An Efficient Non-dominated Sorting Method for Evolutionary Algorithms

Article *in* Evolutionary Computation · February 2008

DOI: 10.1162/evco.2008.16.3.355 · Source: PubMed

An Efficient Non-dominated Sorting Method for Evolutionary Algorithms

Hongbing Fang hfang@uncc.edu
Department of Mechanical Engineering and Engineering Science, University of North Carolina at Charlotte, Charlotte, NC 28223, USA

Qian Wang qwang2@engineering.uiowa.edu
Kal Krishnan Consulting Services, Inc., 300 Lakeside Drive # 1080, Oakland, CA 94612

Yi-Cheng Tu tuyc@cs.purdue.edu
Department of Computer Science, Purdue University, West Lafayette, IN 47907, USA

Mark F. Horstemeyer mfhorst@me.msstate.edu
Department of Mechanical Engineering, Mississippi State University, Starkville, MS 39762, USA

Abstract

We present a new non-dominated sorting algorithm to generate the non-dominated fronts in multi-objective optimization with evolutionary algorithms, particularly the NSGA-II. The non-dominated sorting algorithm used by NSGA-II has a time complexity of $O(MN^2)$ in generating non-dominated fronts in one generation (iteration) for a population size N and M objective functions. Since generating non-dominated fronts takes the majority of total computational time (excluding the cost of fitness evaluations) of NSGA-II, making this algorithm faster will significantly improve the overall efficiency of NSGA-II and other genetic algorithms using non-dominated sorting. The new non-dominated sorting algorithm proposed in this study reduces the number of redundant comparisons existing in the algorithm of NSGA-II by recording the dominance information among solutions from their first comparisons. By utilizing a new data structure called the dominance tree and the divide-and-conquer mechanism, the new algorithm is faster than NSGA-II for different numbers of objective functions. Although the number of solution comparisons by the proposed algorithm is close to that of NSGA-II when the number of objectives becomes large, the total computational time shows that the proposed algorithm still has better efficiency because of the adoption of the dominance tree structure and the divide-and-conquer mechanism.

Keywords

Genetic algorithm, evolutionary algorithm, non-dominated sorting, divide-and-conquer, dominance tree.

1 Introduction

Design optimization for many engineering applications is multi-objective in nature. For most multi-objective problems, the objectives conflict with each other, and it is impossible to obtain a single set of values for the design variables that corresponds to

the optima of all the objectives. In this situation, an optimal solution represents a certain level of trade-offs among all of the objectives, and a set of trade-off solutions exists for a multi-objective optimization problem. The set containing all the trade-off solutions is formally called the Pareto front (Coello, 1999), and the solutions on the Pareto front are also called non-dominated solutions. Therefore, solving a multi-objective optimization problem refers to obtaining a subset of the solutions on the Pareto front instead of getting each objective's optimum.

If the relative importance of all the objectives is known, we can assign a weight coefficient to each objective, combine all the weighted objective functions into a single objective, and solve it as a single-objective problem. Alternatively, if we know the limits of all but one objective, we can change those objectives with known limits into constraints and find the optimum of the objective with no limit. In both cases, the multi-objective optimization problem can be solved using single-objective optimization methods. However, selecting the weight coefficients for each objective and/or determining the objective limits are largely based on the designers' preferences and experiences. The single solution thus obtained may not be satisfactory. It is desirable to obtain a set of solutions with different levels of trade-offs from which the designers can select the final design(s).

Although we can obtain multiple trade-off solutions by using different combinations of the weight coefficients in the aforementioned single-objective approach, solutions can be lost on the concave region of the solution space (Das and Dennis, 1997). To this end, the genetic algorithm, which was first introduced by Holland (1975) belonging to the family of evolutionary algorithms, has been used to obtain solutions with good spread on the Pareto front even when the solution spaces contain concave regions. Over the past ten years, there has been an increasing interest in solving multi-objective optimization problems using evolutionary algorithms such as NSGA-II (Deb et al., 2002a), PESA (Corne et al., 2000), PAES (Knowles and Corne, 2000), SPEA (Zitzler and Thiele, 1999), and Micro-GA (Coello and Pulido, 2001).

The non-dominated sorting genetic algorithm (NSGA) developed by Srinivas and Deb (1995) has been reported to converge to the Pareto front with a good spread of solutions. However, NSGA was also criticized for its lack of elitism and high computational cost, a time complexity of $O(MN^2)$ for M objectives and N solutions. This time complexity is determined by the non-dominated sorting algorithm for generating the non-dominated fronts in each generation; therefore, improving the computational efficiency of non-dominated sorting will improve the overall efficiency (the cost for fitness evaluation is excluded). The NSGA-II was subsequently developed to reduce the computational cost as well as to include elitism (Deb et al., 2002a). We refer to the non-dominated sorting algorithm of NSGA-II as Deb's algorithm hereafter in this paper. Although Deb's algorithm adopts a better bookkeeping scheme and is faster than NSGA, its time complexity remains the same as that of NSGA. The computational cost of NSGA-II is still high due to the repetitive comparisons between the same pairs of solutions when generating the non-dominated fronts.

Jensen proposed a non-dominated sorting algorithm to improve the computational efficiency of NSGA-II and other evolutionary algorithms that perform non-dominated sorting (Jensen, 2003). He showed that the time complexity of his algorithm is $O(MN \log^{M-1} N)$. However, Jensen's algorithm does not generate the same non-dominated fronts as those generated by Deb's algorithm when there are duplicate solutions in the population. Jensen's algorithm puts duplicated solutions into the same front; this is not desirable for evolutionary algorithms. We compared the processing times of

Jensen's algorithm with that of Deb's algorithm and found different speedups from those reported by Jensen (2003).

In this paper, we present a new non-dominated sorting algorithm that generates the same non-dominated fronts as Deb's algorithm. The new algorithm adopts the divide-and-conquer mechanism as well as a data structure called dominance tree to improve both computational and memory efficiencies of NSGA-II. This new algorithm was found to preserve satisfactory speedups when the number of objectives becomes large, which was not true for Jensen's algorithm. In the remaining portion of this paper, we give an overview of Deb's and Jensen's algorithms. Then we present details of the dominance tree and the divide-and-conquer mechanism used in our new algorithm. Finally, we compare the performance of the new algorithm with those of Deb's and Jensen's algorithms on the number of operations and processing time; this is followed by some concluding remarks.

2 Overview of Non-dominated Sorting Algorithms

For multi-objective optimization problems solved by genetic algorithms, an initial set of solutions called a population is randomly generated. A set of genetic operators, selection, crossover, and mutation, are then applied to the solutions to generate a new set of solutions for the next iteration called a generation. The final set of solutions is obtained after a certain number of generations. After performing the Pareto non-dominance check, the final solutions are all non-dominated or trade-off solutions such that no solution is strictly better than any other solutions in the set. We now provide some definitions that will be used in the following discussions.

We define that Solution x dominates Solution y in a minimization problem of m objectives by

$$x \prec y \mid \forall_i : f_i(x) \leq f_i(y) \quad \text{and} \quad \exists_j : f_j(x) < f_j(y) \tag{1}$$

where $f_i(x)$ and $f_i(y)$ are the values of the i-th objective corresponding to x and y, respectively. The above definition means that all the objectives corresponding to Solution x are smaller than or equal to those corresponding to y, and there exists at least one objective whose value for x is smaller than that for y. If x does not dominate y and vice versa, the two are said to be non-dominated. A set of non-dominated solutions is called a non-dominated front.

For solutions of a given population, there may be multiple non-dominated fronts. We assign each front a unique number and represent it by $Front_k$, where k ($k > 1$) is the front number. In this study, we rank all the fronts by their front numbers with smaller numbers representing higher ranks. For example, $Front_1$ ranks higher than $Front_2$ and $Front_3$. The non-dominated fronts have the following properties.

1. A solution in $Front_{k+1}$ must be dominated by at least one solution in $Front_k$.
2. A solution in $Front_{k+1}$ may or may not dominate solutions in $Front_{k+2}$.

Solutions in a higher-ranked front have higher preference (fitness) in the selection process than those in a lower-ranked front, because the latter is dominated by the former. We now introduce NSGA-II and briefly discuss Jensen's sorting algorithm.

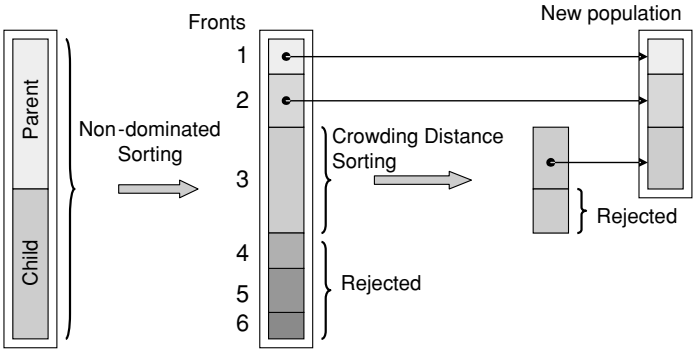


Figure 1: Schematic procedure of NSGA-II.

2.1 The Non-dominated Sorting Algorithm of NSGA-II

In each generation in NSGA-II, genetic operators are applied to the parent population to obtain an equal sized child population. The parent and child are then combined to form a temporary population on which the non-dominated sorting algorithm is applied to generate the non-dominated fronts. Therefore, the population size for non-dominated sorting is twice as large as the parent population.

The population for the next generation has the same size as the parent population; its solutions are selected from non-dominated fronts with the highest ranks. The solutions within a front are sorted by crowding-distances so that those with the largest crowding-distances are selected into the next generation, if only a portion of the front can be selected. Details about the definition and sorting algorithm of crowding-distance sorting can be found in the literature (Deb et al., 2002a). Figure 1 illustrates the schematic procedure of NSGA-II in one generation.

In NSGA-II, non-dominated fronts are obtained one after another starting from the one with the highest rank to the one with the lowest rank. The procedure for generating the first front of a population of N solutions is as follows.

