Accelerating MOEA Non-dominated Sorting by Preserving Archival Relationships

Joseph Axilrod Craig
West Virginia University

Follow this and additional works at: https://researchrepository.wvu.edu/etd

Recommended Citation
https://researchrepository.wvu.edu/etd/193

This Thesis is brought to you for free and open access by The Research Repository @ WVU. It has been accepted for inclusion in Graduate Theses, Dissertations, and Problem Reports by an authorized administrator of The Research Repository @ WVU. For more information, please contact ian.harmon@mail.wvu.edu.
Accelerating MOEA Non-dominated Sorting by Preserving Archival Relationships

by

Joseph Axilrod Craig

Thesis submitted to the Benjamin M. Statler College of Engineering and Mineral Resources at West Virginia University in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science

Bojan Cukic, Ph.D.
Cynthia Tanner
Timothy Menzies, Ph.D., Chair

Lane Department of Computer Science and Electrical Engineering

Morgantown, West Virginia
2013

Keywords: non-dominated sorting, parallelization, concurrency, multi-objective optimization

Copyright 2013 Joseph Axilrod Craig
Abstract

Accelerating MOEA Non-dominated Sorting by Preserving Archival Relationships

by

Joseph Axilrod Craig
Master of Science in Computer Science
West Virginia University
Timothy Menzies, Ph.D., Chair

Non-dominated sorting is an important part of many multi-objective evolutionary algorithms (MOEAs). It is used to determine which individuals to keep in the archive of best individuals between generations and to evaluate fitness for breeding. Because this sorting is performed after every generation, it can contribute significantly to the running time of an MOEA. If the population size is sufficiently large, the time spent sorting the population can come to dominate an MOEA's running time.

The non-dominated sorting algorithms in the literature propose optimizations that focus on skipping unnecessary comparisons among individuals. While these algorithms provide different means for skipping these comparisons, none of them have considered adding mechanisms to preserve knowledge of relationships among the individuals from the archive that was sorted in an MOEA’s previous generation. Considering that the primary use for non-dominated sorting algorithms is in MOEAs, this seems a gross oversight.

In this thesis, I propose three new algorithms can accept partially, internally sorted populations: the Non-dominated Insertion Sort (NIS), the Non-dominated Merge Sort (NMS), and the Modified Dominance Tree Sort (MDTS). The strengths and weaknesses against each other and the algorithms in the literature are explored by testing how they impact the running time of the well-known MOEA NSGA-II [1] on several multi-objective problems (MOPs) and constrained optimization problems (COPs).
Acknowledgements

I thank my parents, David and Elaine Craig, for having raised me, provided me with a solid primary education, having encouraged me to excel, and even providing some proof-reading for this thesis.

I thank my grandmother, Gertrude Axilrod, for having such a vested interest in my development, the least expression of which was helping to pay for my college education.

I thank my parents-in-law, Stephen and Marie Guice, for providing encouragement and insight to the murky depths of graduate school, and for helping when the graduate student budget drew too tight.

I thank my wife, Simone Craig, for having joined me in life before graduate school, staying by my side through it, and for showing unsurpassed support, love, and friendship.

I thank my closest friends: Caleb Waggoner, John Guice VI, Norman Guice, and Stefan Guice, who all listened to me talk about my research ad nauseam.

I thank my advisor, Dr. Tim Menzies, for directing me to an interesting area of research and helping me know when to stop.

This thesis is proof that, with God, all things are possible.
# Contents

Acknowledgements .......................................................... iii  
List of Figures ............................................................... vi  
List of Tables ............................................................... vii  
List of Algorithms ........................................................ viii  
Notation ................................................................. ix  

## 1 Introduction  .............................................................. 1  

## 2 Background and Related Works ........................................... 4  
### 2.1 MOO and Pareto Domination ............................................. 4  
### 2.2 Non-dominated Fronts and Optimality .................................. 8  
### 2.3 MOEAs and NSGA-II ..................................................... 9  
### 2.4 Review of Non-dominated Sorting Algorithms .......................... 13  
#### 2.4.1 Goldberg’s Non-dominated Sort .................................... 13  
#### 2.4.2 Deb et al.’s Fast Non-dominated Sort ............................... 14  
#### 2.4.3 Jensen’s Divide-and-Conquer Sort ................................. 17  
#### 2.4.4 Shi et al.’s Better Non-dominated Sort ............................. 24  
#### 2.4.5 Fang et al.’s Dominance Tree Sort ................................. 29  
#### 2.4.6 Mazurek and Wesolkowski’s Limiting Index Sort .................. 36  
#### 2.4.7 Verma et al.’s Novel Non-dominated Sort ......................... 43  
#### 2.4.8 McClymont and Keedwell’s Climbing and Deductive Sorts ........ 46  
### 2.5 Conclusion .......................................................... 50  

## 3 NIS, NMS, and MDTS ......................................................... 51  
### 3.1 Non-dominated Insertion Sort (NIS) .................................... 52  
### 3.2 Non-dominated Merge Sort (NMS) ...................................... 56  
### 3.3 Modified Dominance Tree Sort (MDTS) ............................... 61  
### 3.4 Conclusion .......................................................... 65
## CONTENTS

4 Experiments and Results 66  
4.1 Experimental Design 66  
4.2 Results 68  
4.3 Conclusion 70  

5 Conclusion 72  

References 73
List of Figures

2.1 A dominance tree node. .................................................. 29
2.2 An example dominance tree. ............................................. 30
2.3 If a node is inserted into a dominance tree that dominates a non-leaf node, then the dominated node’s entire subtree can be set as the child of the new node. .................................................. 31
4.1 Threaded results. ............................................................ 69
4.2 Unthreaded results. ........................................................ 69
List of Tables

2.1 Descriptions and computational complexities of the NSGA-II pieces as described in Algorithm 1. ......................................................... 12
2.2 Pareto domination transitivity ....................................................... 25
4.1 The definition for the new MOP called Craig intended to stress algorithms that perform poorly with a large number of objectives. ......................... 67
# List of Algorithms

<table>
<thead>
<tr>
<th></th>
<th>Algorithm</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NSGA-II</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>NS</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>FNS</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>DCS Two-objective Sweep</td>
<td>17</td>
</tr>
<tr>
<td>5</td>
<td>DCS: rankPopulation</td>
<td>20</td>
</tr>
<tr>
<td>6</td>
<td>DCS: initializeWorseRanksByBetter</td>
<td>21</td>
</tr>
<tr>
<td>7</td>
<td>DCS: sweepWorseRanksAgainstBetter</td>
<td>23</td>
</tr>
<tr>
<td>8</td>
<td>Better function</td>
<td>26</td>
</tr>
<tr>
<td>9</td>
<td>BNS comparison logic</td>
<td>27</td>
</tr>
<tr>
<td>10</td>
<td>DTS</td>
<td>32</td>
</tr>
<tr>
<td>11</td>
<td>LIS</td>
<td>40</td>
</tr>
<tr>
<td>12</td>
<td>NNS</td>
<td>45</td>
</tr>
<tr>
<td>13</td>
<td>CS</td>
<td>48</td>
</tr>
<tr>
<td>14</td>
<td>DS</td>
<td>49</td>
</tr>
<tr>
<td>15</td>
<td>NIS</td>
<td>54</td>
</tr>
<tr>
<td>16</td>
<td>NMS, part I</td>
<td>57</td>
</tr>
<tr>
<td>17</td>
<td>NMS, part II</td>
<td>58</td>
</tr>
<tr>
<td>18</td>
<td>NMS, part III</td>
<td>59</td>
</tr>
<tr>
<td>19</td>
<td>MDTS, part I</td>
<td>62</td>
</tr>
<tr>
<td>20</td>
<td>MDTS, part II</td>
<td>63</td>
</tr>
</tbody>
</table>
# Notation

The following notations are used in this thesis. Many of these notations are common, but some are unique.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\bar{x}$</td>
<td>a vector</td>
</tr>
<tr>
<td>$</td>
<td>x</td>
</tr>
<tr>
<td>$x_i$</td>
<td>if $\bar{x}$ is a vector, $\bar{x}$’s $i^{th}$ element</td>
</tr>
<tr>
<td>:</td>
<td>“such that”</td>
</tr>
<tr>
<td>$\bar{u} \prec \bar{v}$</td>
<td>decision vector $\bar{u}$ dominates decision vector $\bar{v}$</td>
</tr>
<tr>
<td>$\bar{u} \not\prec \bar{v}$</td>
<td>decision vector $\bar{v}$ is non-dominated in reference to decision vector $\bar{u}$</td>
</tr>
<tr>
<td>$\bar{u} \bowtie \bar{v}$</td>
<td>decision vectors $\bar{u}$ and $\bar{v}$ are mutually non-dominated</td>
</tr>
<tr>
<td>nil</td>
<td>an empty linked list</td>
</tr>
<tr>
<td>list1 + list2</td>
<td>list concatenation</td>
</tr>
<tr>
<td>$\langle 1, 2, 3 \rangle$</td>
<td>a literal vector in equations or linked list in algorithms</td>
</tr>
<tr>
<td>{ $x \in A$</td>
<td>set builder notation, the set of $A$’s items such that $\text{cond}(x)$ is true</td>
</tr>
<tr>
<td>$\langle \text{expr}(i) \forall i \in P \rangle$</td>
<td>list comprehension, similar to set builder notation</td>
</tr>
<tr>
<td>$x \leftarrow y$</td>
<td>the variable $x$ is assigned the value of $y$</td>
</tr>
<tr>
<td>$(a, b, c)$</td>
<td>a tuple</td>
</tr>
<tr>
<td>$(a, b, c) \leftarrow (1, 2, 3)$</td>
<td>tuple extraction; $a \leftarrow 1$, $b \leftarrow 2$, and $c \leftarrow 3$</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Life is complicated. Tasks from every area of our lives vie for our limited time, each with their own advantages and disadvantages for us taking them on or leaving them for later or never. We are constantly playing a balancing game, trying to meet multiple competing goals as we choose what tasks to address, how we attack them, and even in what order. This complexity seems fractal, as it permeates every level from the most trivial to the most dire of situations. A lot of humanity’s thought and research has been focused on trying to determine The Best Way to solve each of our problems. However, even this effort has birthed new complexity; people are not able to agree on The Best Way in very many, if any, facets of life. The complexity of life is such that most problems have multiple equally valid solutions that trade off on achieving different goals within the problem, and which ones makes sense depend upon factors outside of the problem.

This is the inspiration for the field of Multi-Objective Optimization (MOO). The goal of MOO is to aid experts in solving real-world problems that have multiple goals to solve simultaneously by modelling the problem, testing many solutions to the problem, comparing the solutions, and generating a set of solutions which cannot be said to be innately worse than any other solution. Experts can then pick the solution from this Pareto optimal set that they see as best suiting their circumstances. By seeking the Pareto optimal set as opposed to trying to find a solution that matches a certain goal, experts can get a sense of the “shape” of their problem and see solutions that they might not have otherwise considered.

A large subset of MOO is the research of Multi-Objective Evolutionary Algorithms (MOEAs). MOEAs use mechanics inspired by macro-evolution to “evolve” a population of solutions (also referred to as “individuals”) in the search for the Pareto optimal set. There
are numerous papers on the subject, but Coello Coello gives a well-written introduction to the concepts and landmark algorithms in [2]. Many of the MOEAs developed within the past decade draw their inspiration from Deb et al.’s Non-dominated Sorting Genetic Algorithm II (NSGA-II) [1]. It evolves successive generations of individuals from an archive of the best solutions found over the course of its execution, and the archive at the end of NSGA-II’s execution is the algorithm’s approximation of the Pareto optimal set. Which solutions are kept in the archive is determined using “non-dominated sorting”.

Non-dominated sorting is a partial ordering of individuals based upon how well each individual accomplishes each of the objectives of the problem it is intended to solve. The dominated refers to the concept of Pareto domination, which is the mechanism used to determine that one solution is better than another. Beyond determining a partial ordering, non-dominated sorting divides individuals into “fronts” of mutually non-dominated individuals so that their domination depth affects how fit the individual is deemed to be for breeding. As such, it is more complicated than typical sorting, because an individual’s position in a listing is not the final word of how valuable that individual is.

Because different non-dominated sorts will divide the same population into the same fronts, an MOEA that uses non-dominated sorting to determine its archive can exchange one sort for another without any change to its end result. Which sort is used does not matter in terms of the functionality of the MOEA. And yet the non-dominated sort represents much, if not most, of the work an MOEA performs each generation\(^1\). This means that developing a faster non-dominated sort will improve the performance of many MOEAs at once and open the door to solving more complicated problems and exploring more solutions simultaneously.

Some research has already been done in the work of making better non-dominated sorts. Researchers have used techniques such as avoiding unnecessary comparisons, using tricks based upon the original ordering of the individuals in a population, divide-and-conquer, and parallelization to reduce the running time and computational complexities of their non-dominated sorting algorithms. Each of these techniques has made great contributions toward solving the problem of non-dominated sorting, but surprisingly none of the work I have surveyed has considered optimizing their algorithms using knowledge about the context in which they are run. If these algorithms are used to update an archive in an MOEA like

\(^{1}\)This statement excludes the computational complexity of scoring individuals as solutions to their problem. While this certainly affects an MOEA’s running time, it is not definitive to the algorithm. In addition, for many of the test problems used to evaluate an MOEA, this computational complexity is still overwhelmed by that of the non-dominated sorting.
NSGA-II, they are being used to compare individuals between two sets of individuals to get a new ordering based upon the two sets. The individuals in the archive have already been compared against each other, so why should a non-dominated sort have to make these comparisons again?

I developed two algorithms to explore this concept: the Non-dominated Insertion Sort (NIS) and the Non-dominated Merge Sort (NMS). Each of these sorts generates the front divisions for the individuals so that new individuals to be compared with the previous results are added into the ordering as opposed to determining all the previously determined relationships again. NIS adds a single individual into a pre-existing ordering and thus is suitable for an MOEA that uses an active archive update policy (making a newly bred child available for breeding as soon as it has been scored). NMS, intended for MOEAs that update the archive with all the children in a generation at once, is a parallelized divide-and-conquer sort that sorts the new individuals to add in reference to each other, then uses that information while inserting them into the archive with the NIS. In addition, I explored the possibility of modifying the usage of Fang et al.’s Dominance Tree Sort (DTS) [3] so that it can take advantage of successive calls to some degree.

The rest of this thesis is as follows. Chapter 2 discusses the background material glossed over in this introduction and describes the non-dominated sorts in the literature in detail. Chapter 3 presents an argument for why maintaining domination information inbetween calls to a non-dominated sort should improve running time performance, detailed design and analysis of my NIS and NMS algorithms, and a brief discussion of how the DTS can provide this functionality to some degree. Chapter 4 describes the experiments I performed to test these ideas and presents my findings. Chapter 5 concludes my thesis with another summary of this thesis and suggestions for future work.
Chapter 2

Background and Related Works

In this chapter, I lay out the foundations for the problem of non-dominated sorting. I explain what the name means and the math behind the problem, the context in which these algorithms are used, and several of the algorithms that appear in the literature. The algorithm sections contain detailed explanations of how these algorithms correctly solve the sorting problem, their strengths and limitations, and how well they perform.

2.1 MOO and Pareto Domination

Multi-objective optimization (MOO) is a field in which researchers design algorithms to solve problems that have multiple goals of equal importance. It is related to optimization, which is solving problems with either only one goal or multiple goals of differing levels of importance. Optimization is relatively simple in comparison to MOO; one can eliminate potential solutions by optimizing each goal in descending order of importance until only one is left. This works because it becomes a trivial matter to determine which solution is better than another: the one that best completes the most important goal (or goals in the case of ties) is the best solution. Without a hierarchy for the goals, comparing solutions becomes difficult, and the end result is that there is frequently more than one “best” answer.

To demonstrate this, consider a scenario in which a student is trying to determine how to divide his study time between two classes to maximize his grades. He believes that by allocating four hours to his first class and six hours to his second, he can get a $B$ in the first and an $A$ in the second. However, if he spends five hours in each, he can get an $A$ in the first and a $B$ in the second. He spends the same amount of time studying to get the same two
grades, but he gets different grades for the two classes. If there is no way of telling which class is more important, there is also no way to say whether getting \(\langle B, A \rangle\) as opposed to \(\langle A, B \rangle\) is better. They are equally good solutions to his problem.

Intuitively the assertion about the solutions above makes sense, but this intuition needs to be generalized to account for multiple dimensions to be useful for MOO. This generalization is called Pareto domination, and is defined as follows:

**Definition 1** (Pareto Domination). Let \(\vec{u}\) and \(\vec{v}\) be vectors of the decisions that represent individuals (solutions) for a multi-objective problem minimization problem. Let \(\vec{f}\) be a vector of functions known as scoring functions that determines how well the solutions accomplish the problem’s objectives, \(\vec{f}(\vec{x})\) be the vector of the individual \(\vec{x}\)’s objective scores, and \(f_i(\vec{x})\) be the \(i^{th}\) element of \(\vec{f}(\vec{x})\). Without loss of generality, \(\vec{u}\) dominates \(\vec{v}\) when

\[
\vec{u} \prec \vec{v} \iff \forall i, f_i(\vec{u}) \leq f_i(\vec{v}) \land \exists j : f_j(\vec{u}) < f_j(\vec{v})
\] (2.1)

This definition supports the intuition mentioned before. Returning to the example above, let’s say that the student believes that if he studies for nine hours for his first class and one for the second, he will get an \(A\) in the first class and a \(D\) in the second. Using the notation from equation 2.1, we can say \(\vec{u} = \langle 5, 5 \rangle\), \(\vec{f}(\vec{u}) = \langle A, B \rangle\), \(\vec{v} = \langle 9, 1 \rangle\), and \(\vec{f}(\vec{v}) = \langle A, D \rangle\). While the student gets an \(A\) for his first class in both solutions, he gets a better grade for the second class if he picks \(\vec{u}\) instead of \(\vec{v}\). Therefore, \(\vec{u} \prec \vec{v}\).²

If a solution does not dominate another, it is said to be non-dominated toward the other. Naturally, the definition of non-domination is the opposite of the definition of domination:

**Definition 2** (Non-domination). Let all the assumptions be as in Definition 1. Without loss of generality, \(\vec{u}\) is non-dominated in relation to \(\vec{v}\) when

\[
\vec{u} \not\prec \vec{v} \iff \exists i : f_i(\vec{v}) < f_i(\vec{u})
\] (2.2)

The intuition behind non-domination is even simpler than domination. If there is even one goal which one solution performs more poorly than the other, it cannot be better than the first.

