Implementation of Simple Multiobjective Memetic Algorithms and Its Application to Knapsack Problems

Hisao Ishibuchi and Shiori Kaige

Dept. of Industrial Engineering, Osaka Prefecture University
1-1 Gakuen-cho, Sakai, Osaka 599-8531, Japan
{hisaoi, shiori}@ie.osakafu-u.ac.jp

Abstract. The aim of this paper is to propose a simple but powerful multiobjective hybrid genetic algorithm and to examine its search ability through computational experiments on commonly used test problems in the literature. We first propose a new multiobjective hybrid genetic algorithm, which is designed by combining local search with an EMO (evolutionary multiobjective optimization) algorithm. In the design of our algorithm, we try to make its algorithmic complexity as simple as possible so that it can be easily understood, easily implemented and easily executed within short CPU time. At the same time, we try to maximize its search ability. Our algorithm makes use of advantages of both EMO and local search for achieving high search ability without increasing its algorithmic complexity. For example, each solution is evaluated based on Pareto ranking and the concept of crowding as in many EMO algorithms. On the other hand, a weighted scalar fitness function is used for efficiently executing local search. A kind of elitism is also implemented using Pareto ranking in the process of generation update. Through computational experiments on multiobjective 0/1 knapsack problems, we examine the search ability of four variants of our algorithm with various parameter specifications. Those variants are different from each other in the implementation of parent selection and local search. While some variants use the weighted scalar fitness function only for local search, others use it for both local search and parent selection. One variant uses Pareto ranking instead of the weighted scalar fitness function in local search. In addition to the comparison among those four variants, our algorithm is also compared with well-known EMO algorithms (i.e., SPEA of Zitzler & Thiele and NSGA-II of Deb et al.) and memetic EMO algorithms (i.e., M-PAES of Knowles & Corne and MOGLS of Jaszkiewicz). We also examine the effect of the balance between genetic search and local search on the search ability of our algorithm using two parameters: a local search application probability and a local search stopping condition. Moreover we demonstrate the usefulness of a weighted scalar fitness function-based greedy repair procedure in the application of memetic EMO algorithms to multiobjective 0/1 knapsack problems. Our experimental results by various EMO algorithms show that there exists a clear tradeoff between CPU time and the quality of solution sets obtained by each algorithm. Since our algorithm is very simple, it can be efficiently executed. As a result, our algorithm outperforms many EMO and memetic EMO algorithms in terms of CPU time for large test problems while it does not always outperform them in terms of the quality of obtained solution sets.

Keywords: evolutionary multiobjective optimization, memetic algorithms, genetic algorithms, local search.

1 Introduction

Since Schaffer (1985), evolutionary algorithms have been applied to various multiobjective optimization problems for finding their Pareto-optimal or near Pareto-optimal solutions (e.g., see Deb (2001) and Coello et al. (2002)). Those algorithms are often referred to as EMO (evolutionary multiobjective optimization) algorithms. Recent EMO algorithms share some common ideas such as elitism, fitness sharing and Pareto ranking. For implementing the concept of elitism, some EMO algorithms have a secondary population that is stored separately from the main population. While the use of the secondary population significantly improves the convergence speed to the Pareto-front of EMO algorithms (e.g., see computational experiments in Zitzler & Thiele (1999) and Zitzler et al. (2000)), usually it also increases algorithmic complexity and CPU time. On the other hand, the concept of fitness sharing (or crowding) is implemented in almost all EMO algorithms for increasing the diversity of solutions. Pareto ranking is also used in almost all EMO algorithms for evaluating each solution with respect to multiple objectives.

In some studies, local search was combined with EMO algorithms for further improving the convergence speed to the Pareto-front. Hybridization of EMO algorithms with local search is often referred to as MOGLS (multiobjective genetic local search) algorithms. Such a hybrid algorithm is also called a memetic EMO al-

gorithm. Memetic EMO algorithms can be roughly classified into two categories according to their solution evaluation mechanisms in local search: One uses a weighted scalar fitness function, and the other uses Pareto ranking. A MOGLS algorithm based on a weighted scalar fitness function with random weight values was first proposed by Ishibuchi & Murata (1996, 1998), and improved by Jaszkiewicz (2002a) and Ishibuchi et al. (2003). On the other hand, Knowles & Corne (2000a) proposed a memetic EMO algorithm called M-PAES (memetic Pareto archived evolution strategy) where each solution was evaluated based on Pareto ranking. Some Pareto ranking-based acceptance rules of neighboring solutions in local search were examined in Ishibuchi et al. (2003) and Murata et al. (2003). The MOGLS of Jaszkiewicz (2002a) and the M-PAES of Knowles & Corne (2000a), which are well-known memetic EMO algorithms with high search ability, have been compared with each other in many comparative studies (e.g., Knowles & Corne (2000b), Jaszkiewicz (2001, 2002b), Ishibuchi & Kaige (2003)). Experimental results in those studies show that the M-PAES has higher convergence speed to the Pareto-front while the MOGLS can find much more solutions with larger diversity. That is, they have their own advantages and disadvantages.

In this paper, we propose a new MOGLS algorithm (i.e., a new memetic EMO algorithm) that shares some advantages with existing EMO and memetic EMO algorithms. While we try to maximize the search ability of our MOGLS algorithm, we also try to minimize its algorithmic complexity so that it can be easily understood, easily implemented and easily executed within short CPU time. In order to emphasize its simplicity, we refer to our MOGLS algorithm as the simple MOGLS (i.e., S-MOGLS) algorithm in this paper. Our S-MOGLS algorithm can be viewed as a hybrid algorithm of the NSGA-II (elitist non-dominated sorting genetic algorithm) of Deb et al. (2002) and local search in our former MOGLS algorithm (Ishibuchi & Murata (1996, 1998) and Ishibuchi et al. (2003)). More specifically, we use the generation update mechanism of the NSGA-II where solutions in the next population are chosen from the current population and the offspring population. We adopt this generation update mechanism because it does not need a secondary population. The use of the secondary population often increases algorithmic complexity and CPU time. Each solution is evaluated by Pareto ranking and the distances from its neighboring solutions. We adopt this solution evaluation mechanism of the NSGA-II because it realizes the concept of crowding in a simple but effective manner. A local search procedure is probabilistically applied to each offspring generated by genetic operations. As in our former MOGLS algorithm, a weighted scalar fitness function with random weights is used for choosing a pair of parent solutions. The same fitness function is used in local search for their offspring. We use the weighted scalar fitness function in parent selection and local search because it can be efficiently calculated, i.e., because its calculation is not time-consuming if compared with Pareto ranking-based schemes. We also examine several variants of our S-MOGLS algorithm. Some variants do not use the weighted scalar fitness function in parent selection (i.e., it is used only in local search). Another variant uses Pareto ranking instead of the weighted scalar fitness function in local search.

This paper is organized as follows. In Section 2, we briefly describe multiobjective optimization problems and the concept of Pareto optimality. Then we explain our S-MOGLS algorithm and its four variants in Section 3. Those four variants are compared with each other through computational experiments on multiobjective 0/1 knapsack problems of Zitzler & Thiele (1999) in Section 4 where they are also compared with the SPEA (strength Pareto evolutionary algorithm) of Zitzler & Thiele (1999), the NSGA-II of Deb et al. (2002), the M-PAES of Knowles & Corne (2000a), and the MOGLS of Jaszkiewicz (2002a). Experimental results show that our S-MOGLS algorithm can find better solutions using less CPU time than the SPEA and similar solutions using less CPU time than the NSGA-II. It is also shown that our S-MOGLS algorithm needs much less CPU time than the M-PAES and almost the same CPU time as the MOGLS for finding solutions of almost the same quality. Moreover we demonstrate that the use of a heuristic solution repair mechanism based on the weighted scalar fitness function significantly improves the search ability of the MOGLS and S-MOGLS algorithms. Finally Section 5 concludes this paper.