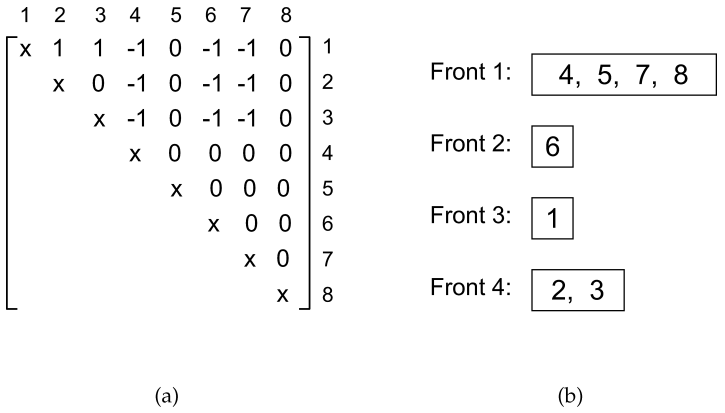
1. Create a population P by combining the parent and offspring populations.
2. Create an empty front F. Remove the first solution s_1 from P and put s_1 into F.
3. Compare the second solution s_2 in P with s_1. If s_1 dominates s_2, s_2 remains in P and go to next step; otherwise, remove s_2 from P and put it into F. If s_1 is dominated by s_2, remove s_1 from F and put it back to P; otherwise, s_1 and s_2 are non-dominated.
4. Compare s_i ($i = 3, 4, \ldots, N$) in P with all solutions in F. If solution s_i is dominated by a solution in F, s_i remains in P. Any solutions in F dominated by s_i are removed from F and put back to P. If s_i is not dominated after all the comparisons, s_i is removed from P and put into F.

After all the comparisons in the above procedure are finished, the solutions in F form the first non-dominated front.

Solutions remaining in P are used to generate the second and subsequent fronts with the same procedure as that for the first front. The number of solution comparisons

Table 1: Objective Values of a Population of Eight Solutions

Solution	f_1	f_2	f_3
1	182.08	100.13	192.21
2	187.53	246.16	203.20
3	197.15	201.57	318.86
4	47.48	74.96	22.69
5	37.05	304.83	381.19
6	126.88	54.58	144.17
7	101.77	49.18	111.91
8	37.47	18.63	446.57



(a) (b)

Figure 2: The dominance array and non-dominated fronts of eight solutions. (a) The dominance array; (b) the non-dominated fronts.

in the above procedure is in the order of $O(N^2)$. For an optimization problem with M objectives, we need M comparisons in the worst case to determine the dominance between any two solutions. The time complexity of the above procedure is therefore $O(MN^2)$ (Deb et al., 2002a).

We now demonstrate the non-dominated sorting algorithm of NSGA-II with the following example to identify where the inefficiency comes from. This example will also be used in Section 3 to demonstrate the new non-dominated sorting algorithm. The objective values of the eight solutions were randomly generated (Table 1).

The dominance relationship between any two solutions can be represented with a matrix as shown in Figure 2(a). For an element A_{ij} in the dominance matrix, a value of 1 means that Solution i dominates Solution j, -1 means that Solution j dominates Solution i, and 0 means that the two solutions are non-dominated. Note that only the upper triangle is needed to keep all the pair relationships. Figure 2(b) shows the final non-dominated fronts for the example population of the eight solutions.

Following the aforementioned procedure, we generated the non-dominated fronts of Figure 2(b) with detailed steps given in Table 2. A total of 18 paired comparisons are needed including some repetitive comparisons between Solutions 1 and 2 and Solutions 1 and 3. Solutions 1 and 2 are compared three times and the same is true for Solutions 1 and 3.

Table 2: An Example of the Non-dominated Sorting Algorithm in NSGA-II

Front No.	Step	Solution(s) in front	Remaining solution(s)	Comparison
	0		1, 2, 3, 4, 5, 6, 7, 8	
	1	1	2, 3, 4, 5, 6, 7, 8	
	2	1	2*, 3, 4, 5, 6, 7, 8	1 & 2
	3	1	2*, 3*, 4, 5, 6, 7, 8	1 & 3
1	4	4	1*, 2*, 3*, 4, 5, 6, 7, 8	1 & 4
	5	4, 5	1*, 2*, 3*, 6, 7, 8	4 & 5
	6	4, 5, 6	1*, 2*, 3*, 7, 8	4 & 6, 5 & 6
	7	4, 5, 7	1*, 2*, 3*, 6*, 8	4 & 7, 5 & 7, 6 & 7
	8	4, 5, 7, 8	1*, 2*, 3*, 6*	4 & 8, 5 & 8, 7 & 8
	0		1, 2, 3, 6	
	1	1	2, 3, 6	
2	2	1	2*, 3, 6	1 & 2
	3	1	2*, 3*, 6	1 & 3
	4	6	1*, 2*, 3*	1 & 6
	0		1, 2, 3	
3	1	1	2, 3	
	2	1	2*, 3	1 & 2
	3	1	2*, 3*	1 & 3
	0		2, 3	
4	1	2	3	
	2	2, 3		2 & 3
	Total number of solution comparisons			18

*Solution found dominated

If the dominance information between Solutions 1 and 2 can be saved after their first comparison, we can reduce the number of repetitive comparisons. The same is true for the case of Solutions 1 and 3. Before we present the new algorithm of this paper, we briefly introduce Jensen's algorithm and compare it with that of NSGA-II.

2.2 Jensen's Non-dominated Sorting Algorithm

Jensen's algorithm was an extension to the algorithm developed by Kung et al. (1975), which only found the first non-dominated fronts. In Jensen's algorithm, the solutions are first sorted by the objective values before the non-dominated fronts are generated. For the case of two objectives, a straightforward approach is taken by sorting all the solutions by their values of the first objective and then comparing their second objective values to determine their fronts. For the cases of three or more objectives, a divide-and-conquer mechanism is adopted such that the solutions are recursively divided into smaller subpopulations and comparisons are recursively performed based on each of the objectives. Details of Jensen's algorithm can be found in Jensen (2003), and the source code of this algorithm, implemented in C++, can be freely downloaded from the Web site (Jensen, 2008).

If duplicate solutions exist in a population, the same duplicates will be sorted into the same fronts by Jensen's algorithm, while only one of the duplicate solutions enters a front by Deb's algorithm. To demonstrate this, we used the DTLZ1 benchmark (Deb et al., 2002b) to obtain the objective values for seven randomly generated populations corresponding to two to eight objectives, respectively. All the populations had a size

Table 3: Sizes of Non-dominated Fronts Generated by Jensen's and Deb's Algorithms

	Number of objectives							
	2		3		5		8	
Front no.	Jensen's	Deb's	Jensen's	Deb's	Jensen's	Deb's	Jensen's	Deb's
1	13	13	25	25	81	79	117	115
2	22	21	34	33	76	75	65	63
3	12	11	43	40	33	35	17	20
4	16	16	44	43	9	10	1	2
5	18	17	32	32	1	1		
6	20	18	17	16				
7	19	19	3	9				
8	25	23	2	2				
9	18	21						
10	13	15						
11	10	10						
12	8	9						
13	4	3						
14	2	3						
15		1						

of 200; they were then used to generate the non-dominated fronts by Jensen's and Deb's algorithms. Table 3 shows the results for two, three, five, and eight objectives.

The results showed that Jensen's algorithms did not generate the same non-dominated fronts as Deb's algorithm due to the aforementioned reason. Correcting this deficiency of Jensen's algorithm is non-trivial, because the comparisons between two solutions were made for each objective and the duplicates can be identified only after sorting based on all the objectives. Since this is not the focus of this study, it is not addressed in this paper. We also compared the processing times of Jensen's algorithm with those of Deb's algorithm and the new algorithm proposed in this paper; the results will be presented in Section 4.

3 The New Non-dominated Sorting Algorithm

We found from the example in Section 2.1 that Deb's algorithm did not save the dominance information among solutions from their first comparisons. As a result, repetitive comparisons may occur between the same pair of solutions and thus increase the computational time.

The basic idea of the proposed non-dominated sorting algorithm is that, if the dominance information from the first comparisons can be saved for any two solutions, we can then eliminate the redundant comparisons between the same pairs of solutions. Furthermore, we do not even need to compare all the pairs of solutions. For example, if Solution A is found to dominate Solution B, all solutions already found being dominated by Solution B do not need to compare with Solution A, because they are all dominated by Solution A. Because the dominance information among solutions is hierarchical, a hierarchical data structure, that is, a tree structure, is preferred to save the dominance information. In this study, we used the dominance tree presented in Section 3.1 for this purpose. We also adopted the divide-and-conquer mechanism to make the new algorithm faster and more efficient in obtaining the dominance information. The idea will

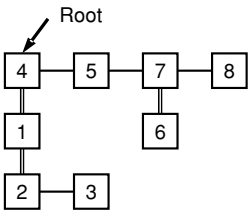


Figure 3: A dominance tree of eight solutions. A single-line connects nodes containing non-dominated solutions, and a double-line connects solutions with dominance.

become clear in the discussion in Section 3.2 when we demonstrate the new algorithm using the example of Section 2.1.

3.1 The Dominance Tree

We developed a data structure called the dominance tree to save the dominance information among solutions in order to reduce the number of repetitive comparisons. The basic unit of a dominance tree is called a node, which keeps a solution and is linked to other nodes. There are two types of links between any two nodes, the non-dominance link and the dominance link. The nodes connected with a non-dominance link are called siblings whose solutions are non-dominated. The nodes connected with a dominance link are called parent and child, respectively, with the solution in the parent node dominating the one in the child node. The dominance tree for the example problem in Section 2 is shown in Figure 3, in which 4, 5, 7, and 8 are sibling nodes and Node 4 (parent) dominates Node 1 (child). If several sibling nodes are dominated by the same parent node, only one dominance link is needed between the parent node and the first sibling node. This is the case for Nodes 1, 2, and 3 in Figure 3.

Similar to the rules for non-dominated fronts given in Section 2, the solution of a child node in a dominance tree cannot dominate any of its parent's siblings. On the other hand, the solution of a node may not necessarily dominate its siblings' children. For example, Node 1 in Figure 3 cannot dominate Nodes 5, 7, or 8, which are the siblings of the parent of Node 1. On the other hand, Nodes 5, 7, and 8 may or may not dominate Node 1. In fact, Node 1 is dominated by Node 7 but is non-dominated with Nodes 5 and 8.

After the dominance tree is generated for the entire population, we can obtain the non-dominated fronts by merging the children of all siblings at the same levels. This process starts from the root of the dominance tree and propagates until no more merge needs to be performed. For example, to obtain the non-dominated fronts in Figure 2(b) from the dominance tree in Figure 3, we start by merging Nodes 1 and 6; this results in Node 6 dominating Node 1. In this example, only one merge is needed to form the non-dominated fronts.

The algorithm for merging the children of siblings is the same as that for merging two dominance trees; this will be discussed in Section 3.2.

