An Algorithm to Find Efficient Supported Solutions of Non-Convex Multiobjective Optimization Problems

by

Nell Kiyoko Elliott

Sc.B., Brown University, 2011

Submitted in partial fulfillment of the requirements for the

Degree of Master of Science in the Department of Computer Science at Brown University

Providence, Rhode Island

May 29, 2011

This thesis by Nell Kiyoko Elliott is accepted in its present form
by the Department of Computer Science as satisfying the thesis requirement
for the degree of Master of Science.

Date _____ _____

Professor Pascal Van Hentenryck, Advisor

Approved by the Graduate Council

Date _____ _____

Peter M. Weber
Dean of the Graduate School

Vita

Nell Kiyoko Elliott was born in Evanston, Illinois. She graduated from Evanston Township High School in 2007, and was awarded the Council of Presidential Awardees in Mathematics Scholarship, the Metropolitan Mathematics Club of Chicago Scholarship, and the Eleanor Kornhaber Award for Women in Mathematics in her senior year. She entered Brown University in the fall of 2007, where she studied mathematics and computer science. She was admitted into Brown's Concurrent Bachelor's/Master's Degree Program in the spring of 2010, and will complete both her Bachelor of Science degree in Applied Mathematics/Computer Science and her Master of Science degree in Computer Science in May 2011. During her time at Brown, she was a teaching assistant in the Computer Science Department for six semesters and a head teaching assistant for one semester. She also coordinated a computer science outreach program for the Rhode Island community during her sophomore and junior years. She completed an internship as a Program Manager at Microsoft Corporation in the summer of 2010 and will be returning as a full-time employee after she graduates.

Abstract

This thesis presents an algorithm to find all efficient supported solutions of two-objective optimization problems. The algorithm is based on the first phase of the two-phase method, which has previously been studied as a technique to solve multiobjective combinatorial optimization problems. I give an implementation of the algorithm that can be used on non-convex problems, a class of problems that the original algorithm could not be used to solve. I also provide a test model that is based on previous work with the Single Commodity Allocation Problem (SCAP) for disaster recovery, and give test data for the model on benchmarks designed by Los Alamos National Laboratory. Finally, I discuss implications of the algorithm for specific use in humanitarian logistics work with the SCAP.

Acknowledgements

I would like to thank Professor Pascal Van Hentenryck, my advisor, for his support and direction of my first research project. His guidance saw me through a completely new experience, for which I am truly grateful. I am also indebted to my de-facto mentor, Carleton Coffrin. He provided invaluable assistance in navigating new systems and unfamiliar technology, and was always available and willing to discuss my results and ideas, and to answer my never-ending questions with patience and diligence. Additional thanks to fellow student Ben Simon for his assistance with this project, but more particularly for being my collaborator on various projects for the past eight years. My academic career has benefited greatly from everything I have learned from working with him, and I could not have tackled this project without the perspective and experience I gained from our collaborations. I am additionally grateful to the Council of Presidential Awardees in Mathematics, the Metropolitan Mathematics Club of Chicago, and the Cherry Scholarship Foundation for their generous financial assistance, which gave me the freedom to pursue my interests in academics and service work throughout my time at Brown. I also owe thanks to my family and friends for their constant moral support. In particular, I want to thank Ben Cohen for always being there to keep my spirits up and encourage me to laugh even when my work was frustrating and difficult. Finally, I extend special thanks to my father, Clark Elliott, to whom I cannot express the depths of my gratitude. He has been an example of perseverence and hard work without peer, and provides the highest standard to which I can hope to hold myself. Without his around-the-clock support, encouragement, and dependability for the past twenty-two years, none of this would have been possible.

Contents

List of Figures

Chapter 1

Background

I will first give definitions and other provide other background information that will be used throughout this thesis. Two references for the material in this section are Ehrgott (2005) [3] and Ehrgott (2006) [2].

1.1 Multiobjective Integer Programming

A single-objective integer program (IP) has the following components:

- Variables

$$x = [x_1, x_2, \ldots, x_n]^T, \tag{1.1}$$

  where $x_i \in \mathbb{Z}$.

- Constraint matrices

$$\begin{aligned} A &= [a^{(1)}, a^{(2)}, \ldots, a^{(m)}]^T \\ B &= [b^{(1)}, b^{(2)}, \ldots, b^{(p)}]^T, \end{aligned} \tag{1.2}$$

  where $a^{(i)}, b^{(i)} \in \mathbb{Z}^n$.

- Objective vector

$$C = [c_1, c_2, \ldots, c_n], \tag{1.3}$$

  where $c_i \in \mathbb{Z}$.

Then for $a \in \mathbb{Z}^m, b \in \mathbb{Z}^p$, the IP is defined as:

$$\begin{aligned} \min \quad & Cx \\ \text{subject to:} \quad & \\ & Ax \leq a^T \\ & Bx = b^T. \end{aligned} \tag{1.4}$$

We denote the set of feasible solutions as F, and say that a solution $\hat{x} \in F$ if and only if $x = \hat{x}$ satisfies (1.4).

A single-objective IP does not allow us to model situations in which there is more than one component over which we want to optimize. Because such situations are common to the real world, it is sometimes necessary to use a multiobjective IP (MOIP) formulation. The mathematical formulation of the MOIP is almost identical to the single-objective IP formulation given in (1.4), where the only difference is in the objective, which now becomes a *matrix* C:

$$C = [c^{(1)}, c^{(2)}, \ldots, c^{(n)}], \tag{1.5}$$

where each $c^{(i)} \in \mathbb{Z}^k$, and k is the number of objective components over which we optimize.

But given this definition of a MOIP, our objective is no longer well-defined. In the single-objective case, the objective value Cx was a scalar, and thus it was easy to compare two feasible solutions x^1 and x^2 by asking whether $Cx^1 \leq Cx^2$. In the MOIP case, the objective value Cx is a vector in $\mathbb{Z}^k$, and for two feasible solutions x^1 and x^2 it is possible to have $(Cx^1)_i < (Cx^2)_i$ while simultaneously having $(Cx^1)_j > (Cx^2)_j$ for $i \neq j$. That is, decreasing the objective in terms of some component $c^{(i)}$ may increase the objective in terms of a different component $c^{(j)}$. Thus, we need to define a way of measuring optimality in the multiobjective case.

1.2 Efficiency and Non-Dominance

Let $f : F \to \mathbb{Z}^k$ be the function $f(x) = Cx$. Note that the value of a MOIP solution $\hat{x}$ can be plotted as a point in k-space, whose coordinates correspond to the components of $f(\hat{x})$. Also let x^1, x^2 be solutions to a MOIP, and let $y^1 = f(x^1)$, $y^2 = f(x^2)$ be the corresponding objective "values" (where y^1, $y^2 \in \mathbb{Z}^k$). We say that $y^1 \leq y^2$ if for all $1 \leq i \leq k$, $y^1_i \leq y^2_i$. Similarly, $y^1 < y^2$ if for all $1 \leq i \leq k$, $y^1_i < y^2_i$.

Now consider a solution $x^* \in F$. We say that x^* is an *efficient* solution (also known as a *Pareto-optimal* solution or as a *non-dominated* solution) if there does not exist another solution $\hat{x} \in F$ such that $f(\hat{x}) \leq f(x^*)$. In the case of an efficient solution x^*, we also say that the k-dimensional point corresponding to $f(x^*)$ is a *non-dominated point*. If we have x^1, $x^2 \in F$ and $f(x^1) \leq f(x^2)$, then we say that $f(x^1)$ dominates $f(x^2)$ and that x^1 dominates x^2.

We differentiate between *weakly efficient* and *efficient* solutions. A solution x^* is weakly efficient if there does not exist another feasible solution $\hat{x}$ such that $f(\hat{x}) < f(x^*)$. A weakly efficient solution corresponds to a *weakly non-dominated* point.

Intuitively, this means that an efficient solution is one in which you cannot improve a single component of the objective without strictly worsening at least one other component: a solution x^* is efficient if for $\hat{x} \neq x^*$, $i \neq j$, $c^{(i)}\hat{x} \leq c^{(i)}x^* \Rightarrow c^{(j)}\hat{x} > c^{(j)}x^*$. A weakly efficient solution is one in which there is no way to improve *every* component of the objective while remaining feasible.

The concepts of efficiency and non-dominance provide us with a way to discuss optimality in a MOIP: a solution is optimal if it is efficient with respect to the k components of the objective function. (The corresponding points are said to form the *Pareto-optimal frontier*.) It is important to note that we no longer have a single optimal value, as we do in a single-objective IP.

1.3 Scalarization

Now that we've defined optimality in a MOIP, we still need a way actually to *find* optimal solutions. *Scalarization* is a commonly used technique: transforming the MOIP into a related single-objective IP. In particular, we are concerned with the *weighted-sum* scalarization method. Given a MOIP as described by (1.4) and (1.5), we want to assign a weight $\lambda \geq 0$ to each of the k objectives. That is, given

$$\lambda = [\lambda_1, \lambda_2, \ldots, \lambda_k] \text{ (where all } \lambda_i \in \mathbb{R}_0^+) \tag{1.6}$$

and variables, constraints, and objectives as in (1.1), (1.2), and (1.5), we want to solve the following IP:

$$\begin{aligned} \min \quad & \sum_{i=1}^{k} \lambda_i c^{(i)} x \\ \text{subject to:} \quad & \\ & Ax \leq a^T \\ & Bx = b^T. \end{aligned} \tag{1.7}$$

We have now transformed our k-component objective into an single-component aggregate objective over the original components. The λs denote the relative "importance" we ascribe to each of the objective components: for example, if $\lambda_i > \lambda_j$, then optimal solutions of (1.7) will favor solutions that are more minimal with respect to $c^{(i)}$ than with respect to $c^{(j)}$.

We say that $x^* \in F$ is a *supported solution* of the MOIP defined by (1.4) and (1.5) if it is an optimal solution to the related IP given by (1.7) for some $\hat{\lambda} > 0$. (If any $\hat{\lambda_i} = 0$, as allowed by (1.6), then x^* is called a *weakly supported* solution.) We are particularly concerned with efficient supported solutions: solutions that are efficient with respect to the MOIP given by (1.4) and (1.5) and supported with respect to the corresponding weighted-sum IP given by (1.7).

Chapter 2

Finding Efficient Supported Solutions

Przybylski, Gandibleux, and Ehrgott [5] presented an algorithm to find efficient solutions of two-dimensional MOIPs with totally unimodular constraint matricies. (Total unimodularity ensures that the efficient supported solutions will form a convex set [3].) Their algorithm is based on the two-phase method, which was originally presented by Ulungu and Teghem [7]. We have adapted the first phase of the two-phase method to generate a "representative set" (defined below) of all efficient supported solutions of MOIPs with finite solution sets.[1] Note that our algorithm does *not* require convexity.