2 Multiobjective Optimization

Let us consider the following *k*-objective maximization problem:

$$\text{Maximize } \mathbf{f}(\mathbf{x}) = (f_1(\mathbf{x}),\ f_2(\mathbf{x}),\ ...,\ f_k(\mathbf{x})) \text{ subject to } \mathbf{x} \in \mathbf{X}, \tag{1}$$

where $\mathbf{f}(\mathbf{x})$ is the objective vector, $f_i(\mathbf{x})$ is the *i*-th objective to be maximized, $\mathbf{x}$ is the decision vector, and $\mathbf{X}$ is the feasible region in the decision space.

When the following two conditions are satisfied, a feasible solution $\mathbf{x} \in \mathbf{X}$ is said to be dominated by an-

other feasible solution $\mathbf{y} \in \mathbf{X}$ (i.e., **y** dominates **x**: **y** is better than **x**):

$$\forall i,\ f_i(\mathbf{x}) \le f_i(\mathbf{y}) \ \text{ and } \ \exists j,\ f_j(\mathbf{x}) < f_j(\mathbf{y}). \tag{2}$$

If there is no feasible solution **y** that dominates **x**, **x** is said to be a Pareto-optimal solution of the multiobjective optimization problem in (1). Our task in this paper is to find Pareto-optimal or near Pareto-optimal solutions as many as possible. The main advantage of EMO and memetic EMO algorithms over other search methods is that multiple solutions can be simultaneously obtained by their single run.

3 Proposed S-MOGLS Algorithm

In this section, we first briefly describe our former MOGLS algorithm (Ishibuchi & Murata (1996, 1998), Ishibuchi et al. (2003)) and the MOGLS algorithm of Jaszkiewicz (2002a). Next we explain the basic framework of our S-MOGLS algorithm. Then we show its variants. Those variants are different from each other in the implementation of parent selection and local search.

3.1 Former MOGLS Algorithms

The following weighted scalar fitness function was used in our former MOGLS algorithm (Ishibuchi & Murata (1996, 1998), Ishibuchi et al. (2003)) and the MOGLS algorithm of Jaszkiewicz (2002a):

$$f(\mathbf{x}, \boldsymbol{\lambda}) = \sum_{i=1}^{k} \lambda_i f_i(\mathbf{x}), \tag{3}$$

where

$$\forall i,\ \lambda_i \ge 0 \ \text{ and } \ \sum_{i=1}^{k} \lambda_i = 1. \tag{4}$$

In our former MOGLS algorithm, the weight vector $\boldsymbol{\lambda} = (\lambda_1, ..., \lambda_k)$ was randomly specified whenever a pair of parent solutions was to be selected based on the weighted scalar fitness function. The roulette wheel selection was used in Ishibuchi & Murata (1996, 1998) while it was outperformed by the binary tournament selection in computational experiments in Ishibuchi et al. (2003). An offspring was generated by crossover and mutation from the selected pair of parents. The same weighted scalar fitness function with the current weight values was used in local search for the generated offspring. In Fig. 1, we show the outline of our former MOGLS algorithm. All the non-dominated solutions among examined ones were stored in the secondary population with no limitation on its size. A pre-specified number of solutions were randomly selected from the secondary population and their copies were added to the main population.

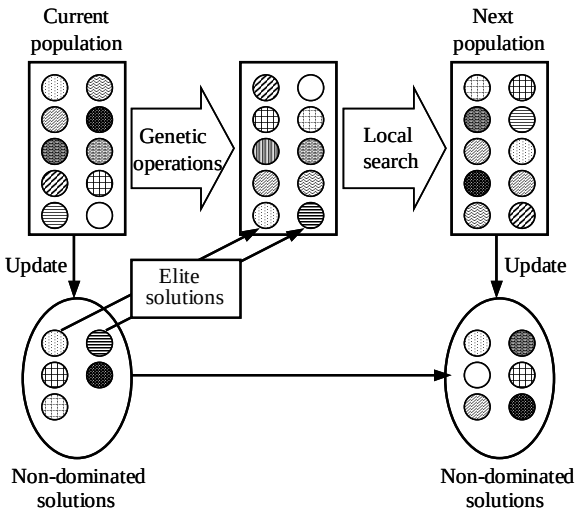


Figure 1. Outline of our former MOGLS algorithm.

In the MOGLS algorithm of Jaszkiewicz (2002a), the weight vector $\boldsymbol{\lambda} = (\lambda_1, ..., \lambda_k)$ was also randomly specified whenever a pair of parent solutions was to be selected based on the weighted scalar fitness function. The best *K* solutions were selected from the current population with respect to the weighted scalar fitness function. Then a pair of parent solutions was randomly chosen from those best *K* solutions in order to generate an offspring by crossover. Local search was applied to the generated offspring using the weighted scalar fitness function with the current weight values. All the non-dominated solutions among the examined ones were stored in the secondary population.

3.2 S-MOGLS Algorithm

As in some other EMO and memetic EMO algorithms (e.g., SPEA, M-PAES, and Jaszkiewicz's MOGLS), our former MOGLS algorithm used the secondary population for storing non-dominated solutions. The use of the secondary population often increases algorithmic complexity and CPU time. Thus we do not use the secondary population in our new MOGLS (i.e., simple MOGLS: S-MOGLS) algorithm. Since some form of elitism is necessary for implementing high performance EMO and memetic EMO algorithms, we use the same generation update mechanism as in the NSGA-II of Deb et al. (2002). That is, solutions in the next population are selected from the current and offspring populations as shown in Fig. 2. More specifically, first the current and offspring populations are merged to form a tentative population. Then a rank is assigned to each solution in the tentative population using Pareto ranking. That is, the first rank is assigned to all the non-dominated solutions in the tentative population. All solutions with the first rank are removed from the tentative population and added to the next population. The second rank is assigned to all the non-dominated solutions in the reduced tentative population. All solutions with the second rank are removed from the reduced tentative population and added to the next population. In this manner, better solutions with respect to multiple objectives are chosen and added to the next population. If the number of the solutions in the next population exceeds the population size, solutions with the worst rank in the next population are sorted using the concept of crowding. Each solution is evaluated by the sum of the distances from adjacent solutions with the same rank. More specifically, two adjacent solutions of each solution are identified with respect to each objective. Then the distance between those adjacent solutions is calculated on each objective and summed up over all the *k* objectives for calculating the measure of crowding. For each extreme solution with the maximum or minimum value of at least one objective among the same rank solutions, an infinite large value is assigned to the crowding measure because one of the two adjacent solutions cannot be identified. Solutions with larger values of the crowding measure are viewed as being better because those solutions are not located in crowded regions in the objective space. Solutions with the worst rank are removed from the next population in the increasing order of the crowding measure until the number of remaining solutions in the next population becomes the population size. For further descriptions of this generation update mechanism, see Deb (2001) and Deb et al. (2002).

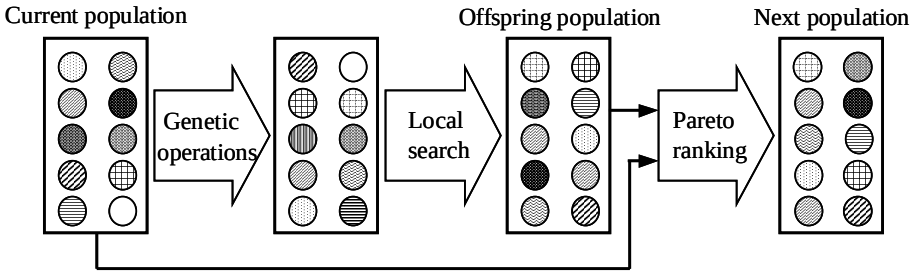


Figure 2. Outline of the proposed S-MOGLS algorithm.

