mathcal{S}_j}, t_j). \tag{1}$$

We adopt the assumption that the value produced by a team $g_{\mathcal{S}_j}$ may be expressed as the sum of the values produced by agents while they are influenced by the team

$$f(g_{\mathcal{S}_j}, t_j) = \sum_{i\in\mathcal{S}_j} v(a_i, g_{\mathcal{S}_j}, t_j). \tag{2}$$

Substituting Eqs. (2) into Eq. (1), we obtain

$$u = \sum_{j=1}^{n}\sum_{i\in\mathcal{S}_j} v(a_i, g_{\mathcal{S}_j}, t_j). \tag{3}$$

A convenient approach to expressing the problem is to introduce the *assignment matrix* $\mathcal{X}=[x_{ij}]_{m\times n}$, where $x_{ij}=1$ if task t_j is assigned to agent a_i and is 0 otherwise. Using this notation, the objective function expressed by Eq. (3) can be reformulated equivalently as the following function:

$$u(\mathcal{X}) = \sum_{j=1}^{n}\sum_{i=1}^{m} v(a_i, g_{\mathcal{S}_j}, t_j)x_{ij}, \tag{4}$$

which should be maximized subject to the constraints

$$\sum_{i=1}^{m} x_{ij} = d_j, \qquad \forall t_j \in \mathcal{T} \tag{5}$$

$$\sum_{j=1}^{n} x_{ij} = 1, \qquad \forall a_i \in \mathcal{A} \tag{6}$$

$$\sum_{j=1}^{n} d_j \leq m \tag{7}$$

$$d_j \geq 1, \qquad \forall t_j \in \mathcal{T} \tag{8}$$

$$x_{ij} \in \{0,1\}, \qquad \forall a_i \in \mathcal{A}, \forall t_j \in \mathcal{T}, \tag{9}$$

where $\mathcal{S}_j = \{i : i \in \{1,\ldots m\}, x_{ij}=1\}$.

Constraint (5) ensures that each task is performed by a group of agents, where each group $g_{\mathcal{S}_j}$ contains d_j agents for the task t_j. Constraint (6) means that no agent is assigned to more than one task. Constraint (7) ensures that the number of agents making up n teams should be less than or equal to the total number of agents. Finally, constraint (8) ensures that each group (team) should contain one or more agents.

In this model, the performance of a team is a non-linear function of its members. Interactions (between individuals within a team) introduce this non-linearity and make the problem intractable. In Section 6, we also consider a flavor of the model without interactions, and solve it in polynomial time using the *Hungarian method*.

4. Genetic algorithms for the assignment of tasks to teams of agents

GAs [17,32], which are recognized metaheuristics based on the evolutionary ideas of natural selection and genetics, were introduced by John Holland [22] in the 1960s. They simulate the "survival of the fittest" principle, which was laid down by Charles

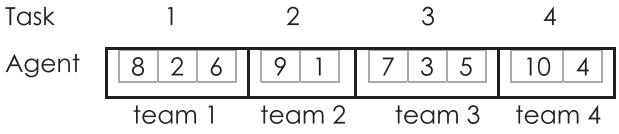

Fig. 1. Chromosome representation for a candidate solution.

Darwin. GAs have frequently been employed to solve many optimization problems [3,8,12,20].

The first step in designing a GA for a particular problem is to devise a suitable representation. Our algorithm adopts an appropriate representation scheme in which an n-dimensional vector of disjoint subsets represents a task set (called a chromosome in the GA literature), as illustrated in Fig. 1. Each task is performed by a team of agents, and here an n-dimensional vector indicates that n tasks are performed by teams of agents, where each team j requires d_j agents. Thus, the total number of agents required to perform the tasks is $\sum_{j=1}^{n} d_j$. For instance, consider a case of four tasks requiring 3, 2, 3, and 2 agents, respectively. Assume that there are 10 agents available. This means that $n = 4$, $d_1 = 3$, $d_2 = 2$, $d_3 = 3$, $d_4 = 2$, and $m = 10$. Fig. 1 illustrates a possible assignment of agents to teams. The designed representation scheme ensures that all the constraints in (5)–(8) are satisfied. A detailed description of our GA is given in [44]. A summary of the steps, which are similar to the GA steps provided by Chu and Beasley [8], is given as follows:

1. Construct N candidate solutions to form an initial population.
2. Calculate the fitness value according to the given fitness function as given by Eq. (4).
3. For reproduction, two parent solutions are selected using the binary tournament selection scheme, which is also employed in [8]. In binary tournament selection, two candidate solutions are randomly picked from the population, and the one with the higher fitness value is selected for reproduction.
4. A crossover operator is applied to the selected parents in order to generate a child solution. The baseline crossover operator used in the GA is a one-point crossover operator in which a point p is selected randomly from the set of integers such that $p \in \{1, 2, \ldots, \sum_{j=1}^{n} d_j\}$. The generated child solution inherits p genes from one parent and the remaining genes from the other parent. However, this operation may render our candidate solution infeasible by violating constraint (6), which states that no agent can be assigned to more than one task. In order to make the solution feasible, the duplicate assignments are replaced by some other agent numbers, which are not part of the chromosome. This operation changes an invalid child chromosome into a valid one. In Section 7, we employ several other well-known crossover operators, modify some of these operators to enhance their performances, and introduce a number of new crossover operators. The modified crossover operators use a shuffle list to repair the infeasible solutions. A shuffle list is a list of all available agents. For instance, if the total number of available agents is 10, then the shuffle list is a permutation of a list of size 10, containing agent indexes from 1 to 10. Before repairing an infeasible solution, we randomly permute the shuffle list.
5. The crossover operator is followed by a mutation procedure with a smaller probability value. In this study, a simple method is employed in which two genes are randomly chosen and their values are swapped.
6. Following crossover and mutation, a child solution needs to be added to the population if it is not already part of the population. First, the fitness value is calculated for the child, and then the chromosome with the smallest fitness value in the population is replaced by the child. This replacement scheme helps to introduce new solutions into the population and eliminate those solutions that are weaker (have lower fitness values).
7. The selection, crossover, mutation, and individual replacement operations (points 3–6 above) are repeatedly performed until a termination criterion is fulfilled. Examples of such criteria are the number of generations created without improving the best solution, or the number of fitness evaluations. Throughout this study, we employ the latter criterion.

5. Implementation overview and computational complexity of GA

This section presents the key design decisions for the implementation of the GA, and estimates the asymptotic complexity of a solution. In the following, the size of a chromosome and the population are denoted by M and N, respectively.

Each chromosome is initialized randomly by shuffling a list of $\{1, \ldots, m\}$ agents. To avoid duplicate chromosomes, a naive $O(MN^2)$ algorithm is employed, because it is only run during the initialization. Moreover, each chromosome stores a fitness value (that is, the fitness value is computed only once, for efficiency).

The parents for the crossover operation are selected by randomly choosing two chromosomes and selecting the fitter one (taking $O(1)$ time). The crossover and repair operations are implemented as discussed in Section 4, requiring $O(M)$ time. Finally, the best and worst chromosomes (with respect to the fitness) are each maintained by a priority queue, where the chromosomes' positions are recorded in the priority queues.

Each iteration of the GA requires $O(\log N)$ time for maintaining the best and worst chromosome in the population, and $O(M)$ for checking that no duplicate chromosome is entered into the population. Hence, the time spent for each iteration is $O(\log N + M)$.

6. Analysis of the genetic algorithm

In this section, various sets of experiments and their results are presented. The first step in any experiment is to employ a model to calculate the value produced by agents in a team when performing a task. We refine the collaborative model described in [23]. In this model, each agent a_i has a set of p real-valued attributes $c_i = \{c_{i1}, c_{i2}, \ldots, c_{ip}\}$, called *capabilities*, which affect the value produced by the agent. Each task t_j has a set of p real-valued attributes $w_j = \{w_{1j}, w_{2j}, \ldots, w_{pj}\}$, henceforth referred to as *weights*, which specify the importance of the capabilities of agents in performing the task. If a task is performed by a single agent, then the performance, or the value produced by the agent, is defined by the weighted sum of the agent's capabilities:

$$v(a_i, t_j) = \sum_{k=1}^{p} c_{ik} w_{kj}. \tag{10}$$

However, if a task is assigned to more than one agent, then the produced value is more than the sum of the values produced by the individual agents.

Modeling the performance of collaborating agents in a team is fairly complex. Here, we introduce a model based on certain assumptions on how the interactions between agents with different levels of capabilities might affect their performance. We note that the presented model is not validated, and does not completely reflect the emerging interactions between agents. However, the main focus of this paper is not to provide such a model, and the model simply serves as a means to test and analyze the GAs. Moreover, the model can readily be replaced by any other, without affecting the main discussions in this paper.

It is assumed that for each capability type, the capabilities of agents are influenced by the maximum capability of that

Table 1
Specification of the assignment problems used in this study.

Prob #	Total agents	Required agents	Number of tasks	Teams (number of agents assigned to the tasks)
1	10	10	4	[2, 3, 2, 3]
2	20	20	4	[3, 4, 6, 7]
3	20	20	8	[3, 2, 4, 3, 2, 2, 2, 2]
4	30	30	8	[4, 5, 2, 3, 4, 6, 3, 3]
5	30	30	12	[2, 2, 3, 3, 1, 4, 5, 2, 3, 2, 2, 1]
6	60	60	12	[6, 4, 8, 2, 7, 3, 5, 5, 3, 7, 1, 9]
7	80	80	20	[5, 4, 8, 2, 3, 3, 4, 3, 4, 7, 1, 6, 3, 6, 2, 6, 3, 1, 5, 4]
8	96	96	30	[5, 4, 8, 2, 3, 2, 2, 3, 1, 4, 7, 1, 5, 3, 2, 6, 2, 6, 3, 1, 5, 4, 2, 2, 4, 2, 1, 2, 3, 1]
9	400	400	100	[4, 4, ..., 4]
10	800	400	100	[4, 4, ..., 4]
11	1600	400	100	[4, 4, ..., 4]
12	1600	1600	100	[16, 16, ..., 16]
13	400	400	200	[2, 2, ..., 2]

type ($c_k^{\max}$) in the team (g_{S_j}), and the new capabilities (c'_{ik}) are calculated by

$$c'_{ik} = c_{ik} + c_{ik}(c_k^{\max} - c_{ik})/c_k^{\max}, \tag{11}$$

where $c_k^{\max} = \max_{i \in S_j}\{c_{ik}\}, \quad \forall k.$

Eq. (11) implies the following:

1. In a team of agents, only those with a lower capability than the maximum capability benefit from collaboration.
2. The capability of the agent with the maximum capability is not affected by cooperation.
3. The capability of an agent equal to 0 is not affected by cooperation.
4. Agents that have a capability equal to $c_k^{\max}/2$ receive the highest benefit from cooperation.

From Eqs. (10) and (11), the value produced by an agent while collaborating with others is obtained as

$$v(a_i, g_{S_j}, t_j) = \sum_{k=1}^{p}(c_{ik} + c_{ik}(c_k^{\max} - c_{ik})/c_k^{\max})w_{kj}. \tag{12}$$

Substituting Eq. (12) into the objective function defined by Eq. (4), we obtain

$$u(\mathcal{X}) = \sum_{j=1}^{n}\sum_{i=1}^{m}\sum_{k=1}^{p}(c_{ik} + c_{ik}(c_k^{\max} - c_{ik})/c_k^{\max})w_{kj}x_{ij}, \tag{13}$$

where $c_k^{\max}(\mathcal{X}) = \max_{1 \le i \le m}\{c_{ik}x_{ij}\}$, subject to the constraints (5)–(9).

To test the algorithm using the above model, a series of experiments with different problem sizes were conducted. In this scenario, each agent a_i has 10 capabilities $\{c_{i1}, c_{i2}, \ldots, c_{i10}\}$, and each task t_j weights the capability c_{ik} as $w_{kj} \in \{w_{1j}, w_{2j}, \ldots, w_{10j}\}$, where $c_{ik}, w_{kj} \in \{0, 1, \ldots, 4.0\}$. The input data is randomly generated, such that $p(c_{ik} = 0) = p(c_{ik} = 4) = 0.1$, $p(c_{ik} = 1) = p(c_{ik} = 3) = 0.15$, and $p(c_{ik} = 2) = 0.5$. The same distribution is applied to generate the weights (w_{kj}).

For the experiments, various problem sizes are formulated as shown in Table 1. For instance, in problem #1 the total number of agents is 10, and the objective is to assign all 10 agents to four teams such that the gain is maximized. The number of agents in the teams are two, three, two, and three, respectively. In problem #10, the total number of agents is 800, and the objective is to assign 400 of the agents to 100 teams, each of which consists of four agents.

The presented algorithm is implemented in Java, and run on a PC with an Intel Core $i5 - 2.60$ GHz and 4 GB of RAM. For each problem size, 10 replications are executed. The initial population size is set to 50, and the mutation probability is 0.2.

6.1. Accuracy

To verify the accuracy of the algorithm, we consider a scenario in which agents work independently in teams and have no interactions. The problem is thus reduced to a variant of the standard AP, for which the well-known Hungarian method efficiently finds the optimal solution although the search space is equally large. We consider the relative deviation of a solution x_i from the optimal solution x^* expressed as a percentage,

$$dev_i = \frac{100(x^* - x_i)}{x^*}, \tag{14}$$

as a measure of the quality of a solution. Furthermore, for a set of n replications of the GA, we use the average and the standard deviation of dev_i as overall measures of the quality of the GA. That is,

$$avgDev = \frac{1}{n}\sum_{i=1}^{n} dev_i, \tag{15}$$

and

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(dev_i - avgDev)^2}. \tag{16}$$

We conduct a set of experiments on the problems specified in Table 1. For each problem, we run 10 replications, where the search process is terminated after 80 000 fitness evaluations have been performed (the same number of offspring are generated). A comparison of the best results obtained by our algorithm with the optimal solutions (Table 2) shows that these are almost the same for the problem instances 1 to 8. That is, for small- and medium-sized problems, the average and standard deviation from the optimal solution are close to 0.0. However, the quality of the solutions considerably degrades as the size of the problem increases. In the remainder of this paper, we introduce more efficient GAs that also produce high-quality solutions for large problem instances.