To define the representative set, consider the following: (1) A MOIP may have non-unique solutions. In particular, $x^1, x^2, \ldots, x^p \in F$ such that for all $i \neq j, 1 \leq i, j \leq p$,

$$\begin{aligned} x^i &\neq x^j \\ f(x^i) &= f(x^j). \end{aligned} \tag{2.1}$$

And (2), a weighted-sum IP transformation of a MOIP may have multiple solutions $x^1, x^2, \ldots, x^q \in F$ such that for a given $\lambda > 0$, for all $i \neq j, 1 \leq i, j \leq q$,

$$\begin{aligned} x^i &\neq x^j \\ \lambda \cdot x^i &= \lambda \cdot x^j. \end{aligned} \tag{2.2}$$

[1] For a MOIP with an infinite number of solutions, we could use a variation of this algorithm to find an arbitrarily large set of solutions by imposing constraints on how similar one solution can be to another in each objective component.

Then a representative set R has three properties:

1. For all sets $S = \{x^1, x^2, \ldots, x^p\}$ that satisfy (2.1), R contains exactly one solution $x^* \in S$.
2. For all sets $T = \{x^1, x^2, \ldots, x^q\}$ that satisfy (2.2) for a given $\lambda > 0$, R contains at least one solution $x^* \in T$.
3. For all other efficient supported solutions x^* such that $x^* \notin S$ and $x^* \notin T$ for any $\lambda > 0$, R contains x^*.

That is, a representative set R will contain "all"[2] efficient supported solutions such that it does not contain multiple non-unique solutions and such that it is not guaranteed to contain more than two of any set of solutions that are collinear with respect to their corresponding two-dimensional points.

For $\hat{x} \in F$, let

$$f_1(\hat{x}) = c^{(1)}\hat{x} \tag{2.3}$$

$$f_2(\hat{x}) = c^{(2)}\hat{x}. \tag{2.4}$$

The algorithm works in two steps:

1. Find the two lexicographically extreme solutions x_{lE} and x_{rE}.
2. Recursively find all efficient supported solutions $\hat{x}$ such that

$$f_1(x_{lE}) < f_1(\hat{x}) < f_1(x_{rE}) \tag{2.5}$$

$$f_2(x_{rE}) < f_2(\hat{x}) < f_2(x_{lE}). \tag{2.6}$$

2.1 Finding Lexicographically Extreme Solutions

To find the lexicographically extreme solution with respect to the objective component f_1, we create a single-objective IP in which we minimize with respect to f_1, and entirely ignore the component f_2. This gives us a solution x_{l_init} such that $f_1(x_{l\_init}) \leq f_1(\hat{x})$ for all other $\hat{x} \in F$. However, since we entirely removed f_2 from the objective function, this solution x_{l_init} may be dominated by another solution x^* such that $f_1(x^*) = f_1(x_{l\_init})$ but $f_2(x^*) < f_2(x_{l\_init})$. In order to ensure that our lexicographically extreme solutions are non-dominated, we then solve a new IP that has the constraints of the original, in additon to a constraint fixing the value of $f_1(x)$ to be equal to the

[2] In the description and discussion of our algorithm, we will say "all efficient supported solutions" to mean all solutions in a representative set of efficient supported solutions.

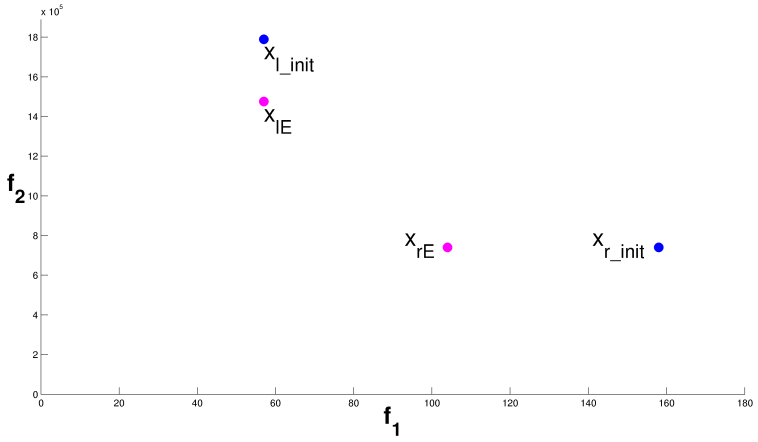

Figure 2.1: Lexicographically Extreme Solutions

minimal value $f_1(x_{l\_init})$. In the new IP, we minimize f_2 and ignore the component f_1. This gives us a solution x_{lE} that is now guaranteed to be minimal in f_1, and further guaranteed to be as minimal as possible in f_2 while maintaining minimality in f_1: a non-dominated, lexicographically extreme solution (see Figure 2.1). The method for finding the second lexicographically extreme solution x_{rE} is analagous.

2.2 Finding Non-Extreme Solutions

Given lexicographically extreme starting solutions x_{lE} and x_{rE} (where $f_1(x_{lE}) < f_1(x_{rE})$ and $f_2(x_{rE}) < f_2(x_{lE})$), we calculate $\lambda = \langle \lambda_1, \lambda_2 \rangle$ such that λ is perpendicular to the vector between the points $(f_1(x_{lE}), f_2(x_{lE}))$ and $(f_1(x_{rE}), f_2(x_{rE}))$. We then transform the MOIP into a weighted-sum single-objective IP defined by the scalars λ: this IP minimizes in the direction of the λ-vector (see Figure 2.2). If solving that IP gives us a solution $\hat{x}$ whose value is distinct from the values of both x_{lE} and x_{rE} and is additionally non-collinear with them, we solve two recursive problems: one in which our starting solutions are x_{lE} and $\hat{x}$, and one in which our starting solutions are $\hat{x}$ and x_{rE}. Otherwise, we know that there are no more efficient supported solutions between the starting solutions and therefore stop that branch of recursion.

The full algorithm is given as the three procedures $findESS(\cdot)$, $findExtremes(\cdot)$, and $solveRecursion(\cdot)$ in Figure 2.3. Note that for these procedures, we assume that our IP solver returns a single optimal-valued solution rather than a set that contains all optimal-valued solutions.

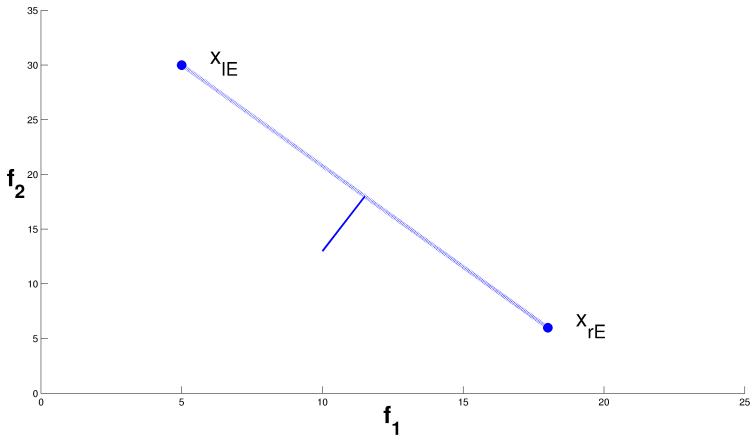

Figure 2.2: Minimize in the Direction of $\langle \lambda_1, \lambda_2 \rangle$

2.3 Correctness of Algorithm

All solutions are supported. Let S be the set of solutions found by the algorithm. First, note that the two lexicographically extreme solutions x_{lE} and x_{rE} are only guaranteed to be *weakly* supported, since we only showed that they were each supported by a set of λs in which one took on the value of 0. (If the IP was guaranteed to have a convex set of efficient supported solutions, then the efficiency of the extreme points, which we show in the next section, would imply that they were non-weakly supported [3]. Since we are not restricting our algorithm to convex IPs, we cannot prove non-weak supportedness of the extreme points in the general case.) However, we will show that the rest of the solutions found by the algorithm are non-weakly supported. By the definition of supported, it is sufficient to show that λ_1, λ_2 calculated from parent solutions $x_l, x_r \in S$ are strictly positive at every iteration of the algorithm (see lines (1–2) of $solveRecursion(\cdot)$). The general proof structure will be to break down into specific cases and show that each case is either impossible or ensures that x_l lies strictly above and to the left of x_r (as x_{lE}, x_{rE} are positioned in Figure 2.2). We use strong induction on the depth of recursion, k:

Base Cases: (1) $x_l = x_{lE}$ and $x_r = x_{rE}$, (2a) $x_l = x_{lE}$ and $x_r = \hat{x_r}$, where $\hat{x_r}$ was found from parents x_{rE} and x_{lE}, and (2b) $x_l = \hat{x_l}$ and $x_r = x_{rE}$, where $\hat{x_l}$ was found from parents x_{lE} and x_{rE}.

(1) **Case 1:** $x_l = x_{lE}$ and $x_r = x_{rE}$. (This is depth $k = 1$ of recursion.) We calculate

$$\lambda_1 = f_2(x_{lE}) - f_2(x_{rE}) \tag{2.7}$$

$$\lambda_2 = f_1(x_{rE}) - f_1(x_{lE}). \tag{2.8}$$

Algorithm 1: findESS(C, M)

Input: Two-objective matrix $C = [c^{(1)}, c^{(2)}]$, model M defining linear constraints.
Result: Compute all efficient supported solutions of the two-dimensional MOIP defined by C, M.
Output: The set S of all efficient supported solutions.

$x_{lE}, x_{rE} \leftarrow$ findExtremes(C, M)
if $(f_1(x_{lE}), f_2(x_{lE})) \neq (f_1(x_{rE}), f_2(x_{rE}))$:
    $S \leftarrow$ solveRecursion$(C, M, x_{lE}, x_{rE}, \emptyset)$
else:
    $S \leftarrow \{x_{lE}\}$
return S

Algorithm 2: findExtremes(C, M)

Input: Two-objective matrix $C = [c^{(1)}, c^{(2)}]$, model M defining linear constraints.
Result: Compute the two lexicographically extreme efficient supported solutions x_{lE}, x_{rE}.
Output: x_{lE}, x_{rE}.