Let N_{pop} be the population size. When a pair of parent solutions is to be selected from the current population with N_{pop} solutions, the weight vector is randomly specified as in our former MOGLS algorithm. We use the standard binary tournament selection in our S-MOGLS algorithm because it was demonstrated in Ishibuchi et al. (2003) that the tournament selection outperformed the roulette wheel selection in computational experiments on multiobjective flowshop scheduling problems using our former MOGLS algorithm. An offspring is generated from the selected pair of parents using crossover and mutation. A local search procedure is applied to the generated offspring with a pre-specified local search application probability P_{LS}. The

weighted scalar fitness function with the current weight values is used in the local search procedure for the generated offspring. The genetic operations (i.e., selection, crossover and mutation) and the local search procedure are iterated for generating N_{pop} offspring. The next population with N_{pop} strings is constructed by choosing good solutions from the current and offspring populations in the above-mentioned manner, i.e., using the generation update mechanism of the NSGA-II.

3.3 Several Variants

We can implement various variants of our S-MOGLS algorithm. The S-MOGLS algorithm in the previous subsection is referred to as the S-MOGLS Version 1 algorithm, which can be written as follows:

S-MOGLS Version 1 Algorithm:

Step 1 (Initialization): Generate an initial population with N_{pop} solutions.

Step 2 (Genetic operations and local search): Generate an offspring population by iterating the following procedures N_{pop} times:

(1) Randomly specify the weight vector.

(2) Choose a pair of parent solutions from the current population using the binary tournament selection based on the weighted scalar fitness function with the current weight values.

(3) Generate an offspring from the selected parents by crossover and mutation.

(4) Apply a local search procedure based on the weighted scalar fitness function with the current weight values to the generated offspring with the local search application probability P_{LS}.

Step 3 (Generation update): Construct the next population from the current and offspring populations by choosing good solutions from them based on Pareto ranking and the crowding measure.

Step 4 (Termination test): If the pre-specified stopping condition is not satisfied, return to Step 2. Otherwise terminate the execution of the algorithm.

In this algorithm, the weighted scalar fitness function is used not only in local search but also in parent selection. One might think that the use of the weighted scalar fitness function in the parent selection degrades high search ability of the NSGA-II algorithm with which local search is combined for designing our S-MOGLS algorithm. So we examine the following variant where the weighted scalar fitness function is used only in local search.

S-MOGLS Version 2 Algorithm (Steps 1, 3, 4 are the same as the S-MOGLS Version 1 algorithm):

Step 2-1 (Genetic operations): Generate an offspring population with N_{pop} solutions in the same manner as in the NSGA-II where the binary tournament selection based on Pareto ranking and the crowding measure is used for parent selection.

Step 2-2 (Local search): Apply the following procedures to each offspring generated in Step 2-1 with the local search application probability P_{LS}.

(1) Randomly specify the weight vector.

(2) Apply a local search procedure based on the weighted scalar fitness function with the current weight values to the current offspring. If the current offspring (i.e., the initial solution in local search) is updated by the local search procedure, the updated solution (i.e., the final solution in local search) is added to the offspring population generated in Step 2-1.

This algorithm is the same as the NSGA-II except for the local search part in Step 2-2 while the above-mentioned Version 1 algorithm uses a different parent selection mechanism from the NSGA-II. One possible flaw of the Version 2 algorithm is the random specification of the weight vector in local search. That is, the local search direction is totally random in the Version 2 algorithm while it is inherited from the parent selection procedure in the Version 1 algorithm. As an intermediate version between these two algorithms, we also examine the following variant:

S-MOGLS Version 3 Algorithm (Steps 1, 3, 4 are the same as the S-MOGLS Version 1 algorithm):

Step 2 (Genetic operations and local search): Generate an offspring population by iterating the following procedures N_{pop} times:

(1) Using the local search application probability P_{LS}, determine whether local search is applied or not.

(2) If local search is applied, generate an offspring from the current population in the same manner as (1)-(4) in Step 2 of the Version 1 algorithm.

(3) If local search is not applied, generate an offspring from the current population in the same manner as Step 2-1 of the Version 2 algorithm.

Instead of the weighted scalar fitness function used in the local search part of the above-mentioned three variants of the S-MOGLS algorithm, we also examine the use of Pareto ranking in local search where the current solution is updated only when it is dominated by its neighboring solution (see Murata et al. (2003)):

S-MOGLS Version 4 Algorithm (Steps 1, 3, 4 are the same as the S-MOGLS Version 1 algorithm):

Step 2-1 (Genetic operations): Generate an offspring population with N_{pop} solutions in the same manner as in the NSGA-II where the binary tournament selection based on Pareto ranking and the crowding measure is used for parent selection (i.e., this step is the same as Step 2-1 of the Version 2 algorithm).

Step 2-2 (Local search): Apply a Pareto ranking-based local search procedure to each offspring in Step 2-1 with the local search application probability P_{LS}. The current solution is updated in local search only when it is dominated by its neighboring solution. If the current offspring (i.e., the initial solution in local search) is updated by the local search procedure, the updated solution (i.e., the final solution in local search) is added to the offspring population generated in Step 2-1.

This algorithm is the same as the Version 2 algorithm except for the use of Pareto ranking instead of the weighted scalar fitness function.

4 Computational Experiments

Through computational experiments on multiobjective 0/1 knapsack problems, we examine the following issues:

(a) Effect of the balance between genetic search and local search on the performance of our S-MOGLS algorithm.
(b) Comparison among the four variants of our S-MOGLS algorithm.
(c) Comparison of our S-MOGLS algorithm with other EMO and memetic EMO algorithms.
(d) Use of the weighted scalar fitness function in a heuristic greedy repair procedure for unfeasible solutions.

4.1 Conditions of Computational Experiments

In our computational experiments, we use the nine multiobjective 0/1 knapsack problems with two, three or four objectives (i.e., knapsacks) and 250, 500 or 750 items in Zitzler & Thiele (1999). The *k*-objective *n*-item problem is denoted as the *k*-*n* problem (e.g., 2-250 and 3-750 problems). Multiobjective 0/1 knapsack problems with *k* knapsacks (i.e., *k* objectives) and *n* items can be written in a generic form as follows:

$$\text{Maximize } \mathbf{f}(\mathbf{x}) = (f_1(\mathbf{x}),\ f_2(\mathbf{x}),\ ...,\ f_k(\mathbf{x})), \tag{5}$$

$$\text{subject to } \sum_{j=1}^{n} w_{ij} x_j \le c_i, \quad i = 1, 2, ..., k, \tag{6}$$

where

$$f_i(\mathbf{x}) = \sum_{j=1}^{n} p_{ij} x_j, \quad i = 1, 2, ..., k. \tag{7}$$

In this formulation, $\mathbf{x}$ is an *n*-dimensional binary vector (i.e., $\mathbf{x} = (x_1, x_2, ..., x_n) \in \{0, 1\}^n$), p_{ij} is the profit of item *j* according to knapsack *i*, w_{ij} is the weight of item *j* according to knapsack *i*, and c_i is the capacity of knapsack *i*.

Our computational experiments on the multiobjective 0/1 knapsack problems are performed in the same manner as in other comparative studies using the same test problems (e.g., Zitzler & Thiele (1999), Knowles & Corne (2000b), Jaszkiewicz (2001), Ishibuchi & Kaige (2003)). Each solution is handled as a binary string of the length *n* in EMO and memetic EMO algorithms. When new solutions are generated by genetic operations, they are often unfeasible. For deriving feasible solutions from unfeasible ones, repair procedures have been used in the literature. Zitzler & Thiele (1999) used a greedy repair procedure where items were removed in the increasing order of the maximum profit/weight ratio q_j over all knapsacks:

$$q_j = \max\{p_{ij} / w_{ij} \mid i = 1,2,...,k\}, \quad j = 1,2,...,n. \tag{8}$$

This maximum ratio greedy repair procedure has been used in many studies on EMO and memetic EMO algorithms (Zitzler & Thiele (1999), Knowles & Corne (2000b), Jaszkiewicz (2001, 2002b), Ishibuchi & Kaige (2003)). This greedy repair procedure is used in our computational experiments of this paper. We also examine a different greedy repair procedure based on the weighted scalar fitness function.