In the collaborative case where agents interact with one another, the Hungarian method cannot be applied. Thus, the accuracy of the algorithm is tested using a simple data set given in Tables 4 and 5. The data set is designed in such a manner that the optimal solution can easily be found, and the correctness of the algorithm can be verified. For instance, problem #1 has four tasks and four teams of agents. The numbers of agents in each team are two, three, two, and three. Each agent has the highest value of 4.0 for exactly one attribute, and for each task the attributes that have the highest weight (4.0) determine which agents should be assigned to the task. For example, for task number 1 only the weights w_1 and w_2 of the attributes c_1 and c_2 are 4.0, while the remaining weights are 1.0. The gain will be maximized if this task is assigned

Table 2
Computational results for the GA with a one-point crossover operator, where agents work in teams without interactions. *avgDev* and σ are the mean and standard deviation (for 10 replication) of the dev_i (relative deviation from the optimal solution obtained by the Hungarian method), respectively. The search process is terminated after 80 000 offspring have been generated.

Prob #	Solution in each of the 10 trials										Average execution time (s)	Best found solution	Hungarian method	*avgDev* (%)	σ (%)
1	466	466	466	466	466	466	466	466	466	466	0.16	466	466	0.00	0.00
2	934	934	934	934	931	934	934	931	934	936	0.29	936	936	0.26	0.15
3	876	873	873	873	875	870	868	873	870	869	0.28	876	876	0.46	0.28
4	1202	1205	1201	1203	1206	1207	1205	1204	1203	1207	0.42	1207	1207	0.22	0.16
5	1290	1289	1291	1291	1285	1287	1291	1283	1288	1290	0.43	1291	1291	0.19	0.20
6	2833	2830	2833	2834	2838	2830	2837	2832	2833	2835	0.78	2838	2842	0.30	0.09
7	3825	3817	3817	3814	3823	3824	3822	3823	3816	3818	1.03	3825	3838	0.47	0.10
8	4548	4551	4539	4536	4546	4533	4541	4548	4554	4543	1.21	4554	4581	0.81	0.14
9	17886	17778	17842	17832	17887	17767	17845	17785	17914	17919	5.79	17919	19107	6.60	0.28
10	18020	17939	18090	18063	18067	18064	18079	18012	18047	17934	6.18	18090	21156	14.77	0.25

Table 3
Comparison of GA results with the optimal solution calculated manually for problem 1 consisting of four tasks. g_{s_j} is a team of d_j agents performing task t_j.

Prob #	Solution in each of the 10 trials (using GA Heuristic)										Average execution time (s)	Agents assignment (obtained by GA)	Optimal solution value (calculated manually)	Optimal assignment (calculated manually)
1	346	346	346	346	346	346	346	346	346	346	0.32	$g_{s_1} = \{1, 2\}$ $g_{s_2} = \{3, 4, 5\}$ $g_{s_3} = \{6, 7\}$ $g_{s_4} = \{8, 9, 10\}$	346	$g_{s_1} = \{1, 2\}$ $g_{s_2} = \{3, 4, 5\}$ $g_{s_3} = \{6, 7\}$ $g_{s_4} = \{8, 9, 10\}$

Table 4
Data set for agent attributes (accuracy test).

Agent #	Capabilities values									
	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}
1	4.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
2	1.0	4.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
3	1.0	1.0	4.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
4	1.0	1.0	1.0	4.0	1.0	1.0	1.0	1.0	1.0	1.0
5	1.0	1.0	1.0	1.0	4.0	1.0	1.0	1.0	1.0	1.0
6	1.0	1.0	1.0	1.0	1.0	4.0	1.0	1.0	1.0	1.0
7	1.0	1.0	1.0	1.0	1.0	1.0	4.0	1.0	1.0	1.0
8	1.0	1.0	1.0	1.0	1.0	1.0	1.0	4.0	1.0	1.0
9	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	4.0	1.0
10	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	4.0

Table 5
Data set for task attributes (accuracy test).

Task #	Weights for attributes									
	w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9	w_{10}
1	4.0	4.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
2	1.0	1.0	4.0	4.0	4.0	1.0	1.0	1.0	1.0	1.0
3	1.0	1.0	1.0	1.0	1.0	4.0	4.0	1.0	1.0	1.0
4	1.0	1.0	1.0	1.0	1.0	1.0	1.0	4.0	4.0	4.0

to agents number 1 and 2 in Table 4, because they have the highest values for the capabilities c_1 and c_2. Optimal assignments for the other tasks are determined similarly. The correctness of the algorithm can be verified by considering the results given in Table 3. The results obtained by executing the proposed GA are consistent with the manually determined optimal values.

6.2. Efficiency

As shown in Table 2, the relative deviations from the optimal solutions for the results obtained using the GA for problems 9 and 10 are (on average) 6.60% and 14.77%, respectively. A proposed method to improve the results of the GA is to increase the number of offspring (fitness evaluations). However, increasing the number of fitness evaluations does not monotonically increase the fitness of the best individual, although it always leads to an increase in the execution time, which sometimes may violate time constraints. In order to study the effect of the number of fitness evaluations on the performance of the GA, two sets of experiments were conducted on the problem instances 9 and 10, as shown in Table 1. In these experiments, the average of the relative deviations of the GA results (from the optimal solution determined by the Hungarian method) is calculated as a function of the number of offspring, which is varied from 10 000 to 2 560 000 by factors of 2. The results are summarized in Tables 6 and 7 for problems 9 and 10, respectively.

As the results show, the average execution time is roughly doubled when the number of offspring is doubled, which is an expected behavior. However, the improvement in the results (decrease of the average relative deviation from the optimal solution) does not change at the same rate. For instance, for problem #10 (Table 7) the rate of the improvement is significantly diminished beyond 640 000 fitness evaluations. These results clearly demonstrate that the GA with the one-point crossover operator is not capable of efficiently solving the problem. In order to improve the efficiency of the GA, in the next section we present several crossover operators and suggest modifications to them, and we introduce two new crossover operators.

7. Genetic crossover operators

Crossover is one of the main operators of a GA, and is responsible for exchanging information between chromosomes and producing new solutions. It plays a central role in determining the performance of an algorithm. Efficient GAs that are able to achieve near-optimal solutions in relatively short execution times require a careful selection of the (crossover) operators. In this section, we present several crossover operators that have been employed in the GA literature, demonstrating why they might not be feasible. Moreover, we introduce modifications to these crossover operators, and introduce new operators to address these shortcomings.

Achieving an appropriate balance between exploration and exploitation is crucial to the success of a GA. A search that is too

Table 6
Computational results of the GA with a one-point crossover operator for problem #9 and for a varying number of offspring (the first column denotes the number of fitness evaluations). Agents work in teams without interactions, and the optimal solution is obtained using the Hungarian method. *avgDev* and σ are the mean and standard deviation (for 10 replications) of the dev_i (relative deviation from the optimal solution), respectively.

Number of fitness evaluations	Solution in each of the 10 trials										Average execution time (s)	Best found solution	Hungarian method	*avgDev* (%)	σ (%)
10000	16902	16881	16850	16761	16821	16820	16785	16862	16838	16857	0.83	16902	19107	11.88	0.21
20000	17120	17071	17059	17046	17159	17214	17020	17054	17082	17139	1.51	17214	19107	10.52	0.3
40000	17419	17511	17448	17468	17457	17437	17381	17366	17413	17389	2.93	17511	19107	8.78	0.22
80000	17886	17778	17842	17832	17887	17767	17845	17785	17914	17919	5.79	17919	19107	6.60	0.28
160000	18271	18233	18277	18249	18272	18275	18305	18261	18241	18263	11.46	18305	19107	4.41	0.10
320000	18611	18602	18589	18629	18648	18576	18622	18626	18638	18630	22.76	18648	19107	2.56	0.11
640000	18855	18857	18875	18835	18860	18825	18843	18830	18861	18862	45.46	18875	19107	1.34	0.08
1280000	18970	18972	18975	18965	18985	18945	18966	18959	18960	18946	88.47	18985	19107	0.75	0.06
2560000	19018	18993	18986	19001	18983	19005	18996	18994	19019	18984	178.49	19019	19107	0.57	0.06

Table 7
Computational results for the GA with a one-point crossover operator for problem #10 and with a varied number of offspring (the first column denotes the number of fitness evaluations). Agents work in teams without interaction, and the optimal solution is obtained using the Hungarian method. *avgDev* and σ are the mean and standard deviation (for 10 replications) of the dev_i (relative deviation from the optimal solution), respectively.

Number of fitness evaluations	Solution in each of the 10 trials										Average execution time (s)	Best found solution	Hungarian method	*avgDev* (%)	σ (%)
10000	16945	16885	16924	17085	17073	17062	16908	17019	17040	16974	0.95	17085	21156	19.68	0.33
20000	17248	17453	17332	17293	17237	17347	17344	17238	17237	17235	1.72	17453	21156	18.24	0.32
40000	17656	17762	17677	17539	17693	17590	17762	17589	17456	17638	3.23	17762	21156	16.64	0.43
80000	18020	17939	18090	18063	18067	18064	18079	18012	18047	17934	6.18	18090	21156	14.77	0.25
160000	18343	18533	18391	18503	18439	18506	18533	18428.0	18400	18466	12.12	18533	21156	12.77	0.29
320000	18831	18766	18772	18880	18830	18820	18843	18909	18836	18768	24.16	18909	21156	11.02	0.21
640000	19079	19132	19034	19110	18997	19040	19029	18998	18989	19061	47.38	19132	21156	9.97	0.22
1280000	19169	19177	19142	19231	19141	19216	19155	19033	19017	18956	92.73	19231	21156	9.61	0.41
2560000	19138	19197	19166	19159	19201	19236	19217	19155	19118	19188	208.20	19236	21156	9.35	0.16

exploratory in nature may degenerate the GA into a random walk, where the advantages of the GA are lost. On the contrary, a search that is too exploitative in nature may converge prematurely to a local optimum, and leave large parts of the search space unexplored.

The crossover operator combines parts of two different parent solutions to construct a new offspring solution. However, the combination of two solutions may lead to a new solution that does not respect the imposed constraints. In general, there are two approaches to this problem. The first is to allow the existence of individuals that violate the Constraints, but discriminate them by a high penalty. The other approach, which is adopted here, is to incorporate the constraints into the crossover operator, in order to keep the offspring in the feasible region. Using this approach, an offspring is not formed only by mixing the clusters of genes of two parents. Therefore, one interesting problem is to study the crossover operators to determine the most successful ones.

In this section, we start by explaining several well-known crossover operators, before proceeding to describe the efficient operators that are introduced in this article. We employ the well-known operators both as a base-line to compare our results with and also as a means to intuitively clarify why the new crossover operators perform better.

Before proceeding to the details of these crossover operators, a simple example is described, which will be referred to when necessary. In this example, 10 agents are assigned to four teams, where teams 1 and 3 require three agents each, whereas teams 2 and 4 each require two. Thus, each chromosome consists of 10 genes (positions), where the value of each gene is the index of an agent. In all crossover operators, two parents (named A and B) are chosen from the population using tournament selection.

7.1. Well-known crossover operators

In the following, several crossover operators that have been widely employed in the literature are described. Later, we discuss the pitfalls of these operators, and present modifications to them that address these problems and enhance their efficiency. Furthermore, we will introduce our new operators.

7.1.1. k-point crossover operator

The k-point crossover operator, where k is usually a fixed small number (e.g. 1, 2, or 3), picks k cutting points from $\{1, \dots, n-1\}$ (where n is the number of genes) uniformly at random without replacement. These cutting points divide both parents into $k+1$ segments, which are used to assemble offspring chromosomes. Here, we only describe the details for the one-point crossover operator. Other k-point crossover operators are implemented similarly, with the only difference being that they have k crossover points.

A chromosome with n genes consists of $n-1$ division points (crossover points). In the one-point crossover operator, one of these division points is randomly selected (with equal probability) to divide one of the parents (parent A) into two parts. The values (agent assignments) of the first p genes of parent A are deleted from parent B (where p is the selected crossover point). The first p genes of the offspring 1 are inherited from parent A, and the remaining $n-p$ genes are taken from parent B, in sequence from left to right [34]. After switching the roles of the parents, the same process is applied to generate the second offspring. This crossover operator is illustrated in Fig. 2. The selected crossover point $p=4$ is represented by a vertical line for parent A. The first four genes assignments (8, 2, 6, and 9) of parent A are copied to the corresponding genes of offspring 1, and are deleted from parent B, as shown by small horizontal lines. The remaining six unfixed genes of the offspring are populated by assigning the remaining six genes of parent B (5, 3, 4, 10, 7, and 1) in sequence from left to right, as shown in Fig. 2.

Two flavors of two-point crossover operators are also shown in Figs. 3 and 4. The difference between these two versions of the operator lies in the segments in parent A from which the genes in the offspring 1 are inherited.

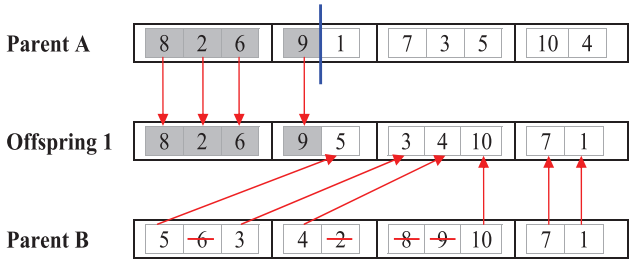

Fig. 2. One-point crossover operator.

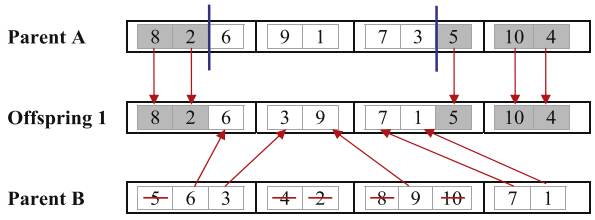

Fig. 3. Two-point v1 crossover operator.

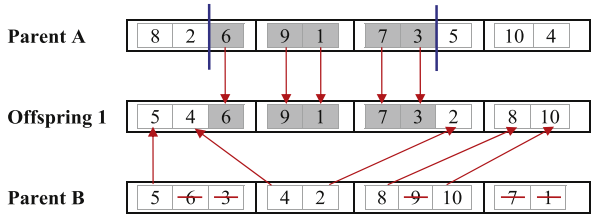

Fig. 4. Two-point v2 crossover operator.

