Deductive Sort and Climbing Sort: New Methods for Non-Dominated Sorting

Kent McClymont km314@exeter.ac.uk
College of Engineering, Mathematics and Physical Sciences,
University of Exeter, EX4 4QJ, UK

Ed Keedwell E.C.Keedwell@exeter.ac.uk
College of Engineering, Mathematics and Physical Sciences,
University of Exeter, EX4 4QJ, UK

Abstract

In recent years an increasing number of real-world many-dimensional optimisation problems have been identified across the spectrum of research fields. Many popular evolutionary algorithms use non-dominance as a measure for selecting solutions for future generations. The process of sorting populations into non-dominated fronts is usually the controlling order of computational complexity and can be expensive for large populations or for a high number of objectives. This paper presents two novel methods for non-dominated sorting: deductive sort and climbing sort. The two new methods are compared to the fast non-dominated sort of NSGA-II and the non-dominated rank sort of the omni-optimizer. The results demonstrate the improved efficiencies of the deductive sort and the reductions in comparisons that can be made when applying inferred dominance relationships defined in this paper.

Keywords

Non-dominated sorting, Pareto optimal, Pareto front, genetic algorithms, evolution strategies.

1 Introduction

In recent years an increasing number of real-world many-dimensional optimisation problems have been identified across the spectrum of research fields. With the rapid increase in the number of recognised real-world multi-/many-objective optimisation problems (MOOPs), multi-/many-objective evolutionary algorithms (MOEAs) are experiencing a growth in interest and application. The ability of evolutionary algorithms (EAs) to optimise MOOPs is well documented (Coello Coello et al., 2007) and for many the MOEA is the algorithm of choice. As the number of objectives increases, the complexity of the non-dominated sorting function in MOEAs becomes a comparatively important factor in the complexity of the optimising algorithms and it should be expected that the impact of this function will continue to rise. Therefore it is important for MOEAs to minimise the computational complexity of this function with respect to the number of objectives to ensure durability.

A multitude of MOEAs have emerged since the first applications of genetic algorithms (GAs) to multi-objective optimisation, such as Schaffer's vector evaluated genetic algorithm (VEGA) (Schaffer, 1984), and later many-objective optimisation problems. The majority of these algorithms implement some form of non-dominated sorting and selection (a concept suggested by Goldberg, 1989, in order to combat some of the weaknesses of VEGA) which in most cases can be shown to be the controlling order of computational complexity. Well established algorithms such as NSGA-II (Deb et al., 2000) and the omni-optimizer (Deb and Tiwari, 2005) which use fast sorts still present high computational complexity as a result of the sort $O(MN^2)$ (where N is the magnitude of the population and M is the number of objectives). Efforts have been made to reduce the complexity arising from the sorting procedures in order to improve the algorithms' performance. Jensen (2003) and Yukish (2004) have both presented algorithms which reduce this complexity to $O(N(log N)^{M-1})$ for MOOPs, applicable to sorting sets of solutions not weakly dominated by any other. However, as discussed later, the application of these algorithms is limited to the definition of dominance that can in the case of some MOEAs, the omni-optimizer included, restrict their application.

This paper investigates the properties of Pareto optimality, dominance, and non-dominance and the possible inherent inferences that can be made based upon the nature of these relationships. Developing this theme, the paper reviews non-dominance in selection and archiving and considers the computational complexities of these operations. Finally, the paper presents two novel approaches to non-dominated sorting and Pareto front generation, known as climbing sort and deductive sort, that exploit natural efficiencies. A thorough theoretical comparison of these algorithms is conducted along with empirical evidence that demonstrates that the new approaches show reduced complexity with regard to population size and number of objectives in comparison with the fast non-dominated sort of NSGA-II (Deb et al., 2000) and non-dominated rank sort of the omni-optimizer (Deb and Tiwari, 2005).

2 Use of Pareto Sorting in Multi-/Many-Objective Evolutionary Algorithms

The algorithm VEGA, proposed by Schaffer (1984) and further developed in later work (Schaffer and Grefenstette, 1985), presented a novel approach to solving problems with conflicting objectives and initiated a series of algorithms focused on solving highly complex MOOPs. Goldberg addressed some of the drawbacks of VEGA and other MOEAs and suggested the use of non-dominated sorting and niching as methods to improve these algorithms (Goldberg, 1989).

Fonseca and Fleming were one of the first to implement a variant of non-dominated sorting as suggested by Goldberg (1989) in an algorithm called the multi-objective genetic algorithm (MOGA; Fonseca and Fleming, 1993) which was shortly followed by the niched pareto genetic algorithm (NPGA) presented by Horn et al. (1994). Both MOGA and NPGA demonstrated the efficacy of non-dominated sorting techniques (Zitzler et al., 2000) and were followed by a number of more advanced evolutionary algorithms (EAs) that further improved upon these early designs. Some of the most notable algorithms that followed are the genetic algorithms (GA) non-dominated sorting genetic algorithm (NSGA; Srinivas and Deb, 1994) and subsequently fast elitist non-dominated sorting genetic algorithm (NSGA-II; Deb et al., 2000). A collection of evolutionary strategies (ESs) also implemented a non-dominated selection approach, such as the Pareto archived evolution strategy (PAES; Knowles and Corne, 1999), strength Pareto evolutionary algorithm (SPEA; Zitzler and Thiele, 1999), the strength Pareto evolutionary algorithm 2 (SPEA2; Zitzler et al., 2002), and the Pareto envelope selection algorithm (PESA; Corne et al., 2000).

These two branches of EAs, GA and ES, are widely applied and proven in many cases to be effective algorithms for solving MOOPs (Coello Coello et al., 2007). Whilst the implementation of the non-dominated sorting differs between GA and ES, the effect of selection based upon non-dominated elitism can be influential in the convergence of populations toward the optimal Pareto front (Coello Coello, 1999; Zitzler et al., 2000). Typically, an ES deals with a special case of non-dominated sorting where a set of unsorted solutions is inserted into a presorted set. Additionally, the new set is often much smaller in magnitude than the presorted archive. As such, general Pareto sorting algorithms like those presented in this paper are not likely to perform as quickly as specialised algorithms. Everson et al. (2002) present a good example of a specialised data structure called Dominance Trees, designed for inserting new solutions into a presorted archive.

Although PAES, SPEA, and NSGA-II are considered successful algorithms, their methods to preserve a good distribution of solutions across the Pareto front have been shown to possibly prevent convergence and thus remove the guarantee that these algorithms will eventually converge on the Pareto-optimal front (Laumanns et al., 2002). The effects of selection and repopulation methods can be greatly affected by the employed definition of dominance and, as suggested by Laumanns et al. (2002), by changing the definition of dominance, the performance (population quality and convergence) can be significantly increased. Laumanns et al. (2002) introduced a method based upon a proposed ϵ-dominance for generating approximate Pareto fronts that ensured convergence and diversity of solutions in objective space.

More recent algorithms utilising non-dominated sorting such as ϵ-MOEA (Deb et al., 2003) and the omni-optimizer (Deb and Tiwari, 2005) use ϵ dominance. Both these algorithms further enforce ϵ dominance as a viable alternative to previous definitions. Knowles (2006) also makes use of ϵ dominance, using counts of dominance of one set over the other as an indicator for performance. Clearly, it cannot be said with any certainty which approach future algorithms will implement and so new sorting methods must accommodate as many definitions of dominance as possible.

3 Review of Previous Work

Jensen (2003) and Yukish (2004) both independently present non-dominated sorting algorithms that extend Kung's divide and conquer vector sorting algorithm (Kung et al., 1975) which identifies only the non-dominated vectors from a set of vectors. The algorithm proposed by Jensen reduced the existing $O(MN^2)$ to $O(Nlog^{M-1}N)$ for three objectives and above (Jensen, 2003); where N is the number of solutions and M the number of objectives. For two and three objectives, Kung's original algorithm offers sorting at $O(NlogN)$ and $O(N(logN)^{M-2})$ for objectives greater than three (Kung et al., 1975). Following Kung's notation, Jensen's algorithm, which produces fronts as well as sorting, can be rewritten as $O(N(logN)^{M-1})$ for objectives greater than two; a polylogarithmic relationship with respect to N. Whilst for low dimensions this presents an improvement over previous polynomial algorithms, the complexity increases exponentially with respect to M, and with higher M the efficiencies are diminished.

In addition, the algorithms presented by Jensen and Yukish both assume a rigid ordering of solutions and thus are applicable only to weak dominance—where a solution dominates another if it is better or equal in all objectives and better in at least one objective. Any alteration to the definition of dominance that breaks this strict ordering, such as strong (where a solution only dominates another if it is better for all objectives),

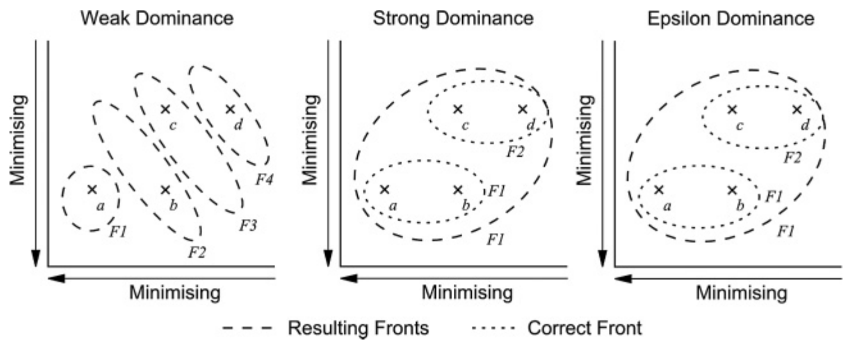

Figure 1: Front allocation based on strict ordering of solutions.