For comparing the search ability of various EMO and memetic EMO algorithms, we use the generational distance and the $D1_R$ measure (see Deb (2001), Coello (2002), Knowles & Corne (2002) for various performance measures). These measures evaluate the quality of an obtained non-dominated solution set using a reference solution set. The reference solution set is a set of Pareto-optimal or near Pareto-optimal solutions. In our computational experiments, the reference solution set for each test problem is constructed by choosing non-dominated solutions among all solutions obtained by computational experiments in Ishibuchi & Kaige (2003). The generational distance is the average distance from each solution in the obtained solution set to its nearest reference solution. This measure evaluates the convergence speed to the Pareto-front. On the other hand, the $D1_R$ measure is the average distance from each reference solution to its nearest solution in the obtained solution set. This measure evaluates both the convergence speed and the diversity of obtained solutions. We also monitor the CPU time of each algorithm in our computational experiments.

In addition to our S-MOGLS algorithm, we use two EMO algorithms (i.e., SPEA and NSGA-II) and two memetic EMO algorithms (i.e., M-PAES and Jaszkiewicz's MOGLS) in our computational experiments of this paper. In the three memetic EMO algorithms, we use two parameters for terminating local search for each solution as in Knowles & Corne (2000b) and Jaszkiewicz (2001). One is the maximum number of local search moves (i.e., *l_opt*) and the other is the maximum number of consecutive fails of local search moves (i.e., *l_ fails*). In other words, *l_opt* is the upper bound on the total number of examined solutions in local search from an initial solution while *l_ fails* is the upper bound on the number of examined neighbors of the current solution. In all the three memetic EMO algorithms, a neighboring solution is generated by applying the standard bit-flip mutation operation with a probability of 4/*n* to each bit of the current solution where *n* is the number of items. This operation is also used as a mutation operation. The standard one-point crossover is used in all the five algorithms. The crossover probability is specified as 0.8 in the two EMO algorithms and our S-MOGLS algorithm while it is specified as 1.0 in the M-PAES and the MOGLS.

Parameter values in our computational experiments are summarized in Table 1. In this table, *max_evals* is the total number of evaluated solutions, which is used as the stopping condition of each algorithm. Our parameter specifications are almost the same as those in Zitzler & Thiele (1999), Knowles & Corne (2000a, 2000b) and Jaszkiewicz (2001).

Table 1. Parameter values in our computational experiments.

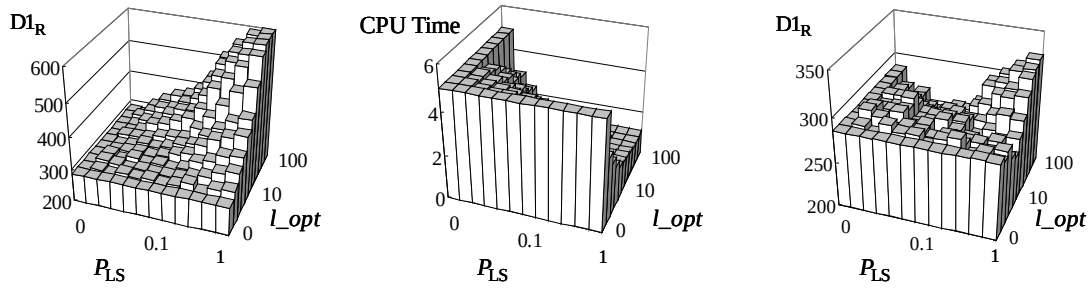
Problem	Population Size					*K*	*l_fails*	*l_opt*	*max_evals*
	SPEA	NSGA-II	M-PAES	MOGLS	S-MOGLS	MOGLS	Memetic EMO		
2-250	120	150	30	3,000	150	20	20	100	75,000
2-500	160	200	40	4,000	200	20	20	100	100,000
2-750	200	250	50	5,000	250	20	5	20	125,000
3-250	160	200	40	4,000	200	20	20	50	100,000
3-500	200	250	50	5,000	250	20	20	50	125,000
3-750	240	300	60	6,000	300	20	5	20	150,000
4-250	200	250	50	5,000	250	20	20	50	125,000
4-500	240	300	60	6,000	300	20	20	50	150,000
4-750	280	350	70	7,000	350	20	5	20	175,000

4.2 Effect of the Balance between Genetic Search and Local Search

For examining the effect of the balance between genetic search and local search on the performance of our S-MOGLS algorithm, we only use *l_opt* as the stopping condition of local search in this subsection. Thus the average number of examined solutions by local search in each generation of our S-MOGLS algorithm can be calculated as $N_{\text{pop}} \cdot P_{\text{LS}} \cdot l\_opt$ while genetic operations generate N_{pop} solutions in each generation. Thus local search examines $P_{\text{LS}} \cdot l\_opt$ times as many solutions as genetic search (i.e., $P_{\text{LS}} \cdot l\_opt$ is the relative

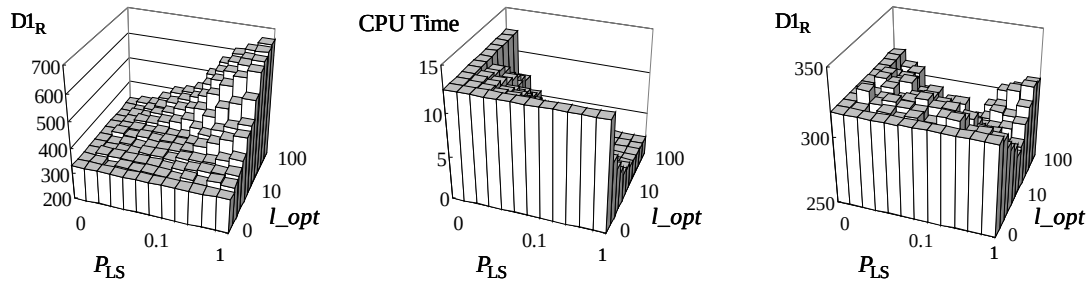
computation load of local search with respect to genetic search). We perform computational experiments using various combinations of P_{LS} and l_opt. More specifically, we examine 12×12 combinations of P_{LS} and l_opt: $P_{LS} = 0$, 0.01, 0.02, 0.04, 0.06, 0.08, 0.1, 0.2, 0.4, 0.6, 0.8, 1 and $l\_opt = 0$, 1, 2, 4, 6, 8, 10, 20, 40, 60, 80, 100. When P_{LS} and/or l_opt are specified as 0, local search is not used. On the other hand, local search examines 100 times as many solutions as genetic search when $P_{LS} = 1$ and $l\_opt = 100$.