3.2 Divide-and-conquer Based Non-dominated Sorting Algorithm

We adopted the divide-and-conquer mechanism in the proposed non-dominated sorting algorithm to generate the dominance tree. In the general divide-and-conquer

algorithm, the original problem is first divided into two sub-problems that are similar to the original problem but smaller in size (Cormen et al., 2001). Each of the sub-problems is further divided into two sub-problems. This "divide" process continues until the solution for the sub-problem is readily available. The "conquer" process is the reversal of the "divide" process; solutions of the two sub-problems that are divided from the same problem are merged to form the solution for the larger problem. This process continues until the solution for the original problem is obtained. The time complexity of sorting N elements using full pair-comparisons is $O(N^2)$, while it is $O(N \log N)$ if the divide-and-conquer algorithm is used.

In the proposed algorithm of this paper, a population (for non-dominated sorting) is recursively divided into two populations by the midpoint until the population contains only one solution. The two neighboring dominance trees, which contain solutions of the same larger population before dividing, are then merged to form a new dominance tree. The process of merging (conquering) dominance trees is the reverse of the divide process and is also recursive. The dominance tree of the entire population is formed when all the dominance trees are merged.

The new algorithm of this paper differs from Jensen's algorithm in that it compares all of the objective values (if necessary) between two solutions and treats one as dominated by the other if the two are duplicates. As a consequence, only one copy of the duplicated solutions exists in the same front, which is the case of using Deb's algorithm. Figure 4 gives the pseudocode of the divide-and-conquer based algorithm for generating the dominance tree. This is a recursive algorithm, because both the "divide" and "conquer" procedures are recursive. When merging two dominance trees, the merge starts at the roots of the two trees. Since there can be sibling nodes at any level of the tree, all the sibling nodes at the same level of both trees need to be merged. All of the non-dominated nodes from both trees form a new sibling group. If a node is found dominated by another node, it is inserted into the dominating node's subtree for dominated nodes. The insertion follows the same procedure as that of the merge, because the dominated node itself is a dominance tree.

We now use the example in Section 2.1 to illustrate the new algorithm. Note that the "divide" and "conquer" procedures in the algorithm of Figure 4 are combined and executed concurrently. However, for illustration purposes, we treat them as separate procedures in the following discussion. Dividing the population is straightforward as shown in Figure 5.

In the first round of the "conquer" phase as shown in Figure 6, four dominance trees were generated after performing four comparisons between Nodes 1 and 2, 3 and 4, 5 and 6, and 7 and 8, respectively. In the four trees of Round 1, Nodes 1 and 4 dominate Nodes 2 and 3, respectively. Nodes 5 and 6 are non-dominated and the same is true for Nodes 7 and 8.

In the second round, the two trees rooted at Nodes 1 and 4, respectively, were merged by a comparison between Nodes 1 and 4. Since Node 4 dominates Node 1, the tree rooted at Node 1 is merged with the tree rooted at Node 3, which was determined in Round 1 being dominated by Node 4. The merge of the trees rooted at Nodes 1 and 3, respectively, followed the same procedure as that for Nodes 1 and 4. The comparisons were made between Nodes 1 and 3, and 2 and 3, with Node 3 dominated by Node 1 but non-dominated by Node 2. The merge between the trees rooted at Nodes 5 and 7, respectively, required comparisons between Nodes 5 and 7, 5 and 8, and 6 and 7. Node 6 was found dominated by Node 7 and therefore not compared with Node 8.

```
Function GetDominanceTree(NodeList, Size)
    If Size > 1 Then
      LeftTree ← GetDominanceTree(NodeList, Size/2)
      RightTree ← GetDominanceTree(NodeList + Size/2, Size − Size/2)
    Else
      Return NodeList[0]
    End If
    Return MergeDominanceTrees(LeftTree, RightTree)
End Function

Function MergeDominanceTrees(LeftTree, RightTree)
    LeftNode ← LeftTree.Root
    RightNode ← RightTree.Root
    While LeftNode ≠ NULL and RightNode ≠ NULL
        If LeftNode ≺ RightNode Then
          TempNode ← RightTree
          RightNode ← RightNode.NextSiblingNode
          RightTree.Remove(TempNode)
          MergeDominanceTrees(LeftNode.DominatedChildren, TempNode)
        Else If RightNode ≺ LeftNode Then
            TempNode ← LeftTree
            LeftNode ← LeftNode.NextSiblingNode
            LeftTree.Remove(TempNode)
            MergeDominanceTrees(RightNode.DominatedChildren, TempNode)
        Else
            RightNode ← RightNode.NextSiblingNode
            If RightNode = NULL Then
              RightNode ← RightTree.Root
              LeftNode ← LeftNode.NextSiblingNode
            End If
        End If
    End While
    LeftNode ← LeftTree.Root
    RightNode ← RightTree.Root
    While RightNode ≠ NULL
        LeftNode.AddSiblingNode(RightNode)
        RightNode ← RightNode.NextSiblingNode
    End While
    Return LeftTree
End Function
```

Figure 4: Divide-and-conquer based non-dominated sorting algorithm, no delayed insertion.

In the third round, the comparisons to merge trees rooted at Nodes 4 and 5 were between Node 4 and Nodes 5, 7, and 8, respectively. The resulting dominance tree is the leftmost one in Round 3 of Figure 6. To generate the non-dominated fronts from the dominance tree, we also need to merge the dominated nodes of all the siblings at the same level. In this example, we merged the dominance trees rooted at Nodes 1 and 6, respectively. This resulted in Node 1 dominated by Node 6. With no more merging needed, we generated the final non-dominated fronts as shown in the rightmost of Round 3 in Figure 6, which is the same as that of Figure 2(b). The total number of comparisons by the new non-dominated sorting algorithm is 14; detailed comparisons are summarized in Table 4.

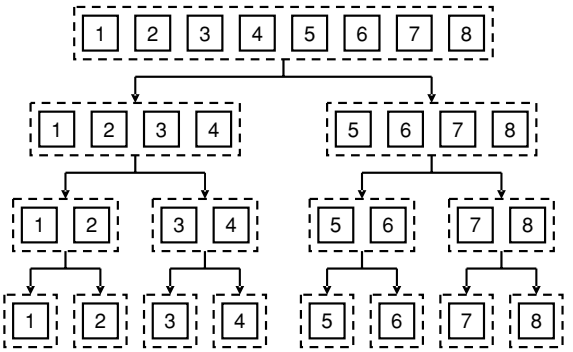


Figure 5: The divide phase: generating the comparison subgroups.

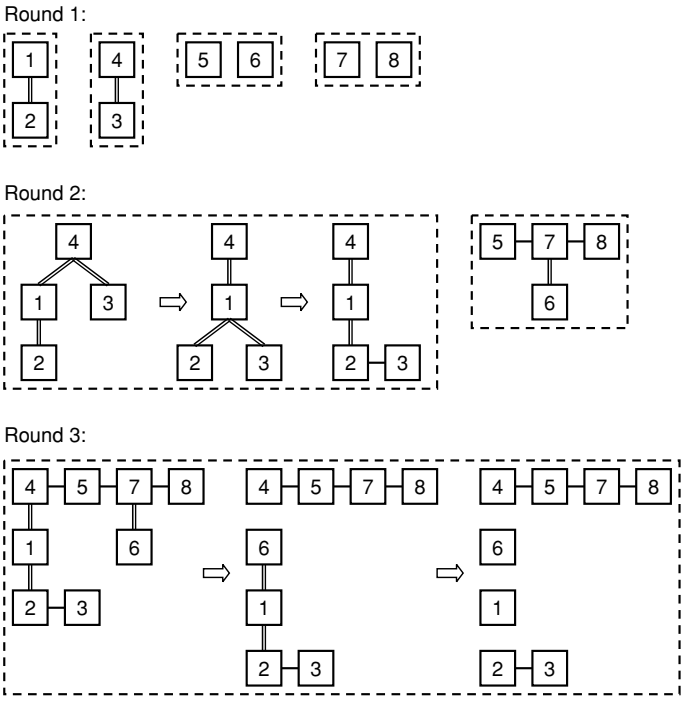


Figure 6: The conquer phase: merging dominance trees to generate non-dominated fronts.

By saving the dominance information among nodes, the new non-dominated sorting algorithm reduces the number of repetitive comparisons among solutions, and therefore is expected to be faster than Deb's algorithm. Since the proposed algorithm is a recursive one and does not require storing the dominance array, it is also memory efficient. In the proposed algorithm, each node in the dominance tree requires $O(M)$ memory to store the values of the M objectives for this solution, and a total of N nodes

Table 4: Number of Comparisons for Obtaining Non-dominated Fronts Using the New Algorithm

Round	Comparison
1	1 & 2, 3 & 4, 5 & 6, 7 & 8
2	1 & 4, 1 & 3, 2 & 3, 5 & 7, 5 & 8, 6 & 7
3	4 & 5, 4 & 7, 4 & 8, 1 & 6
Total number of comparisons	14

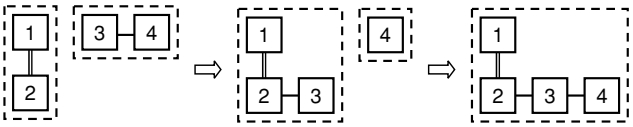


Figure 7: Merging two dominance trees without delayed insertion.

are needed for a population of size N. Therefore, the space complexity of the new algorithm is $O(MN)$, while it is $O(MN + N^2)$ for Deb's algorithm.

3.3 Improvement on the New Algorithm

Although the new algorithm presented in Section 3.2 is generally efficient, a preliminary performance evaluation showed that it sometimes required more comparisons than Deb's algorithm to generate the non-dominated fronts. Through a close examination, we identified the problem being the way of handling dominated nodes while merging two dominance trees. Consider the merge of two dominance trees shown in Figure 7 in which Nodes 2, 3, and 4 are non-dominated but all dominated by Node 1.

In the ideal situation, a total of four comparisons are needed to merge the two trees, that is, the comparisons between Nodes 1 and 3, 1 and 4, 2 and 3, and 2 and 4. During the merge process using the algorithm of Section 3.2, Node 3 is first compared with Node 1 and inserted into the latter's dominated node list after comparing with Node 2. Node 4 is then compared with Node 1 and also inserted into the dominated node list of Node 1. During this insertion, Nodes 4 and 3 are compared again due to the loss of their relationship as non-dominated solutions. Therefore a total of five comparisons are required including one repeated comparison between Nodes 3 and 4.