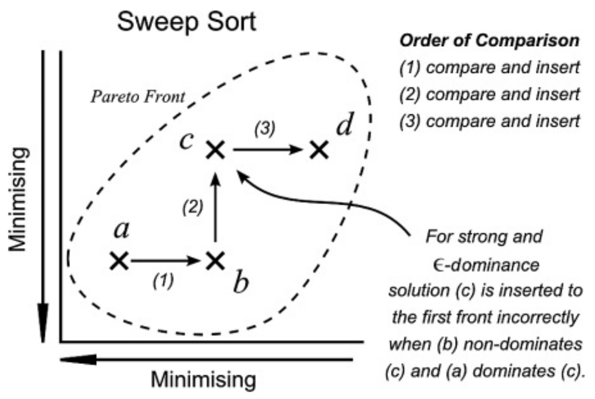

Figure 2: Order of comparison for Jensen's algorithm highlighting incorrect addition of solutions to the first front.

constrained dominance (Deb, 2000; Kukkonen and Lampinen, 2005), and ϵ-dominance as defined by Laumanns et al. (2002), renders Jensen and Yukish's approaches inapplicable in their current form. Readers are directed to Laumanns et al. (2002) for more information on ϵ-dominance. This is demonstrated in Figure 1 where, for example, a sweep line algorithm would incorrectly assign solutions c and d to the first front for cases 2 and 3; strong and ϵ-dominance. The demonstrated problem arises due to the method by which a solution is compared with respect to a given front, as shown in Figure 2. An $O(MN^2)$ approach compares a solution against all solutions in a given front, thus ensuring that any dominating solutions are identified and that solutions are not incorrectly added to the wrong front. Jensen's sweep algorithm compares the next ordered solution to the last solution added to a given front and determines the whole front's relationship based upon the last added solution. For cases 2 and 3, solution c is dominated by solution a but not b; however, b being the last solution added to the first front results in the incorrect addition of c to the first front.

The field of EAs is constantly evolving as definitions are developed and existing principles altered. It is therefore important that any base algorithm, such as non-dominated sorting, is flexible in its approach. Algorithms that restrict their application to a specific dominance relation run the risk of being prohibitively stringent and so less accessible to researchers and new technologies. In addition, for any algorithm performing a relatively small task to be widely accessible to developers, the practical implications of developing the algorithm must be reduced where possible. This paper proposes such a flexible and simple approach through the use of deduction.

4 Preliminaries

For the purpose of this paper the term Pareto front defines the set of all non-dominated points in the population. Dominated solutions are sorted into subsequent fronts, where each front is non-dominated by the population of remaining solutions once all preceding fronts are removed. The term solution will be used synonymously with point and individual, where a solution represents both the genotypically encoded string and phenotypically represented point as given by the vector of values for each objective (fitness) function. A set of solutions is referred to as a population. For the following inferences, dominance will be assessed in terms of strong dominance; however, the inferences apply to weak and ϵ-dominance also. The sorting algorithms only sort solutions by the objective values and not the parameters (the genotypically encoded string).

Dominance of a over b is given as $a \prec b$ and can be said to be transitive ($a \prec b \wedge b \prec c \vdash a \prec c$) (Deb, 2001). In addition, a distinction is made between non-dominance ($a \not\prec b$) and mutual non-dominance ($a \curlywedge b$) where non-dominance of a over b does not imply the dominance of b over a, but permits it, and the mutual non-dominance of a over b implies the non-dominance (and so mutual non-dominance) of b over a and so not permitting the dominance of b over a.

Direct dominance can be defined as the dominance of a over b where no solution dominated by a dominates b. Direct dominance can also be referred to as first order domination and is denoted as $D(1)$. Second order domination, $D(2)$, occurs when a dominates b and c, and c also dominates b. In this instance, a is said not to directly dominate b but to indirectly dominate b through c; ($a \overset{c}{\prec} b$). From order of domination it is possible to build up a string of direct dominance relationships describing the indirect relationship between two solutions, which will be referred to as a dominance chain. A chain is denoted as $a \overset{b,c,d}{\prec} e$; or alternatively, as $f_{\text{dominates}}(a, \{b, c, d\}, e)$, where the order of solutions represents the order of direct dominance. Whilst order of dominance is always given as the shortest possible chain of directly dominating relationships, it is valid to describe dominance in longer chains.

The set of all solutions dominated by a is given as D_a and directly dominated solutions by $\bar{D}_a$. The order of domination of a over b can also be defined as follows: a is said to directly dominate b if $a \prec b$ and b is not dominated by any solution from the set of all other solutions dominated by a; ($b \not\prec (D_a \backslash b)$).

Using these definitions, it is possible to infer relationships between solutions without comparing the solutions directly. In theory, it is therefore possible to reduce the number of comparisons to the bare minimum (only direct dominance relationships) and still produce a full set of relationships if required, for example, when producing a full set of fronts. Representing these relationships in graphical form (see Figure 3), it is possible to see how the structure of direct dominance represents a graph or network, which will be referred to as a dominance graph. The deductive principle can be seen working here because the dominating relationship between a and d can be deduced rather than calculated.

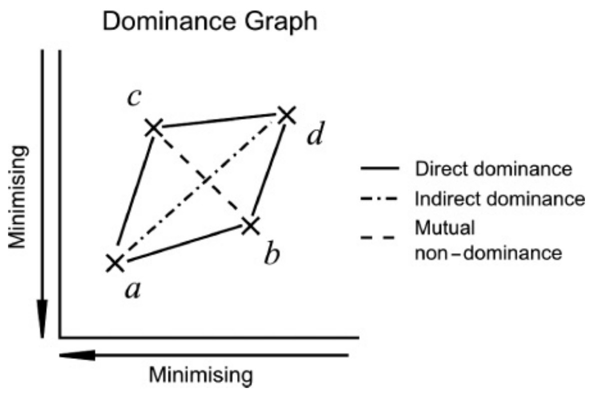

Figure 3: Dominance relationships between solutions.

4.1 Dominance Function

For the algorithms proposed in this paper, a fast comparison function (given in Appendix A) will be used to compare the relationship between two solutions. In the best case the comparison function used in Section 5 returns a value after comparing only two objectives and in the worst case after all objectives have been compared.

5 Methodology

For the experiments conducted in this paper, the following method was used to ensure a fair and consistent comparison of all algorithms. The experiments were conducted over two tests with differing types of datasets. The first experiment aims to assess the general ability of each algorithm given a random population of solutions, a cloud. The second experiment aims to assess the relationship between each algorithm's performance against a dataset containing differing numbers of fronts. Both tests were conducted over a range of population sizes and number of objectives to highlight the effect of population size changes and increased objective dimensionality.

The tests were implemented in C$^{\#}$ and conducted on a 2.66 GHz Intel Core 2 Quad CPU with 3.2 GB RAM running Microsoft Windows XP SP3. To ensure the tests were not affected by external processes or operating system interruptions, the test application process was executed on one dedicated, isolated core. All the results were collected in a single run with no recorded process interruptions. Further, to minimise the possible variations in execution times, each algorithm sorted each dataset five times and the runtime results were averaged. To prevent possible unexpected advantages that may occur in a population's distribution or ordering, the algorithms sorted five equivalent datasets with varying order and distribution, and the runtime and comparison results were then averaged.

To maintain a platform for fair comparison of results, the following parameters were applied to both tests. Each algorithm was presented with identical population datasets and where the number of objectives ranged from 2 to 20, incrementing by 1. Each objective dataset contained populations ranging from 100 to 5,000 solutions at strict 100 increments. The solutions were randomly ordered for each population. The same generated populations were used for all algorithms. For the second test, each

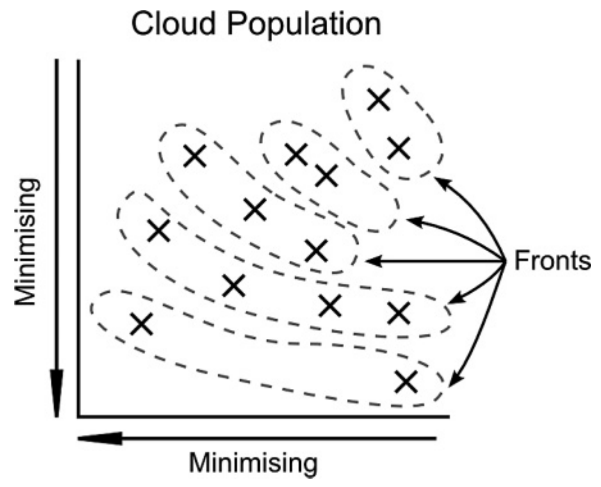

Figure 4: A population containing a cloud of solutions.

population size dataset contained datasets with varying numbers of fronts that ranged from 1 to 20, incrementing by 1.

For both tests, the following metrics were recorded: number of comparisons and total execution time to sort a single population. Each algorithm was presented with the same dominance function, as defined above, and each call to the function was recorded and counted once an algorithm finished sorting a population. The execution time was recorded using the .Net system diagnostics timer (System.Diagnostics.Stopwatch) accurate to 1 processor tick and validated by the elapsed time recorded from the .Net system DateTime structure (System.DateTime), accurate to 0.8 ms, checking for results differing by over 1.6 ms. All execution time results were successfully validated.

The tests were conducted on all algorithms presented in this paper and also on two additional benchmark algorithms; the fast non-dominated sort of NSGA-II (Deb et al., 2000) and the non-dominated rank sort of the omni-optimizer (Deb and Tiwari, 2005). In addition, to ensure a reliable runtime result, the lists used by the sorting algorithms in the experiments were implemented specifically for these tests to prevent any unexpected changes to the overall computational complexity of the algorithms that may occur from using existing code.

5.1 Test One: Cloud Populations

To compare each algorithm's performance when sorting cloud populations, a simple test was devised using randomly generated objective values and randomly ordered solutions. The test was intended to mimic populations from early states of an EA's run in addition to the general ability of an algorithm to identify efficient methods of sorting mixed populations. Each solution's objective values were randomly sampled from a uniform distribution in the range [0, 1] producing an unstructured population of solutions across the whole space with varying numbers of solutions in the Pareto front and subsequent non-dominated fronts. The randomly sampled population then consists of a random arrangement of solutions in a random order. Each solution also dominates a random number of other solutions with the population containing a varying number of fronts.

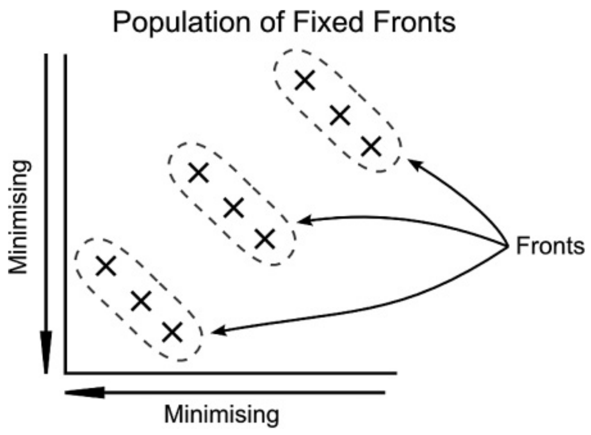

Figure 5: A population containing discrete fronts.