$x_{l\_init} \leftarrow argmin_x(c^{(1)}x : x$ satisfies the constraints of model $M)$
$x_{r\_init} \leftarrow argmin_x(c^{(2)}x : x$ satisfies the constraints of model $M)$
$x_{lE} \leftarrow argmin_x(c^{(2)}x : x$ satisfies: [the constraints of model M and $c^{(1)}x = f_1(x_{l\_init})]$)
$x_{rE} \leftarrow argmin_x(c^{(1)}x : x$ satisfies [the constraints of model M and $c^{(2)}x = f_2(x_{r\_init})]$)
return x_{lE}, x_{rE}

Algorithm 3: solveRecursion(C, M, x_l, x_r, S)

Input: Two-objective matrix $C = [c^{(1)}, c^{(2)}]$, model M defining linear constraints, solutions $x_l, x_r \in F$, set of solutions S.
Result: Calculate all efficient supported solutions $\tilde{x}$ that such that $c^{(1)}x_l < c^{(1)}\tilde{x} < c^{(1)}x_r$.
Output: Set S of efficient supported solutions.

$\lambda_1 \leftarrow f_2(x_l) - f_2(x_r)$
$\lambda_2 \leftarrow f_1(x_r) - f_1(x_l)$
$L \leftarrow$ constraints of model $M \cup \{f_1(x_l) \leq f_1(x) \leq f_1(x_r)\}$
$\hat{x} \leftarrow argmin_x(\lambda_1 c^{(1)}x + \lambda_2 c^{(2)}x : x$ satisfies the constraints given in $L)$
$oldVal \leftarrow \lambda_1 f_1(x_l) + \lambda_2 f_2(x_l)$
$newVal \leftarrow \lambda_1 f_1(\hat{x}) + \lambda_2 f_2(\hat{x})$
if $newVal < oldVal$:
    $S_1 \leftarrow$ solveRecursion$(C, M, x_l, \hat{x})$
    $S_2 \leftarrow$ solveRecursion$(C, M, \hat{x}, x_r)$
    $S \leftarrow S \cup S_1 \cup S_2 \cup \{\hat{x}\}$
return S

Figure 2.3: Finding Efficient Supported Solutions

The solution x_{lE} is guaranteed to have the smallest possible value of $f_1(x)$ over all $x \in S$, since it was found by an IP whose objective was to minimize f_1 (see lines (1–4) of $findExtremes(\cdot)$). Similarly, our other solution x_{rE} is guaranteed to have the smallest possible value of $f_2(x)$ over all $x \in S$. As seen in line (2) of $findESS(\cdot)$, the procedure $solveRecursion(\cdot)$ is only called if the condition

$$(f_1(x_{lE}), f_2(x_{lE})) \neq (f_1(x_{rE}), f_2(x_{rE})) : \; f_1(x_{lE}) \neq f_1(x_{rE}) \vee f_2(x_{lE}) \neq f_2(x_{rE}) \tag{2.9}$$

is met. However, we can achieve a stronger result from (2.9):

$$f_1(x_{lE}) \neq f_1(x_{rE}) \wedge f_2(x_{lE}) \neq f_2(x_{rE}) \tag{2.10}$$

To prove (2.10), assume that $f_1(x_{lE}) \neq f_1(x_{rE})$. Then $f_1(x_{lE}) < f_1(x_{rE})$, since x_{lE} is minimal with respect to f_1. We also know that $f_2(x_{lE})$ is minimal for fixed minimal value of $f_1 = f_1(x_{lE})$. Now assume that $f_2(x_{lE}) = f_2(x_{rE})$. Then $f_2(x_{lE})$ is the minimum possible value taken on by f_2. Also, $f_1(x_{rE})$ is the minimum possible value of f_1 given the fixed minimal $f_2 = f_2(x_{rE})$. But $f_1(x_{lE}) < f_1(x_{rE})$, and so $f_1(x_{rE})$ is not the minimum value of f_1 for a fixed value of f_2, which is a contradiction. We can similarly show that $f_2(x_{lE}) \neq f_2(x_{rE}) \implies f_1(x_{lE}) \neq f_1(x_{rE})$, which concludes our proof of (2.10). □

Given (2.10) and the minimality of $f_1(x_{lE})$ and $f_2(x_{rE})$, we see that:

$$f_1(x_{lE}) \neq f_1(x_{rE}) \quad \implies \quad f_1(x_{lE}) < f_1(x_{rE}) \tag{2.11}$$

$$f_2(x_{lE}) \neq f_2(x_{rE}) \quad \implies \quad f_2(x_{rE}) < f_2(x_{lE}) \tag{2.12}$$

(see Figure 2.4). Then by (2.8), $\lambda_1, \lambda_2 > 0$, as we wanted to show.

(2) **Case 2a:** $x_l = x_{lE}$ and $x_r = \hat{x_r}$, where $\hat{x_r}$ was found from parents x_{rE} and x_{lE}. (This is depth $k = 2$ of recursion.) We calculate

$$\lambda_1 \;=\; f_2(x_{lE}) - f_2(\hat{x_r}) \tag{2.13}$$

$$\lambda_2 \;=\; f_1(\hat{x_r}) - f_1(x_{lE}). \tag{2.14}$$

By lines (4–7) of $solveRecursion(\cdot)$, we see that $x_{lE} \neq \hat{x_r}$. Combining this with the minimality of $f_1(x_{lE})$, we have that

$$f_1(x_{lE}) < f_1(\hat{x_r}). \tag{2.15}$$

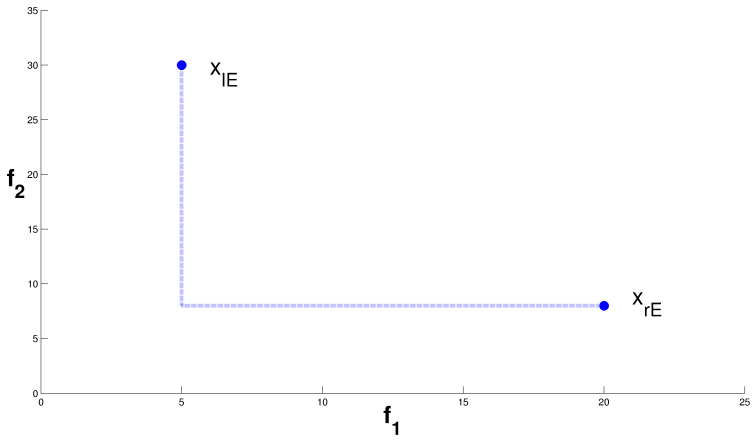

Figure 2.4: Base Case (1)

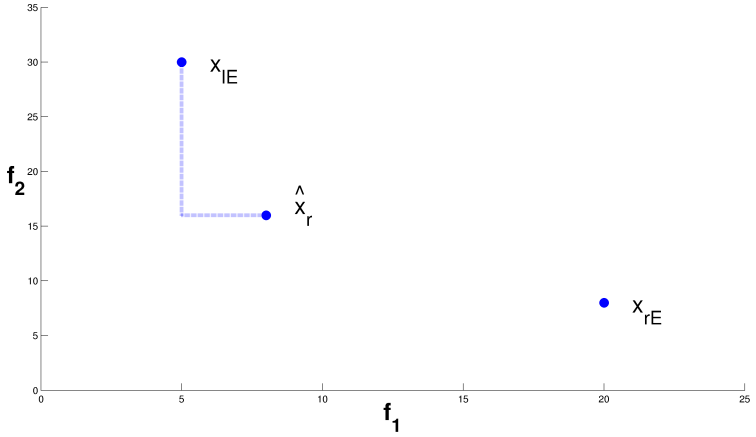

Figure 2.5: Base Case (2a)

By (2.14), this implies that $\lambda_2 > 0$. Furthermore, as shown by Ehrgott [3], if x^* is an optimal solution of the weighted-sum IP

$$\min \qquad \sum_{i=1}^{m} \alpha_i c^{(i)} x \tag{2.16}$$

with $\alpha_m \in \mathbb{R}^+$, then x^* is an efficient solution of the original corresponding MOIP. Since $\hat{x_r}$ was found from parents x_{lE} and x_{rE}, Case (1) shows that $\hat{x_r}$ satisfies the properties of x^* given in (2.16). Therefore, $\hat{x_r}$ is an efficient solution of the MOIP. Given (2.15), the efficiency of $\hat{x_r}$ implies that $f_2(\hat{x_r}) < f_2(x_{lE})$ (see Figure 2.5), and so by (2.13), $\lambda_1 > 0$. Thus, $\lambda_1, \lambda_2 > 0$, as we wanted to show.

(3) **Case 2b:** $x_l = \hat{x_l}$ and $x_r = x_{rE}$, where $\hat{x_l}$ was found from parents x_{lE} and x_{rE}. (This is also depth $k = 2$ of recursion.) This is symmetric to Case (2a), and so the same proof holds.

This concludes our proof of the base case. □

In order to prove the inductive step, we introduce the following lemma:

Lemma 1: If x_l is the left parent of solution $\hat{x}$, where $\hat{x}$ was found at recursive depth $k+1$, then either $x_l = x_{lE}$ or x_l was found at recursive depth i of some branch of $solveRecursion(\cdot)$ such that $i \leq k$. Similarly, if x_r is the right parent solution at recursive depth $k+1$ then either $x_r = x_{rE}$ or x_r was found at recursive depth i of some branch of $solveRecursion(\cdot)$ such that $i \leq k$.

Proof of Lemma 1: The first time x_{lE} is given as an argument to $solveRecursion(\cdot)$ (at recursive depth 1), it is given as the left parent solution (see line (3) of $findESS(\cdot)$). By lines (8–9) of $solveRecursion(\cdot)$, we see that left parent solutions are only ever used as left parent solutions in successive calls to $solveRecursion(\cdot)$; that is, a solution that is used as a left parent at some iteration of $solveRecursion(\cdot)$ will never be used as a right parent at a different iteration. Similarly, solutions that are used as right parents will never be used as left parents. Thus, x_{lE} may be used as a left parent at some recursive depth j such that $j > 1$. If this is true for $j = k+1$, then $x_l = x_{lE}$ and we are done. To complete the proof of the lemma, assume that $x_l \neq x_{lE}$. Since x_l is therefore not an extreme point, we know that x_l must have been found using $solveRecursion(\cdot)$. Assume that it was found at a depth l, where $l > k$. Then any successive calls to $solveRecursion(\cdot)$ that use x_l as a left parent must occur at recursive depths that are all greater than l. Since x_l is a left parent of $\hat{x}$, this means that $\hat{x}$ must have been found at some recursive depth d such that $d > l > k \implies d > k+1$. However, we know that $\hat{x}$ was found at depth $d = k+1$, which is a contradiction. The proof of the lemma for x_r is analagous, and will be ommitted. □

Inductive step: For $k \geq 1$, assume that any solution $\hat{x}_i$ found at recursive depth i such that $1 \leq i \leq k$ is supported. We wish to show that any solution $\hat{x}_{k+1}$ found at recursive depth $k+1$ is also supported. Let x_r and x_l be the parent solutions that generate $\hat{x}_{k+1}$. To show that $\hat{x}_{k+1}$ is supported, it is sufficient to show that $\lambda_1, \lambda_2 > 0$, where $\lambda_1 = f_2(x_l) - f_2(x_r)$ and $\lambda_2 = f_1(x_r) - f_1(x_l)$ (see lines (1–2) of $solveRecursion(\cdot)$). Given Lemma 1, there are three cases to consider:

(1) $x_l = x_{lE}$ and $x_r \neq x_{rE}$, where x_r was found at recursive depth j such that $j \leq k$.