Note that \(\vec{u} \prec \vec{v}\) requires that \(\vec{v} \not\prec \vec{u}\), but \(\vec{u} \not\prec \vec{v}\) does not require that \(\vec{v} \prec \vec{u}\). It is possible that \(\vec{u} \not\prec \vec{v}\) and \(\vec{v} \not\prec \vec{u}\). This is called mutual non-domination and is defined below.

²Note that this problem is a maximization problem instead of a minimization problem as stated in Definition 1. Usually, authors whose research focus on maximization problems would denote the domination relationship as \(\vec{u} \succ \vec{v}\). Because the problems I use in this thesis are all minimization problems, I decided to consistently use the minimization symbol \(\prec\).
Definition 3 (Mutual Non-domination). Let all the assumptions be as in Definition 1. Without loss of generality, $\vec{u}$ and $\vec{v}$ are mutually non-dominated when

$$\vec{u} \rightleftharpoons \vec{v} \iff \exists i : f_i(\vec{u}) < f_i(\vec{v}) \land \exists j \neq i : f_j(\vec{v}) < f_j(\vec{u})$$ (2.3)

Algorithm 8 in Section 2.4.4 shows that it is not difficult to perform comparisons for a pair of individuals simultaneously. What is important is that both relationships must be known to make any determinations about how two individuals compare. This point will be very important later in this section.

A problem that has multiple objectives and no other considerations is referred to as a multi-objective problem (MOP). However, most problems have an additional concern of constraints; very few problems have only solutions that are feasible or valid. For example, a company may try to increase its revenue by allocating some of its capital to advertising, but it cannot allocate more capital than it has. Such a problem is called a constrained optimization problem (COP). The definitions for domination, non-domination, and mutual non-domination are similar between MOPs and COPs, but COPs add the following considerations:

- A solution that violates none of the problem’s constraints is valid.
- A solution that violates any of the problem’s constraints is invalid.
- A valid solution dominates every invalid solution, regardless of their respective objective scores.
- The relationships between two valid solutions are decided solely by their objective scores.
- The relationships between two invalid solutions are decided first by the extent to which each one violates their constraints. Objective scores are only considered if the two individuals’ degree of constraint violation is equivalent.

Note that the last bullet point is not as specific as the others. While MOP domination is solidly defined, incorporating constraints is not, and is an area of research in its own right called constraint handling [4–9]. The approaches proposed for handling constraints are rather complicated, and constraint handling is not a focus of this paper, so I instead use the following organic definition.
CHAPTER 2. BACKGROUND AND RELATED WORKS

**Definition 4** (Constrained Domination). Let all the assumptions be as in Definition 1. In addition, let $\vec{g}$ be a vector of constraint functions that determines to what degree an individual violates each of the problem’s constraints; an individual must score 0 for each $g$ to be valid. Then

$$
\vec{u} \prec \vec{v} \iff \left( \forall i, g_i(\vec{u}) \leq g_i(\vec{v}) \land \exists j : g_j(\vec{u}) < g_j(\vec{v}) \right) \lor \\
\left( \forall k, g_k(\vec{u}) = g_k(\vec{v}) \land \forall l, f_l(\vec{u}) \leq f_l(\vec{v}) \land \right) \\
\exists m : f_m(\vec{u}) < f_m(\vec{v}))
$$

(2.4)

This approach treats the constraints as objectives that must be optimized before the objective scores are considered. This naturally ensures that valid individuals dominate invalid individuals. Other techniques in the literature allow for different types of constraints, such as hard and soft (whether the constraint must be met or can be relaxed) or inequality, strict inequality, or equality constraints (the score must be at most a certain value, or less than a certain value, or equal to a certain value). In the end, though, they all entail ensuring that valid individuals dominate invalid individuals and that the degree of invalidity determines which of the two invalid individuals dominates the other. Which technique is chosen may change some of the relationships determined, but a consistent scheme is all that is required for the rest of this paper.

Pareto domination is the technique used for comparing individuals throughout this paper, but it is not the only one used in MOO. Pareto epsilon dominance is a variation of Pareto domination that loosens the conditions for domination such that a small number (denoted as $\epsilon$) is subtracted (or added in the case of maximization objectives) to the second individual’s objective scores before any comparisons are made [10,11]. Epsilon dominance is useful because it recognizes the constraints on floating-point number representation. Another proposed scheme is $\alpha$-domination which imposes further considerations beyond Pareto domination. The ratio in trade-offs among the solutions’ objectives must meet a certain ratio or else they cannot be considered to be non-dominated [12]. For example, the grade vectors $\langle A, F \rangle$ and $\langle B, C \rangle$ are mutually non-dominated, but clearly getting $\langle B, C \rangle$ is better; $\alpha$-domination can determine this through proper parameterization. In the case of non-dominated sorting, it does not matter whether one of these schemes or one of the many others is used so long as it generates consistent comparisons among individuals; which domination scheme should be used depends upon the context of the non-dominated sorting algorithm’s usage. Pareto dom-
CHAPTER 2. BACKGROUND AND RELATED WORKS

ination with constraint handling as described in equation 2.4 is used throughout the paper because it is intuitive, does not require parameterization, and some of the non-dominated sorts from the literature described in this paper and used in my experiments are optimized by relying on Pareto domination.

2.2 Non-dominated Fronts and Optimality

As mentioned in Chapter 1, non-dominated sorting entails imposing a partial ordering on a collection of solutions to a MOP or COP. Practically, the ordering desired is one in which the best solutions are listed first, so the ordering is determined by listing solutions such that no solution is listed after a solution that it dominates. However, this has the side effect that an individual’s position in the listing provides no real clue to the relative quality or desirability of any solution. For example, if one has a collection in which every pair of individuals is mutually non-dominated, then every permutation of the collection conveys the same level of meaning. This means that non-dominated sorting needs to generate some information beyond the ordering to represent the hierarchy of solutions based upon their individual levels of quality.

This hierarchy divides solutions into non-dominated fronts. A collection of solutions is divided into mutually exclusive subsets called fronts such that every solution in a front is mutually non-dominated with every other solution in that front. Furthermore, these fronts are ranked and ordered in descending order of quality, and every solution is placed in the highest-ranked front possible while maintaining the requirement that no solution can be ordered after a solution it dominates. This concept can be represented mathematically as follows.

Definition 5 (Ranking). Let \( P \) be a collection of solutions for any MOP or COP. Every solution \( \vec{u} \in P \) is placed into a non-dominated front based upon its rank as determined by the following formula.

\[
\text{rank}(\vec{u}, P) = \begin{cases} 
1 & \neg \exists \vec{v} \in P : \vec{v} \prec \vec{u}, \\
\max \{ \text{rank}(\vec{v}, P) | \vec{v} \in P \land \vec{v} \prec \vec{u} \} + 1 & \text{otherwise.}
\end{cases}
\] (2.5)
The highest ranked non-dominated front in $P$, and all the solutions that populate it, are ranked 1. Any lower non-dominated fronts and the solutions that populate them are ranked with larger natural numbers.

A collection’s ordering is determined by listing the solutions from each front in descending order. Because solutions in the same front are mutually non-dominated with every other solution in that front, solutions from the same front can be listed in any permutation.

A solution’s rank as determined by formula 2.5 represents that solution’s relative quality as compared to the rest of the solutions in the collection. The individuals that are ranked 1 are not dominated by any other solution in the collection. Theoretically, if the set of solutions in question is the set of all possible solutions to a given problem, then the solutions that are ranked 1 are non-dominated in respect to the entire solution space; there are no better solutions. These solutions are called optimal, and the set of optimal solutions are the Pareto optimal set. The Pareto optimal set’s mapping into objective space is referred to as the Pareto optimal front. It is unlikely that any given collection to be sorted will actually be the entire set of solutions, so it is difficult to say that the solutions in its first front are optimal, but we can say that they are locally optimal.

Now non-dominated sorting can be defined.

**Definition 6** (Non-dominated Sorting). *Non-dominated sorting is the process of ranking every solution to a given problem in a collection using equation 2.5 and ordering the solutions in ascending order of their ranks.*

This definition is properly agnostic in regards to comparison scheme. The case could be made that the ordering of the solutions may not strictly be necessary since the rankings are more informative than the final ordering, but having the solutions sorted as well as ranked can improve the performance of MOEAs.

### 2.3 MOEAs and NSGA-II

Non-dominated sorting is an interesting problem in its own right, but in the literature, it is constantly paired with MOEAs. *Multi-objective evolutionary algorithms* are algorithms that approximate the Pareto optimal set through successive applications of operators inspired by natural processes to a collection of random solutions to a problem. Encoded solutions to
Algorithm 1 NSGA-II

Require: a population size $N$, a number of generations $G$; a problem’s decision sets or domains $\vec{d}$; functions to choose parents, crossover parents, mutate individuals, score individuals, sort a population, and select the $N$ best individuals from a population; and any necessary configurations for those functions

Ensure: an approximation of the Pareto optimal set, their associated constraint and objectives scores, and their relative ranks

1: archive $\leftarrow$ nil
2: for $i = 1$ to $N$ do
3: \hspace{1em} archive $\leftarrow$ archive + generate($\vec{d}$)
4: archiveScores $\leftarrow$ map(score, archive)
5: (archive, archiveScores, archiveRanks) $\leftarrow$ sort(archive, archiveScores)
6: for $i = 1$ to $G$ do
7: \hspace{1em} children $\leftarrow$ nil
8: \hspace{2em} repeat
9: \hspace{3em} parents $\leftarrow$ choose(archive, archiveScores)
10: \hspace{3em} children $\leftarrow$ children + map(mutate, crossover(parents))
11: \hspace{1em} until $|\text{children}| = N$
12: \hspace{1em} childrenScores $\leftarrow$ map(score, children)
13: \hspace{1em} joined $\leftarrow$ archive + children
14: \hspace{1em} joinedScores $\leftarrow$ archiveScores + childrenScores
15: \hspace{1em} (joined, joinedScores, joinedRanks) $\leftarrow$ sort(joined, joinedScores)
16: \hspace{1em} (archive, archiveScores, archiveRanks) $\leftarrow$ select(joined, joinedScores, joinedRanks, $N$)
17: return (archive, archiveScores, archiveRanks)

a given problem, called individuals, are collected in what is called a population and are transformed over successive iterations called generations to progress the population as a whole toward the Pareto optimal set. Evolutionary algorithms use strategies of mutation, making small random changes to individuals, and selection, culling the population to hold only the most fit or interesting individuals, as operators to guide the search of the solution space. Genetic algorithms, a common subset of evolutionary algorithms, add a crossover operation
CHAPTER 2. BACKGROUND AND RELATED WORKS

which is used to randomly combine two or more individuals in the hopes of encouraging the
strengths of a population to continue to the next generation [13]. EAs are commonly used
to stochastically search problem spaces too large for brute force and for which there is little
or no prior knowledge about which solutions are probably best to heuristically guide the
search. MOPs and COPs tend to fit this description nicely.

One of the best known MOEAs is Deb et al.’s Non-dominated Sorting Genetic Algorithm
II (NSGA-II) [1]. It is a highly influential algorithm; IEEE Xplore listed [1]’s citation count
at 3,384 at the time this was written [14]. As its name suggests, it is an MOEA that uses
genetic algorithm crossover and mutation to create subsequent populations to explore and
uses non-dominated sorting as part of its selection. It also serves as an excellent example as
to why non-dominated sorts need to be optimized, so it is prudent to describe the algorithm
here.

Algorithm 1 shows a simplified, modular description of NSGA-II. Table 2.1 describes what
each of the pieces does and its computational complexity. NSGA-II maintains a population of
N individuals in an archive, which is a term in elitist genetic algorithms for a collection of the
best solutions for a problem found over the course of the execution [1]. It uses the fitnesses
of the solutions in the archives as determined by its previous iteration to choose enough sets
of parents to breed N new children using the chosen crossover and mutation operators. The
children are scored, and then the archive and children are combined into one population and
sorted with a non-dominated sorting algorithm to determine which individuals to keep. The
best N individuals are kept for the archive for the next generation based upon their ranks
and whatever selection scheme is used to encourage diversity. After a preselected number
of generations $G$ has passed, the archive is returned as the collection of solutions that were
found to be best.

Algorithm 1 is written modularly because NSGA-II is a higher-level approach to problem
solving than any of the pieces used to solve the MOP or COP in question. State-of-the-art
research in any of the fields represented by these pieces, such as crossover operators or
non-dominated sorting, can almost certainly be incorporated into NSGA-II to improve its
performance without being able to say that the MOEA is no longer NSGA-II. Perhaps this
modularity and simplicity is part of the reason for its success in the literature.

Table 2.1 shows that non-dominated sorting dominates NSGA-II’s computational com-
plexity. [1] describes NSGA-II’s running time as $O(GMN^2)$ with $M$ as the number of objectives,
the constraints normalized into a single score in one $O(|\vec{g}|N)$ pass per generation, and
CHAPTER 2. BACKGROUND AND RELATED WORKS

<table>
<thead>
<tr>
<th>Piece</th>
<th>Description</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>generate</td>
<td>builds an individual by randomly selecting a value for each decision from that decision’s set or range</td>
<td>$O\left( N \mid \vec{d} \mid \right)$</td>
</tr>
<tr>
<td>score</td>
<td>returns an individual’s constraint and objective scores for the problem’s $\vec{f}$ and $\vec{g}$</td>
<td>depends on problem</td>
</tr>
<tr>
<td>sort</td>
<td>returns a reordered population with the corresponding scores and ranks ordered to match the population based upon the passed population and individual scores</td>
<td>best known worst case is $O \left( \left( \mid \vec{f} \mid + \mid \vec{g} \mid \right) N^2 \right)$</td>
</tr>
<tr>
<td>choose</td>
<td>selects the number of parents required for one call of the crossover operator</td>
<td>$O \left( 1 \right)$</td>
</tr>
<tr>
<td>crossover</td>
<td>returns one or more new individuals by combining the decisions of the passed parents</td>
<td>$O \left( \mid \vec{d} \mid \right)$</td>
</tr>
<tr>
<td>mutate</td>
<td>returns a copy of the passed individual with possibly slightly modified decision scores</td>
<td>$O \left( \mid \vec{d} \mid \right)$</td>
</tr>
<tr>
<td>select</td>
<td>returns the best $N$ individuals based first upon as many of the best fronts can fit completely within the archive and the remaining slots filled with the least crowded individuals from the next best front</td>
<td>$O \left( \mid \vec{f} \mid N \log N \right) \left[ 1 \right]$</td>
</tr>
</tbody>
</table>

Table 2.1: Descriptions and computational complexities of the NSGA-II pieces as described in Algorithm 1.

the constant multiple of time spent scoring ignored. With the more flexible constraint handling described in Definition 4, the running time changes slightly to $O \left( G \left( \mid \vec{f} \mid + \mid \vec{g} \mid \right) N^2 \right)$. As experts use larger $G$’s or deal with problems with more constraints and objectives or require nontrivial time to score each individual, the only sure way to decrease NSGA-II’s running time effectively is to reduce the running time of the non-dominated sort. Depending upon the running time improvement, problems that are currently too prohibitive to be solved by MOEAs might become manageable.
CHAPTER 2. BACKGROUND AND RELATED WORKS

2.4 Review of Non-dominated Sorting Algorithms

This section presents a thorough review of many of the non-dominated sorts in the literature. Each section will explain how the authors intended for their algorithm to improve upon the state of the art, provide pseudocode for their algorithms, explain how the algorithms work, and conclude with notes about their computational and memory complexities. For simplicity, I will refer to $|\vec{f}| + |\vec{g}|$ as $M$ and the number of decisions plus $M$ as $C$.

2.4.1 Goldberg’s Non-dominated Sort

Goldberg published the first non-dominated sorting algorithm in 1989 [15]. Srivinas and Deb reiterated this algorithm in their paper presenting the Non-dominated Sorting Genetic Algorithm in 1994 [16]. This algorithm, to which I will refer as NS, is an intuitive algorithm that follows the structure Equation 2.5: it finds all the non-dominated solutions, then the individuals dominated only by the non-dominated individuals, and so on until all the individuals are ranked. Pseudocode for NDS is given in Algorithm 2.

A more thorough explanation of Algorithm 2 is as follows. It starts by assuming that every individual in the passed population is part of the first front. It then compares every individual that has not been successfully placed into a rank yet against every other individual that could be a member of the current front. If an individual that is assumed to be in the current front is found to dominate the current individual, it is then assumed to be in the next front instead. If no individual is found that dominates it in the current front, then it is part of the current front, and is added to the ordering as such. After all the individuals that have not been added to the ordering have been tested, the front counter is incremented, and it starts checking for membership in the second front. This process continues until all the individuals are placed.

NS has a running time of $O(MN^3)$ and a memory requirement of $O(CN)$. In the best case, all the individuals are mutually non-dominated, and it compares each pair of individuals once in each direction for a running time of $O(MN^2)$. If the population has a lot of fronts, then the same pairs of individuals may be compared several times throughout NS’s execution.
CHAPTER 2. BACKGROUND AND RELATED WORKS

2.4.2 Deb et al.’s Fast Non-dominated Sort

Deb et al. published NSGA-II in 2002 [1]. NSGA-2 was developed as an improvement on NSGA in both results generated and running time efficiency. Getting a speed boost required a faster non-dominated sort, and Deb et al. included such an algorithm in [1]. Their Fast Non-dominated Sort (referred to as FNS from now on) addresses the weakness of NS by guaranteeing that each pair of individuals is checked for domination relationships exactly once.