5.2 Test Two: Populations with Fixed Fronts

The second test provided a structured dataset intended to examine the effect to the algorithms' performance based on changing the number of fronts in a population. The populations were generated with discrete fronts where every solution in a given front dominates all solutions in subsequent fronts. The purpose of the test was to highlight the change in performance of each algorithm as the number of fronts diminish. The tests simulate the changing nature of an EA population as it converges on the Pareto front. As the population converged on one front, the simpler algorithms presented in this paper were expected to perform similarly to existing algorithms, however, the more sophisticated algorithms were expected to continue outperforming existing algorithms in both the number of comparisons and overall execution time.

To generate fixed fronts, the population was split into equal sized sets, one set for each front; that is, for three fronts, a population of 12 would be split into three sets of four solutions. When the population did not divide exactly into the number of sets, then the last front was increased or decreased accordingly to ensure the other fronts remained equal; that is, for three fronts, a population of 10 would be split into two sets of three solutions and one of four solutions. The distribution of the solutions was adjusted to ensure the distance between the extremal solutions within the set was equal for all sets. The solutions were distributed on a line or plane. The line or plane was oriented orthogonally with respect to the vector **1**—a vector of 1s. For example, in a 2D space, the solutions were distributed equally along a line perpendicular to the vector (1, 1). This is demonstrated in Figure 5.

For spaces of three or more dimensions, the solutions were evenly projected onto the plane in a grid arrangement. Each front was wholly separated so that every solution in a given front dominated all solutions in the subsequent fronts. As such, the worst value for each dimension in a dominating front is always better than the best value for each dimension in the next front, where the arrangement of solutions ensures a solution will always dominate every solution in all subsequent fronts.

6 Climbing Sort

The climbing sort algorithm, shown in Algorithm 1, follows dominating relationships between solutions and climbs up the graph toward the Pareto front. This is done by

Algorithm 1 Climbing sort

```
climbingSort(S):
   x = 0                                      //set first front
   nf = 0                                     //set first solution
   F = {}                                     //initialise fronts
   while ns < |S|:                            //while not all are sorted
    F += {}                                 //append a new (empty) front
      s = nf                                  //set current solution to nf
      D = {}                                  //set D to empty set
      while s >= 0 & ns < |S|:                //while not all are sorted
         n = -1                               //set n to invalid index
         for i = 0 to (|S| - 1):              //iterate through solutions
            //if s != i and S[i] not in a previous front or dominated
            if s != i & S[i] !in F[0 to x-1] & S[i] !in D:
               d = dominates(S[i], S[s])      //compute relationship
               if d = 3:                      //if S[i] dominates S[s]
                  D += S[s]                   //insert S[s] into set D
                  if S[i] !in F[x]:           //if S[i] not sorted
                     n = i                    //set next to assess to i
                     nf = s                   //start for next front to s
                     break                    //break out of for loop
               else if d = 1:                 //else if S[s] dominates S[i]
                  nf = i                      //start of next front to i
                  D += S[i]                   //insert S[i] into set D
               else:                          //else if mutual
                  if S[i] !in D:              //and S[i] not dominated
                     n = i                    //set next to assess to i
         if S[s] !in D:                       //if S[s] not dominated
            F[x] += S[s]                      //insert S[s] into F[x]
            ns = ns + 1                       //increment number sorted
         s = n                                //set next to be assessed
      x = x + 1                               //increment current front
   return F                                   //return the set of fronts
```

moving from any dominated solutions to the dominating solution. The process is repeated until a non-dominated solution (on the front) has been located. In moving to dominating solutions, the climbing sort aims to assess the points on the Pareto front in as few moves as possible. As the climbing sort encounters dominated solutions, it marks and discards them. After a solution has been compared to all remaining unmarked solutions and found to be non-dominated, the algorithm inserts the solution into the current front. Next, the climbing sort moves to the last mutually non-dominating solution with the assumption that this solution is also likely to occupy the current front. The movement of the climbing sort is restricted to solutions that either dominate the current solution or are mutually non-dominating. Therefore, it is impossible for any solution dominated by the current solution (and thus any solution indirectly dominated) to dominate the next solution chosen by the algorithm. By discarding dominated solutions, the algorithm infers the non-dominance of the dominated solution over any future solution to be assessed.

Once all non-dominated solutions have been assigned to the current front and there still exist unsorted solutions, a new front is added and set as the current front. The last dominated solution assessed is selected as the first solution to assess for the next front as this is likely to occupy the next front. All solutions assigned to a front are now ignored

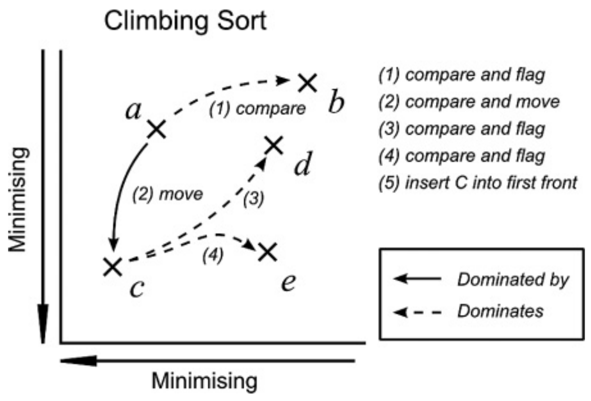

Figure 6: Illustration of the climbing sort. The above diagram highlights the savings made for the comparisons *a-d*, *a-e*, *b-d*, *b-c*, *b-e*, and *d-e*: the algorithm makes 4/10 comparisons. After this first front, the algorithm will move to *e* (the last compared solution) and begin the process again, hopefully already on the next front.

as the algorithm repeats the climbing and sorting until all solutions are assigned. This process is shown in Figure 6 where one iteration is shown.

The climbing sort takes advantage of dominating relationships and attempts to reduce the number of dominated solutions that are assessed in each loop. The algorithm performs better on populations containing a large number of solutions in a large number of fronts; i.e., a cloud. In contrast, for single front populations, the advantage is lost and the algorithm is no better than the fast non-dominated sort (Deb et al., 2000) or non-dominated rank sort (Deb and Tiwari, 2005), making $N(N-1)$ comparisons.

Given Algorithm 1, it is possible to deduce that the algorithm sorts in a worst case $O(N^2)$, performing $N(N-1)$ comparisons. For the climbing sort, the worst case can be seen as the case where all solutions are non-dominating with respect to all other solutions (they occupy a single front). The climbing sort reduces the number of comparisons when it finds dominated solutions and discards them—benefiting from the transitive features of the dominance relationship. However, if no solution is dominated, then no solutions can be discarded and all comparisons must be made. The worst case is computed as follows.

Given $S \forall s \in S(\neg\exists t \rightarrow t \prec s)$ such that to compute the mutual non-dominance of s with respect to the set S/s, s must be compared to all $t \in S$ to verify $\neg\exists t \rightarrow t \prec s$. The computation of each s in S can be seen to be $O(N)$ where $N-1$ comparisons are made. To compute the mutual non-dominance of all $s \in S$, $N \times (N-1)$ comparisons must be made, giving a total order of $O(N^2)$ performing a total of $N(N-1)$ comparisons.

6.1 Experiment 1: Climbing Sort

The first test, on nonstructured clouds of solutions, demonstrated promising performance for the climbing sort compared to the non-dominated sort and non-dominated rank sort in terms of comparisons, outperforming both methods for all population sizes and number of objectives (see Figure 7). However, in terms of execution time, the climbing sort was marginally slower than the non-dominated rank sort for two objectives. The performance in execution time decreased as the number of objectives

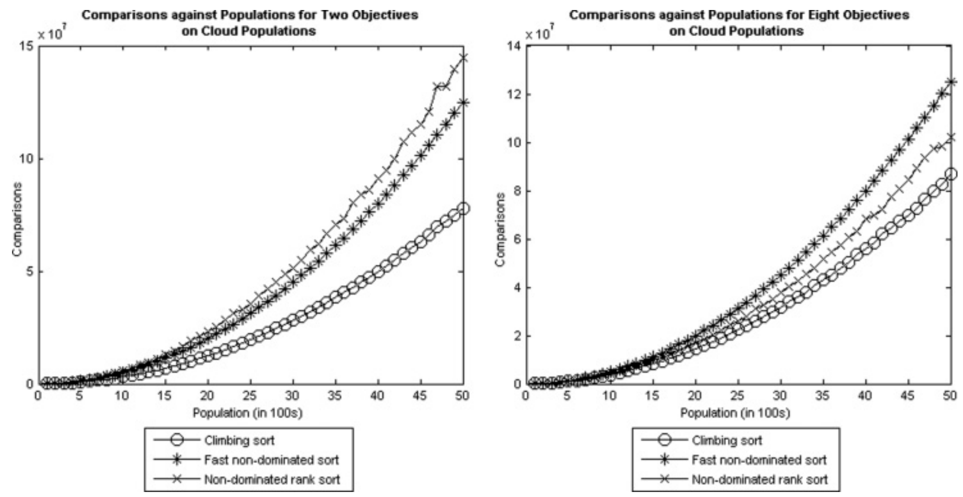

Figure 7: Plot of comparisons for climbing sort, fast non-dominated sort and non-dominated rank sort over random populations.