(2) $x_l \neq x_{lE}$ and $x_r = x_{rE}$, where x_l was found at recursive depth j such that $j \leq k$.

(3) $x_l \neq x_{lE}$ and $x_r \neq x_{rE}$, where x_l and x_r were found at respective recursive depths j, l such that $j, l \leq k$.

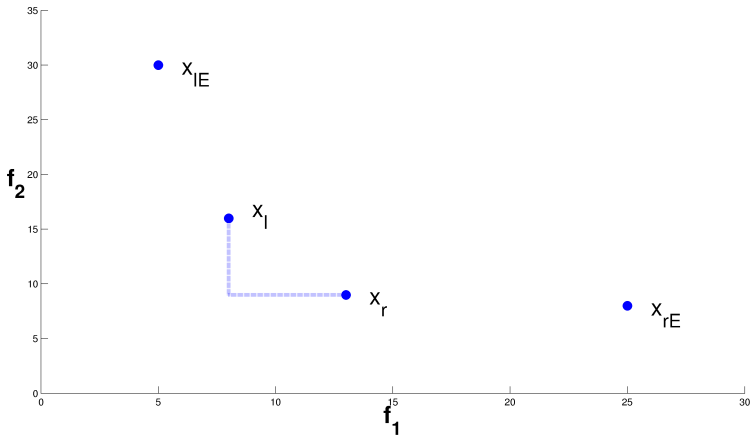

Figure 2.6: Inductive Step (3)

The proofs that $\lambda_1, \lambda_2 > 0$ for Case (1) and Case (2) are analagous to the proofs given in Case (2a) and Case (2b) of the base case and will therefore be ommitted. The proof of Case (3) is as follows:

It is clear that $\lambda_1, \lambda_2 > 0$ if and only if

$$f_2(x_l) > f_2(x_r) \tag{2.17}$$

$$f_1(x_r) > f_1(x_l) \tag{2.18}$$

(see Figure 2.6). We prove (2.17) and (2.18) by showing that the following three cases are impossible:

(3a) $f_2(x_l) \leq f_2(x_r)$ and $f_1(x_r) \leq f_1(x_l)$.

(3b) $f_2(x_l) \leq f_2(x_r)$ and $f_1(x_r) > f_1(x_l)$.

(3c) $f_2(x_l) > f_2(x_r)$ and $f_1(x_r) \leq f_1(x_l)$.

The proofs are as follows:

(3a) **Case 3a:**

$$f_2(x_l) \leq f_2(x_r) \tag{2.19}$$

$$f_1(x_r) \leq f_1(x_l). \tag{2.20}$$

As seen in lines (4, 8–9) of $solveRecursion(\cdot)$, all recursive calls are given as input the new solution found at the current iteration and exactly one of its parent solutions. (That is, in all recursive calls to $solveRecursion(\cdot)$, exactly one of the input solutions is the parent of the other.) Thus, either x_l is the *left* (see discussion in the proof of Lemma 1) parent solution of

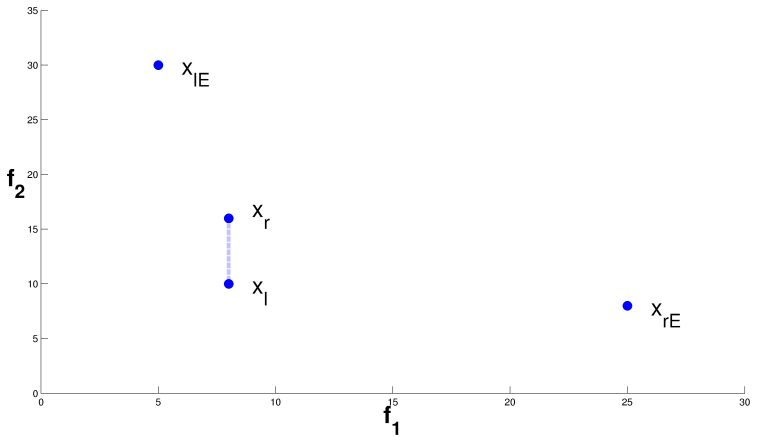

Figure 2.7: Inductive Step (3a)

x_r or x_r is the *right* parent solution of x_l. In either case, line (3) of $solveRecursion(\cdot)$ gives us that $f_1(x_l) \leq f_1(x_r)$. With (2.20), this gives that $f_1(x_r) = f_1(x_l)$. But then by (2.19), x_r is dominated by x_l (see Figure 2.7). However, by the inductive hypothesis and assumptions of Case (3), we know that x_r is a supported solution, and so (2.16) implies that x_r is also an efficient solution, and therefore cannot be dominated. This is a contradiction, and so Case (3a) must be impossible.

(3b) **Case 3b:**

$$f_2(x_l) \leq f_2(x_r) \tag{2.21}$$

$$f_1(x_r) > f_1(x_l). \tag{2.22}$$

In this case, x_r is dominated by x_l (see figures 2.8 and 2.9). However, since by the inductive hypothesis and assumptions of Case (3), x_r is a supported solution, and so (2.16) implies that x_r must be efficient. This is a contradiction, and so Case (3b) is also impossible.

(3c) **Case 3c:**

$$f_2(x_l) > f_2(x_r) \tag{2.23}$$

$$f_1(x_r) \leq f_1(x_l). \tag{2.24}$$

By an argument symmetric to the one given in Case (3b), we show that x_l is dominated by x_r, which is a contradiction to the efficiency of x_l. Thus, Case (3c) is also impossible.

Since cases (3a-3c) are impossible, (2.17) and (2.18) must be true and $\lambda_1, \lambda_2 > 0$ at depth

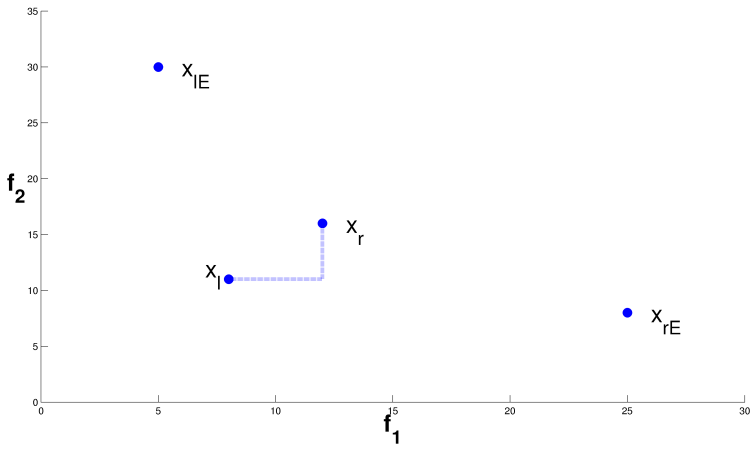

Figure 2.8: Inductive Step (3b) – Inequality

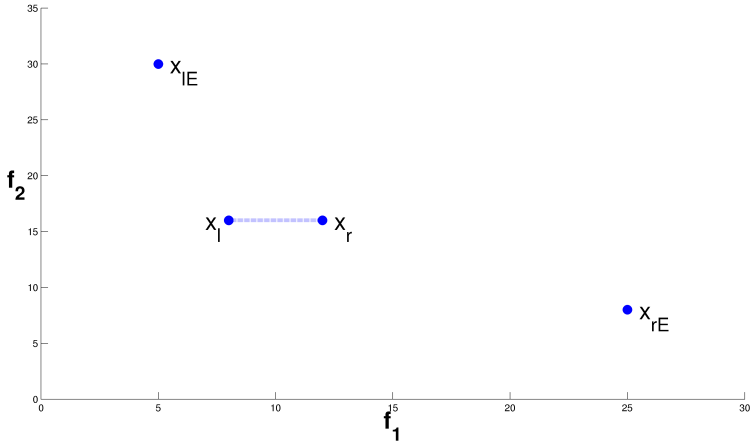

Figure 2.9: Inductive Step (3b) – Equality

$k+1$ of recursion. Combining with our base case, this gives us that all solutions x_k found during $solveRecursion(\cdot)$ for $k \geq 1$ are supported, as we wanted to show. □

All solutions are efficient. Let S be the set of solutions found by our algorithm. For any $x \in S$ such that $f_2(x) \leq f_2(x_{lE})$, we know that $f_1(x_{lE}) < f_1(x)$, since x_{lE} is minimal with respect to f_1 and is furthermore the *only* solution in S that takes on the minimal f_1-value. Thus, x_{lE} is an efficient solution by definition. Similarly, x_{rE} is efficient by definition. We know that all solutions $\hat{x}$ calculated with the procedure $solveRecursion(\cdot)$ are supported, and thus by (2.16) we know they are also efficient. This proves that all solutions found by $findESS(\cdot)$ are efficient. □

We have found all solutions in a representative set. The general structure of this proof will be to consider an efficient supported solution $\hat{x}$, and then to look at the solutions x_l and x_r that lie

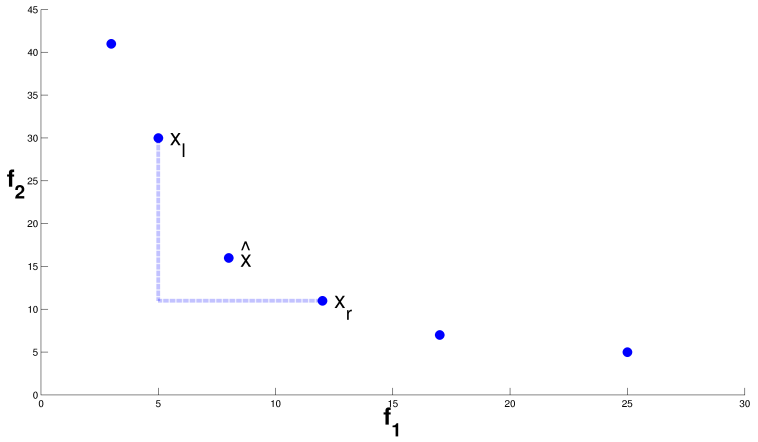

Figure 2.10: Tightest Region Containing $\hat{x}$

directly to the left and right, respectively, of $\hat{x}$. We then show by contradiction that x_l and x_r must be considered as parent solutions at some iteration of $solveRecursion(\cdot)$. Finally, we use this fact to show that if some solution y is efficient and supported but is not found by our algorithm, then we arrive at a contradiction by looking at the solutions that *are* found by our algorithm and that lie directly to the left and right of y. We begin with the following lemma:

Lemma 2: Let S be the set of solutions found by our algorithm, and let $\hat{x}$ be an efficient supported solution of the MOIP. Let $x_l, x_r \in S$ be the solutions found by our algorithm such that

$$f_1(x_l) < \quad f_1(\hat{x}) \quad < f_1(x_r) \tag{2.25}$$

$$f_2(x_r) < \quad f_2(\hat{x}) \quad < f_2(x_l) \tag{2.26}$$

and such that $f_1(x_r) - f_2(x_l)$ is minimal (see Figure 2.10). That is, x_l and x_r define the "tightest" region in which $\hat{x}$ is contained with respect to its f_1-value. Then x_l, x_r are used as the two parent solutions at some iteration of $solveRecursion(\cdot)$.