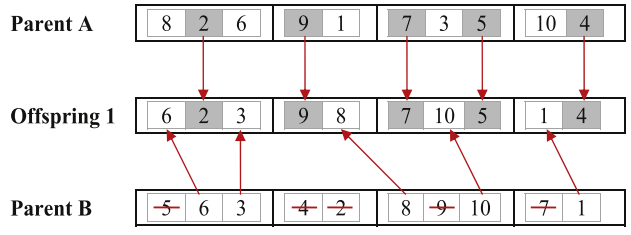

Fig. 5. Position-based crossover operator.

7.1.2. Position-based crossover operator

In the position-based crossover operator, each position (gene) in parent A is selected randomly with a probability of 0.5. The values of these selected positions (agent numbers) are copied into the corresponding positions of the offspring. The values of the already selected genes are deleted from parent B. The remaining values of the genes from parent B are copied into the empty positions in the offspring, in sequence from left to right [16]. For instance, in Fig. 5, the genes numbered by 2, 4, 6, 8, and 10 are randomly selected from parent A. The corresponding values (2, 9, 7, 5, and 4) of the selected genes are copied into the corresponding positions of the offspring. These selected values (agent numbers) are deleted from parent B, and starting from the left the remaining values (6, 3, 8, 10, and 1) are copied into the empty positions of the offspring in sequence. Similarly, another offspring is generated by switching the roles of the parents.

7.1.3. Order-based crossover operator

The order-based crossover operator is a variation of the position-based crossover operator, and was proposed by Syswerda [39]. A set of positions in parent A are selected with a probability of 0.5 for each position (gene). The values (assigned agent numbers) in these selected genes are deleted from parent B, and the remaining values of B are copied to the corresponding genes of the generated offspring 1. Then, the selected genes values of parent A are copied to the remaining unfixed genes of offspring 1 in sequence from left to right. In Fig. 6, the gene numbers 2, 4, 6, 8, and 10 of parent A are selected randomly with a probability of 0.5. The corresponding values of the selected genes (2, 9, 7, 5, and 4) are deleted from parent B. The remaining values (6, 3, 8, 10, and 1) are copied to the corresponding positions (genes) of the offspring. Subsequently, the selected agent numbers (2, 9, 7, 5, and 4) are copied to the unfixed positions of the offspring in sequence. Similarly, another offspring is generated by switching the roles of the parents.

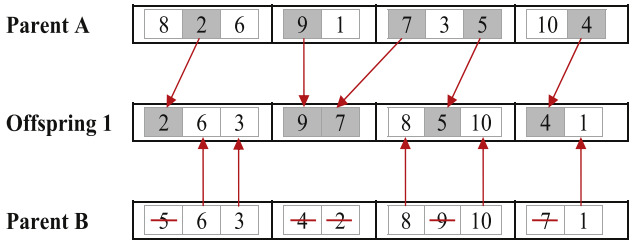

Fig. 6. Order-based crossover operator.

7.2. Modified and new crossover operators

The crossover operators presented above suffer from at least two problems:

- As the genes from parent B are copied to the offspring after deleting the genes that are already represented in parent A, the order of the genes is largely disturbed, and valuable information stored in parent B is lost. The GA is based on the survival of the fittest, and the assumption that the solutions that survive will pass their characteristics to new generations. Disturbing the order of the genes in one of the parents appears to weaken the GA algorithm.
- In the instances of the problem where the number of agents is considerably larger than the number of required agents, i.e., problem instances #10 and #11, agents that are not part of the initial population will not efficiently be incorporated in new solutions (other than through mutation). Therefore, a more powerful exploration mechanism is required to reach the part of the search space where agents not chosen in the initial population are also involved.

In the following, we suggest modifications to the standard crossover operators presented in Section 7.1 and also introduce two new crossover operators to address these shortcomings.

7.2.1. k-point shuffled list crossover operator

The *k*-point shuffled list crossover operator represents an extension of the *k*-point operator, and uses a shuffled repair list to doctor the unfixed genes in the generated offspring. The principle is illustrated in Fig. 7 for the case of the one-point crossover operator. The crossover point p is selected, and the first $p = 4$ gene assignments (8, 2, 6, and 9) of parent A are copied to the corresponding genes of offspring 1, similarly as above for the simple one-point crossover. These selected values are deleted from parent B. Instead of populating the remaining six genes of the offspring from parent B from left to right in sequence, the values of the undeleted genes of B from $p = 4$ onward (which are 10, 7, and 1 in Fig. 7) are copied to the corresponding positions of the offspring 1. The remaining unfixed genes of the offspring are marked

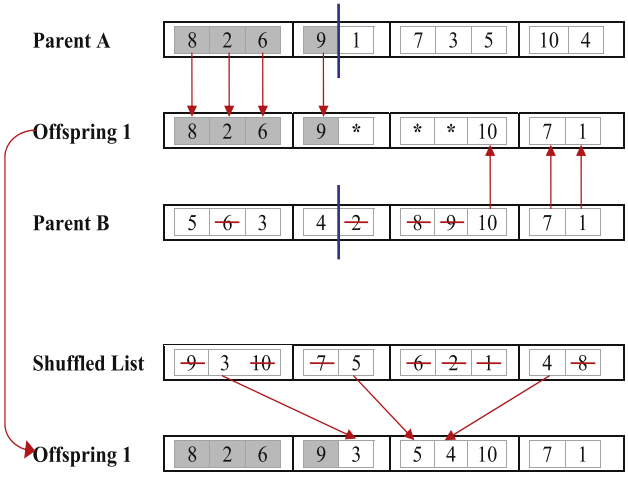

Fig. 7. One-point shuffled list crossover operator.

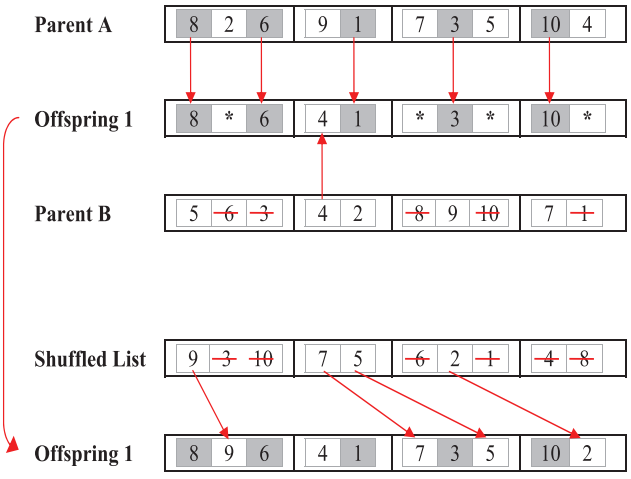

Fig. 8. Position-based shuffled list crossover operator.

by symbol *. In order to fix these marked genes, the operator employs a shuffled repair list. The length of the shuffled repair list is equal to the total number of agents available A_t. It is a shuffled list containing the agent numbers $\{1, 2, \ldots A_t\}$. The gene symbols (8, 2, 6, 9, 10, 7, and 1) that are already present in the offspring are deleted from the shuffled repair list. The values of the leftover genes are copied in sequence (from left to right) from the shuffled repair list to the generated offspring's genes marked with *, as shown in Fig. 7. The other k-point shuffled list crossover operators follow the same principle, and they differ only in the number of crossover points.

7.2.2. Position-based shuffled list crossover operator

In the position-based shuffled list crossover operator, similarly to the simple position-based crossover operator, each position (gene) in parent A is randomly selected with a probability of 0.5. The symbols in these selected positions (agents assigned to tasks) are copied into the corresponding positions of the offspring. The values of genes (agent numbers) that have already been selected are deleted from parent B, and the remaining genes of parent B are copied to the corresponding genes of the offspring. The remaining unfixed genes of the offspring are marked by symbol *, as shown in Fig. 8. In order to fix these marked genes, the operator employs a shuffled repair list, as described in Section 7.2.1. The gene symbols that are already present in the offspring are deleted from the shuffled list. The values of the leftover genes are copied in sequence (from left to right) from shuffled list to the generated offspring's genes marked with *. This method is illustrated by Fig. 8. Similarly, another offspring is generated by switching the roles of the parents.

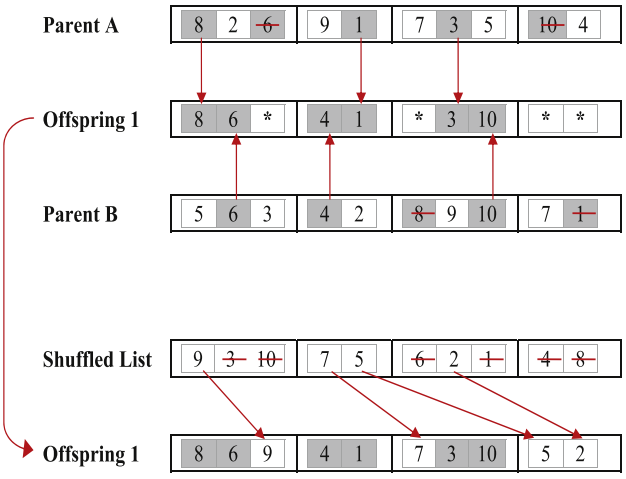

Fig. 9. Uniform shuffled list crossover operator.

7.2.3. Uniform shuffled list crossover operator

In the uniform shuffled list crossover operator, a random number is generated for each position (gene) with a uniform distribution. If the random number is less than 0.5, then the corresponding gene (agent assignment) of parent A is inherited by a new offspring. Otherwise, the corresponding gene is taken from parent B. While inheriting from parent A or B, there is a need to check for duplicate assignments (the inherited gene symbol should not already be present in the offspring). If the gene symbol is already present, then the corresponding position in the offspring is marked by symbol * (called an unfixed gene). In order to fix these marked genes, the operator employs the same shuffled list as discussed earlier. The procedure is illustrated by Fig. 9, where the second offspring is generated by switching the roles of the selected parents.

7.2.4. Random point shuffled list crossover operator

In the random point shuffled list crossover operator, the number of crossover points is selected randomly (with a uniform distribution) between two and $n-1$, where n is total number of genes. If the selected number of crossover points is two, then the operator is similar to the two-point shuffled list described in Section 7.2.1. Similarly, it will be a three-point shuffled list if the selected number of crossover points is three, and so on.

7.2.5. Team-based crossover operator

The team-based crossover operator takes into the account the special characteristics of the problem and the fact that the order of the genes does not affect the solution as long as they correspond to the same task. The goal of this operator is to mitigate the problem of disturbing the order of the genes in parent B during the crossover operation as far as possible, by attempting to assign agents to the same task in the offspring if possible.

In the team-based crossover operator, each position (gene) in parent A is randomly selected with probability p_s (in our experiments, we set $p_s = 0.5$). The symbols in these selected positions (agents assigned to tasks) are copied into the corresponding positions of the offspring, as shown in Fig. 10. The values of already selected genes are deleted from parent B, and the remaining genes of parent B are copied according to the relations of the agents to the tasks. We prioritize copying the remaining genes of parent B

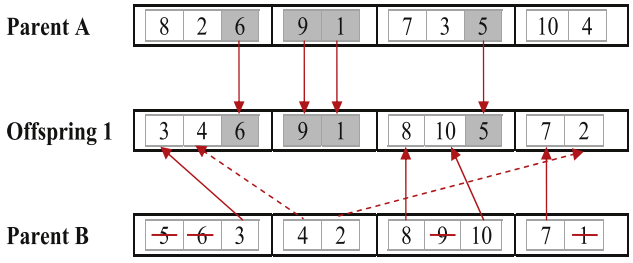

Fig. 10. Team-based crossover operator.

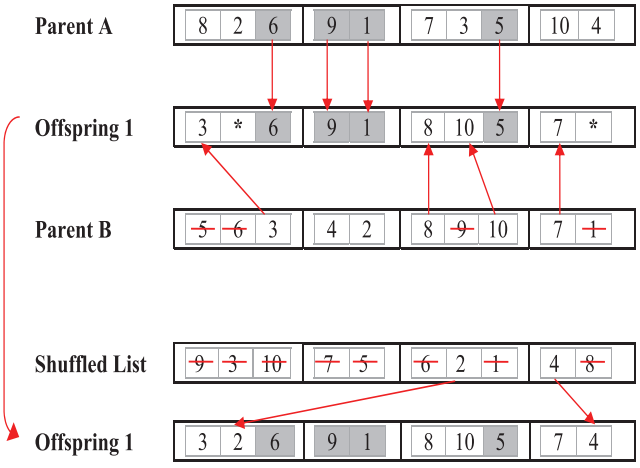

Fig. 11. Team-based shuffled list crossover operator.

to the empty positions of the corresponding tasks in the offspring. The remaining genes that cannot be copied to the corresponding tasks in the offspring are copied in sequence (from left to right) to the empty positions in the offspring. For example, in Fig. 10, after copying the agents 6, 9, 1, and 5 from parent A to offspring 1, agent 3 should be assigned to task 1, agents 8 and 10 to task 3, and agent 7 to task 4. The remaining values (4 and 2) of parent B are copied to the empty positions of the offspring in sequence. Similarly, another offspring is generated by switching the roles of the parents.

7.2.6. Team-based shuffled list crossover operator

In the team-based shuffled list crossover operator, similarly to the simple team-based crossover operator, each position (gene) in parent A is randomly selected with probability p_s (in our experiments we set $p_s = 0.5$). The symbols in the selected positions (agents assigned to tasks) are copied into the corresponding positions of the offspring. The values of already selected genes (agent numbers) are deleted from parent B, and the remaining genes of parent B are copied according to the relations of the agents to the tasks, in the same manner as described in Section 7.2.5. The remaining unfixed genes of the offspring are marked by symbol *, as shown in Fig. 11. In order to fix these marked genes, the operator employs the shuffled repair list. The gene symbols that are already present in the offspring are deleted from the shuffled list. The values of the leftover genes are copied in sequence (from left to right) from the shuffled list to the generated offspring's genes marked with *. Similarly, another offspring is generated by switching the roles of the parents.

8. Experimental results for crossover operators

In this section, a series of computational experiments on large instances of the problem (instances 9, 10, 11, 12, and 13 in Table 1) are presented to compare and analyze the performances of the crossover operators proposed in Section 7.

The aim of these experiments is to first identify the most efficient crossover operators, which are able to yield near-optimal solutions to our AP within a short computing time (on different population sizes and a small number of fitness evaluations), and to fine-tune the relevant GA parameters to further improve the results.