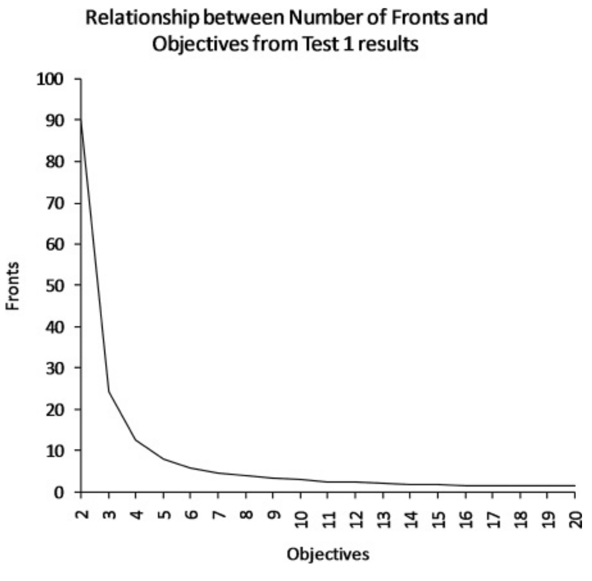

Figure 8: Plot of number of fronts against number of objectives from generated data for Test 1.

increased, tending toward the performance times of the fast non-dominated sort. As the number of objectives increases, the relative efficiencies of the climbing sort are reduced due to the reduction in the number of fronts in the datasets as shown in Figure 8. For an increasing number of objectives, the proportion of space dominated by a solution decreases and thus the likelihood of dominating another solution also decreases. It follows that the number of fronts will naturally be reduced for a fixed population size as the objective-space dimensions increase. Over two to 16 objectives, the climbing

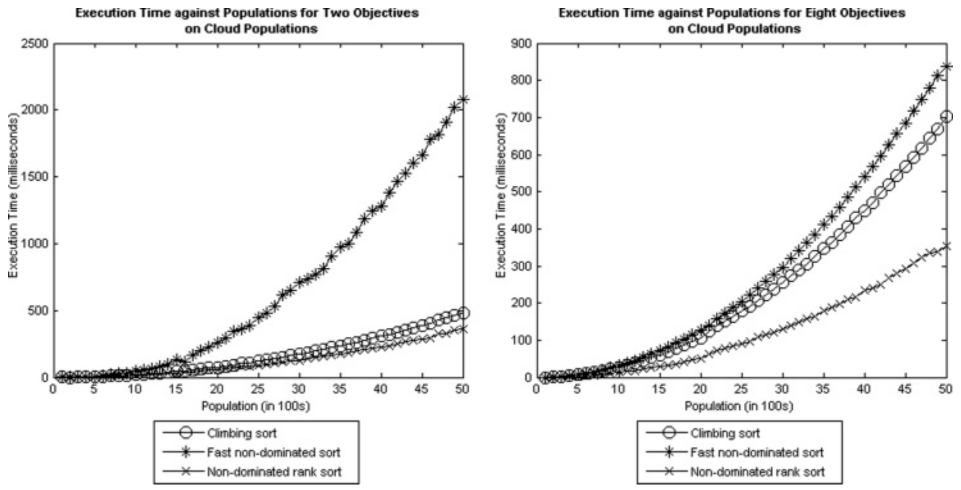

Figure 9: Plot of execution time for climbing sort, fast non-dominated sort and non-dominated rank sort over random populations.

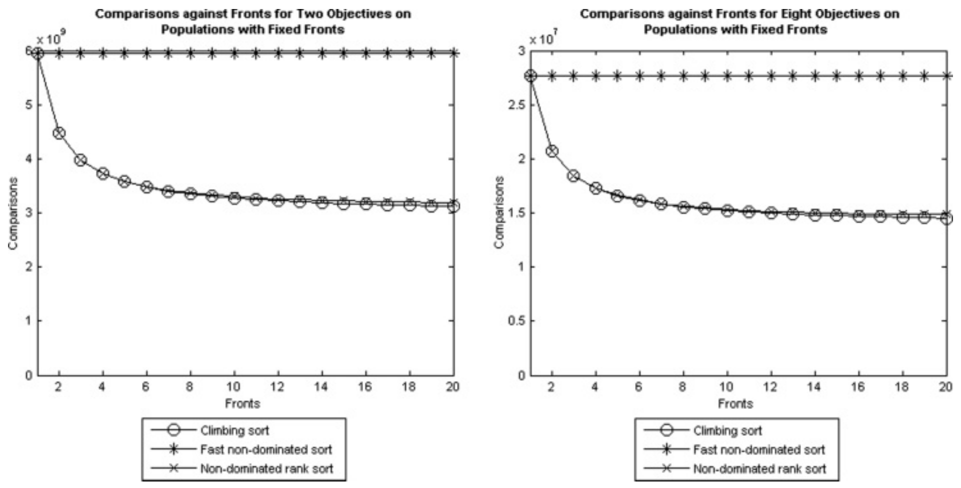

Figure 10: Plot of comparisons on populations of 75,000 for climbing sort, fast non-dominated sort and non-dominated rank sort over populations with fixed fronts.

sort took less time to execute than the fast non-dominated sort, but the improved performance became reduced in higher objective space (see Figure 9). As the number of objectives and the size of the population increased to 18 objectives and 4,000 solutions, the climbing sort and non-dominated rank sort experienced an increase in execution time.

The second test, on structured fronts, also produced mixed results, as shown in Figures 10 and 11. The climbing sort consistently performed better than the fast non-dominated sort in terms of comparisons for all objectives and population sizes. However, in terms of execution time, the climbing sort again consistently performed worse than the non-dominated rank sort for all objectives and population sizes. With

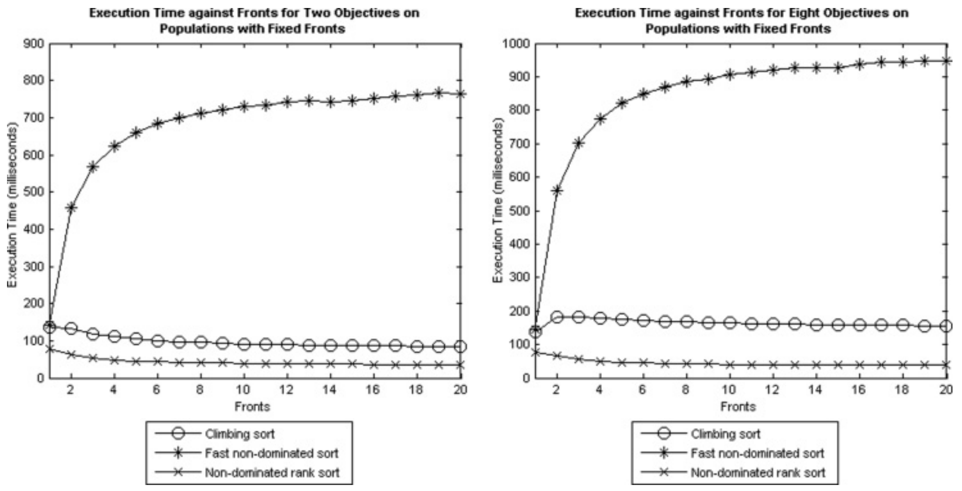

Figure 11: Plot of execution time on populations of 75,000 for climbing sort, fast non-dominated sort and non-dominated rank sort over populations with fixed fronts.

respect to the number of comparisons, the fast non-dominated sort performed poorly—consistently performing $N^2 - N$ comparisons, as expected. The climbing sort performed marginally better than the non-dominated rank sort, with the algorithm performing relatively fewer comparisons as the number of fronts increases. The non-dominated rank sort also experienced a performance increase as the number of fronts increased but eventually plateaued.

6.2 Improved Climbing Sorts

Two alterations can be made to the climbing sort to produce two algorithms of differing style and effect. Firstly, each solution dominated by a dominated solution can be flagged as dominated by at least $D(2)$ and so should remain marked as dominated when processing the next front. This method reduces the need to reassess a solution already known not to be a member of the next front. The algorithm is given in Appendix B, called the flagging sort. However, the method still allows unnecessary reassessment in later fronts, for example, a solution a may be found to be dominated by b and so is marked dominated. Solution b may then be found to be dominated by c and so marked b dominated and a dominated by order $D(2)$ and thus ignoring a when computing the next front. However, if a is dominated by b, b is dominated by c, and c is dominated by d, the above policy would mark both a and b as dominated by order $D(2)$, and not $D(3)$, $D(2)$, respectively. The solution a would incorrectly be reassessed at the same time as b, even though it is in a longer dominance chain than b: $d \overset{c,b}{\prec} a$.

To avoid these reassessments, a second alternative alteration, as implemented in the listing sort given in Appendix C, records the known order of all dominated solutions and thus further reduces the unnecessary reassessments. Each solution is assigned a list of solutions it dominates, which is used as a memory process for removing unnecessary repeat comparisons which, in effect, maintains the known order of dominance for all dominated solutions. The list is copied from one solution to all solutions that dominate it, removing further unnecessary comparisons and implementing the inferred dominance. Thus, the algorithm builds a full set of dominating relationships for each solution

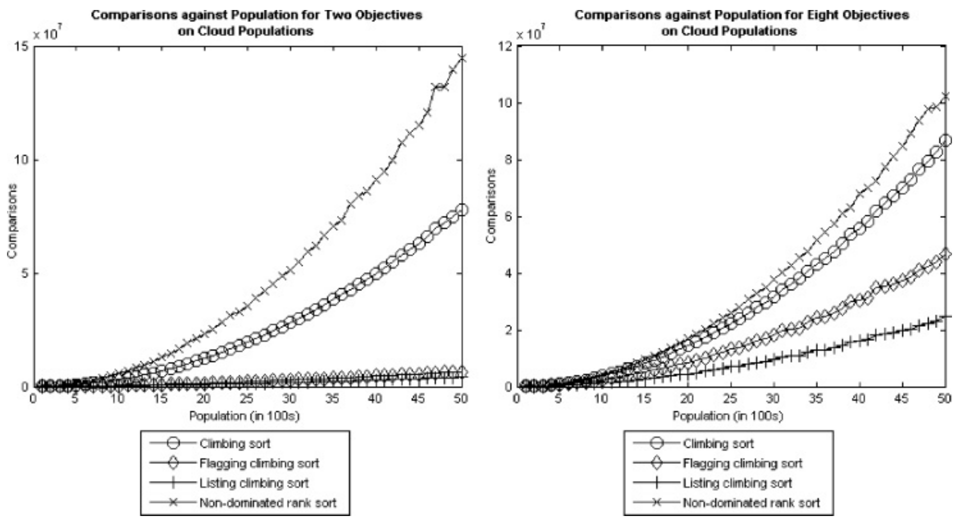

Figure 12: Plot of comparisons for climbing sorts over random populations.

without making every comparison. In addition, it is prudent to record all mutually non-dominated solutions and thus prevent further repetition.