Proof of Lemma 2: If x_l is a parent solution of x_r or x_r is a parent solution of x_l, then the lemma is true, as seen by lines (4, 8–9) of $solveRecursion(\cdot)$). Then assume that neither solution is a parent of the other, and also assume without loss of generality that that x_l is found before x_r. Let $\dot{x_r} \neq x_r$ be the solution with which x_l is used as a parent such that $f_1(x_l) < f_1(x_r) < f_1(\dot{x_r})$ and such that $f_1(\dot{x_r}) - f_1(x_l)$ is minimal. That is, x_l and $\dot{x_r}$ define the "tightest" region in which x_r is contained with respect to its f_1-value such that some iteration of the algorithm uses $x_l, \dot{x_r}$ as parents. By reasoning given in the proof of Lemma 1, we know that x_l is used as the left parent. Let λ_1, λ_2 be

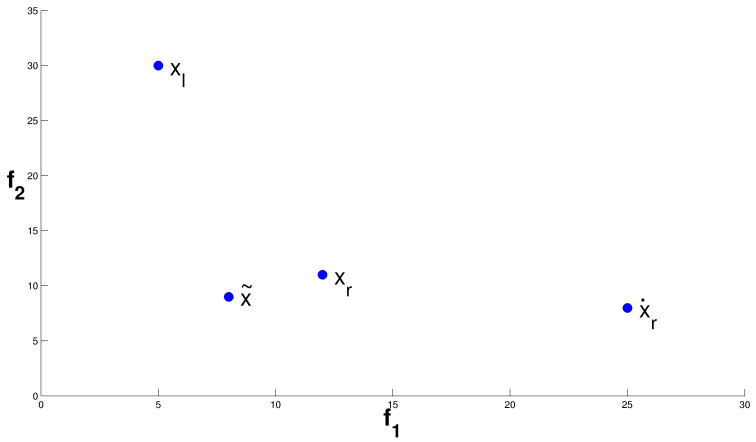

Figure 2.11: x_r is Dominated by $\tilde{x}$

the weights computed from $x_l, \dot{x_r}$. Since by assumption x_l is not a parent of x_r, it must be the case that some other solution $\tilde{x}$ is found from parents $x_l, \dot{x_r}$. Then one of the following must hold:

$$\lambda_1 f_1(\tilde{x}) + \lambda_2 f_2(\tilde{x}) < \lambda_1 f_1(x_r) + \lambda_2 f_2(x_r) \tag{2.27}$$

$$\lambda_1 f_1(\tilde{x}) + \lambda_2 f_2(\tilde{x}) = \lambda_1 f_1(x_r) + \lambda_2 f_2(x_r). \tag{2.28}$$

Assume that (2.27) holds. Since $\lambda_1, \lambda_2 > 0$, at least one of the following must be true:

$$f_1(\tilde{x}) < f_1(x_r) \tag{2.29}$$

$$f_2(\tilde{x}) < f_2(x_r). \tag{2.30}$$

If both hold, then x_r is dominated by $\tilde{x}$ (see Figure 2.11), which would mean that x_r cannot be efficient. This is impossible, since we have previously stated that $x_r \in S$. Therefore, *exactly* one must hold. Suppose that (2.29) holds. We also have that $f_1(x_l) < f_1(\tilde{x})$, since x_l is the left parent of $\tilde{x}$. (The reasoning behind this claim was given in the proof that all solutions found by the algorithm are supported.) Then $\tilde{x}$, $\dot{x_r}$ define a tighter region within which x_r is contained than that defined by $x_l, \dot{x_r}$. Since $\tilde{x}, \dot{x_r}$ are used as parents (see line (9) of $solveRecursion(\cdot)$), this is a contradiction to our assumption that x_l, $\dot{x_r}$ define the tightest such region (see Figure 2.12). An analagous argument shows why it also cannot be the case that (2.30) holds.

Now assume that (2.28) holds. Then $\tilde{x}$ and x_r are both optimal solutions to the weighted-sum problem defined by parents $x_l, \dot{x_r}$. It cannot be true that $f_1(\tilde{x}) = f_1(x_r)$ and that $f_2(\tilde{x}) = f_2(x_r)$, since we find exactly one solution within any set of non-unique solutions and we have already said

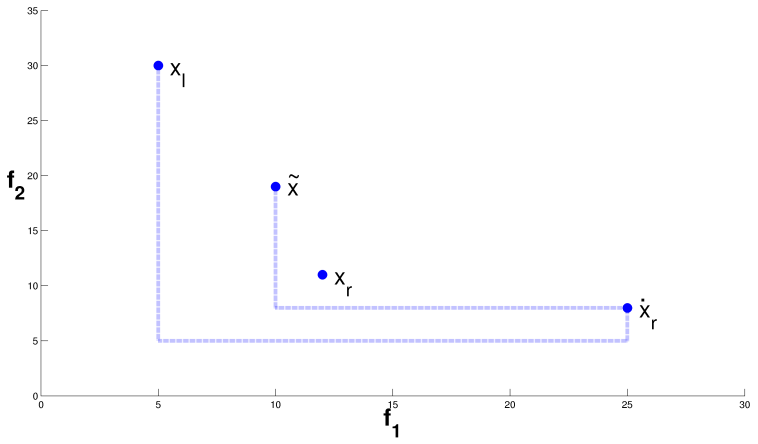

Figure 2.12: Contradictory Tightest Region Containing x_r – Inequality

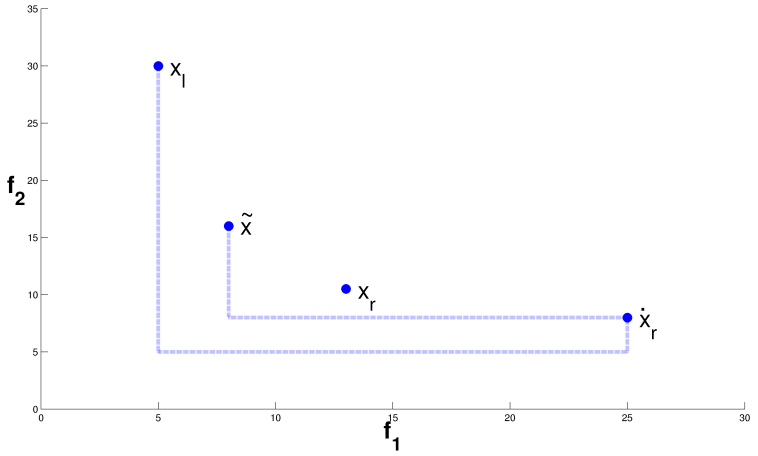

Figure 2.13: Contradictory Tightest Region Containing x_r – Equality

that our algorithm finds x_r at some point. Therefore, we have one of the following:

$$f_1(\tilde{x}) < f_1(x_r) \quad \text{and} \quad f_2(\tilde{x}) > f_2(x_r) \tag{2.31}$$

$$f_1(\tilde{x}) > f_1(x_r) \quad \text{and} \quad f_2(\tilde{x}) < f_2(x_r). \tag{2.32}$$

By lines (4, 9) of $solveRecursion(\cdot)$, we know that we will use $\tilde{x}, \dot{x_r}$ as parents. But because $f_1(x_l) < f_1(\tilde{x})$, then (2.31) shows that $\tilde{x}, \dot{x_r}$ defines a smaller region within which x_r is contained than that defined by $x_l, \dot{x_r}$ (see Figure 2.13). Since $\tilde{x}, \dot{x_r}$ are used as parents, this is a contradiction to the assumption that $x_l, \dot{x_r}$ defined the smallest such region. We can similarly show that (2.32) poses a contradiction. Thus, we have proven Lemma 2. □

Now we will prove that our algorithm finds all solutions in a representative set. Let S be the set

of solutions found by our algorithm, and assume that $\hat{x} \notin S$ is an efficient supported solution such that $\hat{S} = S \cup \{\hat{x}\}$ is a representative set but S is not. Let $x_l, x_r \in S$ be the solutions found by our algorithm such that

$$f_1(x_l) < \quad f_1(\hat{x}) \quad < f_1(x_r) \tag{2.33}$$

$$f_2(x_r) < \quad f_2(\hat{x}) \quad < f_2(x_l) \tag{2.34}$$

and such that $f_1(x_r) - f_2(x_l)$ is minimal. That is, x_l and x_r define the "tightest" region in which $\hat{x}$ is contained with respect to its f_1-value.

By Lemma 2, we know that x_l, x_r are used as the two parent solutions at some iteration of $solveRecursion(\cdot)$. Therefore, let λ_1, λ_2 be the weights defined by the iteration of $solveRecursion(\cdot)$ that uses x_l and x_r as parent solutions. Since by assumption our algorithm does not find $\hat{x}$, it must be the case that there is some other solution $\bar{x}$ that is found from parents x_l and x_r. Then it must hold that

$$\lambda_1 f_1(\bar{x}) + \lambda_2 f_2(\bar{x}) \leq \lambda_1 f_1(\hat{x}) + \lambda_2 f_2(\hat{x}). \tag{2.35}$$

If the above equation gives equality, then we cannot make any guarantees that our algorithm will in fact find $\hat{x}$. However, in this case the point defined by $\hat{x}$ would be collinear with those defined by x_l and x_r, and we have already stated our algorithm is not guaranteed to find more than two out of any set of collinear points. Therefore, assume that we have a strict inequality:

$$\lambda_1 f_1(\bar{x}) + \lambda_2 f_2(\bar{x}) < \lambda_1 f_1(\hat{x}) + \lambda_2 f_2(\hat{x}). \tag{2.36}$$

By analagous reasoning to that given in the proof of Lemma 2, we can show that this poses a contradiction to either the assumption that $\hat{x}$ is efficient or to the assumption that x_l, x_r define the tightest region containing $\hat{x}$. Therefore, we conclude that $findESS(\cdot)$ finds all solutions in a representative set of all efficient supported solutions, as we wanted to show. □

2.4 Finding All Efficient Supported Solutions.

It may be of interest to ensure that we find *all* efficient supported solutions, rather than only those that are guaranteed to be members of a representative set. Given an IP solver that could enumerate all optimal solutions rather than give just one, this would be easy to accomplish. We would modify $solveRecursion(\cdot)$ to become $solveRecursion2(\cdot)$, which is given in Figure 2.14. As seen in line

Algorithm 4: solveRecursion2(C, M, x_l, x_r, S)

Input: Two-objective matrix $C = [c^{(1)}, c^{(2)}]$, model M defining linear constraints, solutions $x_l, x_r \in F$, set of solutions S.
Result: Calculate all efficient supported solutions $\tilde{x}$ that such that $c^{(1)}x_l \leq c^{(1)}\tilde{x} \leq c^{(1)}x_r$.
Output: Set S of efficient supported solutions.