Algorithm 3 shows pseudocode explaining the FNS. It maintains records for each individual, tracking how many individuals dominate it and a list of the indices of the individuals that it dominates. Starting with zeroes for the dominated counts and empty lists for the
Algorithm 3 FNS

**Require:** A population $P$ and the individuals’ scores $S$

**Ensure:** a list of the individuals from the initial population ordered by front, a list of the individuals’ scores, and a list of the individuals’ ranks, all correlated by index

```plaintext
1: dominatedByCounts ← \{0 for $i = 1$ to $|P|$\}
2: dominatesLists ← \{nil for $i = 1$ to $|P|$\}
3: (currentFront, nextFront, order, scores, rank, ranks) ← (nil, nil, nil, nil, 1, nil)
4: for $i = 1$ to $|P|$ do
5:     for $j = i + 1$ to $|P|$ do
6:         if $P_i \prec P_j$ then
7:             dominatedByCounts$_j ←$ dominatedByCounts$_j + 1$
8:             dominatesLists$_i ←$ dominatesLists$_i + j$
9:         else if $P_j \prec P_i$ then
10:            dominatedByCounts$_i ←$ dominatedByCounts$_i + 1$
11:            dominatesLists$_j ←$ dominatesLists$_j + i$
12:     if dominatedByCounts$_i = 0$ then
13:         currentFront ← currentFront + $i$
14:     repeat
15:     for all $i$ in currentFront do
16:         order ← order + $P_i$
17:         scores ← scores + $S_i$
18:         ranks ← ranks + rank
19:     for all $j$ in dominatesLists$_i$ do
20:         dominatedByCounts$_j ←$ dominatedByCounts$_j - 1$
21:     if dominatedByCounts$_j = 0$ then
22:         nextFront ← nextFront + $j$
23: (currentFront, nextFront, rank) ← (nextFront, nil, rank + 1)
24: until currentFront = nil
25: return (order, scores, ranks)
```
dominating lists, FNS then compares every pair of individuals to see whether one individual in the pair dominates the other. If so, then the dominated individual’s dominated count is incremented, and the dominating individual adds the index of the dominated individual to its dominating list. Any individual that has a dominated count of 0 after it has been compared with every other individual, then its index is added to a list of indices of individuals that belong in the current front.

After all the comparisons, the algorithm begins processing each front as can be determined by the dominated counts and dominating lists. It loops over each member of the current front, starting with the one determined during the comparisons, and adds them to the ordering. Then, it iterates over the dominating lists of those members, decrementing the dominated counts for each individual the members dominate. Whenever an individual’s dominated count drops to 0, that means that the lowest-ranked individual that dominates it has been placed into the current front; by Equation 2.5, that means that the individual belongs in the next front. So, as individuals’ dominated counts reach 0, their indices are added to a list for the next front. Once a front has been processed, it is replaced by the next front, the next front is reset to an empty list, and the rank is incremented. Thus, the loop will stop once the last rank has been processed, because the current front will receive an empty list after that pass.

FNS requires $O(MN^2)$ running time and $O(N^2)$ memory for storing the dominating lists [1]. It provides a solid improvement over the performance NS by eliminating superfluous comparisons. However, Wong showed that the design of this algorithm is such that a major performance boost beyond the computational complexity can be achieved by splitting the comparisons into threads [17]. This works because comparing individuals does not need to access any mutable memory, and updating the dominated counts and dominating lists can be done easily with lightweight compare-and-swap (CAS) loops. It seems that a larger population is likely to decrease the likelihood of two threads competing for the same shared memory due to the decreased probability of two threads operating on the same individual simultaneously, which means that larger populations might get better performance boosts from threads than smaller ones. Wong wrote that the competition for shared memory for the second part of FNS would probably be too high to gain a performance boost through parallelization, and I have determined this to be true through my own experimentation. For the purpose of differentiation, I will call the parallelized FNS “PNS”, or Parallelized Non-dominated Sort, from now on.
CHAPTER 2. BACKGROUND AND RELATED WORKS

Algorithm 4 DCS Two-objective Sweep

Require: indices: a list of indices referring to individuals’ decisions, scores, and ranks in $P$, $S$, and ranks; the same index identifies the same individual in all three lists

Require: ranks: a list of values indicating into which non-dominated front an individual in $P$ is expected to go; by default, it should be $\langle 1 \forall x \in \text{indices} \rangle$

Ensure: ranks now holds the appropriate ranks for each individual referred to in indices based upon the first two objectives and the initial ranks passed in

1: function rankByTwoObjectiveSweep($P$, $S$, indices, ranks)
2: ordered $\leftarrow$ sortByFirstTwoObjectives($S$, indices)
3: fronts $\leftarrow$ empty map of integers to integers
4: for all $i$ in ordered do
5: \hspace{1em} $j \leftarrow \max (\{ x \in \text{fronts keys} \mid \text{a\_dominates\_b}(S[\text{fronts}[x]], S[i]) \})$
6: \hspace{1em} if $k \neq \text{null} \land \text{ranks}[i] \leq k$ then ranks$[i] \leftarrow k + 1$
7: \hspace{1em} fronts[ranks$[i]$] $\leftarrow i$
8: function a\_dominates\_b(a, b)

2.4.3 Jensen’s Divide-and-Conquer Sort

In 2003, Jensen published [18], recommending improvements to many MOEAs, including NSGA-II. In this paper, he proposed a divide-and-conquer sort for MOPs that modifies algorithms by Kung et al. that find the first non-dominated front of a population [19] to be able to determine all the fronts at once. This sort takes advantage of a geometric analysis of Equations 2.1 and 2.2 to avoid domination comparisons that can be quickly determined to yield that one individual is not dominated by another.

Jensen’s sort, which I will call DCS, starts from a very basic principle. From Equation 2.2, we know that an individual cannot be dominated by any individual with a worse score in that objective. This means that sorting a population by any objective from best to worst score means that no individual can dominate any individuals that come before it in the ordering (except for the case of a tied score). This allows for checking domination in only one direction and reduces the number of objectives that have to be checked when making the comparison. In the special case that there are only two objectives, it is possible to place individuals in the correct fronts by sorting them by one objective then comparing them by
the other. Algorithm 4 does this with what Jensen called a sweep and forms the base case of DCS.

\texttt{rankByTwoObjectiveSweep} in Algorithm 4 modifies a list of individuals’ ranks as follows. First, it orders a list of indices referring to individuals by the first two objectives in that order. It then builds an empty map to enable looking up the index of the last individual found in a front by the front’s rank. Then it iterates over the indices list, trying to find the rank of the most dominated front that contains an individual that dominates the current individual. This can be done by comparing only with the last individual inserted into the front because of the ordering by the first objective; the new individual has at best a tied first objective score with any individual in the fronts, and the last individual in any front will therefore have the lowest second objective score in that front. If a front is found that dominates the current individual, and its rank is greater than or equal to the individual’s current rank, then the individual’s rank is set to one plus the dominating front’s rank. Then the individual is set as the last individual found for its rank.

The idea that sorting on one objective helps eliminate unnecessary comparisons and the ability to determine rankings quickly for two objectives work together to form the recursive DCS algorithm, which is presented in Algorithms 5 through 7. An oversimplified summary of the algorithm is to split the population so that no individual in the first part can be dominated by any individual in the second part, determine the ranks of the individuals in the first part, use those ranks to initialize the ranks of the individuals in the second part, then determine the final ranks for the individuals in the second part.

The DCS’s entry point is the \texttt{DCS} function in Algorithm 5. It first builds an indices list that uses the same number to access an individual’s decision scores, objectives scores, and ranks in the three lists \textit{P}, \textit{S}, and \textit{ranks}. It then builds the list \textit{ranks} to hold the currently known ranks for each individual, starting with the value 1 for each. It calls the \texttt{rankPopulation} function to determine all the ranks for the individuals in the passed population using all the objectives. Finally, it builds the sort of ordering that NS and FNS return by sorting the population, the individuals’ scores, and the ranks by the ranks in ascending order (from the best rank, 1, on up).

\texttt{rankPopulation} is the function that determines the rankings for a population subset with the assumption that either the rest of the population will not impact this subsets’ rankings at all or that any impacts the rest of the population would have is already reflected in the passed rankings list. It starts by checking for the three base cases. If the passed
population subset, represented by the received indices list $i$, has only one individual, then there is nothing more to be done. If the population subset has two individuals, they can be compared directly for the first $M$ objectives to establish their rankings (the structure of this function allows objectives to be safely ignored if $M < |P|$). If one of the individuals dominates the other, the $note_i.dominated_by_j$ function sets the dominated individual’s rank to the other individual’s rank plus 1 only if doing so increases its rank. If there are more than two individuals, but only two objectives remain for consideration, then the sweep presented in Algorithm 4 is used to update the rankings.

If the population subset does not fall into any of the base cases, then rankPopulation checks to see if every individual in the subset has the same score for the $M$th objective. If so, then the $M$th objective cannot be used within this subset to determine any dominations, so rankPopulation calls itself with the same arguments as it received except that it removes the $M$th objective from consideration by passing $M - 1$ for the objective count. Otherwise, it uses splitForObjectiveM to divide the population subset into two new subsets. These subsets are split using the individuals’ $M$th objective scores so that all of the individuals in the first subdivision have a better score than any individual in the second, each subdivision is as large as possible, and that any individuals with the same score are in the same subdivision. The better part can be sorted without checking against the worse part for dominations, so it is passed to another call to rankPopulation. This guarantees that the ranks in that part are correctly determined, so a call to initializeWorseRanksByBetter in Algorithm 6 ensures that every individual in the worse part starts with a rank of one greater than the rank of the worst-ranked individual from the better part that dominates it. Because it is already known from the splitting that no individual in the worse part has a better score in the $M$th objective than the better part, this call is told to use $M - 1$ objectives for the comparisons. Then, the function recursively calls itself again for the worse part to finalize the rankings for the population subset.

initializeWorseRanksByBetter in Algorithm 6 starts by checking for base cases. If either of the population subdivisions has only one individual, it performs a straightforward domination check using the first $M$ objectives it has available for use and updates every individual in the worst part so that its rank is one greater than the rank of the worst-ranked individual in better. If it has been allowed to use two objectives, then it uses the sweepWorseRanksAgainstBetter function in Algorithm 7, a function inspired by Algorithm 4, to perform any necessary ranking updates.
Algorithm 5 DCS: rankPopulation

1: function DCS($P$, $S$)
2:     indices ← ⟨1 .. |$S$|⟩
3:     ranks ← (1 for $i$ in $P$)
5:     return result of ordering $P$, $S$, and ranks by ranks
6: function RANKPOPULATION($P$, $S$, i, ranks, $M$)
7:     if |$i$| = 1 then pass
8:     else if |$i$| = 2 then
9:         if $P$[i[1]] $\prec$ $P$[i[2]] then NOTE_1_DOMINATED_BY_1(i[2], i[1])
10:        else if $P$[i[2]] $\prec$ $P$[i[1]] then NOTE_1_DOMINATED_BY_1(i[1], i[2])
11:        else if $M$ = 2 then RANKBYTWOOBJECTIVESWEEP($P$, $S$, i, ranks)
12:        else if BEST($S$, i, $M$) = WORST($S$, i, $M$) then
13:            RANKPOPULATION($P$, $S$, i, ranks, $M$ - 1)
14:        else
15:            (better, worse) ← SPLITFOROBJECTIVEM($S$, i, $M$)
16:            RANKPOPULATION($P$, $S$, better, ranks, $M$)
17:            INITIALIZEWORSERANKSBY Better($P$, $S$, worse, ranks, $M$ - 1)
18:            RANKPOPULATION($P$, $S$, worse, ranks, $M$)
19: function NOTE_1_DOMINATED_BY_1(ranks, i, j)
20:     ranks[i] ← max (ranks[i], ranks[j] + 1)
21: function SPLITFOROBJECTIVEM($S$, i, $M$)
22:     sorted ← i ordered by each $S[i]$’s $M$th objective score
23:     split ← sorted $\left\lceil \frac{|\text{sorted}|}{2} \right\rceil$
24:     if $S$[sorted[-1]][M] $\neq$ split then
25:         return $(\{x \in i \mid S[x][M] \leq \text{split}\}, \{x \in i \mid S[x][M] > \text{split}\})$
26:     else
27:         return $(\{x \in i \mid S[x][M] < \text{split}\}, \{x \in i \mid S[x][M] \geq \text{split}\})$
28: function BEST($S$, indices, $M$) return min ($\{S[i][M] \mid i \in \text{indices}\}$)
29: function WORST($S$, indices, $M$) return max ($\{S[i][M] \mid i \in \text{indices}\}$)
Algorithm 6 DCS: initializeWorseRanksByBetter

30: function initializeWorseRanksByBetter(P, S, better, worse, ranks, M)
31:     if |better| = 1 then
32:         for all i in worse do
33:             if \( P[better[1]] \prec P[i] \) then note_i-dominated_by_j(ranks, i, better[1])
34:     else if |worse| = 1 then
35:         for all i in better do
36:             if \( P[i] \prec P[worse[1]] \) then note_i-dominated_by_j(ranks, worse[1], i)
37:     else if \( M = 2 \) then
38:         sweepWorseRanksAgainstBetter(P, S, better, worse, ranks)
39:     else
40:         (best_b, worst_b) ← (best(S, better, M), worst(S, better, M))
41:         (best_w, worst_w) ← (best(S, worse, M), worst(S, worse, M))
42:         if worst_w < best_b then pass
43:     else if worst_b \( \leq \) best_w then
44:         initializeWorseRanksByBetter(P, S, better, worse, M - 1)
45:     else if best_b = worst_b then
46:         \( w \leftarrow \{x \in worse \mid S[x][M] \geq best_b\} \)
47:         initializeWorseRanksByBetter(P, S, better, w, M - 1)
48:     else if best_w = worst_w then
49:         \( b \leftarrow \{x \in better \mid S[x][M] \leq best_w\} \)
50:         initializeWorseRanksByBetter(P, S, b, worse, M - 1)
51:     else
52:         if |better| > |worse| then
53:             (b_1, b_2) ← splitForObjectiveM(S, better, M)
54:             \( w_1 \leftarrow \{x \in worse \mid S[x][M] < b_2[1]\} \)
55:             \( w_2 \leftarrow \{x \in worse \mid S[x][M] \geq b_2[1]\} \)
56:         else
57:             (w_1, w_2) ← splitForObjectiveM(S, worse, M)
58:             \( b_1 \leftarrow \{x \in better \mid S[x][M] \leq w_2[1]\} \)
59:             \( b_2 \leftarrow \{x \in better \mid S[x][M] > w_2[1]\} \)
60:         initializeWorseRanksByBetter(P, S, b_1, w_1, ranks, M)
61:         initializeWorseRanksByBetter(P, S, b_1, w_2, ranks, M - 1)
62:         initializeWorseRanksByBetter(P, S, b_2, w_2, ranks, M)
CHAPTER 2. BACKGROUND AND RELATED WORKS

If initializeWorseRanksByBetter’s base cases do not hold for the population subdivisions, then the comparisons that need to be performed is determined by the degree of overlap of the two divisions’ $M$th objective scores (remember that the $M$ here is at least one less than the $M$ from the initial call from rankPopulation). The best and worst objectives scores for each subdivision are determined to check for the several cases that the overlap may have. If the worse division’s worst score is better than the better division’s best score, then it is impossible for any individual in the better division to dominate any individual in the worse partition by Equation 2.2 and the behavior of rankPopulation and initializeWorseRanksByBetter. In this case, the call does nothing and exits to allow the DCS to continue.

If the better partition’s worst score is better or equal to the worse partition’s best score, then the $M$th objective does not contribute anything to determination of domination relationships between the two partitions. initializeWorseRanksByBetter recurses, restricting itself to the previous $M - 1$ objectives for determining the dominations.

If the better partition has uniform scores for the $M$th objective, then only the worse partition individuals that have a score worse than or equal to the better partition’s common $M$th objective score can be dominated by any of the better partition’s individuals. It extracts the individuals from the worse partition that can be dominated by the individuals in the better partition and recurses over the better partition and the new worse subdivision for the previous $M - 1$ objectives.

If the worse partition has uniform scores for the $M$th objective, then only the better partition individuals that have a score better than or equal to the worse partition’s common $M$th objective score can dominate any of the worse partition’s individuals. It extracts the individuals from the better partition that can dominate the individuals in the worse partition and recurses over the new better subdivision and the worse partition for the previous $M - 1$ objectives.

Otherwise, the better and worse partitions $M$th objective scores overlap. By splitting both of them at a common point on the current objective, we get a better and a worse subdivision for each of the original partitions. The better partition’s better subdivision can dominate any individual in both of the worse partition’s subdivisions, but the better partition’s worse subdivision can only dominate individuals in the worse partition’s worse subdivision. This is because the worse division’s better partition will have better scores for the $M$th objective than the better division’s worse partition. To attempt to come as close to
CHAPTER 2. BACKGROUND AND RELATED WORKS

Algorithm 7 DCS: sweepWorseRanksAgainstBetter

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:</td>
<td><code>function sweepWorseRanksAgainstBetter(P, S, better, worse, ranks)</code></td>
<td>Function to sweep worse ranks against better ranks</td>
</tr>
<tr>
<td>64:</td>
<td><code>b ← sortByFirstTwoObjectives(S, better)</code></td>
<td>Sort by first two objectives for the better partition</td>
</tr>
<tr>
<td>65:</td>
<td><code>w ← sortByFirstTwoObjectives(S, worse)</code></td>
<td>Sort by first two objectives for the worse partition</td>
</tr>
<tr>
<td>66:</td>
<td><code>bFronts ← empty map of integers to integers</code></td>
<td>Initialize an empty map for better fronts</td>
</tr>
<tr>
<td>67:</td>
<td><code>j ← 1</code></td>
<td>Initialize iterator j</td>
</tr>
<tr>
<td>68:</td>
<td><code>for all i in w do</code></td>
<td>Loop through all individuals in the worse partition</td>
</tr>
<tr>
<td>69:</td>
<td>`while j ≤</td>
<td>b</td>
</tr>
<tr>
<td>70:</td>
<td><code>bFronts[ranks[b[j]]] ← b[j]</code></td>
<td>Add the current individual to the map</td>
</tr>
<tr>
<td>71:</td>
<td><code>j ← j + 1</code></td>
<td>Increment j</td>
</tr>
<tr>
<td>72:</td>
<td>`k ← max ({x ∈ bFronts keys</td>
<td>A domina\tes_b(S[bFronts[x]], S[i])})`</td>
</tr>
<tr>
<td>73:</td>
<td><code>if k ≠ null ∧ ranks[i] ≤ k then ranks[i] ← k + 1</code></td>
<td>Update the rank if the current individual is dominated by a front</td>
</tr>
</tbody>
</table>

An even division of the work as is possible, the splitting point for both partitions is chosen to be roughly the median of the larger partition. The resulting subdivisions are then passed to recursive calls to `initializeWorseRanksByBetter` to finish the the updating. The first call operates on the better subdivisions of both partitions for all currently available objectives because the splitting does not reveal anything about their relationships. The second call operates on the better subdivision of the better partition and the worse subdivision of the worse partition for $M-1$ objectives so that the unexplored objectives can be given a chance to reveal something useful. Finally, the worse subdivision of the worse partition is tested against the worse subdivision of the better partition for all the currently available objectives.