The three algorithms proposed above all make use of dominance information and some inference of dominance to improve the efficiency of sorting a population into a complete set of fronts. However, all three algorithms have increased memory requirements with, at the extreme, the listing sort requiring storage of $O(N^2)$ in the worst case, that is, when all solutions are members of the first front. In addition, the algorithms fail to implement all of the possible inferences, not utilising all possible inferences for non-dominance and mutual non-dominance. Further research into developing a fast algorithm for inferring mutual non-dominance may produce an improved algorithm.

6.3 Experiment 2: Comparing Climbing Sorts

In the first test, for a low number of objectives, the flagging sort and listing sort consistently outperform the climbing sort in terms of comparisons (see Figure 12). Both the flagging sort and listing sort experience a significant performance increase as a result of the increased number of fronts. However, as the number of objectives increase, the flagging sort performs less competitively and produces little improvement over the climbing sort. The additional sharing and retention of mutual non-dominance information not present in the climbing and flagging sorts ensures that the listing sort performs in the worst case 50% fewer comparisons; guaranteeing at least half the number of comparisons of the previous sorts.

The non-dominated rank sort produced the most reliable processing time with a gradual increase in relation to the increase in number of objectives (see Figure 13). The listing sort consistently took longer to execute compared to all other algorithms, however, the processing time reduced as the number of objectives increased. The flagging sort executed the fastest for two objectives. However, as the number of objectives increased, the flagging sort's performance degraded, eventually performing worse than the non-dominated rank sort. Whilst the flagging sort continued to make fewer comparisons than the non-dominated rank sort, the execution time was higher for eight

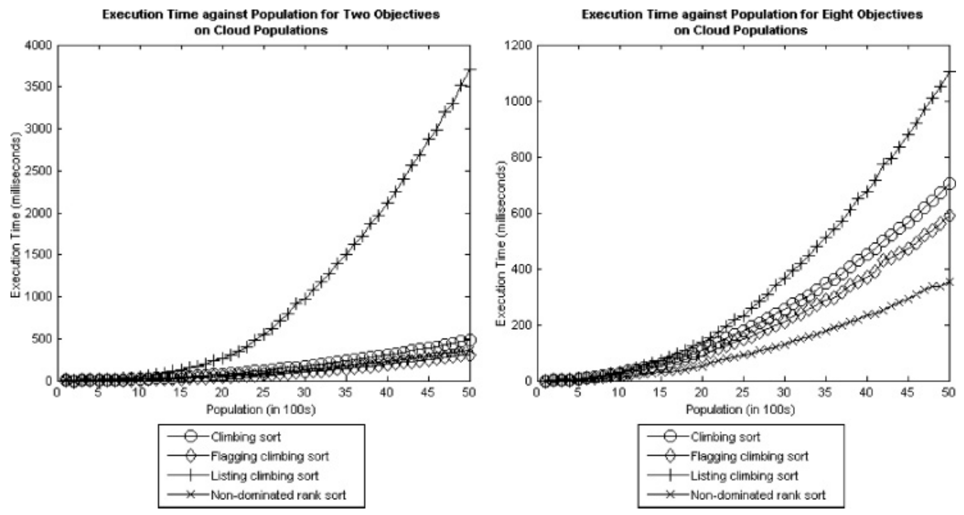

Figure 13: Plot of execution time for climbing sorts over random populations.

objectives. This is due to a loss in the reduction of comparisons as a result of fewer dominated solutions in the population and also in combination with the overheads of the bookkeeping used by the flagging sort: primarily the creation and destruction of lists. The bookkeeping used in the listing sort was significantly more expensive than the flagging sort. Each time a new dominating solution is located, the dominance list of the current solution is replicated and merged with the new dominating solution. In addition, the dominance list of any dominated solution is copied to the dominance list of the current solution. Although fewer comparisons were made by the listing sort, the time taken to maintain and duplicate these lists significantly degraded the performance of the algorithm. Clearly, whilst bookkeeping can be beneficial to reducing the number of comparisons in both the listing and flagging sorts, the cost of bookkeeping can be prohibitively expensive when compared to the simple calculation of the dominance relationship.

As expected, in the second test the basic climbing sort performs worst out of the proposed sorts in terms of comparisons (see Figure 14), making many more comparisons than both the flagging sort and the listing sort. For populations on one front, the climbing sort and flagging sort compute the same number of comparisons. The efficiencies of the flagging sort only take effect for populations with more than one front, whereas the listing sort, due to the transfer of mutual non-dominance relationships, always computes at least 50% fewer comparisons than the climbing sort. Similarly to the flagging sort, the listing sort introduces an additional efficiency with respect to the number of fronts.

When considering computational cost as opposed to comparisons, the results are quite different (see Figure 15). It is interesting to note that the improvement in comparisons seen in the listing sort comes at the cost of increased processing time. The listing sort requires a significant increase in bookkeeping compared to the climbing sort and flagging sort and this bookkeeping bloats the overall processing time. For a small number of fronts, the algorithm shows improvement in processing time until a point where the bookkeeping costs outweigh the improved efficiencies; after this point the algorithm's performance is degraded. Although the above algorithms greatly reduce the number of comparisons when compared to the non-dominated rank sort, the processing

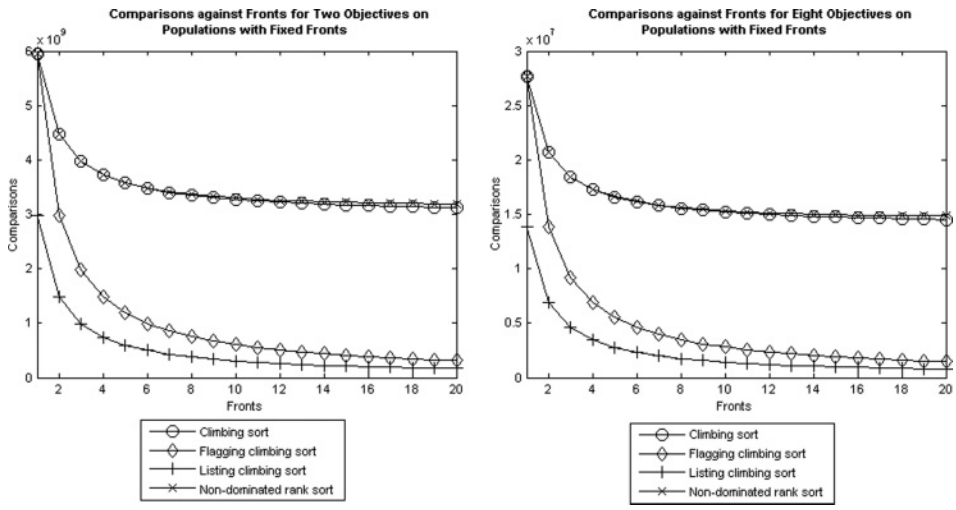

Figure 14: Plot of comparisons on populations of 75,000 with fixed fronts for climbing sorts.

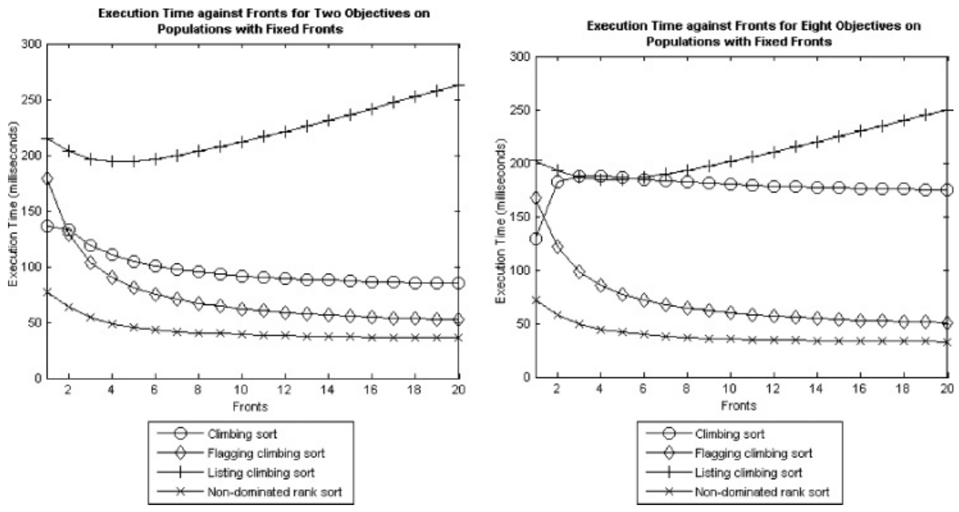

Figure 15: Plot of execution time on populations of 75,000 with fixed fronts for climbing sorts.

time is still slower. It is interesting to note that the listing sort performs better as the number of objectives increases. As this happens, the number of comparable solutions naturally decreases and therefore the number of solutions maintained in any one list also decreases. This reduces the overall bookkeeping costs and results in an increase in performance.

7 Deductive Sort

The deductive sort, given in Algorithm 2, is the most efficient of the proposed algorithms, as demonstrated in the following experiments. The deductive sort implements inferred dominance differently to the previous climbing sort approach. The above

Algorithm 2 Deductive sort

```
deductiveSort(S):
   x = 0                                //set first front
   f = 0                                //set number sorted to 0
     F = {}                               //initialise fronts
   while f < |S|:                       //while not all are sorted
      F += {}                           //append a new (empty) front
      D = boolean[|S|]                  //clear dominated flag array
      for i = 0 to (|S| - 1):           //iterate through solutions
         if !D[i] & S[i] !in F:         //if S[i] not dominated or sorted
            for j=(i+1) to (|S| - 1):   //from next solution to end
               if !D[j] & S[j] !in F:   //if S[j] not dominated or sorted
                  d = dominates(S[j], S[i])   //compute relation
                  if d = 1:             //if S[j] dominated
                     D[j] = true        //flag S[j] as dominated
                  else if d = 3:        //else if S[i] dominated
                     D[i] = true        //flag S[i] as dominated
                     break              //break the inner for loop
            if !D[i]:                   //if S[i] is not dominated
               F[x] += S[i]             //insert S[i] into F[x]
               f++                      //increment number sorted
      x = x + 1                         //increment current front
   return F                             //return the set of fronts
```

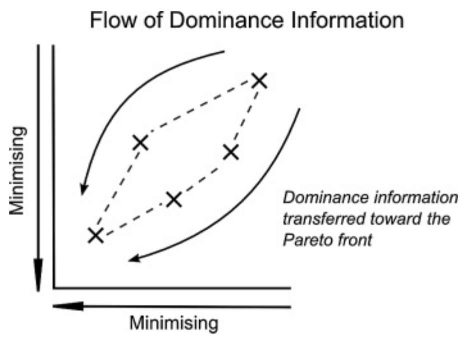
Flow of Dominance Information
Minimising
Minimising
Dominance information transferred toward the Pareto front

Figure 16: The flow of information for climbing sorts.

algorithms, listing sort in particular, transfer the information on known dominated solutions up the chain of dominance by means of flags or lists assuming that efficiencies can be made by building from the most dominated solution up to the non-dominated solutions. In this approach, the flow of information is moving toward the Pareto front as shown in Figure 16.