8.1. Experiment setup

Computational experiments are performed on the problem instances #9 – #13 (given in Table 1). In the first problem instance (problem #9), the total number of agents is equal to the required number of agents, i.e., all available agents are assigned to teams. The numbers of agents and tasks are 400 and 100, respectively. Each task requires four agents. In problem #10, the total number of agents is greater than the required number of agents. The numbers of agents and tasks are 800 and 100, respectively. Each task again requires four agents. In problem #11, the numbers of agents and tasks are 1600 and 100 respectively, where each task requires four agents. Problem #12 consists of 1600 agents and 100 tasks, where a team of 16 agents is assigned to each task. In the final problem (problem #13), the numbers of agents and tasks are 400 and 200, respectively, where each task requires two agents.

These five APs are motivated by real-world scenarios, where the aim is to assign recruited soldiers to different position types. In Sweden, new recruits are trained in basic military training (GMU in Swedish) for a period of three months. The number of recruits on each occasion is to the order of several hundred, and they should be placed in different working areas and geographical garrisons. The combination of working areas and geographical placements is used to cluster the available positions into around 100 clusters. Each recruit has several capabilities (e.g., physical strength and high school grades), and for different type of positions these capabilities may be differently weighted. Although the recruits do not necessarily work as a team, the added value cannot be calculated as a sum of the values of the agents. In the same manner as described in Section 6, one may define a function and calculate $v(a_i, g_{s_j}, t_j)$, i.e., the value added when a_i is assigned to t_j, within the group g_{s_j}.

For these problem instances, we compare the performances of the crossover operators for solving the problem defined in Eq. (13). The search process is terminated after M offspring (called M fitness evaluations) have been generated. For each experiment, 10 independent runs are conducted for each crossover operator introduced in Section 7. The GA is started from the same randomly generated initial population. However, because in the general case the optimal solution is unknown, a modified version of the definition presented in Section 6.1 is adopted to define the relative deviation (Eq. (14)), i.e.,

$$dev_i = \frac{100(x^b - x_i)}{x^b}, \tag{17}$$

where the optimal solution x^* is substituted for the best global solution found, x^b. In the following experiments, unless otherwise specified the value x^b is the best solution value obtained by executing our GA with all the discussed crossover operators. For each crossover operator, the GA terminates after 320 000 fitness evaluations. The best solution values obtained for the problem instances 9, 10, 11, 12, and 13 are 22 632.08, 24 731.92, 25 737.67, 96 201.58, and 21 006.42, respectively. The values of *avgDev* and σ are calculated accordingly, as defined in Eqs. (15) and (16), respectively.

Obviously, the mean and standard deviation of the dev_i only indicate how close the solution of an algorithm is to the best global solution found, and do not provide a rigorous analysis of

Table 8
Computational results of the GA with the one-point shuffled list crossover operator for problem #10 and a varying number of offspring. Agents work in teams without interactions, and the optimal solution is obtained using the Hungarian method. *avgDev* and σ are the mean and standard deviation (for 10 replications) of the dev_i (relative deviation from the optimal solution), respectively.

Number of fitness evaluations	Solution in each of the 10 trials										Average execution time (s)	Best found solution	Hungarian method	*avgDev* (%)	σ (%)
10000	18254	18213	18285	18316	18301	18275	18201	18317	18268	18273	0.99	18317	21156	13.64	0.18
20000	18913	18936	18864	19057	18983	19000	18973	19002	18958	18905	1.85	19057	21156	10.38	0.25
40000	19574	19453	19491	19471	19504	19608	19532	19451	19575	19555	3.57	19608	21156	7.73	0.25
80000	19985	19997	20049	19982	19969	19952	19976	19958	20004	19977	6.97	20049	21156	5.54	0.12
160000	20372	20297	20383	20308	20355	20307	20341	20399	20348	20328	13.71	20399	21156	3.84	0.15
320000	20670	20719	20646	20596	20646	20652	20659	20630	20669	20648	27.03	20719	21156	2.38	0.14
640000	20868	20863	20845	20883	20869	20833	20870	20847	20846	20864	53.15	20883	21156	1.40	0.07
1280000	20971	20966	20969	20984	20947	20946	20940	20974	20991	20972	106.33	20991	21156	0.90	0.08
2560000	20991	20994	21022	20985	20993	21015	20976	20987	20974	20988	214.38	21022	21156	0.77	0.07

the efficiency of the algorithm, unless the best solution found is sufficiently close to the optimal solution (i.e., is a near-optimal solution).

In Section 6.2, we showed that the GA with the one-point crossover operator is not capable of providing near-optimal solutions for at least problem instance 10, even after a large number of fitness evaluations.

In order to ensure that at least some of the presented algorithms can find near-optimal solutions, so that we can obtain a global near-optimal solution to compare the results with, we test the algorithm with the one-point shuffled crossover operator for solving problem instance 10 in the case that there are no interactions between agents. Then, we compare the results with the optimal solution provided by the Hungarian method. The results are summarized in Table 8, showing that after 320 000 fitness evaluations the average relative deviation from the optimal solution is 2.38. The ability of the one-point shuffled list to find a near-optimal solution for the special case where agents do not collaborate indicates that the algorithm is also capable of finding near-optimal solutions in the general case when the agents collaborate. This result can be compared with 11.02 for the same number of fitness evaluations using the original one-point crossover (Table 7).

8.2. Comparison and discussion

In the following, we compare the efficiency of our suggested crossover operators with the baseline operators (on different population sizes, numbers of fitness evaluations, and so on), and elaborate on how other parameters such as the mutation rate influence the efficiency of the best operators found for the AP.

8.2.1. Performance comparison for different crossover operators on a varying population size

In the first set of experiments, we study how different crossover operators perform on problem instances 9–13 (see Table 1) on a varying population size (given a fixed number of fitness evaluations). The crossover operators considered in this set of experiments are the one-point crossover (ONE_POINT), two-point crossover (TWO_POINT), three-point crossover (THREE_POINT), position-based crossover (PBX), and their corresponding versions with a shuffled repair list, denoted by (operator_name_SHUFFLE). The mutation is applied by applying one swap operation, where two genes are randomly chosen and their values are exchanged. The mutation probability is set to 20%.

The results of these experiments are shown in Figs. 12–17. The first three figures depict the experiments on problem #9, where the numbers of fitness evaluations are set to 10 000, 40 000, and 80 000, respectively. In Figs. 15–17, the results of experiments on the problem instances 10 – 12 are illustrated, with the number of fitness evaluations set to 10 000. In Figs. 12–17 the initial population size, depicted on the horizontal axis, is varied from 2 to 1024. The vertical axis represents the quality of the solutions, which is defined as the average percentage deviation from the best solution value obtained after 320 000 fitness evaluations.

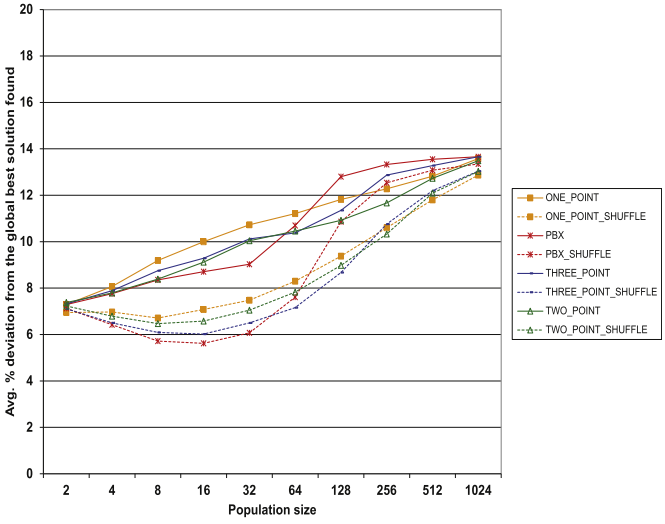

Fig. 12. Effect of the initial population size on the quality of solutions (results on problem #9 with the number of fitness evaluations = 10 000).

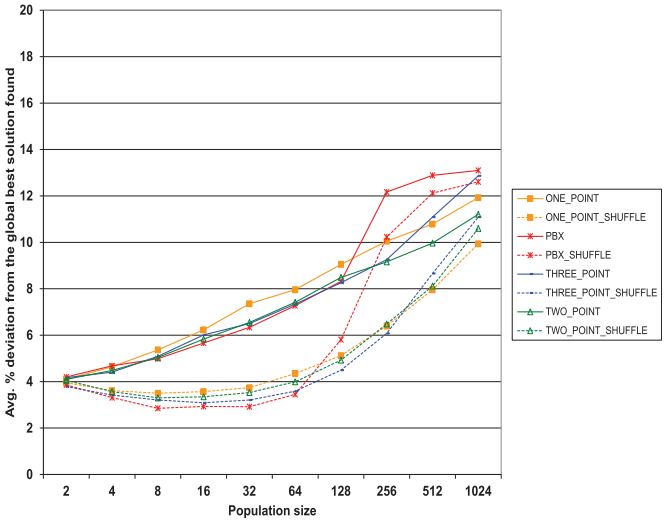

Fig. 13. Effect of the initial population size on the quality of solutions (results on problem #9 with the number of fitness evaluations = 40 000).

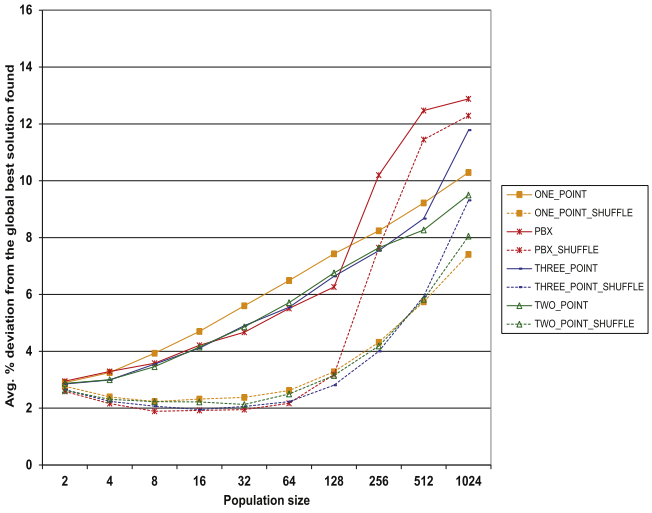

Fig. 14. Effect of the initial population size on the quality of solutions (results on problem #9 with the number of fitness evaluations = 80 000).

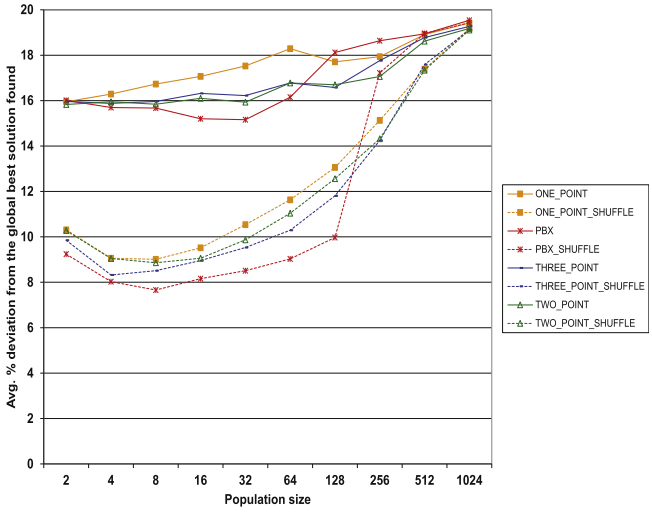

Fig. 15. Effect of the initial population size on the quality of solutions (results on problem #10 with the number of fitness evaluations = 10 000).

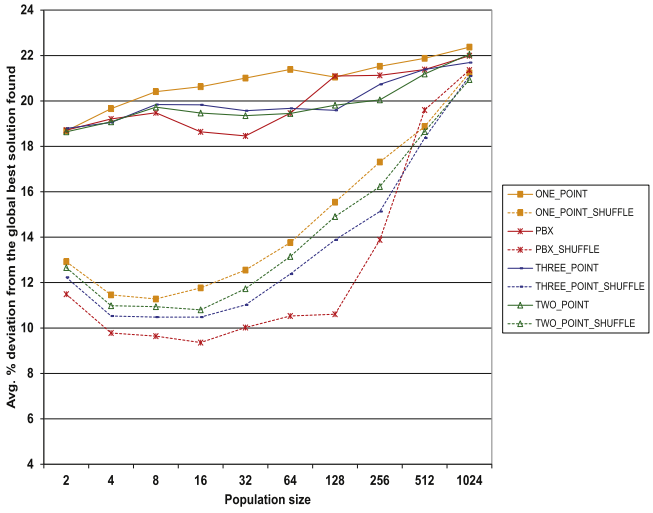

Fig. 16. Effect of the initial population size on the quality of solutions (results on problem #11 with the number of fitness evaluations = 10 000).

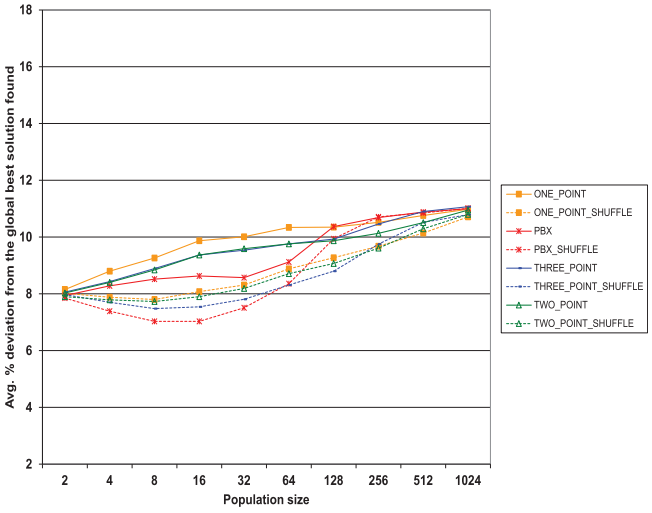

Fig. 17. Effect of the initial population size on the quality of solutions (results on problem #12 with the number of fitness evaluations = 10 000).

Figs. 12–17 show that the simple crossover operators (without any shuffled list) continue to degrade as we increase the initial population size for a given number of fitness evaluations. For example, in Fig. 12 the average percentage deviations for the position-based crossover operator (PBX) are 7.29, 8.35, and 10.7 when the initial population sizes are 2, 16, and 64, respectively. In case of operators with a shuffled list, the quality of the solutions improves as we increase the initial population size to a certain limit, and thereafter it continues to degrade. For example, in the case of the position-based shuffled list crossover operator (PBX_SHUFFLE), the quality of solutions improves when the initial population size varies from 2 to 16, and then degrades thereafter. The average percentage deviations are 7.13, 5.62, and 7.61, when the initial population sizes are 2, 16, and 64, respectively (Fig. 12).