We apply the S-MOGLS Version 1 algorithm to the 2-250 test problem. The execution of the algorithm is terminated when 75,000 solutions are examined. Our computational experiment is performed 30 times (i.e., 30 independent runs) for each combination of P_{LS} and l_opt. We also perform the same computational experiment under the same CPU time for all combinations of P_{LS} and l_opt. More specifically, the S-MOGLS Version 1 algorithm is executed for 5.0 seconds on a PC with a Pentium 4 (2.80 GHz) processor. Experimental results are summarized in Fig. 3. Fig. 3 (a) shows the average value of the $D1_R$ measure over 30 runs for each of the 12×12 combinations of P_{LS} and l_opt where the stopping condition is the evaluation of 75,000 solutions. Fig. 3 (b) shows the corresponding average CPU time. On the other hand, Fig. 3 (c) shows the average value of the $D1_R$ measure where the stopping condition is the CPU time of 5.0 seconds. This stopping condition is almost the same as the average CPU time of the S-MOGLS algorithm around the bottom-left corner in Fig. 3 (b). In these figures, the relative computation load of local search (i.e., $P_{LS} \cdot l\_opt$) is small near the bottom and left sides while it assumes the maximum value at the top-right corner. On the other hand, experimental results on the 3-250 test problem are summarized in Fig. 4. The stopping condition is the examination of 100,000 solutions in Fig. 4 (a) while it is the CPU time of 15.0 seconds in Fig. 4 (c).



(a) $D1_R$ measure (75,000 solutions). (b) CPU time (seconds). (c) $D1_R$ measure (5.0 seconds).

Figure 3. Experimental results on the 2-250 problem by the S-MOGLS Version 1 algorithm.



(a) $D1_R$ measure (100,000 solutions). (b) CPU time (seconds). (c) $D1_R$ measure (15.0 seconds).

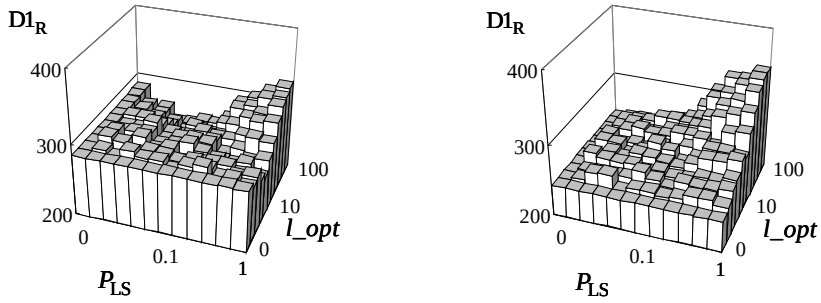
Figure 4. Experimental results on the 3-250 problem by the S-MOGLS Version 1 algorithm.

We can see from Fig. 3 (b) and Fig. 4 (b) that the hybridization with local search significantly decreases the average CPU time. In this sense, our intention to design a simple hybrid algorithm has been realized in our S-MOGLS algorithm. On the other hand, we can see from Fig. 3 (a) and Fig. 4 (a) that the performance of our S-MOGLS algorithm is very poor around the top-right corner where almost all solutions are examined by local search (i.e., where the total number of generation updates is very small). From Fig. 3 (c) and Fig. 4

(c) where various parameter specifications are examined under the same CPU time (i.e., 5.0 seconds and 15.0 seconds), we can see that the hybridization with local search improves (i.e., decreases) the average value of the $D1_R$ measure when the balance between genetic search and local search is appropriate. In Fig. 3 (c), we observe a deep valley (i.e., a parameter region with high performance) where the value of $P_{LS} \cdot l\_opt$ can be roughly viewed as being constant (i.e., the relative computational load of local search is roughly the same). We also observe such a valley in Fig. 4 (c).

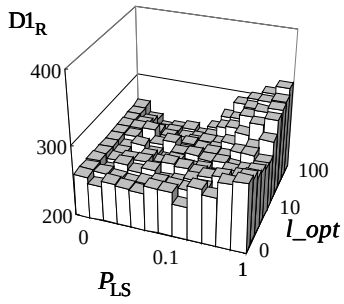
4.3 Comparison among Four Variants

As in Fig. 3 (c) with the stopping condition of the same CPU time (i.e., 5.0 seconds), we compare the four variants of our S-MOGLS algorithm. In Fig. 5, we show the average value of the $D1_R$ measure over 30 runs on the 2-250 problem for each of the 12×12 combinations of P_{LS} and l_opt where the stopping condition of each run is the CPU time of 5.0 seconds.

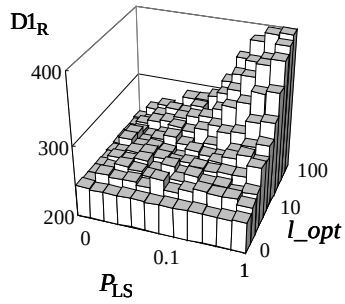


(a) S-MOGLS Version 1 algorithm.

(b) S-MOGLS Version 2 algorithm.



(c) S-MOGLS Version 3 algorithm.



(d) S-MOGLS Version 4 algorithm.

Figure 5. Experimental results on the 2-250 problem by the four variants. Each variant is executed for 5.0 seconds.

While the S-MOGLS Version 1 algorithm uses the weighted scalar fitness function for parent selection, Version 2 uses the parent selection mechanism of the NSGA-II, which is based on Pareto ranking and the crowding measure. The difference between Fig. 5 (a) and Fig. 5 (b) shows the effect of the use of the weighted scalar fitness function for parent selection. We can see that Fig. 5 (a) is inferior to Fig. 5 (b) near the bottom and left sides where the relative computation load of local search is zero or very small. This means that the weighted scalar fitness function does not work well for parent selection in pure EMO algorithms. When the balance between genetic search and local search is appropriate, good results are obtained from the weighted scalar fitness function as shown by the deep valley in Fig. 5 (a).

The S-MOGLS Version 3 algorithm can be viewed as an interpolation of its Version 1 and Version 2 algorithms. The larger the value of P_{LS}, Version 3 becomes more similar to Version 1. Version 3 with $P_{LS} = 1$ is the same as Version 1 while Version 3 with $P_{LS} = 0$ is the same as Version 2. Experimental results in Fig. 5 (c) are consistent with this algorithmic nature of the S-MOGLS Version 3 algorithm. That is, experimental

results with large values of P_{LS} in Fig. 5 (c) are similar to Fig. 5 (a) while those with small values of P_{LS} are similar to Fig. 5 (b).

The difference between Fig. 5 (b) and Fig. 5 (d) is due to the difference in the implementation of local search. The weighted scalar fitness function is used for local search in Fig. 5 (b) while Pareto ranking is used in Fig. 5 (d). From the comparison between Fig. 5 (b) and Fig. 5 (d), we can see that the weighted scalar fitness function works better in local search than Pareto ranking. Similar observations have been often reported in the literature (e.g., see Ishibuchi et al. (2003)). It should be noted that Pareto ranking works better than the weighted scalar fitness function in parent selection as shown in Fig. 5 (a) and Fig. 5 (b).

4.4 Comparison with Other Algorithms

Using the parameter specifications in Table 1 in Subsection 4.1, we compare our S-MOGLS algorithm with the two pure EMO algorithms (i.e., NSGA-II and SPEA) and the two memetic EMO algorithms (i.e., M-PAES and MOGLS of Jaszkiewicz). In this subsection, we use both *l_opt* and *l_ fails* in our S-MOGLS algorithm in the same manner as in the other memetic EMO algorithms. The value of P_{LS} is specified as $P_{LS} = 0.01$ in our S-MOGLS algorithm. We implement the M-PAES and the MOGLS with no restriction on the secondary population size. Each algorithm is applied to each test problem 30 times. Then the average values of the generational distance, the $D1_R$ measure and the CPU time are calculated over those 30 runs of each algorithm on each test problem. The standard deviation is also calculated for each average value over those 30 runs. Experimental results are summarized in Tables 2-4 where the best result (i.e., the smallest average value) in each row is highlighted by boldface. The standard deviation corresponding to each average value is shown in parentheses in those tables. In our computational experiments, we implement all algorithms by ourselves. Thus our experimental results (especially CPU time) by each algorithm in this paper can be further improved by more sophisticated implementations.

Table 2. Average values of the generational distance by the four variants of our S-MOGLS algorithm and the other four algorithms. The standard deviation is shown in parentheses. The best average result for each test problem is highlighted by boldface.