`sweepWorseRanksAgainstBetter` in Algorithm 7 is the final piece of the DCS. It is very similar to `rankByTwoObjectiveSweep`. For each individual in the worse partition, it ensures that all the individuals in the better partition that have first objective scores better than or equal to the current objective’s score, and in the case of ties it also adds any better partition individuals with better second objectives scores. It then finds the most dominated front that contains an individual that dominates the current worse partition individual. If one is found, and its rank is greater than or equal to the current individual’s rank, then the current individual’s rank is set to one plus the dominating front’s rank.

Jensen described his algorithm as having a computational complexity of $O(N \log^{M-1} N)$ and a memory requirement of $O(MN)$ [18]. Its design does not allow for parallelization as FNS’s does. So long as the number of objectives is not too large in proportion to the pop-
ulation, this is indeed an improvement over the FNS. However, DCS cannot sort solutions for COPs due to the implicit hierarchy that breaks the geometric approach that can assume that an individual is better than another if its objective scores are better. It is also designed around Pareto domination, and thus is too inflexible to accommodate new domination techniques. But there is also a potential problem with this computational complexity that I uncovered trying to implement the algorithm myself.

Jensen asserts that the sweeping algorithms can determine the most dominated front that dominates an individual in $O(\lg N)$ time by using a binary sort. In the simple case that the problem has only two objectives to start, the entire population can be sorted in this way without error. But if the problem has at least three objectives, then the sweep will be encountered in situations where the ranks cannot be assumed to start at one for all individuals. In fact, Jensen notes that fronts may be missing entirely. Furthermore, it is possible that an individual from a more dominated front will be found with a lower second objective score than individuals from less dominated fronts. In this case, Jensen writes that the less dominated fronts should be discontinued for consideration as dominating fronts until a new member is found with an even lower second objective score. For illustrations of this concept, see Figure 5 in [18].

The problem this introduces is that the searching can no longer be done in $O(\lg N)$ time. If one takes the approach of removing less dominated fronts that are now “dominated” by a more dominated front when this situation is discovered and then checking to see whether a front may be added back in after every individual has been ranked, it can take $O(N)$ to go through all the fronts to do this. If the easier tack of just checking all fronts with some binary-search style speed-ups every time, this still can take $O(N)$ time, because regardless of whether the branch for the less dominated fronts determines, the branch for the more dominated fronts must still be checked.

2.4.4 Shi et al.’s Better Non-dominated Sort

In 2005, Shi et al. published a non-dominated sort called Better Non-dominated Sort (BNS) [20]. It is the first non-dominated sorting algorithm in the literature to try avoiding comparisons between individuals by exploiting Pareto dominance’s property of transitivity. They also introduced the Better function, for which the algorithm is named, that deter-
mines which of two individuals dominates the other by performing both domination checks simultaneously.

As a partial ordering operator, Pareto domination has the property of transitivity. If \( \vec{u} \prec \vec{v} \) and \( \vec{v} \prec \vec{w} \), then \( \vec{u} \prec \vec{w} \) must also be true. A very simple proof for this appears in [20]. This means that, whenever \( \vec{u} \prec \vec{v} \) and \( \vec{v} \prec \vec{w} \) are known, it is unnecessary to also compare \( \vec{u} \) and \( \vec{v} \). This reduces the number of comparisons needed to rank all the individuals in a population. Table 2.2, based on Table II in [20], shows all the inferences that can be made for the relationship between individuals \( \vec{u} \) and \( \vec{w} \) when the relationships of the \((\vec{u}, \vec{v})\) and \((\vec{v}, \vec{w})\) are already known. The cells with \(-\) are situations in which no inferences can be made.

This method requires a method for caching the results of domination comparisons for efficient look-up. This is easily done with a table with a column and row for each individual in a population, where each cell contains the result of comparing the row’s individual and the column’s individual (does left \( \prec \) right?). However, this approach requires that two cells in the table must be checked to determine a relationship between two individuals. If \( \vec{u} \prec \vec{v} \), then \( \vec{v} \not\prec \vec{u} \) and checking the table’s entry for \((\vec{v}, \vec{u})\) will only hold the value \text{False} to represent \( \vec{v} \not\prec \vec{u} \). It is preferrable to instead store the direction of a domination relationship between the two individuals. For this purpose, Shi et al. introduced the Better function to encode the direction of the relationship:

**Definition 7 (Better function).** Let \( \vec{u} \) and \( \vec{v} \) be solutions to any MOP or COP. The direction of \( \vec{u} \)'s relationship with \( \vec{v} \) can be encoded as follows:

\[
\text{Better} (\vec{u}, \vec{v}) = \begin{cases} 
0 & \vec{u} \preceq \vec{v} \\
1 & \vec{u} \prec \vec{v} \\
2 & \vec{v} \prec \vec{u}
\end{cases}
\]

Algorithm 8 shows a pseudocode representation to perform the Better comparison by comparing the two individuals simultaneously. Shi et al. show that when \( \text{Better} (\vec{u}, \vec{v}) \)
Algorithm 8 Better function

```
1: function Better(fu = \vec{f}(\vec{u}), \ fv = \vec{f}(\vec{v}), \ gu = \vec{g}(\vec{u}), \ gv = \vec{g}(\vec{v}))
2: result ← (|gu| > 0 ? WHICHLISTISBETTER(gu, gv) : -1)
3: if result = -1 then result ← WHICHLISTISBETTER(fu, fv)
4: return (result = -1 ? 0 : result)

5: function WHICHLISTISBETTER(a, b)
6: (aDoms, bDoms) ← (0, 0)
7: for i, j in zip(a, b) do
8: if i < j then aDoms ← aDoms + 1
9: else if j < i then bDoms ← bDoms + 1
10: if aDoms > 0 ∧ bDoms > 0 then break
11: if aDoms = 0 ∧ bDoms = 0 then return -1 \quad \triangleright \text{lists equal; needed for hierarchy}
12: else if aDoms = 0 then return 2
13: else if bDoms = 0 then return 1
14: else return 0
```

returns 1, \textit{Better}(\vec{v}, \vec{u}) will return 2. Storing these encodings in a domination table as described above will require only one access to the table per pair. Because a non-zero value in any cell means the other non-zero value will be in the cell with that cell’s coordinates transposed, it is possible to store only half of the comparison results in the table.

BNS is a modification of FNS to take advantage of Table 2.2 and Equation 2.6. It builds dominating lists and accumulates dominated-by counts and then uses them to determine the fronts into which each individual belongs. The difference is that it adds a domination table to store relationships so that future comparisons can attempt to benefit from Pareto domination transitivity by skipping some unnecessary comparisons.

Algorithm 9 shows BNS’s modified comparison portion. It builds a table to hold the comparison results, the dominated-by counts list, and the list of dominating lists. It then compares every individual with the individual that directly follows it in the population’s current ordering to ensure that every individual has at least one comparison in the table to use for inference. This fills a diagonal line of cells immediately above the line of cells with the same column and row numbers (the cells which would hold the results of comparing each individual against itself). It then fills each diagonal line above that one toward the
Algorithm 9 BNS comparison logic

1: function BNSCompare($P$, $S$)
2:   $t \leftarrow$ new $|P| \times |P|$ table for holding integers
3:   dominatedByCounts $\leftarrow$ $\{0 \forall i \in P\}$
4:   dominatesLists $\leftarrow$ $\{\text{nil} \forall i \in P\}$
5:   for $i \leftarrow 1$, $|P| - 1$ do
6:     $t[i][i + 1] \leftarrow \text{BetterWithInference}(S[i], S[i + 1], \text{NOTHING})$
7:     UPDATECountsAndLists($i$, $i + 1$, $t$, dominatedByCounts, dominatesLists)
8:   for offset $\leftarrow 2$, $|P| - 1$ do
9:     for $u \leftarrow 1$, $|P| - offset$ do
10:    $w \leftarrow u + offset$
11:    $v \leftarrow$ random integer from exclusive range $(u, w)$
12:    known $\leftarrow (t[u][v], t[v][w])$
13:    if known $=$ $(1, 1)$ then $t[u][w] \leftarrow 1$
14:    else if known $=$ $(2, 2)$ then $t[u][w] \leftarrow 2$
15:    else
16:       if known $\in \{(0, 2), (2, 0)\}$ then known $\leftarrow \text{U\_CANT\_DOMINATE\_W}$
17:       else if known $\in \{(0, 1), (1, 0)\}$ then known $\leftarrow \text{W\_CANT\_DOMINATE\_U}$
18:       else known $\leftarrow \text{NOTHING}$
19:       $t[u][w] \leftarrow \text{BetterWithInference}(S[u], S[w], \text{known})$
20:     UPDATECountsAndLists($u$, $w$, $t$, dominatedByCounts, dominatesLists)
21:   firstFront $\leftarrow \{i \in \{1..|P|\} \mid \text{dominatedByCounts}[i] = 0\}$
22:   return (dominatedByCounts, dominatesLists, firstFront)
23: function BetterWithInference($u$, $w$, known)
24:   if known is U\_CANT\_DOMINATE\_W then return ($w \not\prec u ? 0 : 2$)
25:   else if known is W\_CANT\_DOMINATE\_U then return ($u \not\prec w ? 0 : 1$)
26:   else return \text{Better}(u’s objectives, u’s objectives, w’s constraints, w’s constraints)
27: function UPDATECountsAndLists($i$, $j$, table, counts, lists)
28:   if $\text{table}[i][j] = 1$ then \text{PERFORMUpdate}($i$, $j$, counts, lists)
29:   else if $\text{table}[i][j] = 2$ then \text{PERFORMUpdate}($j$, $i$, counts, lists)
30: function PERFORMUpdate($\text{better}$, $\text{worse}$, counts, lists)
31:   counts[worse] $\leftarrow$ counts[worse] + 1
32:   lists[better] $\leftarrow$ lists[better] + worse
Each individual is compared with the one that comes \textit{offset} places after the current individual (where \textit{offset} is the counter showing which line is currently being filled). A random individual between the two is picked. Since both individuals have been compared against that individual, the dominance table has the required information to make use of Table 2.2. BNS checks to see whether any part of the comparison between the individuals can be skipped. If one of the two individuals that need to be compared are shown to have a relationship such that one dominates the other, this relationship is added to the table without comparing them at all. If the two have a relationship such that one individual cannot dominate the other, then it notes that that comparison need not be made. It uses this information to call \texttt{BetterWithInference} to determine the final relationship. After each comparison, BNS uses the \texttt{updateCountsAndLists} function to ensure that any domination information that the front-building step will need has been cached. Finally, it determines which individuals belong in the first front by listing the indices of all individuals that are not dominated, and it returns the dominated counts, dominating lists, and first front for the front-building portion of FNS to complete the work.

\texttt{BetterWithInference} provides a small improvement to the time required to compare two individuals if the inference shows that one individual cannot dominate the other. Assuming that it is known that an individual \(\vec{w}\) cannot dominate another individual \(\vec{u}\), then all that needs to be found is one score at which \(\vec{u}\) is better than \(\vec{w}\) to determine that \(\vec{u} \prec \vec{w}\). If it can’t be found, then \(\vec{u} \not\prec \vec{w}\). Without any inferences to guide the function, it uses the Better function directly to determine the two individuals’ relationship. Note that if the problem is a COP, the constraint scores for the two individuals must be identical before the objective scores will be compared according to Equation 2.4.

Shi et al.’s BNS has the same computational complexity and memory requirements as FNS: \(O(MN^2)\) and \(O(CN^2)\). For populations with many dominance relationships in the population, this algorithm can provide some improvement in performance. However, for populations that have few dominance relationships, the algorithm performs extra work for nearly every comparison in the population to no effect. Since MOEAs work to decrease the dominance relationships in their populations by finding an approximate Pareto optimal set, BNS will end up being slower than FNS overall. In addition, BNS’s optimization cannot be parallelized due to the reliance on other comparisons having been made before two individuals can be completed; this would likely be even slower than the synchronous version presented here.
2.4.5 Fang et al.’s Dominance Tree Sort

Fang et al. published the Dominance Tree Sort (DTS) in 2008 [3]. It is a divide-and-conquer sort that tries to reduce unnecessary comparisons while ranking a population by omitting comparisons between individuals that cannot be placed into the same front. It does this through the construction and merging of instances of a data structure they call a dominance tree. A dominance tree is a node-based structure that contains partial front approximations. Figure 2.1 shows the structure of the dominance tree node. It can store either an address or index for an individual as the data. Its sibling field is used to hold the address of a node that refers to an individual with which this node’s individual is mutually non-dominated. The child field is used to hold the address of a node that refers to an individual that is dominated by this node’s individual.

Dominance trees are built from dominance tree nodes to form structures that look something like Figure 2.2. These trees are structured with the following rules:

- A node may only have or be an immediate or transitive sibling with another node if the two nodes in question refer to individuals that are mutually non-dominated.

- A node may only be an immediate or transitive parent to another node whose referred individual is dominated by its individual.

- No node may be an immediate or transitive child to another node whose referred individual is dominated by its individual.

This says nothing about non-dominated individuals restricting sibling or child relationships because, as explained in Section 2.2, mutual non-domination does not inform the ranking of individuals anyway. Figure 2.2 shows a diagram of a simple dominance tree that can be interpreted as follows. Nodes A through D refer to mutually non-dominated individuals. All the individuals referred to by A’s child subtree (E, F, and J) are dominated by A’s referred
individual, and all the individuals in C’s child subtree (G, H, I, and K) are dominated by C’s referred individual. This process for interpreting the dominance tree is recursive, holding for all subtrees.

In a dominance tree, the root node and all its siblings are the first non-dominated front for the population subset represented by that tree. If the tree represents an entire population, then the root node and its siblings are the population’s first non-dominated front. This relationship does not hold for subtrees because each node in the root-and-sibling list may have their own subtrees, and these subtrees contain individuals that have not been compared with each other. But this means that ignoring the tree’s root node and its siblings leaves the subset of the population excluding the non-dominated individuals, so merging those subtrees together will reveal the next non-dominated front. This property does hold recursively, and is how DTS ultimately works.

DTS can be said to take the opposite approach to eliminating unnecessary comparisons of BNS. BNS tries to build a complete directed graph showing which individuals are dominated by which. DTS instead tries to use inferences to avoid comparing any individual with any other individual that will not inform its placement into the final front orderings in any way. Two examples can be drawn from Figure 2.2. First, assume that every node but D is in the tree. When testing its node against the individuals referred to by A, B, and C, it is found to be mutually non-dominated with all of them. There is currently no point in comparing D’s individual with any other node; all of the others are already known to not be in the same front as A through C, so performing those comparisons is not going to say anything new about any individual’s position. Now assume that every individual but J is in the tree. It is found to be dominated by A, so comparing it against B through D will only tell what is already known: J must be in a lower front than A. After that, it is found to be dominated by E, so it does not need to be compared against F. It is placed as E’s child.

![Figure 2.2: An example dominance tree.](image-url)
The most useful part of this structure is that dominance subtrees can be demoted to lower partial fronts automatically when the individual that dominates them is demoted. Figure 2.3 shows an example of this. Suppose that a new node L refers to an individual that dominates A’s individual. So long as the original tree has not made any passes through the procedure to turn the tree into fronts, then E, F, and J are A’s children because their individuals are dominated by A’s. By Table 2.2, it is clear that L must dominate all of these individuals as well. Rather than making the comparisons with the transitively dominated individuals or even looking up existing relationships in a table, A and its children subtree can simply be removed from the tree and added as L’s child. The end result is a structure that shows L’s transitive dominance over those individuals for free.

Shi et al. described two versions of DTS in [3]: one that inserts dominated subtrees as shown in Figure 2.3 as soon as they are discovered, and one which delays the insertion in the hopes of possibly finding more dominated subtrees to combine together before performing the insertion. Shi et al. explain that the delayed insertion version is computationally more efficient because it is less likely to perform duplicate comparisons, so this is the version that will be discussed.

Algorithm 10 shows the delayed insertion algorithm for DTS. Its starting point is the DTS function. DTS has the buildDominanceTree function build a dominance tree for the entire population. The tree generated by this process is passed to getOrderFromTree to build the ordering and rankings required of a non-dominated sort, which DTS then returns.

buildDominanceTree is a very simple recursive function that divides and conquers the tree-building problem. If its passed population subset has only one individual in it (as noted by having the same value for its lowIndex and highIndex arguments), then it wraps that index in a new dominance tree node to start a new tree. Otherwise, it divides the population
Algorithm 10 DTS

1: function DTS(P, S)
2:   tree ← buildDominanceTree(P, S, 1, |P|)
3:   return getOrderFromTree(P, S, tree)
4: function buildDominanceTree(P, S, lowIndex, highIndex)
5:   if lowIndex = highIndex then return new DOMINANCE_TREE(data ← lowIndex)
6:   else
7:      split ← ⌈(lowIndex+highIndex)/2⌉
8:      left ← buildDominanceTree(P, S, lowIndex, split - 1)
9:      right ← buildDominanceTree(P, S, split, highIndex)
10:     return mergeDominanceTrees(P, S, left, right)
11: function mergeDominanceTrees(P, S, left, right)
12:   (leftNode, leftPrev, rightNode, rightPrev) ← (left, right, NULL, NULL)
13:   (leftDom, rightDom) ← (NULL, NULL)
14:   while leftNode ≠ NULL ∧ rightNode ≠ NULL do
15:      if P[leftNode.data] ≺ P[rightNode.data] then
16:         HANDLE_DOMINATION(P, S, rightDom, rightNode, rightPrev, right, leftDom)
17:      else if P[rightNode.data] ≺ P[leftNode.data] then
18:         HANDLE_DOMINATION(P, S, leftDom, leftNode, leftPrev, left, rightDom)
19:      else
20:         addChild(rightNode, rightDom)
21:         (rightPrev, rightNode) ← (rightNode, rightNode.sibling)
22:   if rightNode = NULL then
23:      addChild(leftNode, leftDom)
24:      (leftPrev, leftNode) ← (leftNode, leftNode.sibling)
25:      (rightPrev, rightNode) ← (NULL, right)
26:   if leftNode = NULL ∧ rightNode ≠ NULL then
27:      addChild(rightNode, rightDom)
28:   if leftPrev ≠ NULL ∧ right ≠ NULL then
29:      leftPrev.sibling ← right
30:   return (left ≠ NULL ? left : right)
31: function HANDLEDOMINATION($P, S, wdRef, wnRef, wpRef, wtRef, bdRef)
32:     ADDCHILD($P, S, wnRef, wdRef)
33:     if wpRef ≠ NULL then wpRef.sibling ← wnRef.sibling
34:     else wtRef ← wnRef.sibling
35:     temp ← wnRef
36:     wnRef ← wnRef.sibling
37:     temp.sibling ← bdRef
38:     bdRef ← temp
39: function ADDCHILD($P, S, parent, childRef)
40:     if childRef ≠ NULL then
41:         if parent.child = NULL then parent.child ← childRef
42:         else parent.child ← MERGEDOMINANCETREES($P, S, parent.child, childRef)
43:     childRef ← NULL
44: function GETORDERFROMTREE($P, S, tree)
45:     fronts ← copy of tree
46:     (front, order, scores, rank, ranks) ← (fronts, nil, nil, 1, nil)
47:     while front ≠ NULL do
48:         node ← front
49:         while node ≠ NULL do
50:             order ← order + $P[node.data]
51:             scores ← scores + $S[node.data]
52:             ranks ← ranks + rank
53:             if node ≠ front ∧ node.child ≠ NULL then
54:                 front.child ← MERGEDOMINANCETREES($P, S, front.child, node.child)
55:             node ← node.sibling
56:         (front, rank) ← (front.child, rank + 1)
57:     return (order, scores, ranks)

into two halves, recursively builds trees for those halves, then builds and returns one tree from those two trees using the mergeDOMINANCETREES function.

mergeDOMINANCETREES is where all the real work is performed. Its basic approach is to iterate through every pair of individuals across the two trees’ root-and-sibling nodes,
searching for any dominations. If a node in one tree refers to an individual that is dominated by an individual referred to by a node in the other tree, the dominated node is removed from its current tree and merged into the dominating node’s subtree. Any nodes that remain in their original root-and-node lists are mutually non-dominated, so the two trees can be merged by extending the left tree’s root-and-sibling list with the right tree’s node. The merged tree is returned so that it can be used by whatever function is calling it, which is most typically `mergeDominanceTrees` itself as it sets a node’s child to a merged tree.