1 $\lambda_1 \leftarrow f_2(x_l) - f_2(x_r)$
2 $\lambda_2 \leftarrow f_1(x_r) - f_1(x_l)$
3 $L \leftarrow$ constraints of model $M \cup \{f_1(x_l) \leq f_1(x) \leq f_1(x_r)\}$
4 set $\hat{X} \leftarrow \{argmin_x(\lambda_1 c^{(1)}x + \lambda_2 c^{(2)}x : x \text{ satisfies the constraints given in } L)\}$
5 set $\hat{X}_l \leftarrow \{argmin_x(f_1(\hat{x}) : \hat{x} \in \hat{X})\}$
6 set $\hat{X}_r \leftarrow \{argmax_x(f_1(\hat{x}) : \hat{x} \in \hat{X})\}$
7 $\hat{x_l} \leftarrow$ some $\hat{x_l} \in \hat{X}_l$
8 $\hat{x_r} \leftarrow$ some $\hat{x_r} \in \hat{X}_r$
9 if $f_1(x_l) < f_1(\hat{x_l})$:
10 $\quad S_1 \leftarrow$ solveRecursion2$(C, M, x_l, \hat{x_l})$
11 $\quad S \leftarrow S \cup S_1 \cup \{\hat{x_l}\}$
12 if $f_1(\hat{x_r}) < f_1(x_r)$:
13 $\quad S_2 \leftarrow$ solveRecursion2$(C, M, \hat{x_r}, x_r)$
14 $\quad S \leftarrow S \cup S_2 \cup \{\hat{x_r}\}$
15 return $S \cup \hat{X}$

Figure 2.14: Finding Non-Unique and Collinear Efficient Supported Solutions

(4), the algorithm now enumerates *all* solutions that are optimal with respect to the weighted-sum scalarization using weights λ_1, λ_2 for parents x_l and x_r. Of that set of optimal solutions, it chooses the two "locally extreme" solutions $\hat{x_l}$ and $\hat{x_r}$ (that is, one that is minimal with respect to f_1 and one that is maximal with respect to f_1 — see lines (5–8)), and solves the recursive problems with parents $(x_l, \hat{x_l})$ and $(\hat{x_r}, x_r)$. Choosing the "extreme" solutions ensures that solutions will not be found multiple times, thus making the recursion more efficient. If there is only a single f_1 value for optimal solutions to the weighted-sum scalarization then $\hat{x_l} = \hat{x_r}$ and the recursive calls are as in the original procedure $solveRecursion(\cdot)$. Note that we must also modify the procedure $findExtremes(\cdot)$ so that it enumerates all solutions that take on the extremal f_1 and f_2 values; this change is similar to the one made to $solveRecursion(\cdot)$, and so we will not rewrite $findExtremes(\cdot)$ here.

If we did not have an IP solver that could enumerate all optimal solutions but were still interested in finding collinear solutions, we could modify our algorithm in a different way, which is shown in Figure 2.15. In this version, we tighten our constraint on the f_1-values such that a child solution $\hat{x}$ must lie strictly in between the f_1-values of its parents, x_l and x_r: $f_1(x_l) < f_1(x) < f_1(x_r)$ (see line (3) of $solveRecursion3(\cdot)$). This allows us to find solutions that are collinear with the two parent solutions. However, with this stricter requirement, the weighted-sum IP solve in line (4) is no longer guaranteed to have any feasible solutions. Therefore, we perform an additional feasibility check in line (5) before proceeding with the algorithm as seen in the original $solveRecursion(\cdot)$.

Algorithm 5: solveRecursion3(C, M, x_l, x_r, S)

Input: Two-objective matrix $C = [c^{(1)}, c^{(2)}]$, model M defining linear constraints, solutions $x_l, x_r \in F$, set of solutions S.
Result: Calculate all efficient supported solutions $\tilde{x}$ that such that $c^{(1)}x_l < c^{(1)}\tilde{x} < c^{(1)}x_r$.
Output: Set S of efficient supported solutions.

$\lambda_1 \leftarrow f_2(x_l) - f_2(x_r)$
$\lambda_2 \leftarrow f_1(x_r) - f_1(x_l)$
$L \leftarrow$ constraints of model $M \cup \{f_1(x_l) < f_1(x) < f_1(x_r)\}$
$\hat{x} \leftarrow argmin_x(\lambda_1 c^{(1)}x + \lambda_2 c^{(2)}x : x$ satisfies the constraints given in $L)$
if $\hat{x}$ exists:
    $oldVal \leftarrow \lambda_1 f_1(x_l) + \lambda_2 f_2(x_l)$
    $newVal \leftarrow \lambda_1 f_1(\hat{x}) + \lambda_2 f_2(\hat{x})$
    if $newVal < oldVal$:
        $S_1 \leftarrow$ solveRecursion$(C, M, x_l, \hat{x})$
        $S_2 \leftarrow$ solveRecursion$(C, M, \hat{x}, x_r)$
        $S \leftarrow S \cup S_1 \cup S_2 \cup \{\hat{x}\}$
return S

Figure 2.15: Finding Collinear Efficient Supported Solutions

Chapter 3

Applications

To test the effectiveness of the algorithm in practice, we use a model that is based on the Single Commodity Allocation Problem (SCAP) previously studied by Van Hentenryck, Bent, and Coffrin [4]. The modified problem is presented here.

3.1 Background

Natural disasters such as hurricanes cause resource shortages in affected areas. One concern of the related SCAP is the pre-allocation of resources among storage locations before the disaster hits such that the total time needed to send the commodity from the storage points to members of the affected population is minimized. However, this is not a single-objective minimization problem: for a fixed percentage of the total demand guaranteed to be satisfied, there is a trade-off between the money spent to satisfy that demand and the time it takes to do so. In previous work, a four-stage algorithm to solve all aspects of the SCAP was presented. Here we focus only on a variation of the first stage, as it is solved by an IP and is therefore useful for demonstrating the effectiveness of the procedure $findESS(\cdot)$. The first stage, in which we use a modification of the *Stochastic Storage Model* (SSM), determines which repositories store the commodity and how much of it they store, as well as which storage locations will serve which demand points.

3.2 The Modified Single Commodity Allocation Problem

In formalizing SCAPs, a populated area is represented as a graph $G = \langle N, E \rangle$, where N represents the locations of interest to the allocation problem: sites requiring the commodity after the disaster (e.g., hospitals, shelters, and public buildings), as well as vehicle storage depots. The required commodity

Given:

- Repositories: $i \in R$
  - Capacity: RC_i
  - Investment Cost: RI_i
  - Maintenance Cost: RM_i
- Scenario Data: $s \in S$
  - Scenario Probability: P_s
  - Available Sites: $AR_s \subset R$
  - Site Demand: $D_{s,i \in R}$
  - Travel Time Matrix: $T_{s,1..l,1..l}$
  - Satisfiable Demand: SD_s
- Weights: W_x, W_y
- Vehicle Capacity: VC
- Demand Percentage: DP

Output:

- The amount stored at each warehouse
- The amount shipped from each warehouse to each demand point

Minimize:

- $W_x *$ Delivery Time +
- $W_y *$ Investment Cost +
- $W_y *$ Maintenance Cost

Subject To:

- Vehicle and site capacities
- Minimum demand satisfaction

Figure 3.1: The Single Commodity Allocation Problem Specification

can be stored at any node of the graph subject to some side constraints. The graph edges in E have weights representing travel times. The weights on the edges form a metric space, but it is non-Euclidean due to the transportation infrastructure. Moreover, travel times can vary in different disaster scenarios due to road damage [4]. In our modified problem, we additionally specify that a minimum amount of demand must be satisfied in any solution. The primary outputs of the portion of the SCAP on which we focus are (1) the amount of commodity to be stored at each node and (2) how much (if any) commodity is sent from each storage location to each demand point. Figure 3.1 summarizes the modified version of the problem, which we now describe in detail.

Objectives. The two-component objective function minimizes: (1) the total time used to move the commodity from the storage locations to the demand points, and (2) the cost of storing the commodity.

Side Constraints. Each repository $i \in R$ has a maximum storage capacity RC_i. It also has a one-time initial cost RI_i (the investment cost) and an incremental cost RM_i per each unit of commodity to be stored. Every repository can act as a warehouse and as a customer and its role changes on a scenario-by-scenario basis depending on site availability and demands. A repository may use its own resources to satisfy its demand. The model is given the maximum amount of demand that can be satisfied in any given scenario: it may be impossible to satisfy all of the demand of a given scenario due to either insufficient repository capacity or unreachability of demand points because of destroyed vehicle routes. The commodity allocation of any feasible solution must guarantee that for all scenarios $s \in S$, a minimum of SD_s DP demand is satisfied — that is, at least the specified percentage of the total satisfiable demand must be satisfied, regardless of which scenario actually happens.