To resolve this problem, a strategy called "delayed insertion" was developed such that all nodes dominated by a node were temporally held without insertion until either all of the comparisons are finished or the dominating node is found dominated by another node. Then all of the dominated nodes were inserted at once using the merge algorithm in Figure 4. This strategy also saves the calls to the merge procedure, and therefore is even more efficient. The new non-dominated sorting algorithm with delayed insertion is given in Figure 8.

The new non-dominated sorting algorithm was then used to merge the dominance trees in Figure 7. The process is illustrated in Figure 9. It can be seen that only four comparisons are needed in the improved algorithm. The redundant comparison between Nodes 3 and 4 is eliminated by preserving their relationship.

```
Function GetDominanceTree(NodeList, Size)
    If Size > 1 Then
      LeftTree ← GetDominanceTree(NodeList, Size/2)
      RightTree ← GetDominanceTree(NodeList + Size/2, Size − Size/2)
    Else
      Return NodeList[0]
    End If
    Return MergeDominanceTrees(LeftTree, RightTree)
End Function

Function MergeDominanceTrees(LeftTree, RightTree)
    LeftNode ← LeftTree.Root
    RightNode ← RightTree.Root
    While LeftNode ≠ NULL and RightNode ≠ NULL
        If LeftNode ≺ RightNode Then
          If RightDelayedInsertionList ≠ NULL Then
              MergeDominanceTrees(RightNode.DominatedChildren,
                                  RightDelayedInsertionList)
          End If
          TempNode ← RightTree
          RightNode ← RightNode.NextSiblingNode
          RightTree.Remove(TempNode)
          MergeDominanceTrees(LeftNode.DominatedChildren, TempNode)
        Else If RightNode ≺ LeftNode Then
            If LeftDelayedInsertionList ≠ NULL Then
                MergeDominanceTrees(LeftNode.DominatedChildren,
                                    LeftDelayedInsertionList)
            End If
            TempNode ← LeftTree
            LeftNode ← LeftNode.NextSiblingNode
            LeftTree.Remove(TempNode)
            MergeDominanceTrees(RightNode.DominatedChildren, TempNode)
        Else
            If RightDelayedInsertionList ≠ NULL Then
                MergeDominanceTrees(RightNode.DominatedChildren,
                                    RightDelayedInsertionList)
            End If
            RightNode ← RightNode.NextSiblingNode
            If RightNode = NULL Then
              RightNode ← RightTree.Root
              LeftNode ← LeftNode.NextSiblingNode
            End If
        End If
    End While
    LeftNode ← LeftTree.Root
    RightNode ← RightTree.Root
    While RightNode ≠ NULL
        LeftNode.AddSiblingNode(RightNode)
        RightNode ← RightNode.NextSiblingNode
    End While
    Return LeftTree
End Function
```

Figure 8: The new non-dominated sorting algorithm with delayed insertion.

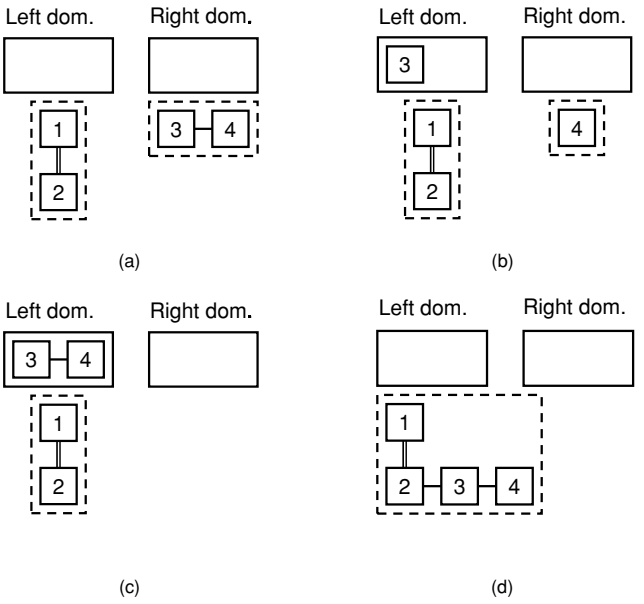


Figure 9: Merging two dominance trees with delayed insertion. (a) Before merge; (b) Node 3 is dominated by Node 1 and inserted into left domination list (Left dom.); (c) Node 4 is dominated by Node 1 and inserted into left domination list; and (d) the left dominated list is merged with Node 2.

3.4 Time Complexity Analysis of the New Algorithm

As aforementioned, the new algorithm improves the efficiency of non-dominated sorting by reducing redundant comparisons, which only exist when there are dominated solutions. In the worst case, when all of the solutions are non-dominant, full comparisons are required and there will not be any redundancy. In this worst-case situation, the time complexity of the new algorithm becomes $O(MN^2)$, which is true for all non-dominated sorting algorithms. The reason is that all pairs of solutions need to be compared before the non-dominated relationships can be fully determined for the entire population. The number of dominated solutions is largely related to the number of objectives; that is, the more the objectives, the less likely a solution dominates another and thus the fewer the dominated solutions. The lower-bound time complexity of the new algorithm is the same as that of the divide-and-conquer algorithm, which is $O(MN \log N)$. When the number of objectives is small and there exist certain number of dominated solutions, the new algorithm is expected to be far away from the worst case and to be close to the lower-bound. This is similar to the case of the Quick-sort algorithm, which has a worst-case time complexity of $O(N^2)$ and an average complexity of $O(N \log N)$.

Although the number of dominated solutions usually decreases with the increase on the number of objectives, there is no quantitative relationship between the number of objectives and the number of dominated solutions. We now adopt a statistical approach to perform quantitative analysis on the time complexity of the new algorithm.

For any two solutions x and y in the population, let p be the probability that either $x \prec y$ or $y \prec x$ and let $q = 1 - p$. We now introduce the following assumption: the cardinalities of the fronts form a geometric progression with parameter $\alpha(\alpha < 1)$, that

is, if the number of solutions in Front 1 is A, there are αA, $\alpha^2 A$, $\alpha^3 A$, ... solutions in Fronts 2, 3, 4, and so on.

According to this assumption, for a given population of N solutions, the number of solutions in Front 1 is approximately βN, where $\beta = (1-\alpha)/(1-\alpha^n) \approx 1-\alpha$. For the same reason, the number of solutions in Front 2 is approximately $\beta N'$ where N' is the total number of solutions in all fronts except for Front 1.

For the *MergeDominanceTrees* routine given in Figure 8, its average time complexity can be described by the following recurrence

$$T(2n) = q^{\beta n}\left[T(2n-\beta n) + O\left(\beta^2 n^2\right)\right] + \left(1-q^{\beta n}\right)\left[T(2n-\beta n-1) + O(\beta n) + \frac{1}{p}\right] \quad (2)$$

where n is the size of both left and right trees. The rationales behind Equation (2) are that the sizes of the first fronts in both trees are β, and that there are two possible cases when comparing solutions from the first fronts of both trees:

1. With a probability of $q^{\beta n}$, we will not see a single dominance between the selected solutions from the two trees before we finish the *while* loop (needs $O(\beta^2 n^2)$ steps) and go to *MergeDominanceTrees* with trees without Front 1 as inputs. The input size for the next round of recurrence becomes $2n - \beta n$;
2. With a probability of $1 - q^{\beta n}$, we will see a dominance between two solutions. This leads to a recursive call to *MergeDominanceTrees* with a smaller (left or right) tree of size $n - \beta n - 1$. The number of comparisons before we see the first dominance follows a geometric distribution with parameter p and the expectation of this distribution is $1/p$. Furthermore, we need to execute the merge on the bottom lines of *MergeDominanceTrees*, which takes $O(\beta n)$ time.

Since the probability $q^{\beta n}$ is an exponential function of n, which is asymptotically smaller than the time complexity of $O(\beta^2 n^2)$, the first item in Equation (2) can be ignored and Equation (2) becomes

$$t(2n) = \left(1-q^{\beta n}\right)\left[t(2n-\beta n-1) + O(\beta n) + \frac{1}{p}\right] \approx t(2n-\beta n-1) + O(\beta n) + \frac{1}{p} \quad (3)$$

On the right-hand side of Equation (3), the sum of $O(\beta n)$ for all iterations is $O(n)$. The third item $1/p$ is counted once at each iteration; and there are $O(\log n)$ iterations according to our assumption. Therefore, the recurrence in Equation (3) can be solved as

$$t(n) = O(n) + O(\log n)\frac{1}{p} \quad (4)$$

The recurrence function for the entire divide-and-conquer algorithm is thus

$$T(N) = 2T\left(\frac{N}{2}\right) + t(N) \quad (5)$$

According to the second case of the Master's Theorem (Cormen et al., 2001), we can

solve the recurrence function in Equation (5) as

$$T(N) = O\left(\frac{1}{p} N \log N\right) \tag{6}$$

In Equation (6), the cost to determine the dominant relationship between any two solutions is assumed to be one comparison. For M objectives, it would require in the worst case M comparisons between two solutions. Since M will appear in $O(\beta n)$ of Equation (3) and in $O(n)$ of Equation (4), we can modify Equation (6) to incorporate M in the time complexity as

$$T(N) = O\left(\frac{1}{p} MN \log N\right) \tag{7}$$

In the above analysis, $1/p$ has been treated as a constant, which is related to M, N, the values of objectives of all solutions, and the distribution of the solution. Since $1/p$ cannot be quantified and due to the assumption made in this analysis, the time complexity given by Equation (7) should not be treated as a strictly quantitative solution. However, Equation (7) does show that when $1/p$ is small or close to 1, the time complexity of the proposed algorithm is close to $O(MN \log N)$. For example, when M is equal to 2, $1/p$ will be approximately 2^{M-1} for a truly random population and $T(N)$ will be $O(MN \log N)$. When the number of objectives increases, it becomes very difficult to determine $1/p$; however, $T(N)$ will always be bounded by $O(MN^2)$.

In the next section, we will experimentally evaluate the performance of the new algorithm and compare it with Deb's as well as Jensen's algorithms.

4 Performance Evaluation of Non-dominated Sorting Algorithms

A direct measurement of the performance of a non-dominated sorting algorithm is to count the number of solution comparisons for a given population size. However, different algorithms have different overheads that may offset the overall efficiencies. Therefore, the total processing time of an algorithm also needs to be examined to obtain the real performance. Because the dominance among solutions varies from one population to another as well as from problem to problem, we used randomly generated solutions for the populations used in our comparisons. The random generator used in this study is the MT random generator (Matsumoto and Nishimura, 1998).