This simple approach is complicated by the delayed insertion design. The idea behind the delayed insertion design is to reduce the chances that individuals from the same tree will be compared against each other again. For example, if a node A from one tree dominates two mutually non-dominated nodes B and C in the other tree, immediately merging B under A when the domination is found will cause C to be compared with B again when C is merged into A’s subtree. Delayed insertion can avoid this by finding multiple dominated nodes in a tree then merging them all at once when the situation requires.

`mergeDominanceTrees` works as follows. It uses pointers to nodes to track which node in each tree is currently being processed and their previous nodes to simplify removing the nodes from the trees. It also uses pointers to store trees built from nodes of individuals dominated by the current node from either tree so that the entire tree can be inserted at once. The current node pointers start at the root nodes for each tree, and the other pointers are set to `NULL`.

The `while` loop works like a nested `for` loop, iterating over the right tree’s non-dominated nodes for each non-dominated node in the left tree. The current nodes are compared. If one of the nodes dominates the either, then the `handleDomination` function is used to manipulate all of the variables as follows. First, `addChild` is used to test whether the dominated node has been determined to dominate any of the other tree’s nodes. If so, those nodes are added to the dominated node’s subtree, either by being set as its child if it does not currently have one or by recursively calling `mergeDominanceTrees` for its current child and the dominated tree. With the dominated tree having been added to the dominated node’s subtree, the pointer to the subtree is set to `NULL` to ready it for handling a dominated tree.

---

2While I describe it as a function, it might not be practical to implement it as such in any given programming language. `mergeDominanceTrees` cannot work unless `handleDomination` can alter the passed-in variables’ values. It is presented as a function in the pseudocode because the logic is identical for the two situations it is used and to save space. It might be best to implement it as a C-style macro, though a function with pointers to pointers or C++-style references would work at the cost of ease of comprehension.
for a different node. Next, the dominated node is removed from its tree by manipulating
the previous node’s sibling pointer to point to its sibling. If the current node is the root
of its tree, then the tree is set to refer to the dominated node’s sibling instead of the NULL
previous node pointer. A temporary pointer is then set to refer to the current node, and
the tree’s current-node pointer is then set to the next node in that tree. Finally, the node
and its subtree are added to the dominating node’s waiting dominated tree by setting the
dominated node’s sibling to whatever address the waiting tree’s pointer holds, then setting
the waiting tree’s pointer to the dominated node’s address. This process makes it so the
iteration through the trees’ non-dominated pairs continues regardless of the removal of a
node and postpones the merging of subtrees until either a node is found to dominate the
node with the waiting tree or the iteration moves past that node.

The iteration moves to the next node in the right tree whenever the current nodes are
mutually non-dominated, and to the next node in the left tree whenever the right tree’s
current-node pointer points to NULL. Moving to the next node in the right tree is simple:
first the right node’s waiting dominated tree is merged with its child, and then the right
tree’s previous and current node pointers are shifted to the current node and the current
node’s sibling respectively. If either this step to the right, or removing a node from the
right tree, causes the right tree’s current-node pointer to be NULL, then the left tree’s node’s
waiting list is merged with its child, then the pointers for all the nodes are set to start from
the next node in the left tree and the first node still in the right tree. Finally, it checks for
the case in which the last node in the left tree was removed due to being dominated to a
right tree node. In that case, no more iterations of the loop will occur, so it merges the right
tree’s child with its waiting tree immediately.

Once the loop has finished, the only remaining nodes in the trees’ root-and-sibling lists
are all mutually non-dominated. If both trees still have any nodes, then the right tree’s node
is set as the sibling of the left tree’s last node in its root-and-sibling list. Finally, it returns
the left tree if it has not been consumed by the right tree; otherwise, it returns the right
tree.

With the dominance tree constructed, it can be used to determine the non-dominated
ranks of its referred individuals. getOrderFromTree does this by processing the passed
dominance tree as follows. First it generates a copy of the tree since determining the fronts
is destructive\(^3\). Then it iterates over the tree’s root-and-node list, adding each referred individual to the ordering and setting its rank. Whenever it encounters a node with a non-root node that has a child, it merges that subtree with the root’s subtree. After it has finished processing that root-and-sibling list, then the processed front can be considered as having been removed from the population, and the root’s child now contains a dominance tree whose root-and-sibling list is the second non-dominated front. This loop continues until a tree’s root node does not have a child. The ordering is then returned.

Shi et al. approximate DTS’s running time complexity as $\theta\left(\frac{1}{p}MN \lg N\right)$, where $p$ is a constant based upon a population’s distribution. Its worst case running time is $O(MN^2)$, when it must compare every individual with every other individual to determine a mutually non-dominated population. It does all this with only a $\theta(N)$ memory requirement for building the tree on top of the $O(MN)$ requirement for building the ordering. Its divide and conquer approach allows the recursive calls to \texttt{buildDominanceTree} to be run in parallel, although the final call to \texttt{mergeDominanceTree} and the entirety of its \texttt{getOrderFromTree} must be run synchronously. It has very good running time and memory requirements, uses an interesting strategy to eliminate comparisons, and can sort both MOP and COP problems. It is difficult to criticize this algorithm\(^4\).

### 2.4.6 Mazurek and Wesolkowski’s Limiting Index Sort

Mazurek and Wesolkowski published the Limiting Index Sort (LIS) in 2010 [21]. LIS, like Jensen’s DCS, is restricted to sorting MOPs with Pareto domination. It also sorts the population by objective scores to eliminate comparisons. However, LIS only performs one sort per objective where DCS performs multiple sorts to find an objective’s median score for each split. LIS avoids comparisons by using the individuals’ indices in those orderings to determine “limiting indices” that reveal which individuals cannot be members in a front.

LIS, like Jensen’s DCS, is a sort that is limited to sorting MOPs as opposed to COPs that requires Pareto domination as the comparison scheme to make the sort work and also

\(^3\)This step is not strictly necessary and is not mentioned in [3]. It is done for my convenience in setting up for Section 3.3.

\(^4\)It is very easy to criticize the paper itself. It has poor scholarship and editing. It makes some claims about the literature that are untrue. For example, it misrepresents how FNS works, then makes the claim that Jensen’s DCS does not work properly. It also presents the NS algorithm as though it is FNS. Furthermore, the pseudocode and accompanying text do not clearly explain the algorithm. The pseudocode for the delayed insertion DTS does not even perform delayed insertion! This paper is so difficult to decipher that DTS was not part of my initial run of experiments.
uses sorting the population on the individual objectives to eliminate comparisons between individuals that cannot be members of the same front. LIS is different from DCS in that LIS performs one sort per objective for the entire population, then uses cached individual positional information for the fronts to determine individual candidacy for fronts before comparing the individuals.

The idea behind LIS is fairly simple. The population is sorted from best to worst score for each objective, with ties in an objective score broken by the other objective scores. This means that, in each ordering, no individual is listed after an individual that can dominate it. If an individual is placed first in any ordering, then that individual cannot be dominated by any other individual because it has the best score for that objective. So the first individuals are added into the first non-dominated front to start the comparisons. Individuals are only compared against individuals that come before them in a list’s ordering to determine whether they belong in the same fronts.

Mazurek and Wesolkowski realized that it is possible to take advantage of a front’s individuals’ indices to eliminate other individuals from consideration for membership in a front. Every individual has a position in every ordering, and it is easy to determine each individual’s minimum and maximum indices. If an individual’s minimum index is less than another’s maximum index, then the first individual has no objective scores that are better than the second’s, and this is very likely caused by the first individual being dominated by the second. If all of the individuals in a front are considered, then the lowest maximum index, referred to from now on as the front’s minimax index, can serve as a cutoff point for membership in that front. This is because any individual whose minimum index comes after the front’s minimax is assumed to be dominated by the minimax individual (the individual whose maximum index is used as the minimax), even if its minimum index comes before the maximum index of another individual in the same front. In general, this is true.

The problem with this approach, not discussed by Mazurek and Wesolkowski in [21], is that individuals can have identical objective score vectors. In MOEAs, this can occur whenever an individual is duplicated in the population, and certain problems may yield identical objective score vectors for different decisions. Since LIS only guarantees that an individual as listed in one of the objective orderings cannot be dominated by any subsequent individuals in that ordering, these positional indices promise nothing about objective vector equality. It is very easy to prove that this is a problem. Assume a population with two individuals with identical objective scores that dominate every other individual in the population. This
means that there is no guarantee of which individual will be listed as the first in any given ordering unless a stable sort is used, but this means that it is possible that one individual will be listed first for every objective, and the other as the second. This makes the first front’s minimax index 1. Since the second individual’s minimum index is 2, the second individual is assumed to be dominated by the first, even though they are identical! This violates Equation 2.5. If used with an MOEA that uses non-dominated ranks to determine fitness of individuals, individuals dominated by duplicate individuals will all be ranked lower than they actually are, which will exert different pressure on the breeding than if they were ranked properly. This problem is discussed in [16] in criticism of Fonseca and Fleming’s MOGA [22], but LIS will cause similar problems. Worse still, because there is no guarantee that the two individuals will be ordered the same way every time in every list (for example, if a randomized sort such as quicksort is used), this sort may generate different rankings and orderings for the same population!

That said, it seems that a minor change can be made to remedy this problem. The limiting index feature could be used as the first of two steps to lock a front as opposed to the only step. LIS could record not only the minimax index for each front, but also the individuals that establish these minimax indices. Note that more than one individual per front could contribute the same minimax index, but only one needs to be tracked. As the iteration progresses through the lists, it will eventually step positionally beyond the minimax index for a given front. This signals that the front will soon be locked. The next individual in the ordering should be checked for domination by the individual that established the minimax. If the individual that should be considered is already ranked, then it is safe to keep the front open to check again for the next pass; this prevents requiring comparing two individuals against each other twice. If it has not been ranked, then LIS performs the domination comparison. If the individual is non-dominated, then it must have an identical objective score vector with the individual, and the front should be kept open to check for another duplicate in the next iteration. If the individual is dominated, then the individual has no more duplicate individuals, and the front can be permanently closed. This process enables duplicates to be placed into the same fronts.

Why only one objective needs to be tracked when multiple individuals could have the same maximum index in a front bears explanation. Simply put, it is impossible for two individuals with different objective score vectors to have the same maximum rank and for either of them to have a duplicate that is not already ranked. Suppose that there are
individuals with the same decision scores and one that has a different objective score vector that is still mutually non-dominated with the other two. Since they have different objective score vectors, that means that the duplicate individuals have a score that is better than the third one, and the third one has a score that is better than the other twos’. This means that, for at least one ordering, both of the two individuals must be listed before the third. Both of them have been ranked, so a duplicate cannot be missed. This means that the only case in which more than one individual contributes the same maximum index to close a front where equality must be checked is when those individuals have the same objective score vectors. Therefore, which objective is checked will not matter, because either the next individual has the same objective score vector as all of the contributing individuals or it is dominated by them all.

Pseudocode for this revised version of LIS is shown in Algorithm 11. Its logic for managing and searching the fronts comes from an LIS variant called LIS$_{\text{half}}$ [21]. The idea is to use a tree to optimize searching through the fronts so that finding the correct front only requires comparisons with individuals in $O(\log F)$ of the fronts.

LIS starts with the epnoymous function LIS. It starts by creating a self-balancing tree, such as the AVL tree I used for my implementation, to hold the fronts as they are built. The tree’s nodes are keyed by the fronts’ ranks so the tree’s depth will be some $O(\log F)$ mentioned above. The tree is initialized with a single node, holding a front ranked ”1” with no individuals. It sets a list of tuples to hold each individual’s indices in each of the orderings, lists to hold each individual’s minimum and maximum indices, and a list for the individuals’ ranks, all intended to be accessed using the same individual indices as in $P$. Also, an empty list is prepared to hold the actual orderings. The indices list enables the sort to skip comparisons with individuals that come after an individual in the current ordering.

With the initialization complete, the orderings must be created. LIS iterates over each of the objectives, sorting the population for the current objective with the help of a function generated by compareByObjective. The returned function determines whether it is safe to list the individual index passed as $a$ before the individual index passed as $b$ based upon the lexicographical ordering described in [21]. The end result is that the ordering contains ordered indices while the original population maintains its original ordering. The current ordering is then processed to store each individual’s index in the ordering in the indices list. If the ordering index is the maximum index encountered thus far, it is cached as such to ease front minimax determination.
Algorithm 11 LIS

1: function LIS(P, S)
2:     fronts ← new self-balancing tree with a new Front with rank 1, members nil,
3:               minimaxIndex 0, minimaxIndividual 0, and locked False
4:     orderings ← nil
5:     indices ← ⟨(0 ∀ j ∈ S[1]) ∀ i ∈ P⟩
6:     maxIndices ← ⟨0 ∀ i ∈ P⟩
7:     (ranked, ranks) ← (0, ⟨0 ∀ i ∈ P⟩)
8:     for i ← 1, |S[1]| do
9:         orderings ← orderings + ⟨1..|P|⟩ sorted using COMPAREByOBJECTIVE(S, i)
10:        for all k ∈ orderings[i] do
11:            if maxIndices[k] = 0 ∨ maxIndices[k] < j then
12:                maxIndices[k] ← j
13:     for i ← 1, |P| do
14:         for j ← 1, |S[1]| do
15:             k ← orderings[i][j]
16:             if ranks[k] = 0 then
17:                 if i = 1 then
18:                     ADDINDIVIDUALToFront(fronts.root.data, k, maxIndices[k])
19:                 else
20:                     front ← GETFront(P, S, k, j, i, tree, indices)
21:                     if front = NULL then
22:                         add new Front to fronts tree with rank |fronts| + 1, members ⟨k⟩,
23:                         minimaxIndex maxIndices[k], minimaxIndividual k, and
24:                         locked False
25:                     else
26:                         ADDINDIVIDUALToFront(front, k, maxIndices[k])
27:                 ranked ← ranked + 1
28:             if ranked = |P| then break
29:         if ranked = |P| then break
30:     iterate through the fronts tree to build the ordering information and return it
28: function compareByObjective($S, i$)
29:     function curriedComparison($a, b$)
30:         (a_before_b, $j$) ← (True, $i$)
31:         repeat
32:             a_before_b ← $S[a][j] \leq S[b][j]$
33:             $j \leftarrow (j + 1 > |S[1]| ? j + 1 : 1)$
34:         until ¬a_before_b \lor i = j
35:         return a_before_b
36:     return curriedComparison
37: function addIndividualToFront(front, indexInPopulation, maxIndexInOrdering)
38:     front.members ← front.members + indexInPopulation
39:     if front.minimax = 0 \lor maxIndexInOrdering < front.minimax then
40:         front.minimax ← maxIndexInOrdering
41:         front.minimaxIndividual ← indexInPopulation
42: function getFront($P, S, i, obj, minIndex, node, indices$)
43:     if node = NULL then return NULL
44:     front ← node.data
45:     if minIndex > front.minimax then
46:         if front.locked \lor front.minimaxIndividual \lt P[i] then
47:             front.locked ← True
48:             return getFront($P, S, i, obj, minIndex, node.right, indices$)
49:         else return front
50:     for all $k \in$ front.members do
51:         if indices$[k][obj] \lt minIndex$ then
52:             relationship ← BETTER($S[k], S[i], nil, nil$)
53:             if relationship = 1 then
54:                 return getFront($P, S, i, obj, minIndex, node.right, indices$)
55:             else if relationship = 2 then break
56:     betterFront ← getFront($P, S, i, obj, minIndex, node.left, indices$)
57:     return (betterFront \neq NULL \? betterFront : front)
With the orderings made, LIS then iterates over the orderings as follows. It iterates over the \( i \)th element of each of the orderings in each pass to build accurate representations of the fronts. On the first pass, each unique population index (the indices stored in the orderings) is added to the first front because those individuals cannot be dominated by any other individual. Individuals are added to the fronts by the `addIndividualToFront` function, which simply appends the passed population index to the front’s members list and ensures that the front’s minimax index and individual are updated if the new individual has a lower maximum index than the current minimax. On the rest of the passes, LIS queries the fronts tree to find the front into which the individual should be placed using the `getFront` function. If a front is found, the individual is added to that front using the `addIndividualToFront` function. If not, it creates a new front that contains only the current individual and adds that to the tree. Every time an individual is added to the tree, LIS checks to see if all the individuals have been placed; if so, it short-circuits the loop. Once all the individuals have been placed, it iterates over the front tree to build the ordering and ranking information that the other sorts return. This is very similar to the logic from the other sorts seen thus far, so I did not repeat it here.