Problem	SPEA	NSGA-II	M-PAES	MOGLS	Version 1	Version 2	Version 3	Version 4
2-250	205 (24)	**61** (10)	73 (15)	157 (19)	74 (14)	76 (14)	89 (12)	78 (15)
2-500	566 (43)	**149** (26)	225 (28)	405 (39)	181 (27)	190 (30)	183 (25)	182 (32)
2-750	923 (71)	**224** (29)	345 (51)	372 (57)	237 (36)	**224** (44)	234 (34)	232 (33)
3-250	367 (51)	75 (12)	**53** (5)	98 (18)	59 (10)	100 (16)	84 (19)	81 (19)
3-500	1079 (85)	306 (39)	301 (45)	424 (44)	**217** (36)	329 (43)	328 (37)	319 (42)
3-750	1945 (121)	560 (60)	609 (92)	497 (82)	**326** (50)	544 (63)	570 (61)	564 (52)
4-250	676 (85)	360 (38)	147 (28)	**130** (20)	158 (24)	342 (25)	337 (31)	376 (33)
4-500	1791 (119)	793 (51)	580 (63)	**447** (68)	456 (58)	795 (50)	788 (61)	847 (72)
4-750	2669 (150)	1133 (84)	658 (117)	**428** (67)	513 (81)	1123 (78)	1111 (84)	1156 (100)

Table 3. Average values of the $D1_R$ measure by the four variants of our S-MOGLS algorithm and the other four algorithms. The standard deviation is shown in parentheses. The best average result for each test problem is highlighted by boldface.

Problem	SPEA	NSGA-II	M-PAES	MOGLS	Version 1	Version 2	Version 3	Version 4
2-250	289 (21)	264 (27)	264 (40)	**231** (17)	293 (36)	263 (39)	246 (25)	260 (23)
2-500	692 (39)	**494** (49)	602 (48)	502 (31)	551 (41)	522 (41)	508 (49)	509 (47)
2-750	1174 (36)	793 (67)	1070 (81)	**620** (49)	846 (66)	798 (65)	810 (47)	778 (47)
3-250	505 (39)	**234** (17)	280 (26)	295 (20)	337 (22)	253 (18)	249 (16)	252 (20)
3-500	1266 (68)	**530** (22)	645 (25)	678 (34)	601 (29)	566 (30)	572 (26)	563 (26)
3-750	2352 (102)	**1194** (42)	1528 (53)	1314 (80)	1436 (54)	1225 (43)	1216 (41)	1216 (57)
4-250	796 (70)	392 (25)	**341** (21)	390 (28)	366 (16)	392 (18)	388 (17)	415 (26)
4-500	2059 (107)	989 (54)	893 (41)	900 (44)	**822** (27)	1002 (50)	983 (61)	1041 (70)
4-750	3267 (134)	1679 (62)	1761 (66)	**1521** (101)	1527 (34)	1687 (53)	1670 (55)	1719 (60)

Table 4. Average values of the CPU time by the four variants of our S-MOGLS algorithm and the other four algorithms. The standard deviation is shown in parentheses. The best average result for each test problem is highlighted by boldface.

Problem	SPEA	NSGA-II	M-PAES	MOGLS	Version 1	Version 2	Version 3	Version 4
2-250	2 (0.06)	5 (0.03)	7 (0.32)	**1** (0.02)	4 (0.04)	4 (0.06)	4 (0.04)	5 (0.04)
2-500	5 (0.08)	10 (0.02)	41 (1.62)	**3** (0.02)	8 (0.06)	9 (0.10)	9 (0.09)	10 (0.06)
2-750	**11** (0.18)	18 (0.02)	83 (5.91)	14 (0.22)	17 (0.08)	19 (0.04)	19 (0.04)	19 (0.05)
3-250	7 (0.38)	10 (0.07)	13 (0.29)	**2** (0.05)	10 (0.09)	9 (0.08)	9 (0.08)	10 (0.08)
3-500	17 (1.11)	16 (0.05)	61 (2.86)	**5** (0.03)	13 (0.09)	14 (0.12)	14 (0.09)	15 (0.09)
3-750	32 (1.49)	26 (0.07)	107 (6.17)	**22** (0.35)	25 (0.05)	27 (0.08)	27 (0.08)	28 (0.07)
4-250	24 (1.41)	15 (0.09)	43 (0.44)	**4** (0.03)	14 (0.10)	16 (0.12)	16 (0.16)	13 (011)
4-500	64 (2.86)	23 (0.17)	134 (3.37)	**7** (0.04)	19 (0.13)	20 (0.14)	20 (0.17)	20 (0.11)
4-750	123 (3.77)	35 (0.16)	250 (14.6)	**30** (0.61)	34 (0.12)	37 (0.22)	37 (0.17)	35 (0.16)

From Table 2, we can see that the S-MOGLS Version 1 algorithm outperforms the other algorithms for many test problems in terms of the generational distance (i.e., in terms of the convergence speed to the Pareto-front) except that it is slightly outperformed by the NSGA-II for the two-objective problems and by the MOGLS for the four-objective problems. Among the four variants of our S-MOGLS algorithm, the best results are obtained by Version 1 in terms of the generational distance in Table 2. On the other hand, the S-MOGLS Version 1 algorithm is somewhat inferior to the MOGLS and the NSGA-II for some test problems, comparable to the M-PAES on average, and superior to the SPEA for many test problems in terms of the $D1_R$ measure in Table 3 (i.e., in terms of both the convergence speed to the Pareto-front and the diversity of obtained solutions). While the best results are obtained by Version 1 among the four variants with respect to the convergence speed to the Pareto-front in Table 2, there is no large difference among their performance in Table 3 with respect to the $D1_R$ measure. This may be because the use of the weighted scalar fitness function in parent selection in Version 1 has a negative effect on the diversity of obtained solutions while it has a positive effect on the convergence speed to the Pareto-front. In terms of CPU time, Table 4 clearly demonstrates that the NSGA-II, the MOGLS and our S-MOGLS are much faster than the SPEA and the M-PAES especially for the four-objective test problems.

As shown in Fig. 3 (b) and Fig. 4 (b), the CPU time of our S-MOGLS algorithm strongly depends on the balance between local search and genetic search. The larger the weight of local search is (i.e., the larger the local search parameters P_{LS}, *l_opt* and *l_ fails* are), the shorter the CPU time is. In our computational experiments, P_{LS} was specified as $P_{\mathrm{LS}} = 0.01$. If we assign a larger value to P_{LS}, our S-MOGLS algorithm can become much faster at the cost of the deterioration in its search ability as shown in Fig. 3 and Fig. 4. The CPU time of our S-MOGLS algorithm (and other algorithms) also strongly depends on the population size. In our computational experiments, we used the same population size in our S-MOGLS algorithm as in the NSGA-II for each test problem (see Table 1). If we use the same population size as in the M-PAES (i.e., 1/5 of the current specification), the CPU time of our S-MOGLS algorithm is significantly decreased (roughly speaking, it is decreased to 1/2 of the reported average values in Table 4).

4.5 Further Discussions on the Hybridization with Local Search

In Jaszkiewicz (2001, 2002b), he used a different repair procedure based on the weighted scalar fitness function in his MOGLS algorithm. In his repair procedure, items were removed in the increasing order of the following ratio using the current weight vector $\boldsymbol{\lambda} = (\lambda_1, \lambda_2, ..., \lambda_k)$ in the weighted scalar fitness function:

$$q_j = \sum_{i=1}^{k} \lambda_i p_{ij} \Big/ \sum_{i=1}^{k} w_{ij} , \quad j = 1,2,...,n . \tag{9}$$

It should be noted that this greedy repair procedure is not always directly applicable to EMO algorithms. Since the MOGLS and S-MOGLS Version 1 algorithms use the weighted scalar fitness function in parent selection as well as in local search, the greedy repair procedure in (9) is directly applicable to these algorithms. On the contrary, this greedy repair procedure cannot be directly used in the following algorithms that are not based on the weighted scalar fitness function: the S-MOGLS Version 4, SPEA, NSGA-II, and M-PAES. In the S-MOGLS Version 2 algorithm, this greedy repair procedure is applicable in the local search

part based on the weighted scalar fitness function. In the S-MOGLS Version 3 algorithm, it can be used in genetic search only when parent solutions are chosen based on the weighted scalar fitness function.