In this section, we first compare the new algorithm with Deb's algorithm to see how the former reduces the number of comparisons among solutions for different numbers of objective functions and population sizes. We then compare the total processing times of the two algorithms for these populations. Note that we do not include the time for fitness evaluation in this study. Finally we compare the processing times of the new algorithm with Jensen's and Deb's algorithms for different numbers of objectives and population sizes.

We implemented the new non-dominated sorting algorithm using the C++ programming language. We also implemented Deb's algorithm in C++ so that it was easy to use the same populations for comparing both algorithms. Since a linked list data structure was used in our implementation of Deb's algorithm, we also reduced its space complexity to $O(MN)$ by removing the dominance array. We obtained the C++ implementation of Jensen's algorithm from the Web site (Jensen, 2008). In order to fairly and

Table 5: Number of Comparisons by the New Algorithm

Population size	Number of objectives						
	2	3	4	5	6	7	8
100	1,029	1,725	2,360	2,894	3,327	3,758	4,130
200	2,884	5,531	8,080	10,240	12,042	13,711	15,312
300	5,241	10,880	16,579	21,281	25,482	29,246	32,763
400	7,997	17,673	27,555	35,966	43,363	49,888	56,175
500	11,067	25,597	40,843	54,008	65,383	75,539	85,296
600	14,465	34,806	56,620	75,162	91,182	106,134	119,905
700	18,133	45,131	74,360	99,531	121,398	141,243	159,961
800	22,052	56,391	93,935	126,794	155,466	181,090	205,506
900	26,134	69,106	116,561	157,697	192,549	225,858	255,926
1,000	30,537	82,095	140,462	190,699	234,095	274,492	311,568
1,500	55,156	162,893	288,949	401,851	494,965	582,709	665,653
2,000	84,017	264,239	483,217	681,678	845,036	995,070	1,140,814
2,500	116,510	386,304	723,348	1,028,637	1,283,479	1,507,914	1,732,510
3,000	152,274	525,249	1,001,560	1,439,041	1,800,746	2,119,951	2,439,385
3,500	191,071	682,395	1,318,348	1,914,981	2,403,096	2,834,552	3,252,353
4,000	232,191	854,760	1,674,172	2,451,167	3,083,480	3,635,253	4,185,300
4,500	276,048	1,044,468	2,066,237	3,042,405	3,850,668	4,531,065	5,213,377
5,000	322,262	1,253,064	2,501,507	3,697,384	4,686,377	5,526,466	6,359,292

accurately evaluate the processing times of the three algorithms, we compiled all of the codes with the same compiler and compiling options, and ran the programs on a single Pentium IV 3.06 GHz processor.

4.1 Evaluation of the Number of Comparisons

We compared the new algorithm with Deb's algorithm using seven populations with two to eight objectives, respectively. For a given number of objectives, we evaluated both algorithms using populations of sizes from 100 to 900 with an increment of 100, and from 1,000 to 5,000 with an increment of 500.

Due to the random nature of populations used in this study, we evaluated both algorithms using the average numbers of comparisons of multiple runs. For a given population size and number of objectives, we randomly generated 1,000 populations, and obtained the average numbers of comparisons of both algorithms for the 1,000 runs. The results are given in Tables 5 and 6 for the new and Deb's algorithms, respectively. The non-dominated fronts generated by both algorithms for each population were compared and shown to be identical.

We define the speedup of the new algorithm to be the ratio of the number of comparisons of Deb's algorithm to that of the new algorithm. By this definition, a speedup value larger than one indicates a performance improvement. The speedup curves for different numbers of objectives and population sizes are plotted in Figure 10.

We can see from Figure 10 that the new algorithm becomes more efficient when the population size increases. For example, for two objectives, the proposed algorithm has a speedup of 4.8 for a population size of 500 and a speedup of 11.3 for a population size of 5,000. This is also true for three to eight objectives, though the speedup increases are smaller than that for two objectives.

Table 6: Number of Comparisons by Deb's Algorithm

Population size	Number of objectives						
	2	3	4	5	6	7	8
100	2,560	2,537	2,731	3,009	3,357	3,761	4,131
200	9,528	9,440	10,178	11,164	12,272	13,716	15,336
300	20,543	20,242	22,216	24,051	26,317	29,405	32,755
400	35,336	34,788	38,773	42,010	45,845	50,387	56,299
500	53,042	53,352	59,326	64,995	70,089	76,482	85,313
600	74,323	75,490	84,463	91,509	98,817	107,974	120,043
700	99,425	100,301	113,216	123,884	133,215	144,680	160,183
800	126,702	129,820	146,639	160,167	172,514	186,466	205,669
900	157,457	161,626	185,383	203,067	216,380	233,751	256,384
1,000	191,295	196,566	225,840	249,479	264,555	286,528	312,751
1,500	404,618	421,811	493,651	548,757	585,041	623,041	675,076
2,000	688,799	725,581	858,121	965,277	1,031,309	1,086,640	1,169,142
2,500	1,031,096	1,092,279	1,315,340	1,494,389	1,602,281	1,677,616	1,793,367
3,000	1,433,723	1,535,311	1,859,812	2,127,181	2,293,666	2,397,675	2,541,498
3,500	1,898,340	2,051,293	2,497,535	2,874,826	3,107,047	3,246,844	3,426,355
4,000	2,428,687	2,623,749	3,233,661	3,738,864	4,036,712	4,228,261	4,444,284
4,500	3,033,176	3,238,717	4,018,232	4,713,659	5,128,822	5,338,033	5,577,683
5,000	3,656,746	3,953,571	4,912,621	5,771,384	6,282,697	6,574,836	6,853,694

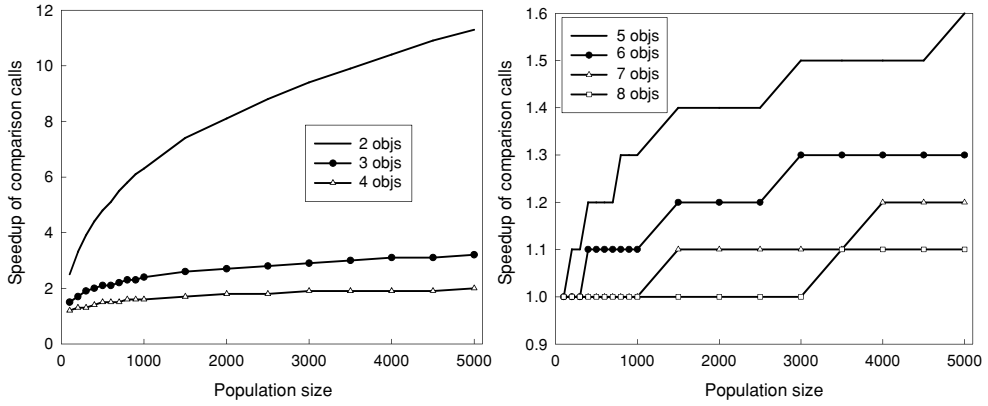


Figure 10: Speedup of the new algorithm on the number of comparisons.

For a given population size, the speedup of the new algorithm decreases as the number of objectives increases. For example, the speedup with a population size of 1,000 is 6.3 for two objectives and is 1.0 for eight objectives. This is true for all population sizes.

We also observed that the new algorithm worked significantly better for two objectives than for three or more objectives. The reason is that the new algorithm takes advantage of the number of dominated solutions, which are significantly reduced when the number of objectives increases. This can be demonstrated by the change of number of non-dominated fronts with the increase on the number of objectives. The new

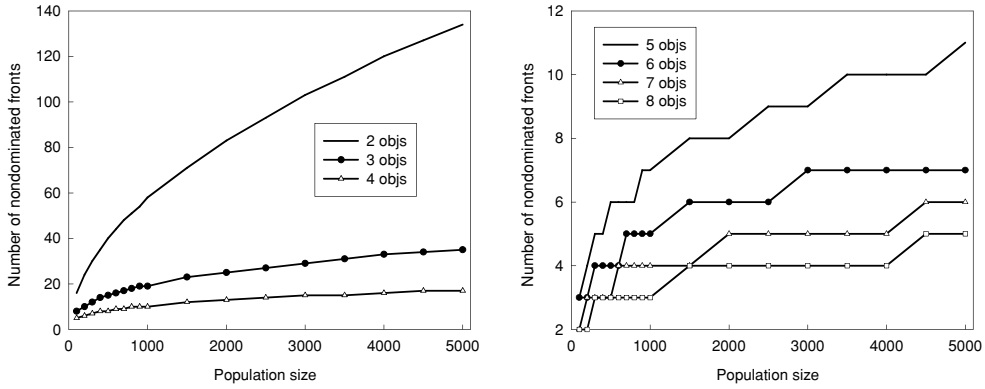


Figure 11: Change of non-dominated fronts with population size and number of objectives.

algorithm is more efficient than Deb's algorithm for large numbers of dominated solutions and non-dominated fronts, which is true when the number of objectives is small. We illustrate this change in Figure 11 for different population sizes used in the above comparisons.

Comparing Figures 10 and 11, we found that the speedup of the new algorithm followed the same trend as the change on the number of non-dominated fronts; that is, when the population size is increased and the number of objectives is decreased, the speedup is increased. Based on evaluations of the number of comparisons, the new algorithm has fewer comparisons than Deb's algorithm for six or less objectives. The new algorithm has no special advantage if the population size is smaller than 1,500 and 3,500 for seven and eight objectives, respectively. Even with a population of 5,000, the speedup for eight objectives is only 1.1, which does not show a significant improvement.