`getFront` allows LIS to find the front into which an individual should be placed with the guarantee that no more than \( O(\log F) \) of the fronts are tested. A recursive function, it is initially called with the tree’s root node and various other arguments pertaining to the population being sorted, the position in the iteration, and the precached information. The base case is simple: if the current node is `NULL`, then the search has descended down a branch where no node currently resides, so `NULL` is returned. If not, then the node has a front stored as its data. If the current individual’s minimum index (the current ordering index for LIS’s outer loop) is greater than the front’s minimax index, then the modifications to LIS I described are used. If the front has been locked, or if the front’s minimax individual dominates the passed individual, then the current individual (and no subsequent individuals) cannot belong in the front. The front is flagged as locked, and `getFront` is called recursively down the node’s right branch. If the front is not locked, and the current individual is not dominated by the front’s minimax individual, it must be because they have identical objective score vectors, so the individual is added to the current front and the front is kept unlocked.

If the current individual’s ordering index is greater than or equal to the front’s minimax index, then the only way to determine potential membership is by comparing the
current individual with the members of the front. LIS automatically excludes any individual from comparison that comes after the current individual in the current objective’s ordering because the ordering condition guarantees that the front’s individual cannot dominate the current individual. If any front individual dominates the current individual, then the current individual must belong in a lower-ranked front, so \texttt{getFront} is called recursively down the tree’s right branch. If the current individual dominates any front member, then the current individual must belong in a higher-ranked front, so \texttt{getFront} is called recursively down the tree’s left branch. If all comparisons made yielded mutual domination, or if all comparisons were skipped, then all that is known is that the current front is the lowest-ranked front in which the current individual can belong. \texttt{getFront} is called recursively down the left branch to see whether there is a better front. The result of any of the recursive calls mentioned is returned as \texttt{getFront}’s result.

Mazurek and Wesolkowski report LIS’s worst-case computational complexity as $O(MN^2)$ with an average computational complexity of $O(MN \lg N)$ [21]. It only requires an additional $O(MN)$ memory in addition to the typical $O(CN)$ for returning the sort’s results [21]. It may be possible to get a slight performance boost by performing the objective orderings in parallel, but this is unlikely to generate much improvement due to competition for setting the individuals’ maximum ranks; it is impossible to parallelize the iteration over the orderings because LIS relies so heavily on its synchronous approach. In addition, it is limited to sorting MOPs and relies on Pareto domination as its mechanism for determining the orderings. However, as MOP sorting algorithms go, it is a very simple algorithm that gets great performance.

\subsection*{2.4.7 Verma et al.’s Novel Non-dominated Sort}

Verma et al. published the Novel Non-dominated Sort (NNS) in 2011 [23]. NNS is a MOP-sorting non-dominated sort that is restricted to using Pareto domination to compare individuals. Like LIS, NNS sorts all the individuals for each of the objectives to determine a comparison order that guarantees that no individual can be listed before any individual that dominates it. However, it uses a different scheme for building the fronts; it can be said that LIS focuses toward the front of the ordering to place the current individual, but NNS focuses toward the back of the ordering to place subsequent individuals in reference to
the current individual. As such, they have different mechanisms for omitting comparisons between individuals.

NNS uses several facts to omit comparisons. Because the ordering of individuals used for the iteration, no individual can be dominated by any individual that follows it, so that means that only one direction of domination needs to be checked. This leads to a nested loop structure where the outer loop focuses on the current individual, and the inner loop compares the subsequent individuals against the current individual for domination. Once the outer loop focuses on an individual, its rank must be finalized, because all of the individuals that can dominate it have already tested their domination against it. And using the transitivity of domination from Table 2.2, it can be shown that only individuals that are assumed to be in the same front as the current individual need to be checked for domination. This can further reduce the number of comparisons made.

Verma et al. also realized another property of the iteration ordering. Assume that tuples across the objective orderings can be used to develop sets they called layers. This means that all the individuals who are placed first in any objective ordering belong in the first layer, any individuals that are placed second in any objective ordering belong in the second layer, and so on until the $N$th, last layer. Just as the individuals in the first layer cannot be dominated by any individual because they have the best objective scores in at least one objective, the individuals in the last layer cannot dominate any individual because they have a worst score. This means that whenever the outer iteration focuses on an individual known to be in the last layer, LIS can skip comparing all subsequent individuals for domination against that individual. If LIS encounters all the individuals in the last front during the outer loop, that means that only individuals in some subset of the layers excluding the last layer remain for the outer loop to process. The new last layer in that subset contains individuals that cannot dominate any other individual, so they then are skipped if they are encountered. NNS therefore keeps track of the current last layer so it can avoid performing the inner loop as many times as possible.

Algorithm 12 shows the pseudocode for NNS as described in [23]. It starts very similarly to LIS; after it performs its initializations, it orders the population lexicographically for each of the objectives. NNS builds a list of sets for the layers, which contains the same information as LIS’s indices list but without duplicates. Then it builds a set of the individual’s population indices as listed in the orderings. The order of insertion is the same as LIS’s nested loop ordering: it iterates across the $M$ ordering lists at the same index before drop-
Algorithm 12 NNS

1: function LIS(P, S)
2: (added, iterOrder, layers, orderings, placed) ←
   (\{\text{False } \forall i \in P\}, \text{nil}, \text{\{\emptyset } \forall i \in P\}, \text{\{\text{False } \forall i \in P\})
3: (end, ranked, ranks) ← (|P|, 0, \{1 \forall i \in P\})
4: for i ← 1, |S[1]| do
5:   orderings ← orderings + (1..|P|) sorted using \text{compareByObjective}(S, i)
6:   for all k ∈ orderings[i] do
7:     layers[k] ← layers[k] \cup \{k\}
8: (i, j) ← (1, 1)
9: while |iterOrder| < |P| do
10:   k ← ordering[i][j]
11:   if added[k] = \text{False} then (added[k], iterOrder) ← (\text{True}, iterOrder + k)
12:   j ← j + 1
13:   if j > |S[1]| then (i, j) ← (i + 1, 1)
14: for i ← 1, |P| − 1 do
15:   current ← iterOrder[i]
16:   placed[current] ← \text{True}
17:   if current \notin layers[end] then
18:     for j ← i + 1, |P| do
19:       other ← iterOrder[j]
20:       if ranks[current] = ranks[other] \land P[current] < P[other] then
21:         ranks[other] ← ranks[current] + 1
22: else if |\{k \in layers[end] | \neg \text{placed}[k]\}| = 0 then end ← end − 1
23: return result of ordering P, S, and ranks by ranks

ping down to the next one. This provides one ordering that guarantees that no individual can be listed before an individual that dominates it, and enables iteration over the list without having to skip over multiple occurrences of individual indices in NNS’s comparison loop. With the iteration ordering built, NNS iterates over it to perform any necessary domination comparisons. Each individual as it is encountered in this outer loop is flagged as having been placed. If the current individual is not a member of the current last layer, then every subsequent individual in the iteration ordering that is assumed to be a member in the same
front is tested for domination by the current individual, and every dominated individual is
demoted to the next rank. If the current individual is a member of the current last layer,
then the current last layer is tested to see whether it still has any individuals that remain to
be placed. If all of the layer’s individuals have been placed, then the layer counter is demoted
to the previous layer. Once all the individuals’ ranks are determined, the population, scores,
and ranks are all sorted by the ranks and returned as the non-dominated sort’s results.

There is a slight modification that can be made to this algorithm that may result in a
speed-up. Rather than simply demoting the last layer counter when a all its individuals have
been determined to be placed, an $O(MN)$ loop can be used to find the highest-indexed layer
that still has unplaced individuals in it. This potentially can skip subsequent unnecessary
$O(MN)$ passes for that front’s members since that front has already been rejected by this
loop. While I would need to validate this assumption empirically to state this as being a
better approach, I did not perform such an experiment in my studies and simply implemented
NNS with this approach. At the very least, it does not hurt the ultimate computational
complexity of the algorithm.

Verma et al. reported NNS’s computational complexity as $O(MN^2)$ [23]. They do not
specifically report the memory requirements, but it requires an additional $O(MN)$ memory
to hold the orderings in addition to the $O(CN)$ memory required to generate the rankings.
Like DCS and LIS, it is a MOP-only, Pareto-domination-dependent algorithm that cannot
be improved through parallelization (other than the initial $M O(N \log N)$ sorts). It is also a
simpler algorithm than either. However, its limitations of incompatibility with COPs and
inflexibility for swapping comparison schemes means that it must be better than both DCS
and LIS to be useful in its niche.

### 2.4.8 McClymont and Keedwell’s Climbing and Deductive Sorts

In 2012, McClymont and Keedwell published a paper introducing four new non-dominated
sorting algorithms: the Climbing Sort (CS), Flagging Sort (FS), Listing Sort (LS), and De-
ductive Sort (DS) [24]. CS is so named because it ”climbs” the domination graph toward
the least dominated front remaining to be ranked in a population. FS and LS are variations
of CS that remember more of the domination graph explored than CS does to decrease the
number of comparisons required. DS is a modification of Goldberg’s NS that takes advantage
of the transitive relationships during NS’s triply nested loop to skip comparisons that can
be guaranteed to be unnecessary. All four of these algorithms were intentionally designed by the authors to be able to handle COPs as well as MOPs and to be capable of using any domination scheme [24]. While FS and CS have their merits, they have been omitted from the literature review and experiments because McClymont and Keedwell found that their running times were worse than CS and DS [24].

CS combines NS's front-building approach with Pareto domination transitivity to reduce the number of comparisons as follows. If any individual is found to be dominated by any other individual in the population subset remaining to be sorted, then that individual does not need to be considered in any further comparisons for the current front. The explanation comes from Table 2.2. If an individual $\vec{w}$ cannot be a member of the current front because it is dominated by an individual $\vec{u}$, another individual $\vec{v}$ that is dominated by $\vec{w}$ will correctly be determined not to be a member of the current front because it will also be dominated by $\vec{u}$. As shown in [20], this must hold for any domination scheme, not just Pareto domination, for non-dominated sorting to work. This can markedly reduce the number of comparisons required to determine the members of each front.

Algorithm 13 shows pseudocode for CS. Rather than using for loops and automatically incremented iteration variables like NS, CS uses while loops with a couple iteration variables that are set as the algorithm runs. same is the index of the next individual to test for membership in the current front. It is will either refer to an individual that was found to be mutually non-dominated with an individual that has been placed in the current front or to test an individual that was found to dominate the current individual. next is the index of an individual that has been found to be dominated by an individual in the current front, and is used to start the testing for the next front. As such, it is initialized to 1, the index of the first individual in the population.

The outer while loop contains the testing for membership in the current front, allowing the fronts to be filled until all the population’s individuals have been ranked. The previous iteration’s last-found dominated individual is used to start the testing for the current rank. This heuristic can help avoid starting testing membership for the front with a highly dominated individual, which in turn is more likely to dominate other individuals and therefore reduce the total number of comparisons required for the current pass. The inner while loop tests individuals for membership in the current front until all the candidates have been placed in the current front or demoted to the next one. It does this by comparing the current individual with every other individual that is currently considered as a candidate for the cur-
Algorithm 13 CS

1: function CS(P, S)

2: \((p, \text{placed}, r, \text{rank}, \text{ranked}, \text{ranks}, s) \leftarrow (\text{nil}, (\text{False} \ \forall \ i \in P), \text{nil}, 1, 0, (1 \ \forall \ i \in P), \text{nil})\)

3: \((\text{next}, \text{same}) \leftarrow (1, 1)\)

4: \textbf{while} ranked < |P| \textbf{do}

5: \hspace{1em} \(i \leftarrow \text{next}\)

6: \hspace{1em} \textbf{while} \(i \geq 1 \land \text{ranked} < |P|\) \textbf{do}

7: \hspace{2em} \text{same} \leftarrow 0

8: \hspace{2em} \textbf{for} \(j \leftarrow 1, |P|\) \textbf{do}

9: \hspace{3em} \textbf{if} \(i \neq j \land \text{ranks}[j] = \text{rank} \land \neg\text{placed}[j] \textbf{then}

10: \hspace{4em} \text{relationship} \leftarrow \text{BETTER}(\vec{f}(P[i]), \vec{f}(P[j]), \vec{g}(P[i]), \vec{g}(P[j]))

11: \hspace{4em} \textbf{if} \text{relationship} = 1 \textbf{then} \ (\text{next}, \text{ranks}[j]) \leftarrow (j, \text{rank} + 1)

12: \hspace{4em} \textbf{else if} \text{relationship} = 2 \textbf{then}

13: \hspace{5em} \ (\text{next}, \text{ranks}[i], \text{same}) \leftarrow (i, \text{rank} + 1, j)

14: \hspace{4em} \text{break}

15: \hspace{4em} \textbf{else} \text{same} \leftarrow j

16: \hspace{4em} \textbf{if} \text{ranks}[i] = \text{rank} \textbf{then}

17: \hspace{5em} (\text{placed}[i], \text{ranked}) \leftarrow (\text{True}, \text{ranked} + 1)

18: \hspace{5em} \ (p, r, s) \leftarrow (p + P[i], r + \text{rank}, s + S[i])

19: \hspace{5em} i \leftarrow \text{same}

20: \hspace{1em} \text{rank} \leftarrow \text{rank} + 1

21: \hspace{1em} \textbf{return} \ (p, s, r)

The testing is rather simple. If any domination is found at all, the next front’s starting index is set to the index of the dominated individual, and its rank is set tentatively to the next rank. If the dominated individual is the current individual for consideration, or if the individuals are mutually non-dominated, then the next individual to consider is set to the index of the dominating individual. If the current individual is demoted from candidacy for
the current front, then it should not be tested against other individuals, because it cannot dominate any more individuals than its dominating individual can. If after all the testing the individual is not dominated, then it is placed solidly into the current front, and the ordering lists that non-dominated sorts should return are updated with the current individual so that the list can be returned without any extra iteration. Once the testing is finished, the ordered lists are returned.

As was said above, DS is a slight modification of NS that employs the same understanding of domination transitivity to reduce comparisons as CS. When DS tests an individual to see whether it can be a member of the current front, it only compares that individual against individuals that follow it in the population’s original ordering. If an individual $\vec{u}$ is found to dominate a subsequent individual $\vec{v}$, any individuals that $\vec{v}$ dominates will also be eliminated from consideration for membership in the front because $\vec{u}$ must dominate them, too. If, instead, $\vec{u}$ is found to be dominated by a subsequent $\vec{w}$, $\vec{w}$ will dominated any other individuals $\vec{u}$ dominates. Whenever an individual is tested for membership in the current

---

**Algorithm 14 DS**

1: function $DS(P,S)$
2: $\langle p, r, \text{rank}, \text{ranked}, \text{ranks}, s \rangle \leftarrow \langle \text{nil}, \text{nil}, 1, 0, \{1 \forall i \in P\}, \text{nil} \rangle$
3: while ranked $< |P|$ do
4:   for $i \leftarrow 1, |P|$ do
5:     if ranks[$i$] $\neq$ rank then continue
6:     for $j \leftarrow i + 1, |P|$ do
7:       if $i = j \lor$ ranks[$j$] $\neq$ rank then continue
8:       relationship $\leftarrow$ Better($\vec{f}(P[i]), \vec{f}(P[j]), \vec{g}(P[i]), \vec{g}(P[j])$)
9:       if relationship $= 1$ then ranks[$j$] $\leftarrow$ rank + 1
10:      else if relationship $= 1$ then
11:         ranks[$i$] $\leftarrow$ rank + 1
12:         break
13:     if ranks[$i$] $= \text{rank}$ then
14:       $\langle p, r, \text{ranked}, s \rangle \leftarrow \langle p + P[i], r + \text{rank}, \text{ranked} + 1, s + S[i] \rangle$
15:     rank $\leftarrow$ rank + 1
16: return $(p, s, r)$
front, it is because no individual that precedes it in the ordering dominates it, so the only individuals that can dominate it are to be found after it. This, perhaps moreso than any other COP-compatible sorting algorithm, reduces the number of comparisons needed to rank individuals. This brief explanation is sufficient to show how Algorithm 14 implements DS and shows that it works.

McClymont and Keedwell report CS and DS both as having $O(MN^2)$ computational complexity and $O(N)$ memory requirements beyond the $O(CN)$ requirements for the result [24]. Neither of these algorithms can be parallelized, but both of these algorithms exploit the relationships among the individuals in a population to reduce the number of comparisons required to complete the sort. These are among the best COP-compatible and domination-scheme flexible algorithms in the literature.

2.5 Conclusion

In this chapter I covered the meaning of non-dominated sorting by explaining the math behind Pareto domination and ranking individuals into non-dominated fronts. Then I discussed MOEAs and presented NSGA-II as a standard in the literature and the algorithm that I used for my experiments. I then discussed nine sorting algorithms from the literature, discussing how they divide individuals into fronts and the approaches they took to reduce the number of comparisons required to determine the fronts. In the next chapter, I introduce a concept for reducing comparisons that does not appear in the literature, and some algorithms to try to exploit this approach.
Chapter 3

NIS, NMS, and MDTS

In Chapter 2, we saw several techniques proposed to optimize non-dominated sorting. DCS, LIS, and NNS use sorting on each objective score to determine individuals that can never dominated by others in the set; their strategies are limited to MOPs and Pareto domination and are criticized as inflexible [24]. BNS exploits previously performed comparisons to make inferences for as many comparisons as possible. DTS uses a divide-and-conquer approach that builds a tree that maintains dominance some information so that demoting the root of a subtree also demotes its children without needing those comparisons. CS and DS use discovered dominations to skip testing individuals for membership in a front with the aim of finding the individuals that dominate the most individuals to thus reduce the required number of dominations. FNS can have its comparisons parallelized as PNS for a massive performance boost, and DTS can also use threading for its divide portion. NS, FNS, PNS, BNS, DTS, LIS, CS, and DS can also use an optimization specifically for use in MOEAs in which in they stop ranking individuals after the count of individuals placed into the top fronts equals or exceeds the size of the archive.