This behavior can be explained by the fact that for a fixed number of fitness evaluations, a larger population size introduces a stronger exploration property to the search method. Operators without a shuffled list already suffer from lack of sufficient exploitation, and this is enhanced as the population size is increased. Most of the operators with a shuffled list provide the best results when the initial population size is between 8 and 16. Their performance begins to degrade as the initial population size increases to 32 and above.

We further analyze the results depicted in Figs. 12–17 to compare the effect of the shuffled list on different operators, especially in the interesting population size range (i.e., 8 to 64).

For instance, in Fig. 12 when the initial population size is 8, the average percentage deviation for the simple one-point crossover operator is 9.19, while this decreases to 6.71 for the one-point crossover with a shuffled list (an almost 27% improvement in the quality of the solution). Similarly, the results for the two-point crossover without and with the shuffled list are 8.39 and 6.15, respectively. Other operators also exhibit similar behavior, i.e., the versions with the shuffled list provide better results compared with those without the shuffled list. This can also be observed in Figs. 13 and 14, where the total number of fitness evaluations is set to 40 000 and 80 000, respectively.

The superiority of the shuffled list operators can be observed more clearly in Figs. 15 and 16, where the total number of agents is greater than the required number of agents (problem #10 and problem #11). For example, in problem #10 when the initial population size is 8, the average percentage deviation for the simple one-point crossover operator is 16.73, while this decreases to 9.01

for the one-point crossover with the shuffled list (an almost 46% improvement in the quality of the solution).

Summarizing Figs. 12–17, we can conclude that the crossover operators with shuffled lists achieve superior performances to the corresponding operators without shuffled lists for all initial population sizes, especially when the population size is in the range between 8 and 64. Moreover, all operators with shuffled lists perform better than all those without shuffled lists (except for the position-based crossover with a shuffled list, which degrades for initial population sizes larger than 128).

For a fixed population size (up to a certain limit), operators within each group of crossover operators (with and without shuffled lists) can be ordered by their performance. Figs. 12–17 confirm that operators with a higher number of crossover points generally exhibit a superior performance. The order is as follows: one-point crossover, two-point crossover, three-point crossover, position-based crossover. The same order is preserved within the corresponding operators with shuffled lists. For the position-based crossover operator, the number of crossover points is approximately $n/2$, where n is the total number of genes. This number is significantly larger than the number of crossover points for other operators.

To explain the inferior performances of the operators with smaller numbers of crossover points, the position-based crossover is compared with the one-point crossover operator. While in the former all genes have the same probability of being chosen, in the latter the starting sequence of genes from the first parent is transferred to future offspring with a very high probability. That is, the inheritance of genes is biased, and depends on their position in the chromosome. As a result, some genes may be transferred to future generations although they are not desirable. This bias decreases when the number of crossover points is increased. The position-based crossover is completely free of this bias.

However, as Figs. 12–17 show, the performances of operators with a larger number of crossover points become worse at higher rates when the population size increases while number of fitness evaluations is kept fixed. For example, in Fig. 12, for an initial population size greater than 64 the position-based shuffled list operator provides poor results compared with the other shuffled list operators. The simultaneous increase in the population size and the number of crossover points results in more exploration of the search space than required, leading to a late convergence.

8.2.2. Performance comparison for different crossover operators with a varying number of fitness evaluations

To ensure that the superior performance of operators with shuffled lists holds for different numbers of fitness evaluations, a second set of experiments is performed. In these experiments, we fix the initial population size and observe how different crossover operators behave under different numbers of fitness evaluations. The results for experiments on problem #9 with population sizes equal to 8, 64, and 128 are presented in Figs. 18, 19, and 20, respectively. The results for experiments on problem instances 10, 11, and 13 with a population size equal to 8 are shown in Figs. 21, 22, and 23 respectively. The studied crossover operators are the same as in the first set of experiments above.

Figs. 18–23 show that for different numbers of fitness evaluations from 5000 to 320 000, the operators with shuffled lists exhibit a globally superior performance to the simple genetic operators (without shuffled lists).

Fig. 20 shows that the position-based shuffled list requires 80 000 fitness evaluations before it can provide high quality results. In other words, operators with high numbers of crossover points are more Sensitive, and their performance may degrade if the initial population size is not carefully chosen.

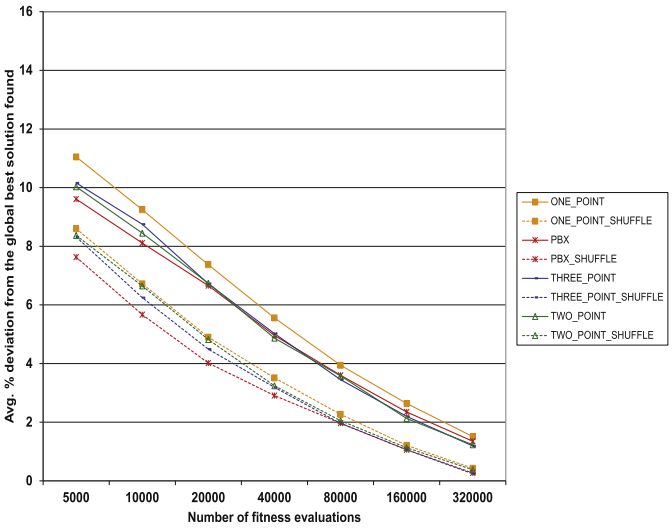

Fig. 18. Effect of the number of fitness evaluations on the quality of the solution (results on problem #9 with an initial population size = 8).

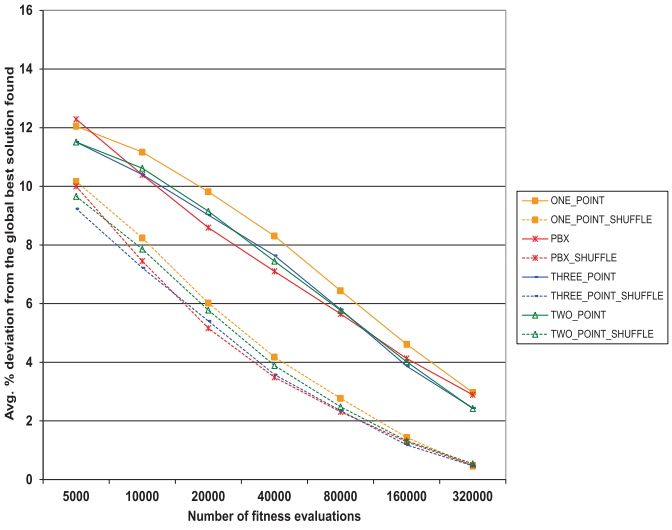

Fig. 19. Effect of the number of fitness evaluations on the quality of the solution (results on problem #9 with the initial population size = 64).

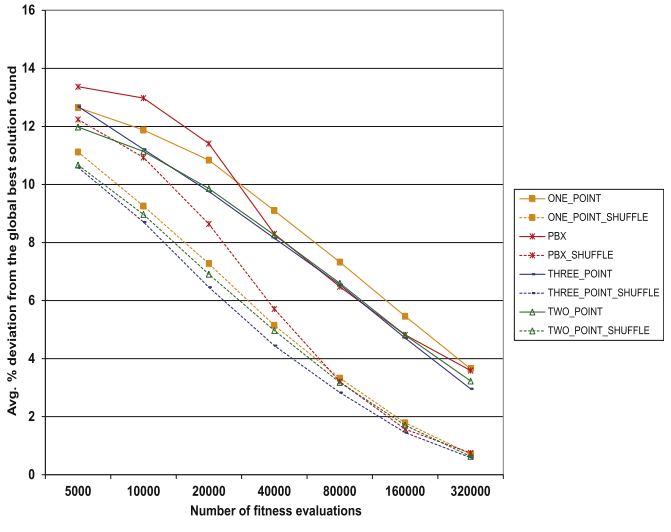

Fig. 20. Effect of the number of fitness evaluations on the quality of the solution (results on problem #9 with the initial population size = 128).

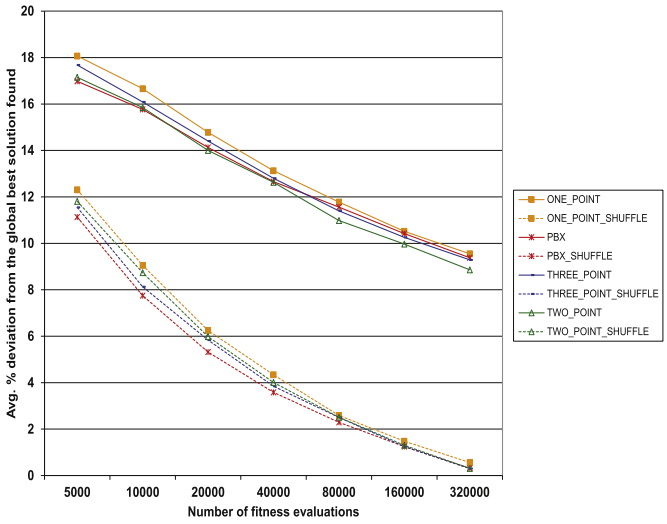

Fig. 21. Effect of the number of fitness evaluations on the quality of the solution (results on problem #10 with the initial population size = 8).

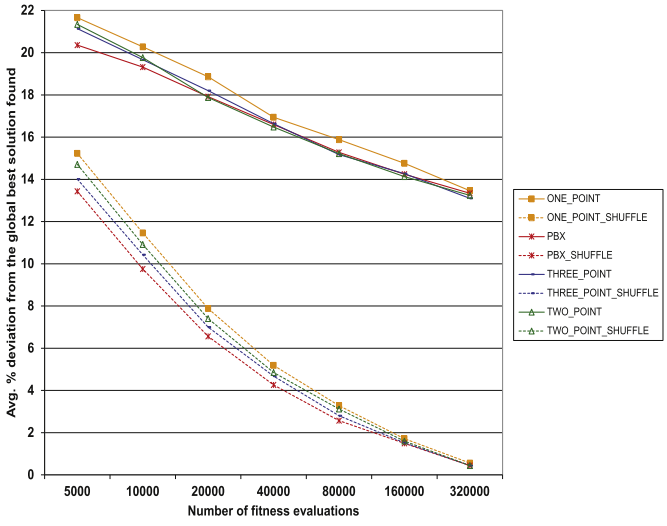

Fig. 22. Effect of the number of fitness evaluations on the quality of the solution (results on problem #11 with the initial population size = 8).

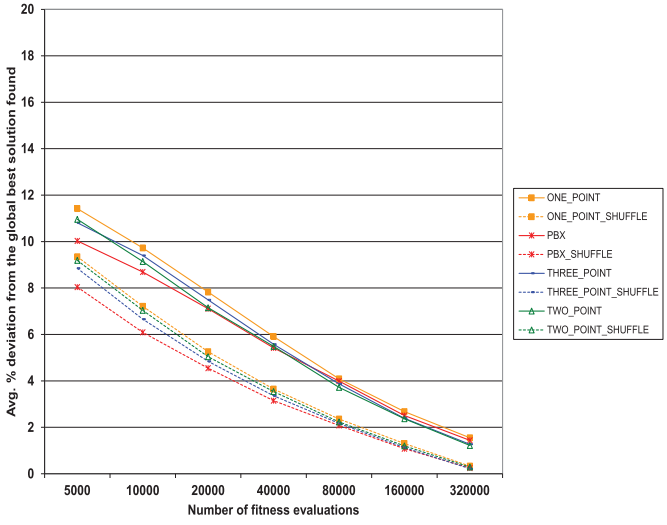

Fig. 23. Effect of the number of fitness evaluations on the quality of the solution (results on problem #13 with the initial population size = 8).

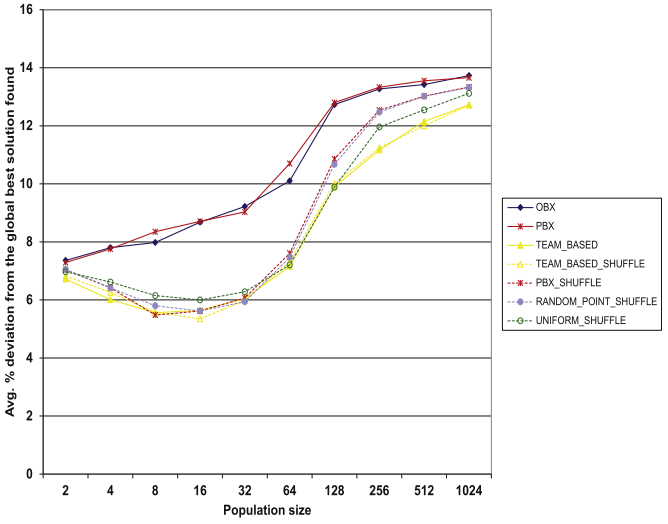

Fig. 24. Comparison of the position-based shuffled list with the other remaining implemented crossover operators (results on problem #9 with the number of fitness evaluations = 10 000).

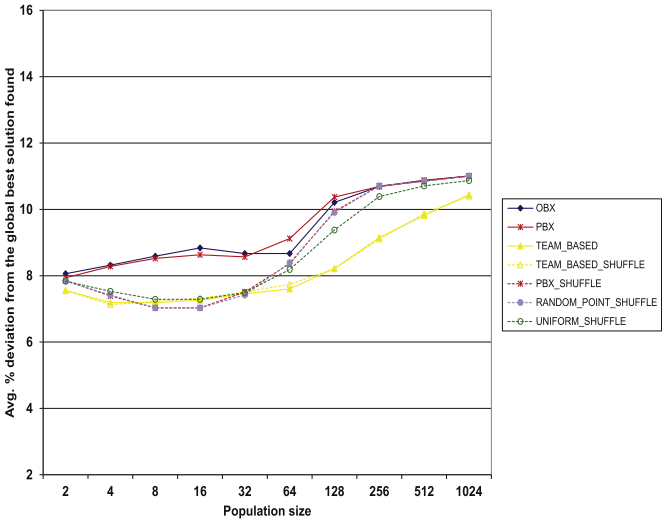

Fig. 25. Comparison of the position-based shuffled list with the other remaining implemented crossover operators (results on problem #12 with the number of fitness evaluations = 10 000).