We now use a statistical method to determine if there are significant differences between the average numbers of comparisons of the two algorithms. The t-statistical test is adopted because the means and variances of both algorithms are unknown (Montgomery, 2001). Let μ_1 and μ_2 be the true means of the number of comparisons of the new and Deb's algorithms, respectively. We made the following two hypotheses

$$\begin{aligned} H_0&: \mu_1 = \mu_2 \\ H_1&: \mu_2 > \mu_1 \end{aligned} \tag{8}$$

where H_0, if true, indicates that there is no significant difference between the two algorithms based on the means of numbers of comparisons; and H_1, if true, indicates that the new algorithm has significantly fewer number of comparisons than the old algorithm. The estimated variances of the means of both algorithms are calculated by

$$S^2 = \frac{1}{n-1}\left[\sum_{i=1}^{n}(y_i - \bar{y})^2\right] \tag{9}$$

Table 7: Standard Deviations of Number of Comparisons by the New Algorithm

Population size	Number of objectives						
	2	3	4	5	6	7	8
100	1	4	6	1	1	5	4
200	2	18	4	19	12	6	34
300	1	17	9	30	15	19	30
400	5	10	82	116	9	73	30
500	29	81	17	58	111	30	26
600	24	69	49	37	70	117	30
700	25	61	65	133	158	190	138
900	12	56	193	1	291	359	110
1000	57	51	34	644	256	138	733
1500	123	170	113	875	954	342	411
2000	33	306	1372	1664	1354	99	59
2500	154	63	1572	759	350	464	50
3000	176	228	250	838	1081	3495	2106
3500	428	136	59	688	2284	3624	1780
4000	126	174	2037	6837	3379	9293	4056
5000	141	2852	940	5063	4919	6703	4614

Table 8: Standard Deviations of Number of Comparisons by Deb's Algorithm

Population size	Number of objectives						
	2	3	4	5	6	7	8
100	49	22	17	2	2	6	4
200	212	141	62	10	5	6	34
300	485	279	188	58	12	24	25
400	870	552	437	75	87	57	26
500	1,299	797	602	289	260	60	26
600	1,918	1,356	831	554	172	59	25
700	2,547	1,685	1,175	636	116	91	122
800	3,319	2,170	1,732	1,188	698	20	133
900	4,167	2,983	2,370	1,437	463	608	124
1,000	5,030	3,673	2,735	2,504	1,220	519	696
1,500	11,180	8,022	6,590	3,773	3,804	934	113
2,000	19,168	14,902	13,234	10,636	7,248	2,798	955
2,500	29,090	22,399	17,158	13,977	9,736	4,905	1,975
3,000	40,719	31,729	27,404	22,610	16,676	12,282	1,125
3,500	53,588	43,174	37,249	31,057	24,556	9,420	3,725
4,000	69,368	56,009	50,363	39,237	32,572	9,780	9,562
4,500	87,439	69,597	63,796	59,713	37,060	16,238	15,582
5,000	105,639	88,293	77,224	60,555	45,586	39,872	11,029

where n is the sample size or number of runs (1,000 in this study), y_i is the number of comparisons in the i-th run, $\bar{y}$ is the estimated sample mean of the number of comparisons, and S is the standard deviation. The standard deviations of both algorithms are calculated for each combination of a number of objectives and a population size; the results are given in Tables 7 and 8.

Table 9: Values of t_0 for the Significance Tests

Population size	Number of objectives						
	2	3	4	5	6	7	8
100	987.84	1148.34	650.78	1626.35	424.26	12.15	5.59
200	991.00	869.63	1067.85	1360.89	559.48	18.63	15.78
300	997.71	1059.16	947.09	1341.44	1374.59	164.26	−6.48
400	993.70	980.32	797.85	1383.64	897.37	170.38	98.77
500	1021.58	1095.60	970.52	1178.71	526.41	444.53	14.62
600	986.82	947.55	1057.70	931.03	1300.17	444.05	111.75
700	1009.25	1034.71	1044.63	1184.85	1321.02	1171.67	42.74
800	997.08	1067.41	961.59	882.82	753.30	889.84	26.89
900	996.59	980.63	915.26	998.42	1378.07	353.50	87.38
1000	1010.59	985.45	987.09	718.93	772.70	708.73	37.01
1500	988.40	1020.43	982.14	1199.44	726.31	1282.28	699.07
2000	997.75	978.78	891.06	833.06	798.88	1034.27	936.24
2500	994.20	996.69	1086.51	1052.21	1034.81	1089.21	974.10
3000	995.18	1006.65	990.34	961.79	932.77	687.76	1352.42
3500	1007.44	1002.64	1001.08	977.09	902.64	1291.76	1332.81
4000	1001.31	998.77	979.00	1037.05	922.92	1412.23	847.86
4500	997.13	997.00	967.08	879.32	1086.13	1363.96	715.50
5000	998.17	966.70	987.26	1079.31	1100.97	819.96	1307.74

The t-value on the difference of the two means is calculated by

$$t_0 = (\bar{y}_2 - \bar{y}_1)\Big/\sqrt{S_1^2/n_1 + S_2^2/n_2} \tag{10}$$

where n_1, S_1^2, and $\bar{y}_1$ are the sample size (1,000), estimated variance, and estimated sample mean, respectively, for the new algorithm; and n_2, S_2^2, and $\bar{y}_2$ are the corresponding values for Deb's algorithm. The values of t_0 calculated by Equation (10) are given in Table 9.

The critical value of t to determine the significance of the two hypotheses in Equation (8) is given by $t_{\alpha,v}$ in which α is the significance level and v is the equivalent sample size calculated by

$$v = \left(\frac{S_1^2}{n_1} + \frac{S_2^2}{n_2}\right)\Big/\left[\frac{\left(S_1^2/n_1\right)^2}{n_1 - 1} + \frac{\left(S_2^2/n_2\right)^2}{n_2 - 1}\right] \tag{11}$$

If $t_0 > t_{\alpha,v}$, we will reject H_0 and accept H_1; this indicates that the mean μ_1 of the new algorithm is significantly smaller than the mean μ_2 of Deb's algorithm. The values of v were calculated for all of the objectives using Equation (11) and they are all greater than 999. Therefore, the values of $t_{\alpha,v}$ at significance levels of $\alpha = 0.05, 0.01, 0.005$, and 0.001 are 1.645, 2.326, 3.09, and 3.291, respectively (Montgomery, 2001).

Comparing the values of t_0 in Table 9 with the critical values of t, we found that hypothesis H_0 should be rejected and hypothesis H_1 should be accepted for almost all the combinations, except for the case with eight objectives and a population size of 300. For populations with six or less objectives, the t_0 values in Table 9 are much larger than the critical t-value corresponding to $\alpha = 0.001$, which is 3.291. This indicates that the

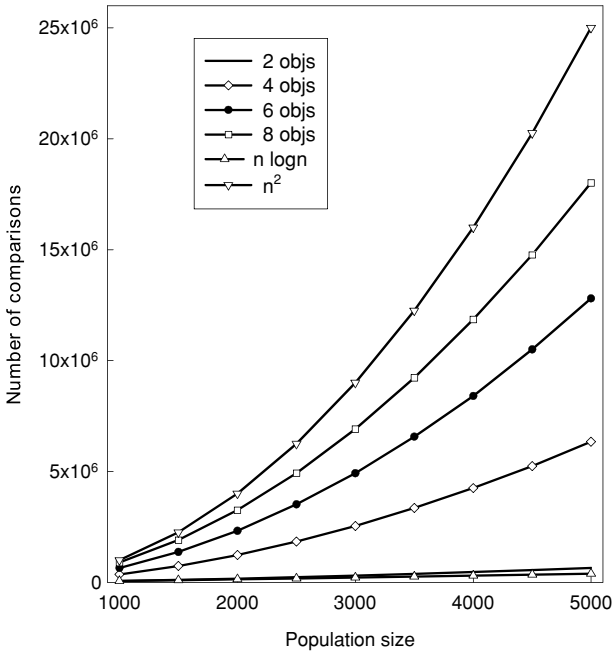


Figure 12: Time complexity of the new algorithm by experimental data.

new algorithm has less number of comparisons than Deb's algorithm at a significance level much higher than $\alpha = 0.001$. This is also true for seven and eight objectives, if the population sizes are larger than 200 and 500, respectively. The above statistical analysis indicates that the new algorithm has in general fewer number of comparisons than Deb's algorithm.

We also compared the experimental data (or the number of comparisons) with the theoretical bounds given in Section 3.4 (Figure 12). It can be seen that experimental data fall in between the two theoretical bounds, $O(MN \log N)$ and $O(MN^2)$, and the numbers of comparisons for two objectives are very close to the lower bound $O(MN \log N)$. With the increase of the number of objectives, the time complexity of the new algorithm approaches asymptotically to the upper bound $O(MN^2)$, which could result in no speedup of the new algorithm over Deb's as aforementioned.

Experiments were also performed for 9 to 20 objective functions. The results show that there is no speedup on the number of comparisons after 10 or more objectives even at a population size of 5,000.

4.2 Evaluation of Total Processing Time

The evaluation in the previous section showed the reductions that the new algorithm had over Deb's algorithm on the number of comparisons for generating the non-dominated fronts. To get the true performance of the new algorithm, we also need to evaluate the total processing time that includes the overhead of an algorithm. Note that the time for fitness evaluation, which can be fairly large or fairly small depending on applications, is excluded in the total processing time in this study.

Table 10: Average Total Processing Time (sec) of 1,000 Runs by the Two Algorithms

Population size	Number of objectives									
	New algorithm					Deb's algorithm				
	2	3	4	6	8	2	3	4	6	8
100	0.1	0.1	0.1	0.1	0.2	0.2	0.1	0.2	0.2	0.2
200	0.2	0.3	0.3	0.5	0.7	0.5	0.5	0.5	0.7	0.8
300	0.2	0.5	0.8	1.1	1.4	1.0	1.0	1.0	1.4	1.8
400	0.5	0.9	1.2	1.9	2.5	1.7	1.6	2.0	2.3	2.8
500	0.5	1.1	1.8	2.6	3.5	3.2	2.4	2.8	3.5	4.4
600	0.7	1.4	2.3	3.6	5.0	3.6	3.4	4.1	5.3	6.2
700	0.9	1.9	3.0	4.8	6.6	4.5	4.5	5.5	6.7	8.2
800	1.1	2.5	3.7	6.4	8.8	5.8	5.6	7.0	8.5	10.5
900	1.3	2.9	4.8	8.4	10.6	7.1	7.2	10.5	13.0	
1000	1.5	3.2	5.5	9.6	13.1	8.5	8.6	10.4	13.4	15.9
1500	2.5	6.2	11.3	20.0	27.6	17.8	18.7	23.4	29.7	35.2
2000	4.2	10.3	19.5	34.8	46.6	31.2	32.8	43.0	55.6	64.0
2500	4.9	15.3	29.6	52.4	72.2	48.3	52.3	70.0	101.4	117.1
3000	7.1	21.2	41.5	77.2	104.4	79.2	82.6	110.5	169.5	217.2
3500	8.9	27.7	55.2	102.7	144.1	110.2	122.8	176.6	275.1	343.1
4000	11.6	35.7	71.3	138.2	188.9	165.3	174.3	256.5	482.4	531.0
4500	13.2	44.3	88.7	166.7	247.9	209.2	258.2	392.5	585.3	774.8
5000	16.3	53.3	107.3	217.8	313.0	301.5	340.1	512.6	806.7	1059.8

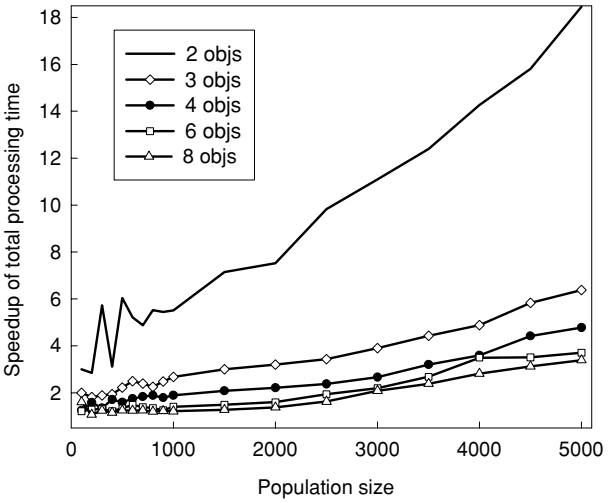


Figure 13: Speedup of the new algorithm on total processing time.