Each of these algorithms share a weakness. If you pass a subset of a population that was sorted by one of these algorithms back into the algorithm, you lose all the results of the comparisons found by the previous sort for that subset. Each of these algorithms assume nothing is known about the relationships among any of the individuals and the ranking is done from scratch. An MOEA like NSGA-II uses a non-dominated sorting algorithm to combine an already internally ranked archive with a new generation of unranked individuals of the same size. This means that a large amount of work is duplicated each call!
For example, FNS makes $2N^2 - N$ comparisons when combining an archive with $N$ individuals and a child generation with another $N$ individuals. The individuals in the archive have already been compared against each other, but will account for $\frac{1}{2}N^2 - \frac{1}{2}N$ of the comparisons FNS will make in a new pass. This means that at most $\frac{3}{2}N^2 - \frac{1}{2}N$ are necessary to determine the new rankings for the individuals in the new, combined population.

In their present forms, none of these algorithms are designed to receive a structure that represents information learned from previous calls to expedite determining the rankings for a population. There is no reason for a sort to not be able to take advantage of this information, especially since the primary use for non-dominated sorting algorithms is in elitist MOEAs. It makes more sense to design non-dominated sorts that can combine transitive domination with a memory to truly combine two populations as opposed to starting from scratch every call.

In this chapter, I present three algorithms that do just that. First, I present my Non-dominated Insertion Sort (NIS), which inserts one new individual into a sorted population at a time while guaranteeing the correct ranking of the individuals at the end. Next, I show my Non-dominated Merge Sort (NMS) which combines NIS, a divide-and-conquer approach, and a domination table to provide a parallelized non-dominated sort\(^1\). Finally, I present a Modified Dominance Tree Sort (MDTS) which returns and receives the dominance tree determined from a call to DTS so that it can avoid at least some of the work performed by the previous call. I explain each of these algorithms, provide pseudocode, and discuss their strengths and weaknesses.

### 3.1 Non-dominated Insertion Sort (NIS)

The Non-dominated Insertion Sort (NIS) is a non-dominated sorting algorithm that works by inserting a new individual into a pre-sorted population to determine the correct final rankings. It is somewhat similar to DTS except that works directly with fronts as opposed to building a dominance tree. If the front an individual is placed into contains individuals that the new individual dominates, then those individuals and all the individuals they directly

---

\(^1\)Upon reconsideration of [3] before writing this paper, I realized that there were remarkable similarities between my NMS and Fang et al.’s DTS. I developed my NIS before having read the DTS paper and NMS about six months after having read it. While I was not thinking about DTS when I developed it, I think it is fair to say that I figured out how to structure NMS because of my knowledge of DTS. Still, I did develop the algorithm on my own and used my NIS as a springboard, so I am claiming the algorithm as my own work.
or transitively dominate are all demoted to the next lower front. DTS instead postpones determining all the fronts until all the individuals in a population are placed in a dominance tree.

NIS works because of Equation 2.5 and Pareto transitivity as shown in Table 2.2. By Equation 2.5, an individual is placed in the best front it can be, either by being placed in the front after its most dominated dominating individual or by being placed in the first front if it is non-dominated in reference to the population. So NIS starts by comparing the new individual against every individual in the first front. If an individual is found to dominate the new individual, then the new individual cannot belong in the first front, so it is tested for membership in the next front. If the individual is mutually non-dominated with every individual with the front in question, then it belongs in the current front. If the individual dominates any individuals in the current front, it belongs in the current front, but the dominated individuals that are currently in the front, and all the individuals they dominate, need to be demoted to reflect the fact that their most dominated dominating individual is now one rank lower than it was previously. It is impossible for the new individual to both dominate one individual in a non-dominated front and to be dominated by another individual in the same front, as shown in Table 2.2 and [20].

Algorithm 15 shows pseudocode for a list-based implementation for NIS\textsuperscript{2}. Its starting point is the \texttt{NIS} function, which receives some arguments beyond the populations’ decision vectors and constraint and objective score vectors. The \texttt{start} parameter gives the index of the first unsorted individual in the population, and the understanding is that \texttt{F} holds a list of index lists representing the fronts determined for the preceding indices. In the default case, the entire population needs to be sorted, and \texttt{start} and \texttt{F} default to 1 and \texttt{nil}. NIS then ranks all the unranked individuals by using the \texttt{insert} function to add them into the previously determined fronts. \texttt{insert} follows the procedure described above, finding the highest front that contains no individuals that dominate the new individual, and cascading demotions of any individuals in that front that the new individual dominates. Once all the individuals have been ranked, it builds and returns not only the ordered lists of decision vectors, constraint and objective score vectors, and ranks that the other non-dominated sorts

\textsuperscript{2}My source code provides an array-based implementation. It requires a little overhead for dividing a single array into fronts, but it is more efficient than the list-based version because it easily handles demoting individuals into the next front.
Algorithm 15 NIS

1: function NIS(P, S, start = 1, F = nil)
2: for i ← start, |P| do F ← INSERT(F, i, P, S)
3: return buildOrderFromFronts(P, S, F)
4: function INSERT(F, i, P, S)
5: (f, placed) ← (1, False)
6: while f ≤ |F| ∧ ¬placed do
7: placed ← True
8: for j ← 1, |F[f]| do
9: relationship ← BETTER(f(P[i]), f(F[f][j]), g(P[i]), g(F[f][j]))
10: if relationship = 2 then
11: (f, placed) ← (f + 1, False)
12: break
13: else if relationship = 2 then
14: dominated ← ⟨F[f][j]⟩ + ⟨k ∈ F[j + 1 : ] | P[i] < P[k]⟩
15: (F[f], h) ← (F[f] − dominated, j + 1)
16: while dominated ≠ nil do
17: if h > |F| then
18: F ← F + dominated
19: break
20: next ← ⟨k ∈ F[h] | ∃m ∈ dominated : P[m] < P[k]⟩
21: F[h] ← F[h] − next + dominated
22: (dominated, h) ← (next, h + 1)
23: break
24: if placed then F[f] ← F[f] + i
25: if ¬placed then F ← F + ⟨i⟩
26: return F
27: function buildOrderFromFronts(P, S, F)
28: (fronts, next, p, s, r, rank) ← (nil, 1, nil, nil, nil, 1)
29: for all f ∈ F do
30: fronts ← fronts + nil
31: for all i ∈ f do
32: (fronts, next, p, s, r) ← (fronts[−1] + next, next + 1, p + P[i], s + S[i], rank)
33: rank ← rank + 1
34: return (p, s, r, fronts)
NIS, NMS, AND MDTS

return, but also a restructured front list so the next call can refer to the correct individuals. This is done with the buildOrderFromFronts function.

NIS is an intuitive non-dominated sorting algorithm that is COP-compatible and domination scheme agnostic while building a structure that preserves some of the information learned from previous passes to the algorithm. Its worst-case computational complexity is $O(MN^3)$ with an average computational complexity of $\theta(MN^2)$ and it has a memory requirement of $O(N)$ for building the fronts beyond the $O(CN)$ return requirements. The computational complexity is dominated by the cascading demotions, as is discussed below. NIS is well-suited for use with steady-state multi-objective genetic algorithms (MOGAs), since they immediately add children to the breeding pool. It is not naturally parallelizable because of the difficulties introduced by the cascading demotions, but as we will see in the next section, this can be gotten around to an extent with a divide-and-conquer approach. Because it returns a front list that has indices that refer to the individuals in the post-sorting order, it is also a very simple matter for an MOEA to perform selection on the archive and truncate the returned lists for the next call. The front that overflows the archive can be sorted in-place using the MOEA’s tie-breaking mechanism. After this sort, the individuals at the end of the list that will not fit can be removed, as can any fronts that follow the truncated front.

NIS is dissatisfactory in one regard. The cascading demotions can potentially require NIS to repeat comparisons when determining which individuals to demote. While it is not shown in the pseudocode, the case in which all the individuals in a front are getting demoted is handled simply by inserting a new front in front of the dominated fronts. In the worst case, a demotion will demote $\left\lceil \frac{1}{2}N \right\rceil$ individuals into a lower front with $N - \left\lceil \frac{1}{2}N \right\rceil - 1$ individuals, which will require $\left\lceil \frac{3}{4}N^2 - \frac{1}{2}N \right\rceil$ duplicated comparisons to perform the cascaded demotions. It is more likely that the running time complexity will be $\sum_i |f_i| \times |f_{i+1}|$ for the $i$ fronts that have individuals to be dominated, where $|f_{i+1}| + \sum_i |f_i| < N$. This means that the more fronts or the fewer dominations there are, the less the cost of the cascading demotions, and there are several cases (such as for completely mutually non-dominated populations) where the cost does not exist or is so small that it can be ignored. Nevertheless, if there were some way to automatically get the demotions the way DTS does, NIS would be a much more reliable algorithm.
3.2 Non-dominated Merge Sort (NMS)

The Non-dominated Merge Sort (NMS) is an improvement on the NIS. A divide-and-conquer algorithm, it splits a population in two, sorts the two partitions independently, then uses a modified version of NIS to insert one of the partitions into the others’ ordering. Because each partition has sorted its own half independently of the other, it will have discovered at least some of the relationships among the individuals in its part. NMS stores this information in a domination table, then inserts individuals in the second, or right, partition into the first, or left, partition in by their order in the second partition’s determined fronts. When an individual’s home front in the left partition is determined, that front’s address or index is cached. So, when an individual’s turn comes to be inserted into the left partition, all the individuals that dominate it have already been added into the left partition. NMS performs a look-up to determine which of an individual’s known dominating individuals is the most dominated in the left partition, then starts the insertion from the front after that one. This can allow for unnecessary comparisons to be skipped. In addition, a dominance table allows for the results of previous comparisons to be looked up as well instead of performing the comparisons again.

Algorithms 16 through 18 show pseudocode for NMS. It begins with the function NMS in Algorithm 16. By default, it assumes that the entire population has been sorted. However, if an initial part of the population has been sorted, then a different starting index and the list of fronts for the sorted part can be supplied. NIS starts by creating a domination table where $-1$ is stored as a code to show that no comparisons have yet been performed, and a list of ranks for the individuals in the population starting with the value 1. If part of the population is already sorted, then the updateRanks is used to set the ranks of the sorted individuals correctly. Whatever part of the population that has not been sorted is then sorted using the NMSrecurse function. If the population had part of the population initially sorted, then the sorting for the two partitions are merged together with NMSmerge. Then the population, its scores, and the fronts are passed to NIS’s buildOrderFromFronts, and its result is returned as the result of the sort.

NMSrecurse performs NMS’s divide-and-conquer. Its base case is that it has been passed the same value for its start and end indices, which means that it is sorting a population of one individual. In this case, it returns a list containing another list whose only element is the passed index. Otherwise, it divides the population into two, calls NMSrecurse to rank the
Algorithm 16 NMS, part I

1: function NMS(P, S, start = 1, F = nil)
2: table ← a $|P| \times |P|$ array with -1 stored in each cell
3: ranks ← (1 ∀ i ∈ P)
4: if start > 1 then UPDATERANKS(P, S, start - 1, F, ranks)
5: $F_r ← \text{NMSRECURSE}(P, S, \text{start}, |P|, \text{table}, \text{ranks})$
6: if start > 1 then $F_r ← \text{NMSMERGE}(P, S, F, F_r, \text{start}, |P|, \text{table}, \text{ranks})$
7: return BUILDORDERFROMFRONTS(P, S, F_r)
8: function UPDATERANKS(P, S, end, F, ranks)
9: for all $i ← 1, |F|$ do
10: for all $j ∈ F[i]$ do ranks ← $i$
11: function NMSRECURSE(P, S, start, end, table, ranks)
12: if start = end then return $\langle\langle\text{start}\rangle\rangle$
13: split ← $\left\lceil \frac{\text{start} + \text{end}}{2} \right\rceil$
14: $F_l ← \text{NMSRECURSE}(P, S, \text{start}, \text{split} - 1, \text{table}, \text{ranks})$
15: $F_r ← \text{NMSRECURSE}(P, S, \text{split}, \text{end}, \text{table}, \text{ranks})$
16: return NMSMERGE(P, S, F_l, F_r, table, ranks)
17: function NMSMERGE(P, S, F_l, F_r, table, ranks)
18: for $f_r ← 1, |F_r|$ do
19: for $i ∈ F_r[f_r]$ do NMSINSERT(P, S, F_l, i, f_r, table, ranks)
20: return $F_l$
21: function NMSINSERT(P, S, F_l, i, f_r, table, ranks)
22: $f_l ← \text{GETSTARTINGFRONT}(P, S, F_r, i, f_r, \text{table}, \text{ranks})$
23: PLACEINDIVIDUAL(P, S, F_l, i, f_l, table, ranks)
24: function GETSTARTINGFRONT(P, S, F_r, i, f_r, table, ranks)
25: $f ← 1$
26: if $f_r > 1$ then
27: for $j ∈ F_r[f_r - 1]$ do
28: if table[i][j] = 2 ∧ ranks[j] > f then $f ← \text{ranks}[j] + 1$
29: return $f$
Algorithm 17 NMS, part II

30: function PLACEINDIVIDUAL($P, S, F_i, i, f_i$, table, ranks)
31: \hspace{1em} ranks[$i$] $\leftarrow f_i$
32: \hspace{1em} if $f_i > |F_i|$ then return $F_i + \langle i \rangle$
33: \hspace{1em} for $j$ $\leftarrow 1, |F_i[f_i]|$ do
34: \hspace{2em} $k$ $\leftarrow F_i[f_i][j]$
35: \hspace{2em} relationship $\leftarrow$ LOOKUPORCOMPARE($P, S, i, k$, table)
36: \hspace{2em} if relationship $= 2$ then return PLACEINDIVIDUAL($P, S, F_i, f_i + 1$, table, ranks)
37: \hspace{2em} else if relationship $= 1$ then
38: \hspace{3em} $F_i$ $\leftarrow$ FINDANDDEMOTE DOMINATED($P, S, F_i, \langle i \rangle, f_i, j$, table, ranks)
39: \hspace{2em} return
40: \hspace{1em} $F_i[f_i] + i$ $\leftarrow F_i[f_i] + i$

41: function LOOKUPORCOMPARE($P, S, i, j$, table)
42: \hspace{1em} if table[$i$][$j$] $= -1$ then
43: \hspace{2em} table[$i$][$j$] $\leftarrow$ BETTER($\bar{f}(P[i]), \bar{f}(P[j]), \bar{g}(P[i]), \bar{g}(P[j])$)
44: \hspace{2em} if table[$i$][$j$] $= 0$ then table[$j$][$i$] $\leftarrow 0$
45: \hspace{2em} else if table[$i$][$j$] $= 1$ then table[$j$][$i$] $\leftarrow 2$
46: \hspace{1em} else table[$j$][$i$] $\leftarrow 1$
47: \hspace{1em} return table[$i$][$j$]

48: function FINDANDDEMOTE DOMINATED($P, S, F_i$, dominating, $f_i$, start, table, ranks)
49: \hspace{1em} if dominating $= \text{nil}$ then return
50: \hspace{1em} else if $f_i > |F_i|$ then $F_i$ $\leftarrow F_i + \text{dominating}$
51: \hspace{1em} else
52: \hspace{2em} dominated $\leftarrow$ FIND DOMINATED BY($P, S, F_i$, dominating, $f_i$, start, table)
53: \hspace{2em} for all $j \in \text{dominated}$ do DEMOTE($F_i, f_i, j$, ranks)
54: \hspace{2em} $F_i[f_i] + \text{dominating}$ $\leftarrow F_i[f_i] + \text{dominating}$
55: \hspace{1em} return

two partitions, and then merges the two partitions’ rankings with NMSmerge, which iterates over all the individuals in the right partition’s fronts and uses NMSinsert to insert them into the left partition. The left partition’s front list, which now represents all the individuals from both partitions, is returned as the result.
Algorithm 18 NMS, part III

```
56: function FINDDOMINATEDBY(\(P, S, F_i, \text{dominating}, f_i, \text{start}, \text{table}\))
57:     dominated ← nil
58:     for \(j \leftarrow \text{start}, |F_i[f_i]|\) do
59:         \(k \leftarrow F_i[f_i][j]\)
60:         for all \(i \in \text{dominating} \) do
61:             if LOOKUPORCOMPARE(\(P, S, i, k, \text{table}\)) = 1 then
62:                 dominated ← dominated + \(k\)
63:             break
64:         return dominated
65: function DEMOTE(\(F_i, f_i, j, \text{ranks}\))
66:     \(F_i[f_i] \leftarrow F_i[f_i] - j\)
67:     \(\text{ranks}[j] \leftarrow f_i + 1\)
```

NMSinsert starts the process that is similar to NIS’s insert function. Instead of directly performing the insertion process starting with the left partition’s first front, it uses the getStartingFronts function to determine the highest front that could contain the current individual based upon where that individual’s dominating individuals were placed. Then it calls placeIndividual to find in which front the individual belongs, perform any necessary cascading demotions, and finally place the individual.

getStartingFront exploits Equation 2.5 to enable skipping comparing an individuals with fronts that we already know contain at least one individual that dominates the individual. If the current individual is not placed in the first front for the right side, then at least one individual in the right side’s previous front must dominate it. By iterating over the individuals in the previous front and checking which fronts on the left side its dominating individuals were placed, NIS can determine which of the dominating individuals was ranked lowest when placed in the left side. The current individual must be placed at best in the front after this individual’s front, so the insertion process should start with this front.

placeIndividual is found in Algorithm 17. It uses a recursive approach for finding the highest-ranked front that contains no individuals that dominate the current individual to be placed. It first sets the individual’s rank to the current left front’s index. It then checks the base case: if the current index is beyond the end of the left partition’s fronts list, then all the left’s fronts contain an individual that dominates the current individual,
so it adds a new front to the end of the fronts list that contains that individual’s index. Otherwise, the individual is compared against the individuals in the current front using the lookupOrCompare function. If the front contains any individual that dominates the current individual, then placeIndividual is called recursively for the next front. If the current individual dominates any individual in the current front, then the individual belongs in the current front, and the cascading demotion is handled by the findAndDemoteDominated function, which also adds the current individual’s index to the current front. If the individual is mutually non-dominated with every individual in the current front, then the individual belongs in the current front, so its index is added to the front with no extra work required.

lookupOrCompare is a very important function for NMS’s efficiency. Before it compares individuals, it checks the domination table to see whether they’ve already been compared. If so, it just returns the cached result. Otherwise, it performs the comparison using the Better function, then stores the results so the relationship can be looked up in either direction. While this does not solve the $O(N^2)$ checks to determine whether an individual should be demoted during the cascading dominate, it does remove the $M$ coefficient by allowing for an $O(1)$ look-up.

findAndDemoteDominated performs cascading demotions until all the individuals that are dominated by the inserted individual have been demoted. It is a recursive function with two base cases. If the passed list of dominating individual is nil, then there are no more dominations to cascade. If the passed front index is beyond the end of the left partition’s front list, then the dominating list is appended as a new front at the end of the left’s fronts list. Otherwise, the dominating individuals need to be compared against the individuals in the current front to find the individuals to demote. A list of dominated individual indices is generated by the findDominatedBy function in Algorithm 18. Each of these individuals is demoted using the demote function from Algorithm 18, which simply removes the index from the current front’s list and increments the demoted individual’s rank. The dominating individuals’ indices are then appended to the current front’s index list. The process continues by recursively calling findAndDemoteDominated for the dominated individuals on the next front.