8.2.3. Performance comparison of other crossover operators

So far, we have shown that for the one-point, two-point (first flavor), three-point, and position-based crossover operators, the crossover operators with shuffled lists exhibit a superior performance compared with those without shuffled lists. In order to confirm that this finding holds for a wider range of operators, in a third set of experiments we compare the position-based operator (both with and without shuffled lists) with two other operators without shuffled lists, namely order-based (OBX) and team-based (TEAM_BASED) crossover operators, and three more operators with shuffled lists, namely random point shuffled list (RANDOM_POINT_SHUFFLE), uniform shuffled list (UNIFORM_SHUFFLE), and team-based shuffled list (TEAM_BASED_SHUFFLE) operators (see Section 7).

The results, illustrated in Figs. 24–27, demonstrate that the operators with shuffled lists (position-based shuffled list, random point shuffled list, and uniform shuffled list) and the team-based operators (with and without shuffled lists) all achieve stronger performances, i.e., smaller average deviations from the best known

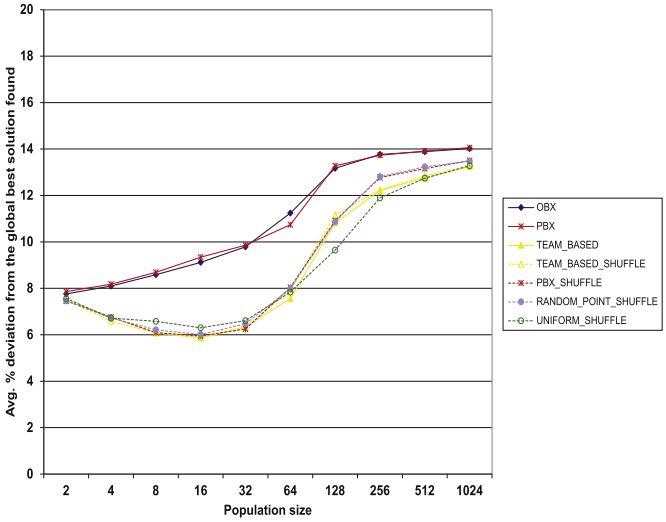

Fig. 26. Comparison of the position-based shuffled list with the other remaining implemented crossover operators (results on problem #13 with the number of fitness evaluations = 10 000).

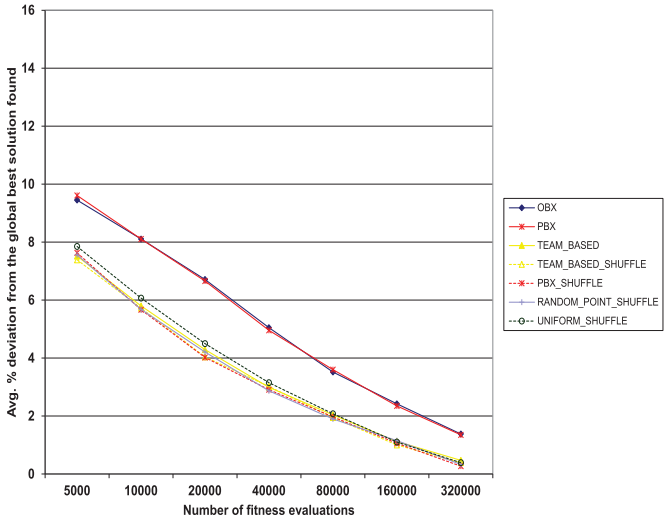

Fig. 27. Comparison of the position-based shuffled list with the other remaining implemented crossover operators (results on problem #9 with the initial population size = 8).

solution. A common feature of these operators is that the offspring at a higher degree inherit the genes from both their parents, either by using a shuffled list or (in the case of team-based operators) by taking advantage of the structure of the problem (see Sections 7.2.5 and 7.2.6).

However, our extensive experiments demonstrate that among the efficient operators, the performance of the position-based shuffled list operator can be enhanced more than the other operators by tuning its parameters. We do not include these experiments in our study, but continue by presenting two sets of experiments in which we tune the parameters of the position-based shuffled list operator for the best results.

8.2.4. Fine-tuning the GA with the position-based shuffled list crossover operator

In the fourth set of experiments, we compare different position-based (PBX) and position-based shuffled list (PBX_SHUFFLE) operators, where each position (gene) in parent A is randomly selected with a different probability (40%, 50%, 60%, 70%, 80%, and 90%). PBX_N means that *N*% of the genes of parent A are copied into

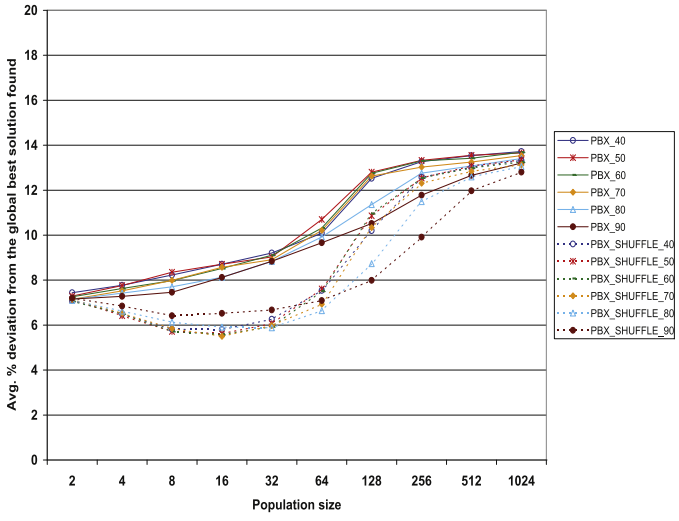

Fig. 28. Effect of the conservative approach (results on problem #9 with the number of fitness evaluations = 10 000).

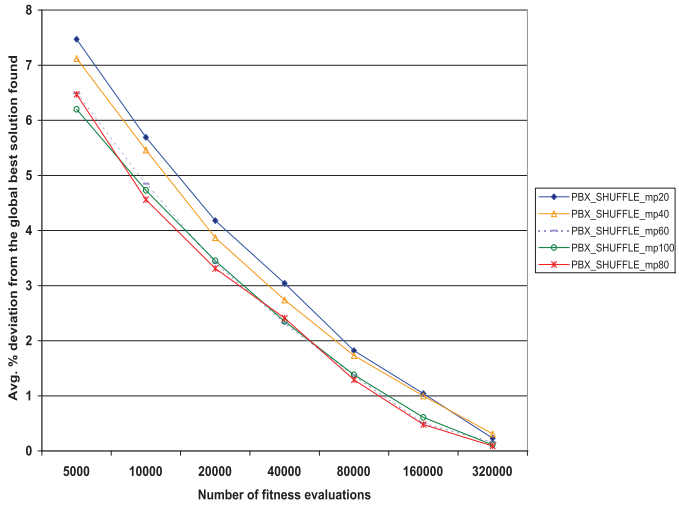

Fig. 29. Comparison of position-based crossover operators with different mutation probabilities (results on problem #9 with the initial population size = 8).

the corresponding positions of the offspring. A higher probability of choosing genes from parent A means that the exploitation property of the search method is increased (at the cost of exploration), i.e., a more conservative search strategy is enacted. PBX_90 is more conservative than PBX_80, and PBX_80 is more conservative than PBX_70, etc.

As seen in Fig. 28, a more conservative search strategy has a positive effect on the position-based operator without a shuffled list, which, as discussed above, is too "exploratory." For the position-based shuffled list operator, a search strategy that is too conservative, such as PBX_SHUFFLE_90, has a moderating effect on the performance, i.e., it degrades the performance if it was good (less than 64) and improves it if it was bad (larger than 64). This behavior is consistent with the observations in the first set of experiments. Fig. 28 shows that when the initial population size is around 16, PBX_SHUFFLE_70 achieves the strongest performance for 10 000 fitness evaluations.

In the fifth set of experiments, the effect of the mutation probability on the quality of the results of position-based shuffled list operator is studied. Five different mutation probabilities (20%, 40%, 60%, 80%, and 100%) are considered, and the results are presented in Fig. 29, where PBX_SHUFFLE_mpN means that the

Table 9
Statistical results for different GA crossover operators using one-tailed t-test with a 99% confidence interval for large-sized problems (9, 10, 11, 12, and 13). Iterations = 10 000, population size = 8, and *avgDev* and σ are the mean and standard deviation (for 10 replications) of the dev_i (relative deviation from the global best solution), respectively.

Problem #	Crossover Operator 1	*avgDev*(%)	σ (%)	Crossover Operator 2	*avgDev*(%)	σ (%)	***p* value**	**Significantly different?**
9	ONE_POINT	9.25	0.17	ONE_POINT_SHUFFLE	6.72	0.23	< 0.001	Yes
	THREE_POINT	8.75	0.21	THREE_POINT_SHUFFLE	6.25	0.34	**< 0.001**	Yes
	PBX	8.11	0.18	PBX_SHUFFLE	5.67	0.22	< 0.001	Yes
	PBX	8.11	0.18	TEAM_BASED_SHUFFLE	5.72	0.19	0	Yes
10	ONE_POINT	9.19	0.32	ONE_POINT_SHUFFLE	6.71	0.24	0.001	Yes
	THREE_POINT	8.75	0.24	THREE_POINT_SHUFFLE	6.09	0.23	< 0.001	Yes
	PBX	8.35	0.31	PBX_SHUFFLE	5.48	0.21	< 0.001	Yes
	PBX	8.35	0.31	TEAM_BASED_SHUFFLE	5.57	0.23	< 0.001	Yes
11	ONE_POINT	9.26	0.12	ONE_POINT_SHUFFLE	7.8	0.07	0	Yes
	THREE_POINT	8.89	0.15	THREE_POINT_SHUFFLE	7.48	0.16	< 0.001	Yes
	PBX	8.52	0.19	PBX_SHUFFLE	7.03	0.13	< 0.001	Yes
	PBX	8.52	0.19	TEAM_BASED_SHUFFLE	7.22	0.09	< 0.001	Yes
12	ONE_POINT	20.41	0.6	ONE_POINT_SHUFFLE	11.28	0.58	0	Yes
	THREE_POINT	19.84	0.66	THREE_POINT_SHUFFLE	10.48	0.33	0	Yes
	PBX	19.48	0.41	PBX_SHUFFLE	9.64	0.19	0	Yes
	PBX	19.48	0.41	TEAM_BASED_SHUFFLE	10.19	0.35	0	Yes
13	ONE_POINT	9.73	0.35	ONE_POINT_SHUFFLE	7.21	0.1	< 0.001	Yes
	THREE_POINT	9.4	0.24	THREE_POINT_SHUFFLE	6.66	0.22	< 0.001	Yes
	PBX	8.69	0.26	PBX_SHUFFLE	6.09	0.29	< 0.001	Yes
	PBX	8.69	0.26	TEAM_BASED_SHUFFLE	6.07	0.22	< 0.001	Yes

Table 10
Parameter values used in different algorithms.

GA and chaotic GA with PBX_Shuffle		PSO		Differential evolution	
Parameter	Value	Parameter	Value	Parameter	Value
Population size	10	Swarm size	10	Population size	10
Crossover rate	1.0	C_1 = social parameter	1.5	CR = crossover rate	0.9
Mutation probability	0.2	C_2 = cognitive parameter	1.5	F = scaling factor	0.5
Mutation operator	Swap	w = inertia weight (upper and lower)	0.9		
Selection operator	Tournament				

mutation probability for each generated offspring is set to *N*%. The initial population size is set to 16, and PBX_SHUFFLE_70 is considered, which is a position-based shuffled list (PBX_SHUFFLE) operator where each position (gene) in parent A is selected with a probability of 70%.

Analyzing Fig. 29 shows that increasing the mutation probability (from 20% to 80%) generally improves the results. For instance, if we consider PBX_SHUFFLE for 10 000 fitness evaluations, the average percentage deviation of the solutions from the best solution decreases from 5.69% to 4.56% upon increasing the mutation probability from 20% to 80%. This behavior can be attributed to the improved exploration property of the search method. However, setting a higher mutation probability than necessary can also degrade the quality of the solution. For example, the 100% mutation probability generally exhibits a higher deviation from the best solution than the 80% mutation probability. An appropriate parameter setting is crucial for maintaining a good balance between exploration and exploitation.

8.2.5. Significance test results for crossover operators

Statistical significance tests were performed to compare the different crossover operators. For each experiment, 10 independent runs were conducted and one tail t-test was adopted to compare the results. Significance tests were only performed for the large-sized problems (9, 10, 11, 12, and 13) presented in Table 1. The number of iterations was set as 10 000, and population size was eight.

Table 9 presents the results (p value) of the one-tail t-test at a 99% confidence interval. For the performance evaluation of the proposed SHUFFLE list operators, the ONE_POINT, THREE_POINT, and PBX crossover operators were compared with the ONE_POINT_SHUFFLE, THREE_POINT_SHUFFLE, and PBX_SHUFFLE counterparts, respectively. The p values in Table 9 show that the ONE_POINT_SHUFFLE, THREE_POINT_SHUFFLE, and PBX_SHUFFLE operators significantly outperform the ONE_POINT, THREE_POINT, and PBX crossover operators, respectively. These statistical test results indicate that the k-point shuffled list crossover operators achieve significantly better results compared with the simple k-point crossover operators.

The results of our proposed TEAM_BASED_SHUFFLE operator are only compared with the PBX operator, because the PBX operator outperforms all other k-point crossover operators. Table 9 shows that TEAM_BASED_SHUFFLE significantly outperforms the PBX operator at the 99% confidence interval, because the p value is less than 0.001 for all large-sized problems.

9. Comparison of GA with other nature-inspired metaheuristics and an approximation algorithm

The “no free lunch” theorem of Wolpert and Macready [42] states that no algorithm is perfect. This means that each algorithm is only effective for particular problems. However, a comparison can provide insight into the algorithm, and demonstrate which algorithm is suitable for which problem.

9.1. Comparison of GA with PSO and differential evolution algorithm

We compared the performances of four different search algorithms, namely PSO [11,26], DE [38], GA (using PBX_SHUFFLE), and chaotic GA (using PBX_SHUFFLE). The aim of the experiments described in this section is to identify the most effective search algorithm for our problem.