We demonstrate the effect of this greedy repair procedure on the search ability of the MOGLS and S-MOGLS Version 1 algorithms in Fig. 6 where "Maximum ratio" and "Weighted scalar" denote the greedy repair procedures in (8) and (9), respectively. Fig. 6 shows a single solution set obtained by a single run of each algorithm with a different repair procedure on the 2-250 test problem. From Fig. 6, we can see that the performance of the MOGLS and S-MOGLS Version 1 algorithms is significantly improved by the use of the greedy repair procedure in (9) based on the weighted scalar fitness function.

Using the weighted scalar greedy repair procedure in (9), we perform the same computational experiments as in Tables 2-4. This greedy repair procedure is used only when it is directly applicable (i.e., only when unfeasible solutions are related to the weighted scalar fitness function). For example, in the case of the S-MOGLS Version 2 algorithm, the maximum ratio greedy repair in (8) is used in genetic search while the weighted scalar greedy repair in (9) is used in local search. Experimental results are summarized in Tables 5-7. From the comparison between Table 2 and Table 5 (and between Table 3 and Table 6), we can see that the performance of the MOGLS and S-MOGLS algorithms (except for Version 4) is improved by the use of the weighted scalar greedy repair procedure. It should be also noted that the use of this greedy repair procedure increases the CPU time of those algorithms. The CPU time of the Version 2 and Version 3 algorithms, however, is still very small in Table 7 while its search ability is improved in Table 5 and Table 6. This means that these two variants of our S-MOGLS algorithm are simple but powerful. That is, our intention to design a simple but powerful memetic EMO algorithm has been realized. Of course, the increase in the CPU time by the use of the weighted scalar greedy repair procedure may be partially remedied by the use of a sophisticated sorting method and/or the use of prespecified weight vectors instead of randomly generated ones.

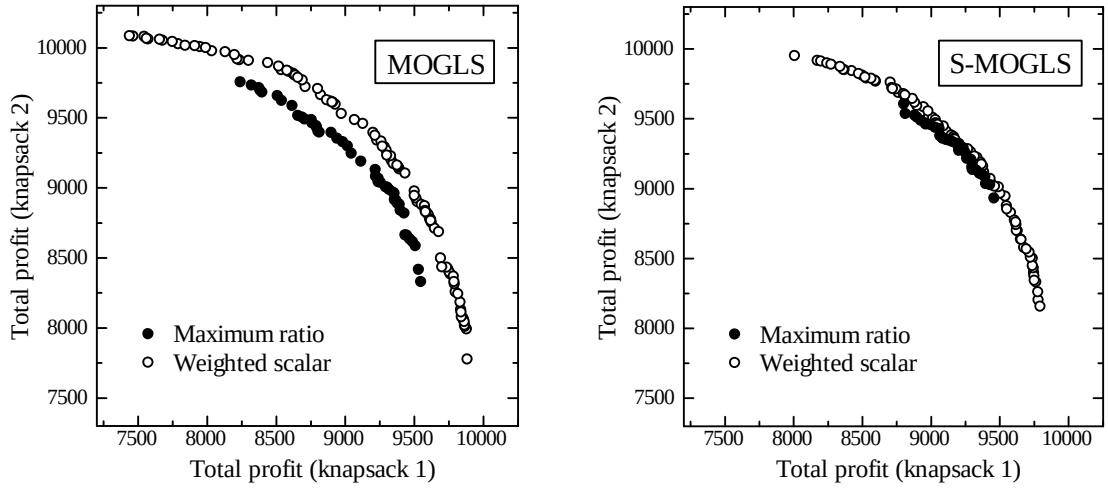


Figure 6. A solution set obtained by a single run of each of the MOGLS of Jaszkiewicz and S-MOGLS Version 1 algorithms with a different greedy repair procedure on the 2-250 problem.

Table 5. Average values of the generational distance by the four variants of our S-MOGLS algorithm and the other four algorithms. The weighted scalar greedy repair procedure is used whenever it is directly applicable.

Problem	SPEA	NSGA-II	M-PAES	MOGLS	Version 1	Version 2	Version 3	Version 4
2-250	205 (24)	61 (10)	73 (15)	24 (2)	**20** (3)	80 (9)	72 (10)	78 (15)
2-500	566 (43)	149 (26)	225 (28)	57 (10)	**29** (6)	163 (18)	145 (18)	182 (32)
2-750	923 (71)	224 (29)	345 (51)	106 (21)	**47** (6)	218 (32)	186 (24)	232 (33)
3-250	367 (51)	75 (12)	**53** (5)	166 (9)	110 (7)	96 (16)	94 (13)	81 (19)
3-500	1079 (85)	306 (39)	301 (45)	246 (44)	**154** (14)	325 (40)	304 (32)	319 (42)
3-750	1945 (121)	560 (60)	609 (92)	350 (50)	**279** (17)	551 (55)	490 (46)	564 (52)
4-250	676 (85)	360 (38)	**147** (28)	162 (10)	225 (22)	330 (28)	364 (46)	376 (33)
4-500	1791 (119)	793 (51)	580 (63)	378 (34)	**347** (35)	715 (48)	710 (62)	847 (72)
4-750	2669 (150)	1133 (84)	658 (117)	658 (72)	**518** (57)	1074 (94)	1055 (97)	1156 (100)

Table 6. Average values of the $D1_R$ measure by the four variants of our S-MOGLS algorithm and the other four algorithms. The weighted scalar greedy repair procedure is used whenever it is directly applicable.

Problem	SPEA	NSGA-II	M-PAES	MOGLS	Version 1	Version 2	Version 3	Version 4
2-250	289 (21)	264 (27)	264 (40)	**30** (4)	63 (13)	145 (30)	119 (19)	260 (23)
2-500	692 (39)	494 (49)	602 (48)	**63** (21)	95 (18)	270 (25)	220 (24)	509 (47)
2-750	1174 (36)	793 (67)	1070 (81)	**107** (38)	210 (28)	729 (55)	609 (69)	778 (47)
3-250	505 (39)	234 (17)	280 (26)	**146** (10)	181 (16)	191 (13)	168 (18)	252 (20)
3-500	1266 (68)	530 (22)	645 (25)	**241** (55)	310 (25)	457 (30)	420 (28)	563 (26)
3-750	2352 (102)	1194 (42)	1528 (53)	**292** (55)	714 (50)	1174 (52)	1061 (64)	1216 (57)
4-250	796 (70)	392 (25)	341 (21)	**173** (18)	268 (19)	359 (20)	380 (39)	415 (26)
4-500	2059 (107)	989 (54)	893 (41)	**363** (51)	500 (28)	880 (61)	871 (78)	1041 (70)
4-750	3267 (134)	1679 (62)	1761 (66)	**525** (70)	949 (51)	1610 (62)	1565 (61)	1719 (60)

Table 7. Average values of the CPU time by the four variants of our S-MOGLS algorithm and the other four algorithms. The weighted scalar greedy repair procedure is used whenever it is directly applicable.