Like NIS, NMS has a worst-case computational complexity of $O(MN^3)$ due to the domination of the demotion function and an average running time of $\theta(MN^2)$. It has a $O(N^2)$ memory requirement for storing the domination table. It is parallelizable because separate population partitions can be sorted simultaneously with no synchronization beyond detecting
when threads end before merging results. It can use the information from the domination table to perform $O(1)$ look-ups during cascading demotions where NIS would repeat the $O(M)$ comparisons. Unfortunately, this does not change $O(MN^2)$ complexity of the cascading demotions because there is no guarantee that individuals being demoted have already been tested against every individual in the next front. In addition, the domination table is prohibitively large for large $N$; I have found that the running time of NMS for large $N$ turns out to be dominated by cache misses as the section of memory that contains the table portion that is needed for the look-up is almost never in memory. There is also no realistic way to preserve a domination table in between calls because of the reordering of the population. Creating the table and reordering a table both require $O(N^2)$ memory, but $O\left(\frac{3}{2}N^2 - \frac{1}{2}N\right)$ cells would have to be overwritten after the truncation anyway. Based upon some timings I saw during the initial development of the algorithm, I concluded that expensive cascading demotions happen infrequently enough that there is little merit in performing the copy. It would be best if the discovered domination information could be preserved in between calls beyond what is represented by the fronts.

### 3.3 Modified Dominance Tree Sort (MDTS)

The final algorithm I propose for preserving information between calls is a Modified Dominance Tree Sort (MDTS). It is almost identical to DTS except that it returns a dominance tree that correlates to the returned ordering that can be accepted as an input for the next call. Building the dominance tree for either a whole or partial population is exactly the same as in DTS, and then a passed in tree can be merged with the tree formed for the rest of the population as a result. The only wrinkle is that a dominance tree needs to be truncated with the population, which requires a different function for building fronts from a dominance tree.

Algorithms 19 and 20 show pseudocode for MDTS. Its MDTS function takes a starting index called `start`, a previously generated dominance tree called `tree`, a selection operator to perform truncation called `select`, and a maximum number of individuals to keep called `individualsToKeep`. `select` is a function that takes a population, its scores, and a list of indices in the population to generate an alternate ordering of that list of indices so the most desirable individuals ordered first. For example, it could order these individuals to maximize Deb et al.’s crowding score [1]. For an initial pass, the default values for the
Algorithm 19 MDTS, part I

1: function MDTS(P, S, start = 1, tree = NULL, select, individualsToKeep = |P|)
2:    tree_all ← BUILDDOMINANCETREE( P, S, start, |P|)
3:    if tree ≠ NULL then
4:        tree_all ← MERGEDOMINANCETREES(P, S, tree, tree_all)
5:    return GETTRUNCATEDORDERFROMTREE(P, S, tree_all, select, individualsToKeep)
6: function GETTRUNCATEDORDERFROMTREE(P, S, tree, select, individualsToKeep)
7:    fronts ← a copy of tree
8:    (front, indices, nextIndex, p, r, s, rank) ← (fronts, (0 ∀ i ∈ P⟩, 1, nil, nil, nil, 1)
9:    function UPDATEORDERINGS(i)
10:    (indices[i], p, r, s) ← (nextIndex, p + P[i], r + rank, s + S[i])
11:    nextIndex + nextIndex + 1
12:    while nextIndex ≤ individualsToKeep ∧ front ≠ NULL do
13:       (frontList, node) ← (nil, front)
14:       while node ≠ NULL do (frontList, node) ← (frontList + node.data, node.sibling)
15:       c ← |frontList| + nextIndex − 1
16:       if c ≤ individualsToKeep then
17:          node ← front
18:          while node ≠ NULL do
19:             UPDATEORDERINGS(node.data)
20:           if c < individualsToKeep ∧ node ≠ front ∧ node.child ≠ NULL then
21:              front.child ← MERGEDOMINANCETREES(P, S, front.child, node.child)
22:          node ← node.sibling
23:       else
24:          frontList ← SELECT(P, S, frontList)
25:          toTake ← individualsToKeep − nextIndex + 1
26:          for i ← 1, toTake do
27:             UPDATEORDERING(frontList[i])
28:       (front, rank) ← (front.child, rank + 1)
29:    return (p, s, r, BUILDTRUNCATEDTREE(tree, indices))
Algorithm 20 MDTS, part II

30: function buildTruncatedTree(tree, indices)
31:     (node, prev, root) ← (tree, NULL, NULL)
32:     while node ≠ NULL do
33:         if indices[node.data] ≠ 0 then
34:             node.data ← indices[node.data]
35:         if node.child ≠ NULL then
36:             node.child ← buildTruncatedTree(node.child, indices)
37:         if root = NULL then root ← node
38:         prev ← node
39:         else if prev ≠ NULL then prev.sibling ← node.sibling
40:         node ← node.sibling
41:     return root

arguments are as shown in Algorithm 19. MDTS sorts the population between start and |P| to get a dominance tree for the unsorted part of the population. If part of the population was already sorted, it then merges the trees together with mergeDominanceTrees to get a dominance tree representing the whole population. Finally, it calls and returns the result of getTruncatedOrderFromTree, which are the truncated decision vector, constraint and objective score vector, and rank lists the other sorts return and a dominance tree that is contains the population’s individual indices in such a way that all the relevant information from the original tree remains.

getTruncatedOrderFromTree’s primary purpose is to build the orderings required and to track what each individual’s index in the final population is so the tree can be modified by buildTruncatedTree. It starts by initializing the same variables as DTS’s getOrderFromTree plus a vector to contain each individual’s final index called indices and a counter showing what the final index of the next placed individual should be. Then it iterates through the fronts tree it builds to determine the fronts until either all the maximum allowed number of individuals have been placed in the ordering or until all the individuals have been placed in the population.

When examining the front, the first thing getTruncatedOrderFromTree is determine whether all of the individuals in the tree will fit in the ordering, so it counts the node in the current subtree’s sibling list. If all of the individuals will fit, then all the nodes are added
to the ordering using the local function or macro `updateOrderings`. This function or macro also stores the individual’s final index and increments the next individual’s index counter. As with `getOrderFromTree`, if any of the nodes other than the front’s root has a child when the current front will not exactly fill the ordering, the front’s root’s child and the node’s child are merged together to help determine the next front. If there are too many individuals in the front to fit in the ordering’s remaining space, then only some of the individuals may be picked. A list of the nodes’ data (the indices for the individuals in the front) is sorted by `select`, and starting with the first element, only as many individuals from that list are added to the ordering by `updateOrderings`. It is unnecessary to merge the children in this situation because no individuals from lower fronts will be processed for addition.

This process builds a list that can map an individual’s current index to its index in the ordering. This makes it a simple matter to manipulate and prune the passed dominance tree to only represent the orderings’ individuals. This is done by Algorithm 20’s `buildTruncatedTree`, a recursive function. Its base case is when it receives a `NULL` reference to a tree, in which case it returns `NULL`. Otherwise, it iterates over the passed tree’s root-and-sibling list to build the final tree. It stores the addresses of the return tree’s root, the current node in the iteration, and the last node kept for the returned tree. If the current node’s data maps to a 0, then its referred individual was not placed in the final population, so it is skipped. Otherwise, its data is changed to the individuals’ final index. If the node has a child, its child is set to the result of recursively calling `buildTruncatedTree` on the child. The node is then hooked into the return tree’s root-and-sibling list before continuing to the next node. Once `buildTruncatedTree` has finished processing a subtree, it returns its address. This process builds a tree that contains all the ordering’s individuals and everything that is known about their relationships based upon the original dominance tree.

MDTS has a computational complexity of $O\left(\frac{p+1}{p} MN \lg N\right)$ and no additional memory requirements beyond DTS because it takes $O(N \lg N)$ time to sort a too-large front when $N$ is the size of the whole population, not just the part that’s kept. It retains all the strengths of DTS, and it adds the strength of ”remembering” what has been processed before by returning a dominance tree. Its weakness lies in the dominance tree’s structure. Each node is placed as the child of a node that refers to the most dominated individual found that dominates that child’s node, but it takes pulling the tree apart with possibly many more merges to determine every individual’s ranking. This means that a tree generated by `buildTruncatedTree` that has many subtrees will need to perform the same mergings in the next call to MDTS as this
call’s `getTruncatedOrderFromTree` performed. NIS and NMS have an expensive cascading
demotion scheme that is likely to be performed infrequently, whereas MDTS has a cheaper
scheme that is early guaranteed to repeat work for each generation.

3.4 Conclusion

In this chapter I presented three algorithms that allow return structures that can be
used to reduce the number of comparisons that need to be performed between generations.
These algorithms are COP-compatible and agnostic to domination scheme, use transitive
domination inferences to further reduce the comparisons, and support selection during sort-
ing (NIS and NMS with minor modifications, MDTS as-is) to save computational effort.
In addition, two of these schemes are parallelizable to further reduce running time speed.
However, none of these schemes optimally preserve information between calls. In the next
chapter, I present the experiment I use to compare these algorithms against each other and
the other algorithms in the literature and present my findings on how they fare.
Chapter 4

Experiments and Results

In this chapter I describe an experiment to determine whether preserving relationship information between generations does improved the performance of MOEAs like NSGA-II. I discuss the results of the experiment and assess whether this approach has validity.

4.1 Experimental Design

The experimental goal was to determine whether my three new algorithms run faster than the algorithms in the literature for the successive sorting calls used in MOEAs. To this end, I designed an experiment that tests my new algorithms against those in the literature by using them in NSGA-II as it is used to solve sixteen problems from the literature plus one new one I designed. I used 100, 200, 300, 400, 500, and 1000 as the archive sizes NSGA-II was allowed for the trials (which means that the sort may be required to sort as many as double the archive size to determine which individuals to return). Each trial for the experiment consisted of choosing a random number seed and then running each of the sorts four times each, reseeding the random number generator each time so the four runs perform identically, where each run is used to collect NSGA-II running times, NSGA-II comparison counts, individual sort call running times, and individual sort call comparison counts. Because each of the algorithms can generate different orderings within the same front, NSGA-II did not run identically for each of the algorithms, but using the same random number seed at least guarantees the same starting point for each trial and that the runs will be at least somewhat similar in what is passed to the sorts each generation. I ran ten trials for the archive sizes 100, 200, 300, 400, and 500 and five trials for the archive size of 1000.
MOP Part | Count | Definitions
--- | --- | ---
Decisions | $D = 4$ | $\forall i \in \{1\ldots 4\}, 0 \leq u_i \leq 1$
Objectives | $M = 10$ | $f_1(\vec{u}) = u_1 + u_2$
 | | $f_2(\vec{u}) = u_1 + u_3$
 | | $f_3(\vec{u}) = u_1 + u_4$
 | | $f_4(\vec{u}) = u_2 + u_3$
 | | $f_5(\vec{u}) = u_2 + u_4$
 | | $f_6(\vec{u}) = u_3 + u_4$
 | | $f_7(\vec{u}) = u_1 + u_2 + u_3$
 | | $f_8(\vec{u}) = u_1 + u_2 + u_4$
 | | $f_9(\vec{u}) = u_2 + u_3 + u_4$
 | | $f_{10}(\vec{u}) = u_1 + u_2 + u_3 + u_4$

Table 4.1: The definition for the new MOP called Craig intended to stress algorithms that perform poorly with a large number of objectives.

DCS, LIS, and NNS were omitted from the experiment due to their MOP and Pareto-domination-only restrictions, so NIS, NMS, and MDTS are compared against NS, FNS, PNS, BNS, DTS, CS, and DS. DTS, MDTS, and NMS are parallelizable, so each was run with one thread through twelve threads (one thread for each processor in my testing system) to get an idea as to how taking advantage of threading changed their performance. Since PNS without threading is just FNS, PNS was run with two threads through twelve threads. I report only the best results for each threaded algorithm because tuning the number of threads to the archive size is a trivial matter.

NSGA-II was configured using the selection, crossover, and mutation operators and their required settings described for COPs in [1]. Rather than running NSGA-II for 500 generations for each call, I used fifty generations to get usable results in a shorter amount of time.

The problems from the literature I used are Schaffer, Fonseca, Kursawe, ZDT1, ZDT2, ZDT3, ZDT6, Ozyczka2, Tanaka, Constr Ex, Srivinas, Golinski, Viennet 2, Viennet 3, Viennet 4, and Water. These are scattered throughout the literature, but detailed descriptions for these problems (number of decisions, constraints, and objectives; formulae for constraints and objectives; and descriptions of the optimal Pareto front) are conveniently collected in [25]. In addition to these, I added a simple minimization MOP which I call “Craig” which is
designed to stress algorithms with a large number of objectives. It has four decisions and ten objectives, where each objective is a sum of a different subset of decisions. The Pareto optimal set is the one decision vector \((0, 0, 0, 0)\). A detailed definition of the problem can be found in Table 4.1.

I wrote the non-dominated sorts and the experiment using Microsoft Visual C. I wrote the sorts presented in the literature considering their associated papers. My implementation of NSGA-II’s operators were largely copied from the official implementation provided at [26]. I used Geoff Kuenning’s Mersenne Twister implementation found at [27] for random number generation, which required using a MSVC-compatible implementation of inttypes found at [28] as well. The statistics and graphs reported here were generated by scripts I wrote in Python 3.2 using unofficial translations of scipy, numpy, and matplotlib found at [29].

The code was compiled using the Microsoft Visual C++ 32-bit compiler in Microsoft Visual Studio 2010. The experiment was executed in Windows 7 64-bit on an Intel Core i7 980X twelve-core processor, each clocked at 3.33 GHz, with 24 GB of DDR3 memory clocked at 1333 MHz.

4.2 Results

When multiple cores are available, PNS is the fastest algorithm. When only a single thread is allowed, MDTS is the fastest algorithm, closely followed by NIS.

Figures 4.1 and 4.2 provide a concise explanation for this conclusion. These graphs show the running time totals for all the sorting calls sent to each algorithm during the experiment. Figure 4.1 shows the results for all the algorithms, and Figure 4.2 shows the results when the threaded algorithm runs were excluded. Both have a logarithmic \(y\) (time) axis to help differentiate the algorithms’ performances at the lower archive sizes.

This running time sum provides a better representation of the differences among the sorts than averages would because of differences in the problems. Problems with low constraint plus objective counts (like Schaffer, with \(0 + 2\)) take less time to perform individual comparisons than problems with higher counts (like Golinski, with \(11 + 2\)). Trying to average running times across these problems does not make sense, because the best distribution one could hope for was a normal distribution with a heavy skew toward the slower problems. Furthermore, the differing number of constraints and objectives combined with the geometry of these functions will make some problems require more generations to converge upon a
CHAPTER 4. EXPERIMENTS AND RESULTS

Figure 4.1: Threaded results.

Figure 4.2: Unthreaded results.
one-front population than others. For many algorithms, a one-front population is the worst case because it requires the full $2N^2 - N$ comparisons, so the “harder” problems will make those algorithms look better because fewer generations were spent in the worst case. The running time sum allows for definitive statements to be made easily while still reflecting all the fluctuations to which each run was subject.

In Figure 4.1, it is clear that PNS is the best performer. Its divide-and-conquer mechanism is the most efficient, since it performs all of its comparisons in parallel, as opposed to DTS, NMS, and MDTS which combine two halves of the population synchronously at the end. MDTS, DTS, and NIS run a distant second, third, and fourth. It is clear that no comparison avoidance mechanism can compete with the performance boost of well-separated parallel tasks.

Figure 4.2 shows the results for only the one-threaded runs of each algorithm. In this case, MDTS is the fastest algorithm overall, with NIS coming in at a close second. However, due to the logarithmic scale, it seems clear that NIS will fall further behind at higher archive sizes. The fact that both these algorithms leaves the others behind shows that preserving archive relationships is a mechanism that successfully reduces the running time needed (e.g., compare MDTS to DTS, the bottom-most line to the third line from the bottom), albeit a less valuable mechanism than was expected.

The biggest surprise in this experiment was NMS. It was specifically designed to improve NIS by enabling it to allow for parallelization. Early trials when I was designing it showed it to be bested only by PNS when sorting random populations and returning the fronts for all of them. It was after I rewrote it to add the features to truncate the population that it slowed down. Two likely causes for the slowdown are the need to release dynamically allocated memory multiple times during the run and the recursive functions I used to simplify freeing the front list structures. Further work may lead to better optimizations that might make it a viable algorithm, but ultimately it seems that MDTS will not be removed as the best of the three new algorithms.

4.3 Conclusion

In this chapter I presented an experiment that generated statistics precise enough to determine minor differences between the sorts if necessary. Ultimately, the PNS’s parallelization mechanism breaks down the comparisons so effectively that no other algorithm,
even the other threaded algorithms, come close. If someone running an MOEA does not have access to multiple cores, then MDTS works the best, followed closely by NIS.
Chapter 5

Conclusion

In this thesis, I described the problem of non-dominated sorting. I discussed the algorithms in the literature for solving this problem, detailing the mechanisms their creators introduced or used to optimize their algorithms’ performances. I proposed a new mechanism for preserving relationships among the individuals between generations, and presented two new algorithms, NIS and NMS, and a modification of one from the literature, MDTS, that do this. After running these algorithms on seventeen problems for different archive sizes for multiple trials, I determined that MDTS and NIS perform better than all algorithms in the literature other than PNS, which has a superior parallelization mechanism. My recommendation is to use PNS whenever multiple cores are available to an experimenter, and MDTS whenever they are not.

There is some further work that could be done to explore my new mechanism. It seems that my NMS implementation was sub-par, so a careful rewrite could possibly lead to different findings. PNS might also be found to be able to benefit by copying the domination table, lists, and counts between generations, though my estimate is that it will create too much overhead to be viable. Barring some breakthrough that I do not foresee, PNS may be the best non-dominated sorting algorithm.
References


REFERENCES