The deductive sort, shown in Algorithm 2, assesses each solution based upon the fixed natural order of the population. Each solution is compared against all following solutions and not previous solutions, as shown in Figure 17. Any solution found to be dominated by the current solution is marked as dominated and is ignored. If the current solution is found to be dominated, then it is marked as dominated and the algorithm proceeds to assess the next solution without completing the assessment of the current solution. Any later solutions not yet assessed are ignored by the algorithm if they

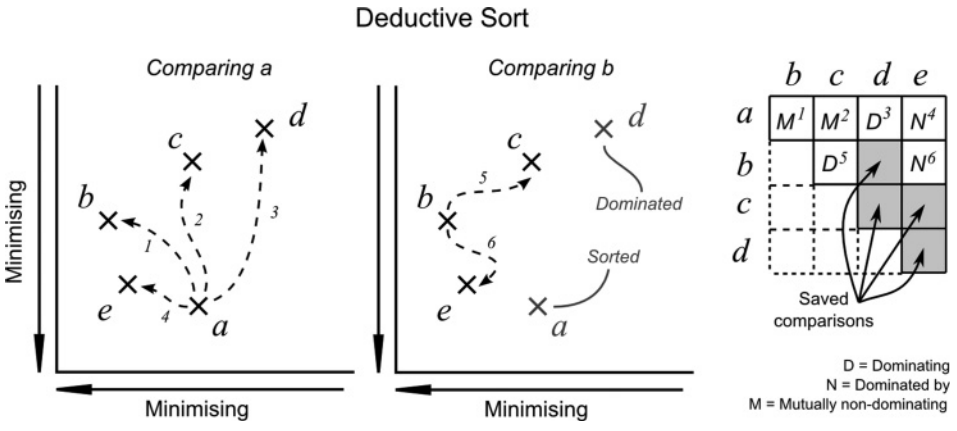

Figure 17: Illustration of the deductive sort.

are marked dominated. The deductive sort works on the premise that any preceding solution that dominates later solutions will mark the later solution as dominated and so skip any comparison with that solution, saving unnecessary comparisons. Any solution dominated by a solution already marked as dominated will also be dominated by the dominating solutions and so any comparison with the marked solution can be superseded by a comparison with the dominating solution that will return the same result. In effect, the deductive sort computes the inferred dominance in reverse. By maintaining a strict order of assessment and comparison, the deductive sort ensures that all dominated solutions are discarded and not incorrectly added to the wrong front. Once a front has been correctly filled, all solutions assigned to a front are then ignored and the process is repeated until all solutions are assigned to a front.

In the example in Figure 17, the algorithm takes a randomly ordered set of solutions $\{a, b, c, d, e\}$ and sorts them into fronts. In practice, the algorithm would produce a complete set of fronts, but only the first front is shown for clarity. Given the set of solutions, the algorithm analyses a first. a is compared against b and found to be mutually non-dominating. Then a is compared to c, the next element in the list, and also found to be mutually non-dominating. Next, a is compared to d and is found to dominate d. d is then marked as dominated and ignored for the remainder of this front. Finally, a is compared to e and found to be mutually non-dominating. Now a has been compared to all other solutions and has not been found to be dominated and so is inserted into the current, first, front. The algorithm now selects b for analysis. As is already known, a and b are mutually non-dominating, so there is no need to compare b to a (or any other preceding solutions) and so b is first compared with c. b dominates c and so c is marked as dominated and continues to d. As d has been marked as dominated, the algorithm skips d and compares b and e. b is mutually non-dominating with e and so e is inserted into the current front. The algorithm now ignores c as it is known that c is dominated. Next, the algorithm ignores d and continues to e. As e is the last item in the list, e does not need to be compared to any other solutions. If e is not marked as dominated, as in this case, the algorithm just adds e to the current front. Now the algorithm discards all sorted solutions and starts the process again for the next front. The algorithm makes 6/10 comparisons.

The deductive sort improves upon the $O(N^2)$ storage requirements, reducing it to $O(N)$. In the best case, where the first front consists of one dominating solution, the first front can be computed in $N-1$ comparisons. Extending this case, for all fronts, where each solution occupies individual fronts, the complete set of fronts can be computed in the worst case $\frac{N(N-1)}{2}$. Similarly, in the algorithm's worst case, where all solutions are members of the first front, the complete set of fronts can be computed in the same number of $\frac{N(N-1)}{2}$ comparisons. Like the climbing sort (and derivatives), the deductive sort utilises the transitive nature of the dominance relationship to reduce the number of comparisons. If no solution is dominated, no solutions can be excluded from the calculations for the current front and no savings are made. In the best case where the population of N solutions evenly occupy the set of fronts F, where each front contains N/F solutions, and where N is a multiple of $|F|$ (in the best case $N = |F|^2$), the set of fronts can be computed in $\frac{NF-1}{2} + \frac{N/F(N/F-1)}{2}$ comparisons; assuming the first solution selected in each iteration is in the current front. As the number of solutions moves away from $|F|^2$, the relative efficiency of the algorithm decreases, limited to $\frac{N(N-1)}{2}$. In addition, efficiency is lost when the first selected solution at each step is further down the set of fronts. From these best and worse cases it is possible to surmise that as a GA progresses, with the population starting as a random population—a cloud of solutions—and slowly converging on a single optimal Pareto front, the relative efficiencies of the algorithm will be reduced for each new generation.

Given the above algorithm, it is possible to deduce that the algorithm sorts in a worst case $O(N^2)$, performing $\frac{N(N-1)}{2}$ comparisons. The worst case, where all solutions are non-dominating with respect to all other solutions (they occupy a single front), is computed as follows. Given $S, \forall s \in S(\neg\exists t \rightarrow t \prec s)$ such that to compute the mutual non-dominance of s with respect to the set S/s, s must be compared to all solutions that have not yet been sorted $t \in (S/F)$, where F is the set of all sorted solutions, to verify $\neg\exists t \rightarrow t \prec s$. The computation of each $s \in S$ can be seen to be $O(N)$ where $N - (|F| + 1)$ comparisons are made; N denotes the number of solutions in the population. To compute the mutual non-dominance of all $s \in S$, $(N-1) + (N-2) + \cdots + (N-N)$ comparisons must be made, giving a total order of $O(N^2)$ performing $\frac{N(N-1)}{2}$ comparisons in total.

7.1 Experiment 3: The Deductive Sort

As shown in Figure 18, all the proposed algorithms performed fewer comparisons than the non-dominated rank sort, utilising the large chains of dominance to remove many of the unnecessary comparisons. However, as discussed earlier, the tendency for the number of fronts to diminish with respect to the increasing number of objectives resulted in the climbing and flagging sort showing less improvement over the non-dominated rank sort in higher objective space. Interestingly, the listing sort tended to perform marginally fewer comparisons than the deductive sort, demonstrating the large decrease in comparisons available by sharing a greater degree of information.

For low to medium numbers of objectives, 2–16, the deductive sort executed faster than all other algorithms, as shown in Figure 19. In addition, the execution time (for all population sizes) remained relatively stable—even as the number of objectives increased. The non-dominated rank sort improved as the number of objectives increased, performing similarly to the deductive sort for 18 objectives. However, as the population reached 4,000, the deductive sort consistently performed better than the non-dominated rank sort. In the second test, as expected, the listing sort and deductive sort performed almost identically, achieving a reasonable level of performance. The non-dominated rank sort and other algorithms performed the worst and computed many more comparisons.

Following the performance pattern of Experiment 2 (see Figure 14), the non-dominated rank sort performed the worst (see Figure 20). The climbing sort

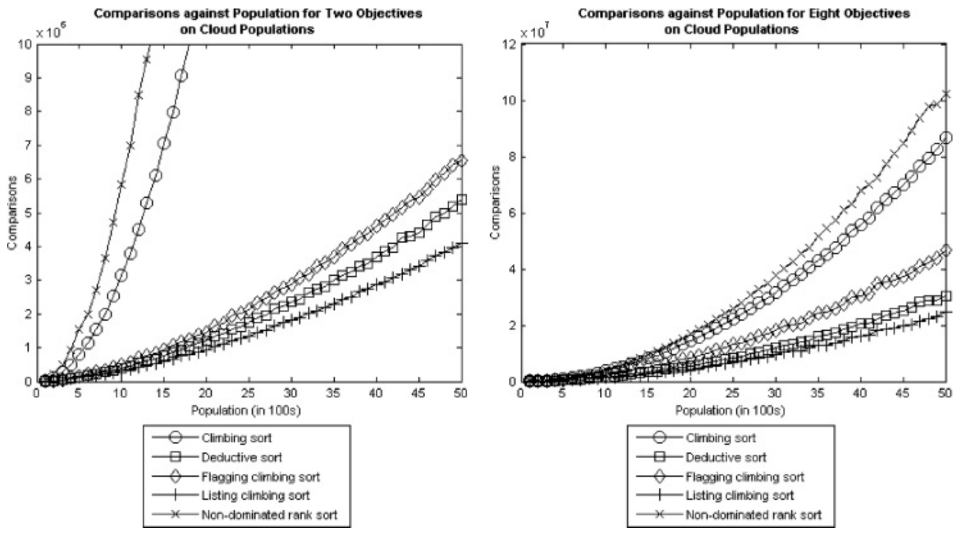

Figure 18: Plot of comparisons for climbing sorts, deductive sort, and non-dominated rank sort over random populations.