Problem	SPEA	NSGA-II	M-PAES	MOGLS	Version 1	Version 2	Version 3	Version 4
2-250	**2** (0.06)	5 (0.03)	7 (0.32)	**2** (0.02)	16 (0.18)	4 (0.07)	4 (0.06)	5 (0.04)
2-500	**5** (0.08)	10 (0.02)	41 (1.62)	**5** (0.06)	66 (0.89)	9 (0.10)	9 (0.06)	10 (0.06)
2-750	**11** (0.18)	18 (0.02)	83 (5.91)	45 (0.47)	227 (0.53)	20 (0.03)	20 (0.03)	19 (0.05)
3-250	7 (0.38)	10 (0.07)	13 (0.29)	**3** (0.03)	27 (0.26)	9 (0.07)	9 (0.06)	10 (0.08)
3-500	17 (1.11)	16 (0.05)	61 (2.86)	**8** (0.06)	90 (0.95)	14 (0.10)	14 (0.10)	15 (0.09)
3-750	32 (1.49)	**26** (0.07)	107 (6.17)	60 (0.49)	277 (0.79)	28 (0.07)	29 (0.07)	28 (0.07)
4-250	24 (1.41)	15 (0.09)	43 (0.44)	**5** (0.03)	36 (0.30)	15 (0.16)	12 (0.09)	13 (011)
4-500	64 (2.86)	23 (0.17)	134 (3.37)	**11** (0.07)	110 (0.84)	20 (0.16)	19 (0.15)	20 (0.11)
4-750	123 (3.77)	**35** (0.16)	250 (14.6)	79 (0.86)	330 (0.72)	38 (0.19)	36 (0.14)	**35** (0.16)

5 Conclusions

In this paper, we proposed a simple but powerful memetic EMO (i.e., S-MOGLS) algorithm. In the design of our S-MOGLS algorithm, emphasis was placed on its algorithmic simplicity as well as its search ability. Our algorithm has the same generation update mechanism as the NSGA-II: Elitism is implemented using Pareto ranking and the concept of crowding without using a secondary population. At the same time, our algorithm uses the weighted scalar fitness function as in other MOGLS algorithms for choosing a pair of parent solutions and executing local search for its offspring. Experimental results showed that our S-MOGLS algorithm is comparable or superior to the well-known EMO and memetic EMO algorithms (i.e., SPEA, NSGA-II, M-PAES and MOGLS) while its CPU time is almost the same as or shorter than the other algorithms for many test problems. It was also demonstrated that the use of a heuristic greedy repair procedure based on the weighted scalar fitness function improved the performance of the MOGLS and S-MOGLS algorithms. When our S-MOGLS algorithm was used together with the weighted scalar greedy repair procedure, its search ability outperformed that of the SPEA, NSGA-II and M-PAES algorithms with the maximum ratio greedy repair procedure.

We also demonstrated the importance of the balance between genetic search and local search in our S-MOGLS algorithm. When this balance was not appropriate, the search ability of our S-MOGLS algorithm was severely deteriorated. We also examined four variants of our S-MOGLS algorithm. Experimental results showed that the use of the weighted scalar fitness function in parent selection had a negative effect on the diversity of solutions and a positive effect on the convergence speed to the Pareto-front. On the other hand, the weighted scalar fitness function worked better than Pareto ranking in local search. When the weighted scalar greedy repair procedure was employed, the use of the weighted scalar fitness function became much more advantageous. In this case, our S-MOGLS Version 2 and Version 3 algorithms outperformed the SPEA, the NSGA-II and the M-PAES from an overall viewpoint with respect to both the search ability and the CPU time. The best results with respect to the generational distance were obtained by our S-MOGLS Version 1 algorithm while the MOGLS worked best with respect to the $D1_R$ measure.

One important issue that was not discussed in this paper is the selection of initial solutions for local

search. It was reported in Ishibuchi et al. (2003) for multiobjective flowshop scheduling problems that the performance of their MOGLS algorithm was significantly improved by choosing good initial solutions for local search from the current population. While we did not report experimental results in this paper, the performance of our S-MOGLS Version 2 algorithm was improved by the choice of good initial solutions for local search.

Acknowledgments

The authors would like to thank the financial support from Japan Society for the Promotion of Science (JSPS) through Grand-in-Aid for Scientific Research (B): KAKENHI (14380194).

References

Coello Coello, C. A., van Veldhuizen, D. A., and Lamont, G. B. (2002), *Evolutionary Algorithms for Solving Multi-Objective Problems*, Kluwer Academic Publishers, Boston, MA.

Deb, K. (2001), *Multi-Objective Optimization Using Evolutionary Algorithms*, John Wiley & Sons, Chichester, UK.

Deb, K., Pratap, A., Agarwal, S., and Meyarivan, T. (2002), "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Trans. on Evolutionary Computation*, vol. 6, no. 2, pp. 182-197.

Ishibuchi, H. and Kaige, S. (2003), "Effects of repair procedures on the performance of EMO algorithms for multiobjective 0/1 knapsack problems," *Proc. of 2003 Congress on Evolutionary Computation* (in press).

Ishibuchi, H. and Murata, T. (1996), "Multi-objective genetic local search algorithm," *Proc. of 1996 IEEE International Conference on Evolutionary Computation*, pp. 119-124.

Ishibuchi, H. and Murata, T. (1998), "A multi-objective genetic local search algorithm and its application to flowshop scheduling," *IEEE Trans. on Systems, Man, and Cybernetics - Part C: Applications and Reviews*, vol. 28, no. 3, pp. 392-403.

Ishibuchi, H., Yoshida, T., and Murata, T. (2003), "Balance between genetic search and local search in memetic algorithms for multiobjective permutation flowshop scheduling," *IEEE Trans. on Evolutionary Computation*, vol. 7, no. 2, pp. 204-223.

Jaszkiewicz, A. (2001), "Comparison of local search-based metaheuristics on the multiple objective knapsack problem," *Foundations of Computing and Decision Sciences*, vol. 26, no. 1, pp. 99-120.

Jaszkiewicz, A. (2002a), "Genetic local search for multi-objective combinatorial optimization," *European Journal of Operational Research*, vol. 137, no. 1, pp. 50-71.

Jaszkiewicz, A. (2002b), "On the performance of multiple-objective genetic local search on the 0/1 knapsack problem - A comparative experiment," *IEEE Trans. on Evolutionary Computation*, vol. 6, no. 4, pp. 402-412.

Knowles, J. D. and Corne, D. W. (2000a), "M-PAES: A memetic algorithm for multiobjective optimization," *Proc. of 2000 Congress on Evolutionary Computation*, pp. 325-332.

Knowles, J. D. and Corne, D. W. (2000b), "A comparison of diverse approaches to memetic multiobjective combinatorial optimization," *Proc. of 2000 Genetic and Evolutionary Computation Conference Workshop Program* (WOMA I), pp. 103-108.

Knowles, J. D. and Corne, D. W. (2002), "On metrics for comparing non-dominated sets," *Proc. of 2002 Congress on Evolutionary Computation*, pp. 711-716.

Murata, T., Kaige, S., and Ishibuchi, H. (2003), "Generalization of dominance relation-based replacement rules for memetic EMO algorithms," *Proc. of 2003 Genetic and Evolutionary Computation Conference*, pp. 1234-1245.

Schaffer, J. D. (1985), "Multiple objective optimization with vector evaluated genetic algorithms," *Proc. of 1st International Conference on Genetic Algorithms and Their Applications*, pp. 93-100.

Zitzler, E., Deb, K., and Thiele, L. (2000), "Comparison of Multiobjective Evolutionary Algorithms: Empirical Results," *Evolutionary Computation*, vol. 8, no. 2, pp. 173-195.

Zitzler, E. and Thiele, L. (1999), "Multiobjective evolutionary algorithms: A comparative case study and the strength Pareto approach," *IEEE Trans. on Evolutionary Computation*, vol. 3, no. 4, pp. 257-271.