Chaotic number generators, developed from chaotic systems, have been employed to generate random numbers, yielding superior results. The use of chaotic sequences in evolutionary algorithms was introduced and proposed by Yang and Chen [43], and Caponetto et al. [5]. We have also employed chaotic sequences to improve the results of the GA algorithm. The logistic map function, characterized by Eq. (18) [30], was adopted in our experiments:

$$x_{k+1} = \alpha . x_k (1 - x_k) \tag{18}$$

where α is a control parameter for generating numbers in (0,1), and is set to 4. We employed chaotic sequences for the initial population generation, parent selection, crossover operators, and mutation probability. Regarding the crossover operators, experiments were conducted using chaotic sequences for all the discussed operators, but only the results for the PBX_SHUFFLE operator (best operator) are reported in Table 11. For the crossover operators, the crossover points are selected using a chaotic number generator. For example, for the position-based crossover operator, each position (gene) in parent A is selected using a chaotic number generator with a probability of 0.5.

In Table 11, the problem instances are the same as those described in the dataset given in Table 1. Because the PBX_SHUFFLE operator exhibits a better performance than the other operators, as discussed in Section 8.2.3, we employed this operator in the experiments presented here. The value of the global best (g_{best}) solution is obtained using the best solution value obtained by executing our GA with the PBX_SHUFFLE operator, and the GA terminates after 320 000 fitness evaluations. The values of *avgDev* and σ are calculated accordingly, as defined in Eqs. (15) and 16, respectively. Ten independent runs were conducted for each experiment, and Avg represents the average over the 10 runs. Each algorithm terminated after 10 000 fitness evaluations. The parameters for all algorithms were optimized empirically, and the values used in our experiments are presented in Table 10. The one-tail t-test was adopted to test the significance of the results obtained by CGA_PBX_SHUFFLE and those of the three compared algorithms (PSO, DE, GA_PBX_SHUFFLE) at a confidence interval of 95% and 99%. Here, ‡ indicates that the compared algorithm is significantly outperformed by CGA_PBX_SHUFFLE according to the one-tail t-test at the 99% confidence interval. Furthermore, †, - and ≈ indicate that CGA_PBX_SHUFFLE performs better, worse, and equally in the comparison according to the one-tail t-test at the 95% confidence interval, respectively.

We analyze the results depicted in Table 11 to compare the performances of the different search algorithms. The results show that GA with the PBX_SHUFFLE operator (GA_PBX_SHUFFLE) outperforms PSO and DE on all the problem instances, as it provides lower values of *avgDev*. As the problem size increases, the difference in the quality of solutions of GA_PBX_SHUFFLE compared with the other algorithms also increases. For instance, the *avgDev* of GA_PBX_SHUFFLE for problem #1 is 0.6, whereas it is 0.7 for PSO and 0.16 for DE. For large problems, we can consider problem #9 as an example. Here, the *avgDev* of GA_PBX_SHUFFLE is 5.75, whereas it is 14.4 for PSO and 14.31 for DE. The superiority of the GA can be observed more clearly for problems where the total number of agents is greater than the required number of agents (problems #10 and #11). For example, in problem #10 the average percentage deviations for PSO and DE are 22.4 and 19.44, respectively, while this decreases to 7.78 for GA_PBX_SHUFFLE (an almost 60% improvement in the quality of the solution). We do not observe any significant differences in the quality of solutions for PSO and DE. It can be concluded from these results that PSO and DE fail to maintain a good balance between exploration and exploitation for the problem.

Table 11
Comparison of chaotic GA (using PBX_SHUFFLE) with the GA (using PBX_SHUFFLE), PSO, and DE algorithms. Iterations = 10 000, population size = 8, and *avgDev* and σ are the mean and standard deviation (for 10 replications) of the dev_i (relative deviation from the global best solution), respectively.

Instance		Chaotic GA with PBX_Shuffle (CGA_PBX_SHUFFLE)			GA with PBX_Shuffle (GA_PBX_SHUFFLE)			Particle swarm optimization (PSO)			Differential evolution (DE)		
Prob #	Global best (g_{best})	Best	Avg	*avgDev* (σ) (%)	Best	Avg	*avgDev* (σ) (%)	Best	Avg	*avgDev* (σ) (%)	Best	Avg	*avgDev* (σ) (%)
1	535	535	535	**0.04 (0.06)**	535	535	0.60 (0.60)[a]	535	531	0.70 (0.72)[a]	535	534	0.16 (0.13)[a]
2	1173	1173	1172	**0.07 (0.08)**	1173	1172	**0.07 (0.08)**[d]	1169	1151	1.86 (0.87)[a]	1170	1161	1.04 (0.63)[a]
3	1013	1013	1012	**0.07 (0.13)**	1013	1010	0.27 (0.30)[b]	982	966	4.64 (1.33)[a]	994	984	2.90 (0.59)[a]
4	1438	1433	1426	0.82 (0.27)	1436	1430	**0.55 (0.34)**[c]	1384	1360	5.38 (0.89)[a]	1401	1385	3.66 (0.83)[a]
5	1484	1481	1477	0.52 (0.20)	1483	1478	**0.42 (0.28)**[d]	1426	1402	5.51 (1.12)[a]	1451	1427	3.88 (0.71)[a]
6	3576	3548	3535	**1.15 (0.29)**	3549	3525	1.43 (0.45)[d]	3326	3300	7.73 (0.53)[a]	3334	3284	8.17 (0.64)[a]
7	4711	4654	4633	**1.66 (0.24)**	4641	4623	1.87 (0.23)[b]	4303	4237	10.07 (1.13)[a]	4364	4232	10.17 (0.95)[a]
8	5512	5422	5387	**2.28 (0.28)**	5392	5376	2.48 (0.24)[d]	4985	4913	10.88 (1.24)[a]	4913	4878	11.50 (0.28)[a]
9	22632	21443	21391	**5.48 (0.21)**	21378	21331	5.75 (0.15)[b]	19499	19374	14.40 (0.50)[a]	19462	19395	14.31 (0.13)[a]
10	24732	23077	22878	**7.50 (0.35)**	22958	22809	7.78 (0.31)[b]	19563	19182	22.44 (0.53)[a]	20451	19924	19.44 (1.24)[a]
11	25738	23443	23345	**9.30 (0.16)**	23419	23279	9.55 (0.41)[b]	19588	19385	24.68 (0.59)[a]	20451	19924	19.44 (1.24)[a]
12	96202	89874	89707	**6.75 (0.13)**	89572	89330	7.14 (0.13)[a]	85532	85135	11.50 (0.34)[a]	85363	85273	11.36 (0.06)[a]
13	21006	19879	19796	**5.76 (0.31)**	19820	19737	6.04 (0.25)[b]	18146	17939	14.60 (0.53)[a]	18027	17982	14.40 (0.11)[a]

[a] CGA_PBX_SHUFFLE shows significantly better performance in the comparison (with alpha = 0.01).
[b] CGA_PBX_SHUFFLE shows significantly better performance in the comparison (with alpha = 0.05).
[c] CGA_PBX_SHUFFLE shows significantly worse performance in the comparison (with alpha = 0.05).
[d] CGA_PBX_SHUFFLE shows equal performance in the comparison.

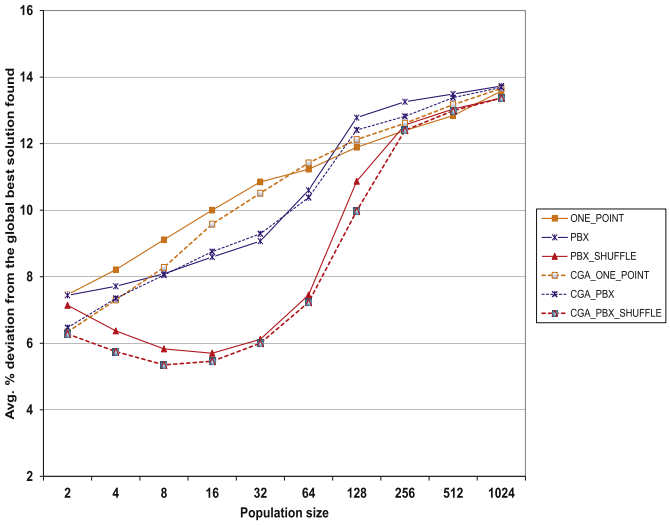

Fig. 30. Comparison of the GA with chaotic GA for different crossover operators (results on problem #9 with the number of fitness evaluations = 10 000).

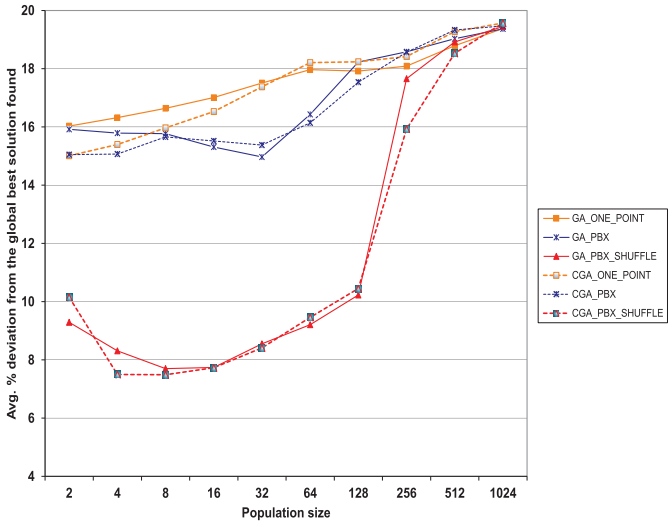

Fig. 31. Comparison of the GA with chaotic GA for different crossover operators (results on problem #10 with the number of fitness evaluations = 10 000).

Another observation from results of GA_PBX_SHUFFLE is that for smaller problems the *avgDev* is less than 1%, but for larger problems the maximum average deviation increases to almost 10%. The reason for this difference is the effect of the number of fitness evaluations on different problem sizes. Figs. 21–23 illustrate that the quality of solutions improves as the number of fitness evaluations increases. For the experiments in this section, 10 000 fitness evaluations were conducted for each algorithm. However, if the number of fitness evaluations is increased from 10 000 to 80 000, the value of *avgDev* drops from almost 10% to 2%, as shown in Figs. 21–23. The number of fitness evaluations has the same effect on PSO and DE, i.e., if the number of fitness evaluations increases the results are improved. However, the GA still outperforms these algorithms, even for a large number of fitness evaluations.

The results of the GA with the PBX_SHUFFLE operator can be further enhanced using chaotic sequences. The best value of the *avgDev* among the four algorithms for each problem is highlighted in bold in Table 11. We can observe that for most of the problems, chaotic sequences help to improve the quality of solutions for the GA. In order to observe the behavior of chaotic sequences on different GA crossover operators, we report the results for the three crossover operators ONE_ POINT (the worst operator), PBX, and PBX_SHUFFLE (the best operator), as shown in Figs. 30 and 31. In general, on small initial population sizes chaotic GA with the PBX_SHUFFLE operator outperforms simple GA with the PBX_SHUFFLE operator for different crossover operators. For example, in Fig. 31 the average percentage deviation for GA with the one point crossover (GA_ONE_POINT) is 8.21, while this decreases to 7.3 for chaotic GA with the one-point crossover (CGA_ONE_POINT). Similarly, the average percentage deviation for the GA with PBX_SHUFFLE (GA_PBX_SHUFFLE) is 6.83, while this decreases to 5.75 for chaotic GA with PBX_SHUFFLE (CGA_PBX_SHUFFLE).

The t-test results in Table 11 show that CGA_PBX_SHUFFLE performs significantly better than PSO and DE at a 99% confidence interval for all problems. CGA_PBX_SHUFFLE is also significantly better than GA_PBX_SHUFFLE at the 95% confidence interval for all large-scale problems (9 to 13). On small problems (1 to 8), the performance of CGA_PBX_SHUFFLE is equivalent to, or sometimes better than, that of GA_PBX_ SHUFFLE.

9.2. Approximation algorithm

To the best of our knowledge, there are no existing approximation algorithms for this particular kind of assignment problem. One possible approximation scheme for assigning tasks to teams of collaborating agents is to employ the Hungarian algorithm [27]. The Hungarian algorithm finds the optimal solution for the case in which agents do not collaborate in teams. Our approximation scheme works as follows:

1. The Hungarian algorithm is employed to find the optimal solution for the case in which agents do not collaborate in teams.
2. New capabilities of agents for the teams formed in step 1 are calculated using Eq. (11).
3. The solution, which represents the weighted sum of these new capabilities, is calculated using Eq. (13).

A loose upper bound for our problem can be computed as follows. At most, an agent's capability can be increased to 50% of its original capability value according to the collaborative model presented in Eq. (11). In the worst case, the proposed approximation scheme will form teams in such a way that none of the agents' capabilities are increased. This means that the proposed approximation scheme will find at least 66.6% of the upper bound. For example, suppose the original capability of an agent is x, and this can be increased to a maximum value of y, where $y = x + (\frac{50}{100}) \times x$. Then, x is 66.6% of y. Therefore, the approximation ratio of the suggested approximation algorithm is two-thirds of the upper bound of the optimal solution.

The aforementioned upper bound is a loose upper bound for our experiments. However, we have computed a more realistic upper bound for the optimal solution of our problem. The previous upper bound assumes that each agent's capability can be increased by 50% of its original value. In reality, the maximum increase in an original capability for most agents will be less than 50% of the original capability, as it depends on the value of the original capability of the agent.

For example, the capabilities of agents used in the above experiments range from 0 to 4, and will increase as shown in Table 12. It is clear from this table that an agent with a capability of 1 or 3 can receive a maximum benefit of 0.75, and an agent with a capability of 2 can receive a maximum benefit of 1. The capabilities of agents are influenced by the maximum capability of that type ($c_k^{\max}$) in the team (g_{s_j}), and the new capabilities (c'_{ik}) are calculated

Table 12
Benefits gained in capabilities of agents according to Eq. (11).

Capability	Max capability in team	Benefit	Increased capability
1	1	0	1
1	2	0.5	1.5
1	3	0.67	1.67
1	4	0.75	1.75
2	1	0	2
2	2	0	2
2	3	0.67	2.67
2	4	1	3
3	1	0	3
3	2	0	3
3	3	0	3
3	4	0.75	3.75
4	1	0	4
4	2	0	4
4	3	0	4
4	4	0	4

Table 13
Comparison of results of the approximation algorithm and upper bound on the optimal solution.