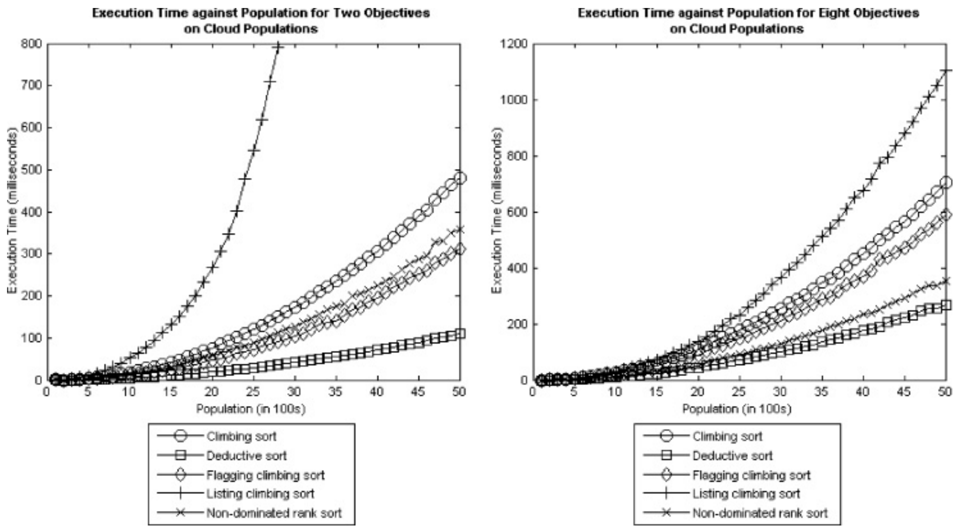

Figure 19: Plot of execution time for climbing sorts, deductive sort and non-dominated rank sort over random populations.

performed marginally better, but still significantly worse than the two augmented climbing sorts: flagging sort and listing sort. The listing sort performed equally as well as the deductive sort, with both algorithms making far fewer comparisons compared to all others. It is interesting that the deductive sort makes an equivalent number of comparisons as does the listing sort when the deductive sort algorithm does not "exchange" dominance information in such a literal and active manner as the listing sort.

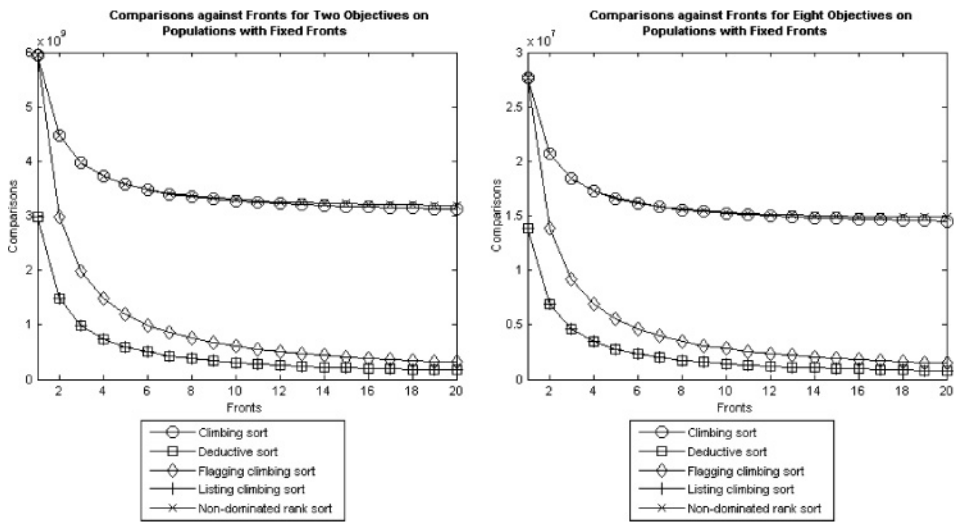

Figure 20: Plot of comparisons on populations of 75,000 for climbing sorts, deductive sort, and non-dominated rank sort over populations with fixed fronts.

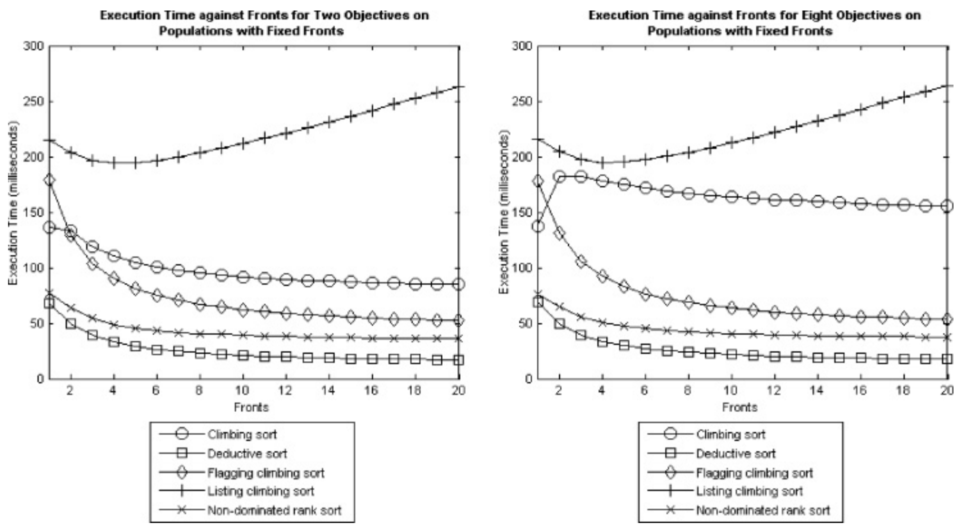

Figure 21: Plot of execution time on populations of 75,000 for climbing sort, deductive sort, and non-dominated rank sort over populations with fixed fronts.

Repeating the performance recorded in the second test of Experiment 2 (see Figure 15), the climbing sort, flagging sort, and listing sort demonstrated varying performance over the range of fronts. When considering processing time, the deductive sort again compared favourably and consistently outperformed all other algorithms, as shown in Figure 21. Counterintuitively, the climbing sort saw only a small improvement when increasing the number of fronts. While sorting populations of one front, all tested algorithms were not affected by dimensionality of the space. The change in dimensionality had the greatest affect on the climbing sort (for populations with more than one front). As shown in Figure 21, the deductive sort executes significantly faster than all other algorithms for all objectives and numbers of fronts. The deductive sort's reduced number of comparisons does not affect the algorithm's overall execution time, unlike the listing sort—which is penalised for copying dominance information. The results suggest that the deductive sort is the most efficient of the algorithms tested, and achieves the desired reduction in number of comparisons.

The tests produced mixed results and immediately highlighted the risk of overcomplicating the data structures used in algorithms such as the listing sort as this increases the runtime. However, the increased runtimes were to the benefit of greatly reduced number of comparisons. These algorithms demonstrate the possible reductions available when implementing inference of dominance. The possibility of improved data structures allows for the future improvement of the listing sort algorithm. The simplicity of the deductive sort ensures that it is not bloated by excessive bookkeeping and so creates a consistently low computational cost. In addition to reducing the runtime, the simplicity of the algorithm greatly reduces the task of implementing the algorithm. Further improvement could be made to the deductive sort by retaining the already computed relationships between solutions. However, in low objective space, the benefits of this storage would be minimal compared to the increased $O(MN^2)$ storage costs.

7.2 Insertion and Merging

In many EAs, a new population of unsorted solutions is generated at each generation and the whole population or selection of solutions is merged with an existing presorted population. Since many algorithms resort the merged population from scratch, the known dominance information from the sorted population is lost. Retaining the dominance information produced by the climbing and deductive sorts presents a potential for faster merging. With many of the comparisons already calculated, an algorithm can quickly sort the new population by filling in the missing information in the dominance graph. Also, when considering the insertion of comparatively small numbers of new solutions into a population, the dominance or non-dominance of a solution can be assessed by intelligently comparing existing solutions in a population and inserting the new solutions into the appropriate position in the dominance graph. The inferences defined in this paper provide the scope for such an algorithm which could, for many EAs, significantly improve the computational complexity of later generations.

8 Conclusion

This paper has presented a number of novel techniques for non-dominated sorting and proposed one algorithm, the deductive sort, that outperforms existing MN^2 approaches such as the non-dominated rank sort of the omni-optimizer. On two different test methods, all the presented approaches performed fewer comparisons than existing techniques and demonstrated the potential of inferring dominance relationships as outlined in this paper. In particular, the deductive sort performed better than all other techniques and in terms of computational complexity for multi- and many-dimensional problems would be the most suitable algorithm to implement, with a guaranteed worst case $\frac{N(N-1)}{2}$ comparisons and storage of $O(N)$.

In addition to examining the number of comparisons in each sorting procedure, this paper highlighted the trade-off between reducing comparisons and increasing the bookkeeping overheads. We examined the effect on the flagging and listing sorts and demonstrated that whilst the listing sort produced relatively few comparisons in our experiments, the overheads of bookkeeping greatly outweighed any savings made. Clearly, the execution cost of such algorithms, and code complexity, should be carefully considered alongside any reduction in comparisons.

In contrast, the efficiency and simplicity of the deductive sort ensures that implementing the non-dominated sorting aspect of an MOEA remains a small task when compared to the greater task of implementing an EA. Finally, in addition to improving on existing sorting algorithms, this paper has outlined an information structure, the dominance graph, that offers the potential for faster insertion of solutions and merging of populations into existing presorted populations. The effects of this result could greatly improve the performance of modern elitist EAs which increasingly rely on the repeated sorting and merging of populations of solutions.

9 Further Work: Mutual Non-dominance

A further inference that could be made in order to preserve the efficiencies produced by the deductive sort's inherent inferences of dominance and non-dominance, is that of mutual non-dominance. Using the previous inferences of dominance, it is possible to define a rule for inferring relationships between mutually non-dominating solutions. This was implemented and tested in a similar fashion to the previous experiments. Whilst the results suggested comparisons can be reduced further, the cost of computing those inferences was found to outweigh the savings made by the reduction in comparisons. If an efficient method could be designed for calculating inferred mutual non-dominance which does not expand the computational complexity, it is feasible that a significantly faster algorithm could be devised. However, as demonstrated by the listing sort, the cost of transferring the information can be expensive and should be avoided.

10 Acknowledgments

This work was supported by an EPSRC CASE award (Grant No. CASE/CNA/07/100) and Mouchel Limited.