We recorded the total processing times for obtaining the non-dominated fronts by both algorithms in Section 4.1. Since we used 1,000 populations for a given number of objectives and population size, we compare both algorithms on the average processing time of 1,000 runs. The total processing times for 2, 3, 4, 6, and 8 objectives are summarized in Table 10. The speedup of the new algorithm is plotted in Figure 13.

The speedup on the total processing time follows the same trend as that on the number of comparisons; that is, the speedup decreases as the number of objectives increases and increases as the population size increases. However, the speedup on the total processing time is larger than that on the number of comparisons for the same number of objectives and the same population size. For example, the speedup for total processing time is 1.37 for eight objectives and a population size of 2,000, where it is 1.02 for the number of comparisons. This is also true for other population sizes and numbers of objectives.

Since the total processing time includes the overhead, the above observations indicate that the new algorithm has smaller overhead and is thus more efficient than Deb's algorithm. It also indicates that the dominance tree structure combined with the divide-and-conquer mechanism is very efficient in obtaining non-dominated fronts. For example, for a population size of 5,000 with eight objectives, the new algorithm reduces the number of comparisons from 6,859,707 to 6,355,722, which has a speedup of 1.08 or a 7.3% reduction. However, the speedup on the total processing time is 3.39, which is equivalent to a 70.5% reduction. Based on the evaluation of the total processing time, the new algorithm shows a significant performance improvement over Deb's algorithm.

4.3 Comparison of Total Processing Time with Jensen's Algorithm

We have shown in Section 2.2 that Jensen's algorithm did not generate the same non-dominated fronts as Deb's algorithm when there are duplicated solutions. We also noticed that Jensen (2003) reported better speedups on the total processing times than the results of this study using Jensen's algorithm. For example, for a population size of 1,000, Jensen reported speedups of 9, 3.8, and 2 for three, five, and eight objectives, respectively; while we found that it had speedups of 2.67, 1.58, and 1.21 in this study. To obtain the true performance of the new algorithm, we compared the three algorithms at the same conditions.

For Jensen's algorithm, we used its C++ implementation downloaded from the Web site (Jensen, 2008). For Deb's algorithm, we implemented it using C++ with improved space efficiency by using the linked list data structure. We then compiled the codes of the three algorithms using the same compiler and compiling options on the same computer. The three algorithms were compared using two population sizes (2,000 and 4,000) and seven different numbers of objectives (from two to eight). For each combination of a certain number of objectives and a certain population size, we randomly generated 100 populations using the DTLZ1 benchmark (Deb et al., 2002a). These populations were then used by all the three algorithms. The recorded total processing time was solely for generating the non-dominated fronts and did not include the time for input/output operations. The sums of the total processing time for the three algorithms are given in Table 11, which also gives the sizes of each of the non-dominated fronts.

The results in Table 11 show that Jensen's algorithm is no better than Deb's algorithm for more than six objectives with a population size of 2,000. When the population size is increased to 4,000, Jensen's algorithm is better than Deb's for up to eight objectives. The new algorithm always has a better performance than Deb's algorithm for all the number of objectives and population sizes. This is consistent with the results in Section 4.2. The new algorithm is better than Jensen's algorithms with four or more objectives for a population size of 2,000. For a population size of 4,000, the new algorithm is better

Table 11: Total Processing Times (sec) of 100 Runs by the Three Algorithms

Number of objectives	Population size = 2,000				Population size = 4,000			
	# fronts	Deb's	Jensen's	New	# fronts	Deb's	Jensen's	New
2	50	3.62	0.07	0.50	71	26.28	0.22	1.38
3	18	4.64	0.87	1.45	22	32.67	2.10	4.59
4	12	5.18	2.31	2.06	14	40.91	5.85	7.17
5	9	5.67	3.76	2.60	11	45.51	10.79	9.13
6	8	5.71	5.54	2.89	10	47.15	16.34	10.56
7	8	6.18	7.53	3.06	9	53.60	22.74	12.01
8	7	6.05	9.42	3.38	8	52.95	28.93	12.32

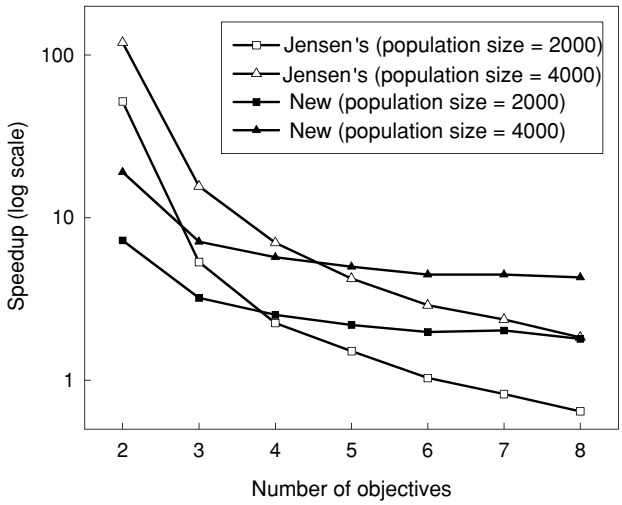


Figure 14: Speedup (log scale) of Jensen's and the new algorithms on total processing time.

than Jensen's when there are five or more objectives. The speedups of Jensen's and the new algorithms are plotted in log scale in Figure 14.

The processing time of Jensen's algorithm grows faster than that of the new algorithm. This also means that the new algorithm is less sensitive to the increase on the number of objectives than Jensen's algorithm. Jensen reported a different performance when comparing his algorithm with Deb's algorithm. This could be due to differences in algorithm implementation, code efficiency, and compiling options.

As mentioned above, Jensen's algorithm sorts duplicated solutions in the same fronts, which is not desirable for evolutionary algorithms. Removing duplicated solutions within a front is nontrivial and doing so will definitely increase the time complexity of Jensen's algorithm. Although removing duplicated solutions from all of the fronts of the population is not the focus of this study, it can be easily implemented in the new algorithm because of the use of the dominance tree. When comparing two solutions for their dominance information, we can simply discard one of them if they are found to be duplicates. Since this happens during the merge process in the new algorithm, the modification of the current algorithm is straightforward.

Table 12: Benchmark Problems Used in this Study

Problem	Objective functions	Parameter setting
SCH	$f_1 = x^2$ $f_2 = (x-2)^2$	$x \in \lfloor -10^3, 10^3 \rfloor$
KUR	$f_1 = \sum_{i=1}^{n-1} \left[-10e^{(-0.2\sqrt{x_i^2 + x_{i+1}^2})} \right]$	$x_i \in [-5, 5],\ i = 1, 2, \ldots, n$
	$f_2 = \sum_{i=2}^{n} \left[\lvert x_i \rvert^a + 5\sin(x_i^b) \right]$	$n = 3$
ZDT2	$f_1 = x_1$	$x_i \in [0, 1],\ i = 1, 2, \ldots, n$
	$f_2 = g\left[1 - (x_1/g)^2\right]$	$n = 30$
	$g = 1 + \left(9\sum_{i=2}^{n} x_i\right) \Big/ (n-1)$	
ZDT3	$f_1 = x_1$	$x_i \in [0, 1],\ i = 1, 2, \ldots, n$
	$f_2 = g[1 - \sqrt{x_1/g} - (x_1/g)\sin(10\pi x_1)]$	$n = 30$
	$g = 1 + \left(9\sum_{i=2}^{n} x_i\right) \Big/ (n-1)$	
DTLZ7	$f_{m=1:M-1} = x_m$	$x_i \in [0, 1],\ i = 1, 2, \ldots, n$
	$f_2 = (1+g)M - \sum_{i=1}^{M-1} f_i\left[1 + \sin(3\pi f_i)\right]$	$n = k + M - 1$
	$g = 1 + \left(9\sum_{i=M}^{n} x_i\right) \Big/ k$	$k = 20$
		$M = 2, 3, \ldots, 10$

4.4 Performance Evaluation Using Benchmark Problems

We used five benchmark problems to evaluate the performance of the new algorithm during the evolutions in solving multi-objective optimization problems. The first four benchmarks, SCH (Schaffer, 1987), KUR (Kursawe, 1990), ZDT2, and ZDT3 (Zitzler et al., 2000), are bi-objective problems. The fifth benchmark is DTLZ7 (Deb et al., 2002a) that is scalable to more than two objectives. The test functions along with parameter settings are summarized in Table 12. All benchmark problems were run on a single Pentium M 2.0 GHz processor.

For the first four benchmarks, we used a population size of 200 and ran for 250 generations with tournament selection, binary crossover (crossover rate 0.9), and mutation at a 1% rate. For each of these benchmarks, 10 simulation runs were performed and the evolutionary histories were recorded. It was observed that the results (comparison history and total number of comparisons) for each benchmark are all similar. Consequently, the results of one simulation run are used for demonstration. Figures 15(a) to 15(d) show the evolution histories of the new and Deb's algorithm on the number of comparisons for SCH, KUR, ZDT2, and ZDT3, respectively.