Problem #	GA	Approximation alg.	Upper bound	% of Upper bound
1	535	518	640	81
2	1172	1145	1304	81
3	1013	992	1215	82
4	1431	1380	1667	83
5	1480	1454	1778	82
6	3557	3451	3783	91
7	4686	4545	5101	89
8	5481	5285	6068	87
9	22277	21877	24915	88
10	24258	23973	26501	90

using Eq. (11). Eq. (11) implies that an agent's capability receives the highest benefit if there is another agent with the maximum capability of that type in the team. The upper bound is computed as follows:

1. Increase each agent's capability to its maximum possible value according to Table 12.
2. Run the Hungarian algorithm on the new increased capabilities.
3. The solution of the Hungarian algorithm will give an upper bound for our problem.

Table 13 presents the results of this approximation scheme. We performed the experiments for this approximation scheme using the dataset given in Table 1. The results given in Table 13 show that the proposed approximation scheme performs reasonably well compared with upper bound on the optimal solution. The last column of the table shows the weighted sum capability of the agents achieved by the approximation scheme as a percentage of the upper bound. Although the approximation ratio computed above constitutes two-thirds of the upper bound, in general the proposed approximation algorithm performs better than this approximation ratio. The values in the last column for different problem sizes range from 81% to 91%. Moreover, our proposed GA is also compared with the approximation algorithm in Table 13, and it can be seen to outperform the approximation algorithm.

10. Summary and conclusions

This paper focuses on a specific class of assignment problems in which agents work collaboratively as teams. It is assumed that each agent has a set of capabilities, and each task has a set of requirements, which are specified as weights for capabilities. The objective is to assign the agents to the teams such that the gain is maximized. We present a mathematical formulation of the problem, and suggest the use of GAs to solve the model. The choice of GAs is motivated by the fact that there are no known methods that can solve this class of APs in polynomial time, and GAs have been widely adopted for a variety of combinatorial optimization problems. We demonstrate that for large instances of the problem, standard crossover operators are not able to efficiently solve the problem.

We suggest modifications to the standard crossover operators by adding shuffled lists to them, and also introduce two new crossover operators: team-based and team-based shuffled list operators. Experiments on synthetically generated data are conducted, which demonstrate that the modified and proposed crossover operators perform significantly better.

Among all the studied operators, the position-based shuffled list crossover operator exhibits the strongest performance. However, this operator is sensitive to a large initial population size, which demands that the size of the population is chosen carefully.

Experiments are performed to fine-tune the parameters of the position-based shuffled list crossover for the best performance. Two parameters are tested: (i) the probability of selecting a gene from one of the parents to be transferred to the offspring, and (ii) the mutation probability. In the first case, the best performance is obtained when the probability of selecting a gene from one of the parents is 70%. In the second case, the highest performance is obtained for an 80% mutation probability. The performance of the GA can be enhanced further by employing chaotic sequences. Moreover, the GA is also compared with the PSO and DE algorithms, and the results demonstrate the superiority of GA over these search algorithms.

Acknowledgments

We would like to thank Prof. Rassul Ayani for supervising this research and Prof. Christian Schulte at the KTH Royal Institute of Technology for his valuable comments and suggestions.

References

[1] F.S. Al-Anzi, K. Al-Zame, A. Allahverdi, The weighted multi-skill resources project scheduling, J. Softw. Eng. Appl. 3 (12) (2010) 1125–1130.
[2] E. Alba, F. Chicano, Software project management with GAs, Inf. Sci. 177 (2007) 2380–2401.
[3] B.M. Baker, M.A. Ayechew, A genetic algorithm for the vehicle routing problem, Comput. Oper. Res. 30 (2003) 787–800.
[4] N. Buchbinder, K. Jain, J.S. Naor, Online primal-dual algorithms for maximizing ad-auctions revenue, in: Proceedings of the 15th Annual European Conference on Algorithms, in: ESA'07, Springer-Verlag, 2007, pp. 253–264.
[5] R. Caponetto, L. Fortuna, S. Fazzino, M.G. Xibilia, Chaotic sequences to improve the performance of evolutionary algorithms, IEEE Trans. Evol. Comput. 7 (3) (2003) 289–304.
[6] P. Cheng, X. Lian, L. Chen, J. Han, J. Zhao, Task assignment on multi--skill oriented spatial crowdsourcing, IEEE Trans. Knowl. Data Eng. 28 (2016) 2201–2215.
[7] F. Chicano, F. Luna, A.J. Nebro, E. Alba, Using multi-objective metaheuristics to solve the software project scheduling problem, in: Proceedings of the GECCO, ACM, 2011.
[8] P.C. Chu, J.E. Beasley, A genetic algorithm for the generalised assignment problem, Comput. Oper. Res. 24 (1997) 17–23.
[9] L. Davis, Applying adaptive algorithms to epistatic domains, in: Proceedings of the 9th International Joint Conference on Artificial Intelligence, 1, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1985, pp. 162–164.
[10] N. Devanur, T.P. Hayes, The adwords problem: Online keyword matching with budgeted bidders under random permutations, in: Proceedings of the 10th ACM Conference on Electronic Commerce, Association for Computing Machinery, Inc., 2009.
[11] R.C. Eberhart, J. Kennedy, A new optimizer using particle swarm theory, in: Proceedings of the Sixth International Symposium on Micro Machine and Human Science, 1995, pp. 39–43.
[12] O. Etiler, B. Toklu, M. Atak, J. Wilson, A genetic algorithm for flow shop scheduling problems, J. Oper. Res. Soc. 55 (8) (2004) 830–835.
[13] J. Feldman, M. Henzinger, N. Korula, V.S. Mirrokni, C. Stein, Online Stochastic Packing Applied to Display Ad Allocation, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 182–194.
[14] M.L. Fisher, R. Jaikumar, L.N.V. Wassenhove, A multiplier adjustment method for the generalized assignment problem, Manag. Sci. 32 (9) (1986) 1095–1103.
[15] A. Frank, On Kuhn's Hungarian method–a tribute from Hungary, Naval Res. Logist. 52 (2005) 2–5.

[16] M. Gen, R. Cheng, Genetic Algorithms and Engineering Optimization, John Wiley Sons Inc., USA, 2000.
[17] D.E. Goldberg, Genetic Algorithms in Search, Optimization and Machine Learning, Addison Wesley, Massachusetts, 1989.
[18] D.E. Goldberg, R. Lingle, Alleles, loci, and the travelling salesman problem, in: Proceedings of the First International Conference on Genetic Algorithms and their Applications, Lawrence Erlbaum, Hillsdale, 1985, pp. 154–159.
[19] D.E. Goldberg, robert, Alleles, loci, and the traveling salesman problem, in: J.J. Grefenstette (Ed.), Proceedings of the First International Conference on Genetic Algorithms and Their Applications, Lawrence Erlbaum Associates, Publishers, 1985.
[20] J.F. Gonçalves, J.J.M. Mendes, M.G.C. Resende, A genetic algorithm for the resource constrained multi-project scheduling problem, Eur. J. Oper. Res. 189 (3) (2008) 1171–1190.
[21] C.-J. Ho, J.W. Vaughan, Online task assignment in crowdsourcing markets, in: Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, AAAI Press, 2012, pp. 45–51.
[22] J.H. Holland, Adaptation in Natural and Artificial Systems, MIT Press, Cambridge, MA, USA, 1992.
[23] F. Kamrani, R. Ayani, F. Moradi, A model for estimating the performance of a team of agents, in: Proceedings IEEE International Conference on Systems, Man, and Cybernetics (SMC), 2011, pp. 2393–2400.
[24] H. Kazemipoor, R. Tavakkoli-Moghaddam, P. Shahnazari-Shahrezaei, A. Azaron, A differential evolution algorithm to solve multi-skilled project portfolio scheduling problems, Int. J. Adv. Manuf. Technol. 64 (5) (2013) 1099–1111.
[25] T. Kellegöz, B. Toklu, J.M. Wilson, Comparing efficiencies of genetic crossover operators for one machine total weighted tardiness problem, Appl. Math. Comput. 199 (2008) 590–598.
[26] J. Kennedy, R.C. Eberhart, Particle swarm optimization, in: Proceedings of the IEEE International Conference on Neural Networks, 1995, pp. 1942–1948.
[27] H.W. Kuhn, The Hungarian method for the assignment problem, Naval Res. Logist. Q. 2 (1955) 83–97.
[28] M. Kuroda, K. Yamamori, M. Munetomo, M. Yasunaga, I. Yoshihara, A proposal for zoning crossover of hybrid genetic algorithms for large-scale traveling salesman problems, in: Proceedings of the IEEE Congress on Evolutionary Computation (CEC), 2010, pp. 1–6.
[29] F. Luna, D.L. Gonzlez-lvarez, F. Chicano, M.A. Vega-Rodrguez, The software project scheduling problem: A scalability analysis of multi-objective metaheuristics, Appl. Soft Comput. 15 (2014) 136–148.
[30] R.M. May, Simple mathematical models with very complicated dynamics, Nature 261 (1976) 459–467.
[31] J.J.M. Mendes, J.F. Gonçalves, M.G.C. Resende, A random key based genetic algorithm for the resource constrained project scheduling problem, Comput. Oper. Res. 36 (1) (2009) 92–109.
[32] M. Mitchell, Introduction to Genetic Algorithms, MIT Press, Cambridge, Massachusetts, 1999.
[33] C. Montoya, O. Bellenguez-Morineau, E. Pinson, D. Rivreau, Integrated Column Generation and Lagrangian Relaxation Approach for the Multi-Skill Project Scheduling Problem, Springer International Publishing, pp. 565–586.
[34] T. Murata, H. Ishibuchi, Performance evaluation of genetic algorithms for flowshop scheduling problems, in: Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence, vol.2, 1994, pp. 812–817.
[35] P.B. Myszkowski, M.E. Skowroński, Ł.P. Olech, K. Oślizło, Hybrid ant colony optimization in solving multi-skill resource-constrained project scheduling problem, Soft Comput. 19 (12) (2015) 3599–3619.
[36] I.M. Oliver, D.J. Smith, J.R.C. Holland, 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, L. Erlbaum Associates Inc., Hillsdale, NJ, USA, 1987, pp. 224–230.
[37] D.W. Pentico, Assignment problems: a golden anniversary survey, Eur. J. Oper. Res. 176 (2) (2007) 774–793.
[38] R. Storn, K. Price, Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces, J. Global Optim. 11 (4) (1997) 341–359.
[39] G. Syswerda, Schedule optimization using genetic algorithms, in: L. Davis (Ed.), Handbook of Genetic Algorithms, Van Nostrand Reinhold, New York, 1990.
[40] L. Wang, X.L. Zheng, A knowledge-guided multi-objective fruit fly optimization algorithmfor the multi-skill resource constrained project scheduling problem, Swarm Evol. Comput. 38 (2017) 54–63.
[41] L.D. Whitley, T. Starkweather, D. Fuquay, Scheduling problems and traveling salesmen: the genetic edge recombination operator, in: Proceedings of the 3rd International Conference on Genetic Algorithms, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1989, pp. 133–140.
[42] D.H. Wolpert, W.G. Macready, No free lunch theorems for optimization and search, IEEE Trans. Evol. Comput. 1 (1) (1997) 67–82.
[43] L.J. Yang, T.L. Chen, Application of chaos in genetic algorithms, Commun. Theor. Phys. 38 (2002) 168–172.
[44] I. Younas, F. Kamrani, C. Schulte, R. Ayani, Optimization of task assignment to collaborating agents, in: Proceedings of the IEEE Symposium on Computational Intelligence in Scheduling, Paris, France, 2011, pp. 17–24.
[45] H.-y. Zheng, L. Wang, X.-l. Zheng, Teaching–learning-based optimization algorithm for multi-skill resource constrained project scheduling problem, Soft Comput. 21 (6) (2017) 1537–1548.

Irfan Younas received the Ph.D. degree from KTH Royal Institute of Technology, Stockholm, Sweden. Currently, he is an Assistant Professor of Computer Science at National University of Computer and Emerging Sciences. His current research interests include Evolutionary Algorithms, Bioinformatics, Combinatorial Optimization, Search Heuristics, Multi-objective Optimization, Many-Objective Algorithms, and Machine Learning. Previously, He has worked on different military and defense related projects carried out at the Swedish Defence Research Agency (FOI). Moreover, he has around 8 years of extensive industry experience in Design, Development, Implementation and Management of systems in wide variety of areas.

Farzad Kamrani is a Scientist at Swedish Defence Research Agency (FOI) in the Department of Decision Support Systems. He has worked for many years on modelling and simulation and his research interest is in the field of evolutionary algorithms, machine learning and artificial intelligence. He has a Masters degree in Computer Science from the University of Gothenburg and a Ph.D. in Electronics and Computer Systems from KTH Royal Institute of Technology.

Maryam Bashir is an Assistant Professor of Computer Science at National University of Computer and Emerging Sciences. She earned her doctorate in Computer Science from Northeastern University in Boston USA. She is recipient of prestigious Fulbright Scholarship for Ph.D. studies in USA. Her research interests include Information Retrieval, Machine Leaning, and Data Sciences.

Johan Schubert is Associate Professor of information and communication technology at the Royal Institute of Technology and Deputy Research Director of information fusion at the Swedish Defence Research Agency. He received the M.Sc. degree in Engineering Physics in 1986 and the Ph.D. degree in Computer Science in 1994, both from the Royal Institute of Technology, Stockholm. He has conducted research in artificial intelligence, decision support, and information fusion for 31 years and published 17 journal articles, 5 book chapters, 52 conference papers, 40 technical reports and 13 popular science articles. He was the technical program chair of the 7th International Conference on Information Fusion, editor of the conference proceedings and guest editor of a double special issue of the journal Information Fusion. He is a board member of the Belief Functions and Applications Society and member of the editorial board of the Information Fusion journal. He received the IJAR Best Paper Award at the 4th International Conference on Belief Functions in 2016. He was the Swedish representative to the NATO S&T Modelling and Simulation Groups Data Farming in Support of NATO 2010–2013 and Developing Actionable Data Farming Decision Support for NATO 2013–2017. He received the NATO Scientific Achievement Award in 2014. His research interests include theoretical and applied aspects of artificial intelligence, soft computing, neural networks, evolutionary algorithms, big data analytics, management of uncertainty, belief functions, modelling and simulation, data farming, military applications of high-level information fusion, decision support, and artificial intelligence for situation and threat assessment and its use in decision support systems.