References

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

Coello Coello, C. A., Lamont, G. B., and Van Veldhuizen, D. A. (2007). *Evolutionary algorithms for solving multi-objective problems*, 2nd ed. New York: Springer.

Corne, D. W., Knowles, J. D., and Oates, M. J. (2000). The Pareto envelope-based selection algorithm for multiobjective optimization. In M. Schoenauer, K. Deb, G. Rudolph, X. Yao, E. Lutton, J. J. Merelo, and H. P. Schwefel (Eds.), *Parallel problem solving from nature. Lecture notes in computer science*, Vol. 1917 (pp. 839–848). Berlin: Springer.

Deb, K. (2000). An efficient constraint handling method for genetic algorithms. *Computer Methods in Applied Mechanics and Engineering*, 186(2–4):311–338.

Deb, K. (2001). *Multi-objective optimization using evolutionary algorithms*. New York: Wiley.

Deb, K., Agrawal, S., Pratab, A., and Meyarivan, T. (2000). A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II. In M. Schoenauer, K. Deb, G. Rudolph, X. Yao, E. Lutton, J. J. Merelo, and H. P. Schwefel (Eds.), *Parallel problem solving from nature. Lecture notes in computer science*, Vol. 1917 (pp. 849–858). Berlin: Springer.

Deb, K., Mohan, M., and Mishra, S. (2003). A fast multi-objective evolutionary algorithm for finding well-spread pareto-optimal solutions. Technical Report 2003002, Indian Institute of Technology, Kanpur.

Deb, K., and Tiwari, S. (2005). Omni-optimizer: A procedure for single and multi-objective optimization. In C. A. Coello Coello, A. Hernández Aguirre, and E. Zitzler (Eds.), *Evolutionary* *multi-criterion optimization. Lecture notes in computer science*, Vol. 3410 (pp. 47–61). Berlin: Springer.

Everson, R. M., Fieldsend, J. E., and Singh, S. (2002). Full elite sets for multi-objective optimisation. In *Proceedings of the Fifth International Conference on Adaptive Computing in Design and Manufacture (ACDM 2002)*, Vol. 5, pp. 343–354.

Fonseca, C. M., and Fleming, P. J. (1993). Genetic algorithms for multiobjective optimization: formulation, discussion and generalization. In S. Forrest (Ed.), *Proceedings of the Fifth International Conference on Genetic Algorithms*, pp. 416–423.

Goldberg, D. E. (1989). *Genetic algorithms in search, optimization and machine learning*. Reading, MA: Addison-Wesley.

Horn, J., Nafpliotis, N., and Goldberg, D. E. (1994). A niched pareto genetic algorithm for multiobjective optimization. In *Proceedings of the First IEEE Conference on Evolutionary Computation, IEEE World Congress on Computational Intelligence*, Vol. 1, pp. 82–87.

Jensen, M. T. (2003). Reducing the run-time complexity of multiobjective EAs: The NSGA-II and other algorithms. *IEEE Transactions on Evolutionary Computation*, 7(5):503–515.

Knowles, J. D. (2006). ParEGO: A hybrid algorithm with on-line landscape approximation for expensive multiobjective optimization problems. *IEEE Transactions on Evolutionary Computation*, 10(1):50–66.

Knowles, J. D., and Corne, D. W. (1999). The Pareto archived evolution strategy: A new baseline algorithm for multiobjective optimisation. In *1999 Congress on Evolutionary Computation*, pp. 98–105.

Kukkonen, S., and Lampinen, J. (2005). GDE3: The third evolution step of generalized differential evolution. Technical Report 2005013, Indian Institute of Technology, Kanpur.

Kung, H. T., Luccio, F., and Preparata, F. P. (1975). On finding the maxima of a set of vectors. *Journal of the ACM*, 22:469–476.

Laumanns, M., Thiele, L., Deb, K., and Zitzler, E. (2002). Combining convergence and diversity in evolutionary multi-objective optimization. *Evolutionary Computation*, 10(3):263–282.

Schaffer, J. D. (1984). *Multiple objective optimization with vector evaluated genetic algorithms*. PhD thesis, Vanderbilt University.

Schaffer, J. D., and Grefenstette, J. J. (1985). Multiobjective learning via genetic algorithms. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence (IJCAI-85)*, pp. 593–595.

Srinivas, N., and Deb, K. (1994). Multiobjective optimization using nondominated sorting in genetic algorithms. *Evolutionary Computation*, 2(3):221–248.

Yukish, M. A. (2004). *Algorithms to identify Pareto points in multi-dimensional data sets*. PhD thesis, College of Engineering, Pennsylvania State University, University Park.

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

Zitzler, E., Laumanns, M., and Thiele, L. (2002). SPEA2: Improving the strength Pareto evolutionary algorithm. In K. Giannakoglou, D. Tsahalis, J. Periaux, P. Papailou, and T. Fogarty (Eds.), *EUROGEN 2001. Evolutionary Methods for Design, Optimization and Control with Applications to Industrial Problems*, pp. 95–100.

Zitzler, E., and Thiele, L. (1999). Multiobjective evolutionary algorithms: A comparative case study and the strength Pareto approach. *IEEE Transactions on Evolutionary Computation*, 3(4):257–271.

Appendix A: Dominance Function

```
dominates(a, b):
   a_dom_b = false          //set flag a_dom_b to false
   b_dom_a = false          //set flag b_dom_a to false
   for m = 0 to (|M| - 1):  //iterate through all objectives
      if b[m] < a[m]:       //if b is less than a in respect to m
         if a_dom_b:        //and a flagged as dominating b
            return 2        //then return mutual non-domination
         if !b_dom_a:       //else if not flagged
            b_dom_a = true  //then flag b as dominating a
      else if a[m] < b[m]:  //else if a is less than b in respect to m
         if b_dom_a:        //and b flagged as dominating a
            return 2        //then return mutual non-domination
         if !a_dom_b:       //else if not flagged
            a_dom_b = true  //then flag a as dominating b
   if a_dom_b:              //if not returned and a dominates b
      return 3              //return dominates
   else if b_dom_a:         //else if b dominates a
      return 1              //return dominated
   else:                    //else if both are equal
      return 2              //return mutual non-domination
```

A value of 2 means a is mutually non-dominated with respect to b; 3, a dominates b; and 1 b dominates a.

Appendix B: Flagging Sort

```
flaggingSort(S):
   x = 0                          //set first front
   ns = 0                         //set the number sorted
   nf = 0                         //set first solution
   F = {}                         //initialise fronts (to empty set)
   DD = {}                        //initialise order D(2) set (set to empty set)
   while ns < |S|:                //while not all are sorted
      F += {}                     //append a new (empty) front
      s = nf                      //set the current solution to nf
      D = DD                      //copy DD set to D
      DD = {}                     //clear DD (set to empty set)
      Q = {}                      //clear the temporary dominated set Q
      while s >= 0 & ns < |S|:    //while s >= 0 and not all are sorted
         n = -1                   //set n to invalid index
         for i = 0 to (|S| - 1): //iterate through all solutions
            //if s!=i and S[i] not in D, Q or a previous front
            if s != i & S[i] !in (D + Q + F[0 to x-1]):
               d = dominates(S[i], S[s])  //compute relationship
               if d = 3:                  //if S[i] dominates S[s]
                  D += S[s]               //insert S[s] into set D
                  if S[i] !in F[x]:       //if S[i] not sorted
                     DD += Q              //add all dominated by S[s] to DD
                     Q = S[s]             //clear Q and insert S[s]
                     n = i                //set next to assess to i
                     nf = s               //set start for next front to s
                     break                //break out of for loop
               else if d = 1:             //else if S[s] dominates S[i]
                 nf = i                   //set start for next front to i
                 D += S[i]                //insert S[i] into set D
                 Q += S[i]                //insert S[i] into set Q
              else:                       //else if mutual
                 if S[i] !in D:           //and S[i] not dominated
                    n = i                 //set next to assess to i
      if S[s] !in D:                      //if S[s] not dominated
```

```
            F[x] += S[s]                   //insert S[s] into current front
            ns = ns + 1                    //increment the number sorted
            Q = {}                         //clear Q (set to empty set)
         s = n                             //set the next to be assessed
      x = x + 1                            //increment the current front
   return F                                //return the set of fronts
```

Appendix C: Listing Sort

```
listingSort(S):
   x = 0                         //set first front
   ns = 0                        //set the number sorted
   nf = 0                        //set first solution
   F = {}                        //initialise fronts (to empty set)
   while ns < |S|:               //while not all are sorted
      F += {}                    //append a new (empty) front
      s = nf                     //set the current solution to nf
      D = {}                     //clear D (set to empty set)
      while s >= 0 & n < |S|:    //while s >= 0 and not all are sorted
         n = -1                  //set n to invalid index
         for i = 0 to (|S| - 1): //iterate through all solutions
            //if s!=i and S[i] not in Q[s], M[s] or a previous front
            if s != i & S[i] !in (Q[s] + M[s] + F[0 to x-1]):
              d = dominates(S[i], S[s])  //compute relationship
              if d = 3:                  //if S[i] dominates S[s]
                 D += S[s]               //insert S[s] into set D
                 Q[i] += S[s]            //add s to dominated by i
                 Q[i] += Q[s]            //copy dominated by s to i
                 if S[i] !in F[x]:       //if S[i] not sorted
                    n = i                //set next to assess to i
                    nf = s               //set start for next front to s
                    break                //break out of for loop
              else if d = 1:             //else if S[s] dominates S[i]
                 nf = i                  //set start for next front to i
                 D += S[i]               //insert S[i] into set D
                 Q[s] += S[i]            //add i to dominated by s
                 Q[s] += Q[i]            //copy dominated by i to s
              else:                      //else if mutual
                 M[s] += S[i]            //insert S[i] into mutual to S[s]
                 M[i] += S[s]            //insert S[s] into mutual to S[i]
                if S[i] !in D:           //if S[i] not dominated
                  n = i                  //set next to assess to i
      if S[s] !in D:                     //if S[s] not dominated
         F[x] += S[s]                    //insert S[s] into current front
         ns = ns + 1                     //increment the number sorted
      s = n                              //set the next to be assessed
   x = x + 1                             //increment the current front
return F                                 //return the set of fronts
```