Variables:

Variables for each repository $i \in R$:

$Stored_i \in [0, RC_i]$	- Units pre-allocated to repository i
$Open_i \in \{0, 1\}$	- Non-zero storage at repository i

Variables for each scenario $s \in S$, each repository $i \in R$:

$Kept_{si} \in [0, D_{si}]$	- Total units kept at repository i
$Sent_{si} \in [0, RC_i]$	- Total units shipped from repository i
$Unsatisfied_{si} \in [0, D_{si}]$	- Demand not satisfied at repository i
$Incoming_{si} \in [0, D_{si}]$	- Total units coming to repository i
$Trips_{sij} \in [0, RC_i]/VC$	- Trips needed from repository i to repository j

Minimize:

$$W_x \sum_{s \in S} P_s \sum_{i \in R} \sum_{j \in R} T_{sij}\ Trips_{sij} \ + \ W_y \sum_{i \in R} (RI_i\ Open_i + RM_i\ Stored_i)$$

Subject To:

$$\sum_{i \in R} Unsatisfied_{si} \leq \sum_{i \in R} (D_{si}) - SD_i\ DP \qquad \forall s \in S \qquad (1)$$

$$RC_i\ Open_i \geq Stored_i \qquad \forall i \in R \qquad (2)$$

$$Incoming_{si} + Kept_{si} + Unsatisfied_{si} = D_{si} \qquad \forall s \in S, i \in R \qquad (3)$$

$$Sent_{si} + Kept_{si} \leq Stored_i \qquad \forall s \in S, i \in R \qquad (4)$$

$$\sum_{j \in R} Trips_{sij} = Sent_{si} \qquad \forall s \in S, i \in R \qquad (5)$$

$$\sum_{j \in R} Trips_{sji} = Incoming_{si} \qquad \forall s \in S, i \in R \qquad (6)$$

$$Sent_{si} + Kept_{si} = 0 \qquad \forall s \in S, i \notin AR_s \qquad (7)$$

Figure 3.2: The IP Formulation for the Stochastic Storage Model (SSM)

Stochasticity. SCAPs are specified by a set S of different disaster scenarios. Scenario $s \in S$ has an associated probability P_s and specifies the set AR_s of sites that remain intact after the disaster. Moreover, scenario s specifies for each repository $i \in R$ the demand D_{si} and site-to-site travel times $T_{s,1..l,1..l}$ (where $l = |N|$) that capture the damages to the transportation infrastructure. It also specifies the maximum satisfiable demand amount SD_s.

3.3 Stochastic Storage

Figure 3.2 presents our variation of the SSM formulation. The meaning of the decision variables is explained in the figure. The objective function sums the investment and maintenance costs for the repositories, and the shipping time for each scenario; each are multiplied by their respective non-negative "importance" weights. Constraint (1) captures the minimum demand satisfaction constraint; constraint (2) ensures that a repository is open if it stores any amount of the commodity; constraint (3) states that the sum of the unsatisfied demand of a repository plus the amount of incoming supply to the repository plus the amount the repository keeps for itself is equal to the repository's demand; constraint (4) expresses that the sum of the supply shipped from repository

Variables:

Variables for each repository $i \in R$:

$Stored_i \in [0, RC_i]$	- Units pre-allocated to repository i
$Open_i \in \{0, 1\}$	- Non-zero storage at repository i

Variables for the given scenario $s^* \in S$, for each repository $i \in R$:

$Kept_{s^*i} \in [0, D_{s^*i}]$	- Total units kept at repository i
$Sent_{s^*i} \in [0, RC_i]$	- Total units shipped from repository i
$Unsatisfied_{s^*i} \in [0, D_{s^*i}]$	- Demand not satisfied at repository i
$Incoming_{s^*i} \in [0, D_{s^*i}]$	- Total units coming to repository i
$Trips_{s^*ij} \in [0, RC_i]/VC$	- Trips needed from repository i to repository j

Minimize:

$$\sum_{i \in R} Unsatisfied_{s^*i}$$

Subject To:

$$RC_i \; Open_i \geq Stored_i \qquad \forall i \in R \qquad (1)$$

$$Incoming_{s^*i} + Kept_{s^*i} + Unsatisfied_{s^*i} = D_{s^*i} \qquad \forall i \in R \qquad (2)$$

$$Sent_{s^*i} + Kept_{s^*i} \leq Stored_i \qquad \forall i \in R \qquad (3)$$

$$\sum_{j \in R} Trips_{s^*ij} = Sent_{s^*i} \qquad \forall i \in R \qquad (4)$$

$$\sum_{j \in R} Trips_{s^*ji} = Incoming_{s^*i} \qquad \forall i \in R \qquad (5)$$

$$Sent_{s^*i} + Kept_{s^*i} = 0 \qquad \forall i \notin AR_{s^*} \qquad (6)$$

Figure 3.3: The IP Formulation for the Maximum Satisfiable Demand Problem

i and the supply that repository i keeps for itself cannot exceed the amount of commodity stored at repository i; constraints (5–6) connect the sent, incoming, and trip variables; and constraint (7) ensures that damaged repositories ship no commodity.

3.4 Finding Maximum Satisfiable Demand

Our IP is given the maximum satisfiable demand SD_s for every scenario s. Because this is a calulated amount rather than one directly specified by the input data, we describe here our method of determining it. To find this amount, we use another IP that is very similar to that given in Figure 3.2. This IP is given a specific scenario rather than a set of scenarios S, and thus is not given the scenario data seen in Figure 3.1. From the model given in Figure 3.2, we eliminate constraint (1), and take constraints (3–7) only over the single scenario $s^* \in S$. We also replace the objective function with one that minimizes the total amount of unsatisfied demand. The entire model is given in Figure 3.3. We run this IP for every scenario $s \in S$, and give the set of satisfiable demands as input to the IP in Figure 3.2.

3.5 The SCAP and Lexicographically Extreme Solutions

We use the IP given in Figure 3.2 as the input model to our algorithm $findESS(\cdot)$. Here, f_1 corresponds to the time component of the objective: $\sum_{s \in S} P_s \sum_{i \in R} \sum_{j \in R} T_{sij}\ Trips_{sij}$, and f_2 corresponds to the cost component: $\sum_{i \in R} (RI_i\ Open_i + RM_i\ Stored_i)$.

Chapter 4

Results

4.1 Implementation, Benchmarks, and Experiments

The algorithm given by precedures $findESS(\cdot)$, $findExtremes(\cdot)$, and $solveRecursion(\cdot)$ was implemented in the Comet system [1], and the experiments were run on an AMD Phenom II X4 955, 3.21GHz processor machine running 32-bit Linux Debian. The benchmarks used to test the results are the same benchmarks that were used by Van Hentenryck, Bent, and Coffrin. They were produced by Los Alamos National Laboratory and are based on the infrastructure of the United States [4]. We ran tests on six of the available benchmarks. Two types of results were obtained: using the procedure $findESS(\cdot)$ on the modified SCAP model given by Figure 3.1 and Figure 3.2, we generated (1) graphs showing the solutions found at each recursive depth with an 80% demand satisfaction constraint, and (2) graphs showing the entire solution sets found with 10% to 100% demand satisfaction constraints in increments of 10%. In set (1), new solutions at each level of recursion are shown in magenta, and all other solutions are shown in blue. Note that the blue points directly to the left and right of each magenta point are the two parent points that were used to generate it. For graphs in set (2), points for each demand percentage are shown in different colors, as described in the graph legends. The full set of results is shown in Section 4.5.

4.2 Implementation Correctness Verification

We ran checks on the algorithm implementation and on our data to ensure that the theoretical results held, to the extent that such checks are possible. For example, we could not check that our algorithm actually found all efficient supported solutions, since we did not know in advance what the efficient supported solutions are, but we *could* perform the following checks:

(1) All λs found during the tests we ran were strictly positive. This verifies that all solutions we found were supported. By the theoretical result discussed in (2.16), this also guarantees that all solutions found by our algorithm are efficient.

(2) None of our solutions were dominated by another solution that we found. This partially verifies our theoretical assertion that all solutions found by our algorithm are efficient. (Note that without a full list of all solutions to the MOIP, we cannot manually verify complete non-domination of our solutions.)

4.3 Implications for the SCAP problem

The results confirm that in the SCAP, there is a definite trade-off between time and money, which is a concern for policy makers who are dealing with real-world disaster situations [4]. The graphs that plot points for multiple demand levels also confirm that there is a trade-off for both time and money as demand increases: solutions with higher demand satisfaction constraints use more money to achieve a specific amount of time than do solutions with lower demand constraints. (Similarly, solutions with higher demand satisfaction contraints use more time to achieve a specific amount of money than do solutions with lower demand constraints.) The algorithm could be of great help to policy makers because for a problem such as the SCAP, there are many feasible solutions from which one must be chosen in a time of crisis. Our algorithm would allow policy makers to narrow down their options to a particular set of non-dominated solutions. This is invaluable because (1) it cuts down the set of options considerably, which is important in time-sensitive situations, and (2) it eliminates a number of solutions that could be improved in at least one objective without worsening another (i.e., it eliminates a number of non-efficient solutions), and that would thus be undesirable. Of particular note is that this algorithm could eliminate the need for policy makers to experiment with different weights to set on the different objective components by providing a systematic way to explore trade-offs between two of the three objective components (where demand was an objective component in the original three-component SCAP [4]).

The algorithm could be further streamlined for optimization in the SCAP by imposing additional constraints on time and budget. Our algorithm was designed to find *all* efficient supported solutions, not just ones that fell within a feasible range for each component. In real-world disaster situations, clearly some of the points found by our algorithm would be unreasonable or infeasible. For example, consider the results for Benchmark 1 seen in Figure 4.2: a number of the left-most points represent solutions in which more money is spent than would be possible in a real situation. Adding temporal

and budgetary constraints would allow us to prune off solutions that would not be of interest to policy makers, thus further cutting down the set of solutions they must consider. There are also areas of the graphs in which solutions are more dense than in other areas. (For example, in Benchmark 2 (see Figure 4.3), solutions are dense around time ≈ 1400, whereas there is a large gap in which there are no solutions between 1500 and 1600.) In the interest of cutting down the number of options, policy makers may want to eliminate solutions that are relatively similar to other solutions. We could achieve this by setting bounds on how close a solution can be in f_1- or f_2-value to either of its parents, therefore ensuring that recursion stops when the graph becomes sufficiently dense.

4.4 Limitations of the Algorithm and Future Research

Our algorithm is limited in two particular ways: (1) it cannot find non-supported efficient solutions, and (2) it can only be used to find solutions of two-dimensional multiobjective problems.

Consider the point x_{ns} in Figure 4.1. x_{ns} is not dominated by either x_l or x_r, but our algorithm would be unable to find it because it is not supported. In general, non-supported but efficient solutions are those that fall in the "upper triangle" between two efficient supported solutions, as seen in the figure: solutions x such that $f_1(x_l) < f_1(x) < f_1(x_r)$ and $f_2(x_r) < f_2(x) < f_2(x_l)$, and such that the point defined by x lies above the line between the points x_l and x_r (our algorithm can find solutions that are *below* that line). For the SCAP in particular, policy makers may be interested in non-supported efficient solutions, due to constraints on time and budget. For example, the graph of results for Benchmark 6 (Figure 4.7) has large gaps (e.g., between time ≈ 1200 and time ≈ 1500) in which there are no efficient supported solutions. Policy makers may be interested in solutions that lie within those gaps because there is more money available than is used by the solution directly to the left of the gap but less money than is used by the solution directly to the right of the gap. (Similarly, they may wish to spend an amount of time that is in between the time spent by the two solutions x_l and x_r.) The second phase of the two-phase method is one existing approach to finding such solutions, but as of yet there is no known algorithm that will efficiently compute all non-supported efficient solutions [5].