The Pareto fronts obtained from the evolution histories in Figure 15 were found matching those in literature; this indicates that the four evolution processes were successful. The evolution histories in Figure 15 show that the number of comparisons by the new algorithm does increase as the number of dominated solutions is reduced. This observation is consistent with the finding in Section 4.1. The number of comparisons for both algorithms tends to become stable after certain numbers of generations,

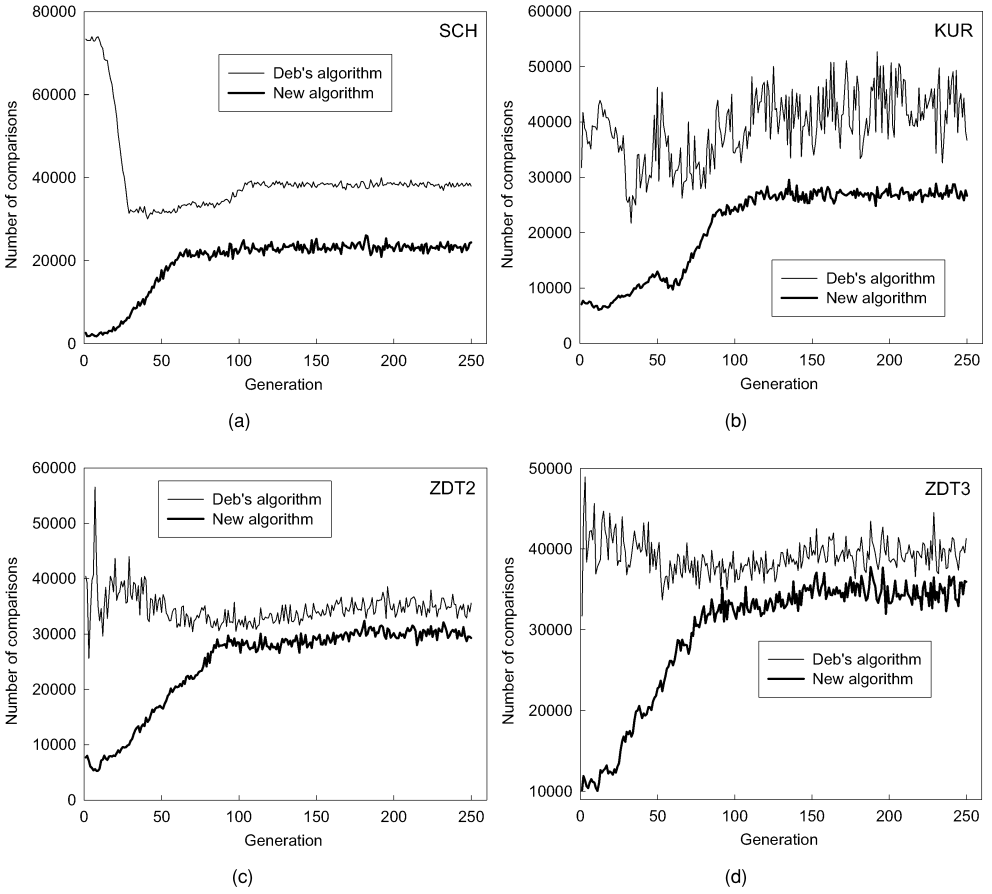


Figure 15: Comparison of the new algorithm and Deb's algorithm on benchmark problems. (a) SCH; (b) KUR; (c) ZDT2; and (d) ZDT3.

with the new algorithm having less comparison than Deb's algorithm. The difference in the number of comparisons exists even when the solutions approach the Pareto front. The reason is that the sorting population in NSGA-II is a combination of the parent and child populations, and that the child population contains dominated solutions. Although savings on the number of comparisons are not as significant as those in Section 4.1, the total processing time on non-dominated sorting by the new algorithm is much less than that of Deb's algorithm. For the total time on non-dominated sorting over 250 generations, the new algorithm has speedups of 4.6, 3.6, 3.6, and 3.1 for SCH, KUR, ZDT2, and ZDT3, respectively. These speedups are the average of 10 runs for each of the four benchmarks.

For the fifth benchmark problem (DTLZ7), we tested two to 10 objectives using a population of 200 solutions and the same setting of genetic operations as previously used. For each combination of a population size and number of objectives, 10 simulation runs were performed and the averages of the 10 runs were compared. Similar performance was obtained for the case of two objectives. The difference in the number

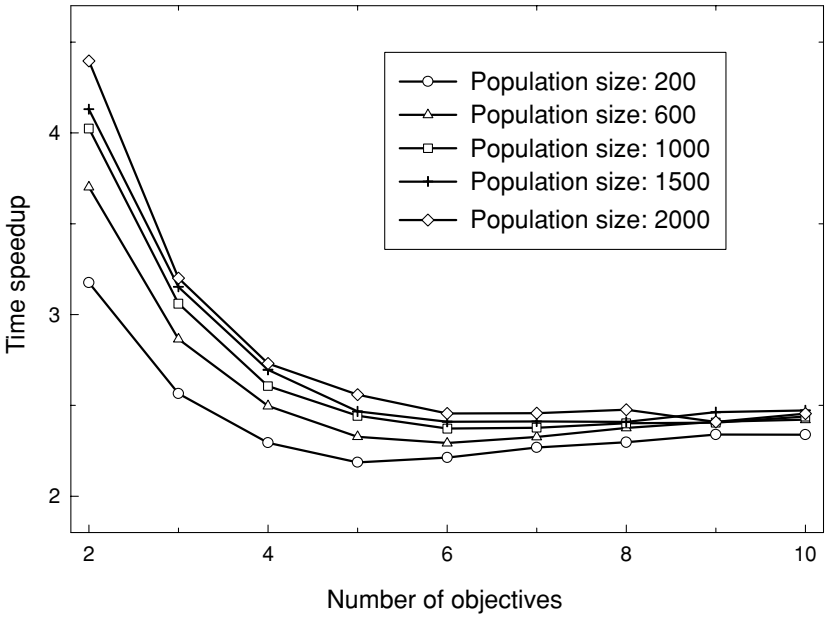


Figure 16: Speedups of the new algorithm on total time.

of comparisons becomes insignificant for three to 10 objectives after approximately 50 generations. Nevertheless, the new algorithm has speedups above two on total processing time over 250 generations for all of the test cases. We also tested for larger population sizes (600, 1000, 1500, and 2000) and the time speedups are all shown in Figure 16.

It can be seen from Figure 16 that the speedup increases as the increase of population sizes, and decreases as the increase of number of objectives. This observation is consistent with results in Section 4.2. This suggests that the new algorithm is more efficient for small number of objectives (two to five) and/or larger population sizes. For large number of objectives (e.g., six to 10), the increase of population size does not significantly increase the time speedup over the entire evolution process. This is not the same as performance evaluation in Section 4.2 using only initial populations. Nevertheless, the new algorithm saves more than 50% on total processing times to generate non-dominated fronts even for a large number of objective functions.

5 Conclusions

In this study, we developed a new non-dominated sorting algorithm that can be used in evolutionary algorithms, particularly the NSGA-II, to efficiently generate the non-dominated fronts. The new algorithm adopts the divide-and-conquer mechanism and uses a new data structure called the dominance tree to achieve both time and space efficiency. The comparisons of the new algorithm with the non-dominated sorting algorithm currently used in NSGA-II showed that the former always performed better for different numbers of objectives and population sizes. Although the reduction on the number of redundant comparisons by the new algorithm becomes less significant

when the number of objectives increases, the speedup of the new algorithm on the total processing time is still significant even for a large number of objectives. The non-dominated fronts generated by the new algorithm for different numbers of objectives and population sizes were also verified and shown to be identical to those generated by the algorithm of NSGA-II.

References

Coello, C. A. C. (1999). A comprehensive survey of evolutionary-based multi-objective optimization techniques. *Knowledge and Information Systems*, 1:269–308.

Coello, C. A. C. and Pulido, G. T. (2001). A micro-genetic algorithm for multi-objective optimization. In Zitzler, E., Deb, K., Thiele, L., Coello, C. A. C., and Corne, D. W., editors, *First International Conference on Evolutionary Multi-Criterion Optimization*, pages 126–140. Springer-Verlag, Berlin.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2001). *Introduction to Algorithms*, 2nd ed. McGraw-Hill, New York.

Corne, D. W., Knowles, J. D., and Oates, M. J. (2000). The pareto envelope-based selection algorithm for multi-objective optimization. In Schoenauer, M., Deb, K., Rudolph, G., Yao, X., Lutton, E., Merelo, J. J., and Schwefel, H., editors, *Parallel Problem Solving from Nature; PPSN VI Proceedings*, volume 1917 of *Lecture Notes in Computer Science*, pages 839–848.

Das, I. and Dennis, J. (1997). A closer look at drawbacks of minimizing weighted sums of objectives for pareto set generation in multicriteria optimization problems. *Structural Optimization*, 14:63–69.

Deb, K., Pratab, A., Agarwal, S., and Meyarivan, T. (2002a). A fast and elitist multi-objective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6:182–197.

Deb, K., Thiele, L., Laumanns, M., and Zitzler, E. (2002b). Scalable multi-objective optimization test problems. In Fogel, D., El-Sharkawi, M., Yao, X., Greenwood, G., Iba, H., Marrow, P., and Shackleton, M., editors, *Proceedings of IEEE Congress on Evolutionary Computation (CEC)*, volume 1, pages 825–830.

Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI.

Jensen, M. T. (2003). Reducing the run-time complexity of multi-objective EAs: The NSGA-II and other algorithms. *IEEE Transactions on Evolutionary Computation*, 7:502–515.

Jensen, M. T. (2008). Available at http://www.daimi.au.dk/ mjensen/research/nsgaiidownload.tar.gz.

Knowles, J. and Corne, D. (2000). Approximating the nondominated front using the pareto archived evolution strategy. *Evolutionary Computation*, 8:149–172.

Kung, H., Luccio, R., and Preparata, F. (1975). On finding the maxima of a set of vectors. *Journal of the Association for Computing Machinery*, 22:469–476.

Kursawe, F. (1990). A variant of evolution strategies for vector optimization. In Schwefel, H.-P. and Männer, R., editors, *Parallel Problem Solving from Nature*, pages 193–197. Springer-Verlag, Berlin.

Matsumoto, M. and Nishimura, T. (1998). Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8:3–30.

Montgomery, D. C. (2001). *Design and Analysis of Experiments*. John Wiley & Sons, New York.

Schaffer, J. D. (1987). Multiple objective optimization with vector evaluated genetic algorithms. In Grefensttete, J. J., editor, *Proceedings of the First International Conference on Genetic Algorithms*, pages 93–100. Lawrence Erlbaum, Hillsdale, NJ.

Srinivas, N. and Deb, K. (1995). Multi-objective function optimization using non-dominated sorting genetic algorithms. *Evolutionary Computation*, 2:221–248.

Zitzler, E., Deb, K., and Thiele, L. (2000). Comparison of multiobjective evolutionary algorithms: Empirical results. *Evolutionary Computation*, 8(2):173–195.

Zitzler, E. and Thiele, L. (1999). Multi-objective evolutionary algorithms: A comparative case study and the strength pareto approach. *IEEE Transactions on Evolutionary Computation*, 3:254–271.