We are also interested in extending the algorithm to MOIPs with more than two objectives. This is known to be a considerably more difficult problem to solve, and has also been studied by Pryzbylski, Gandibleux, and Ehrgott [6]. Of particular interest to the SCAP is an algorithm for three-objective MOIPs: the original SCAP includes satisfied demand as a third objective, rather than setting it as a constraint under which time and budget can vary [4]. Although we can easily vary the required

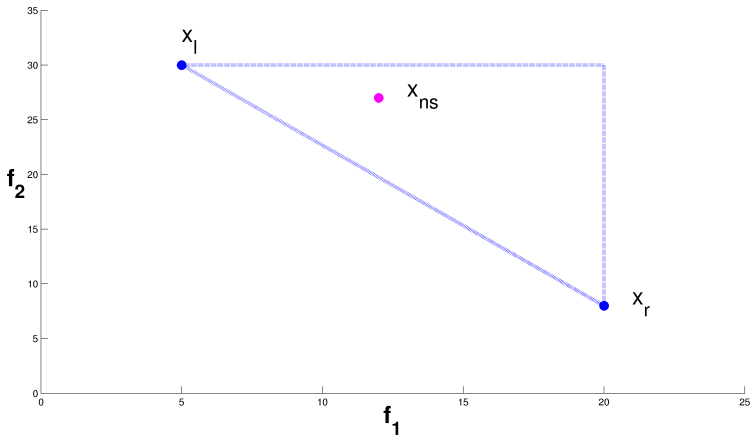

Figure 4.1: Non-Supported Efficient Solutions

satisfiable demand percentage, the structure of the problem means that the model will always try to satisfy as little demand as possible such that it will still meet the requirements: it minimizes over time and money, which are both trade-offs with demand. Thus, although we set lower bounds on the amount of demand that must be satisfied, they effectively become exact specifications rather than bounds. A three-dimensional model would give us more flexibility in exploring trade-offs among all three objectives without restricting the space within which one objective value will fall. In our preliminary work to formulate the algorithm to find efficient supported solutions in three dimensions, we have already encountered some aspects that are more complicated than in the two-dimensional case. For example, six initial extreme points rather than two are required to begin the algorithm (one for every permutation of the order in which to minimize the three objectives). Additionally, using three points to define a plane and then minimizing in the direction of the vector that is perpendicular to the plane no longer guarantees that the λ-values remain strictly positive. Further study will be required to formulate and implement the three-dimensional algorithm correctly.

4.5 Data

We present here the data obtained from our experiments on six different benchmarks. The first set of tests fixes demand at 80%, and the second set of graphs varies demand from 10% to 100% in increments of 10%.

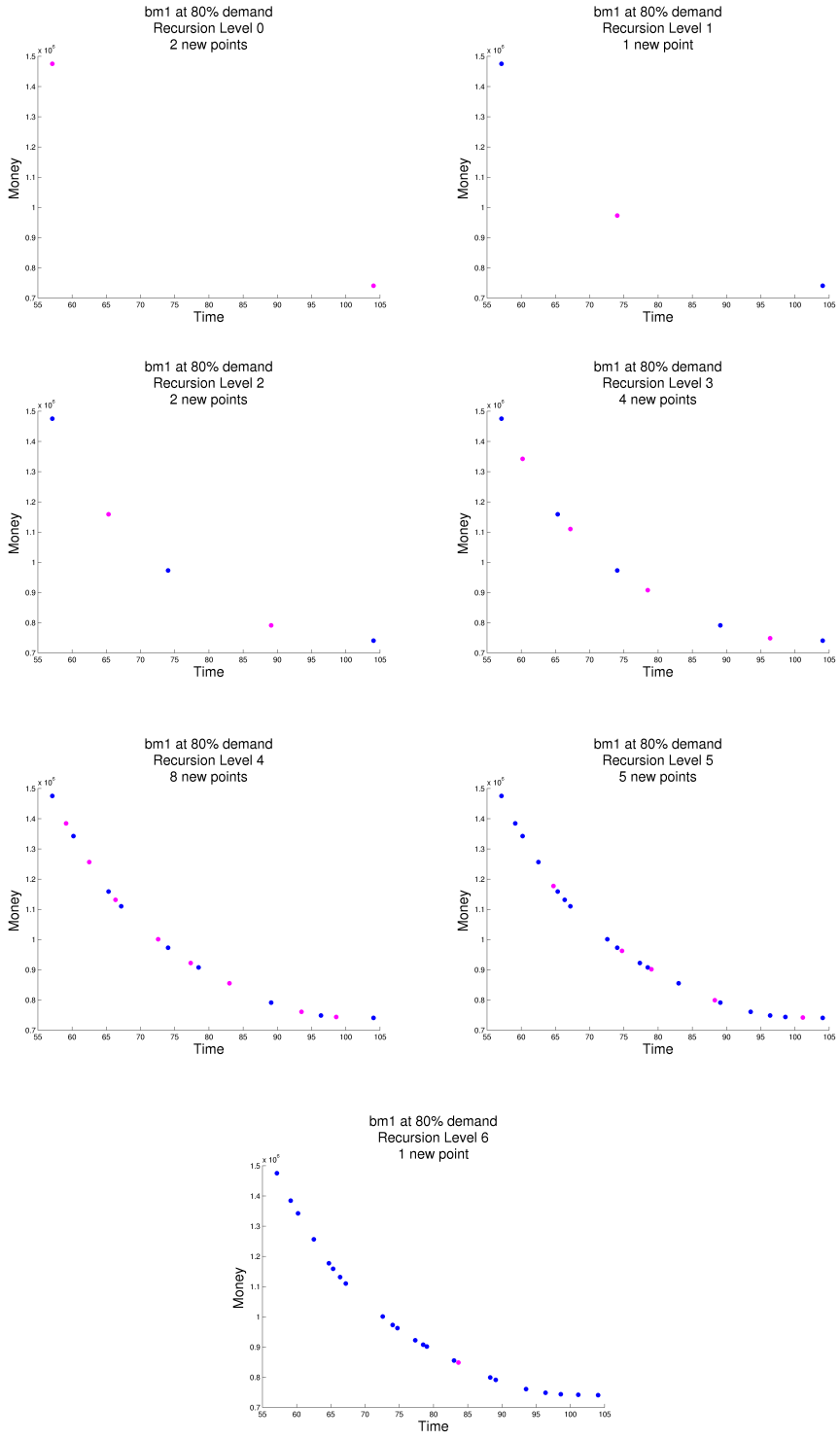

Figure 4.2: Benchmark 1 at 80% Demand Satisfaction

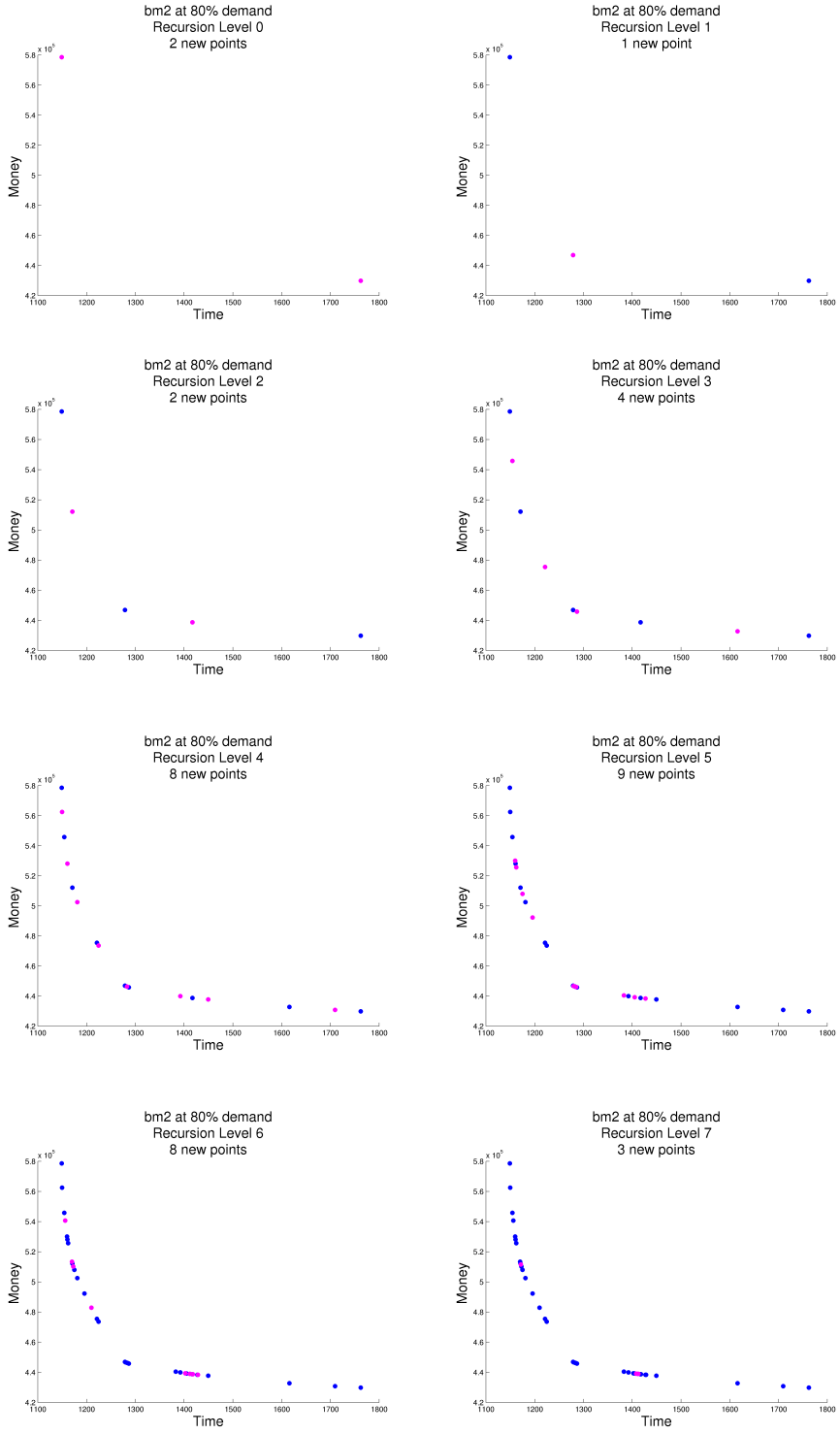

Figure 4.3: Benchmark 2 at 80% Demand Satisfaction

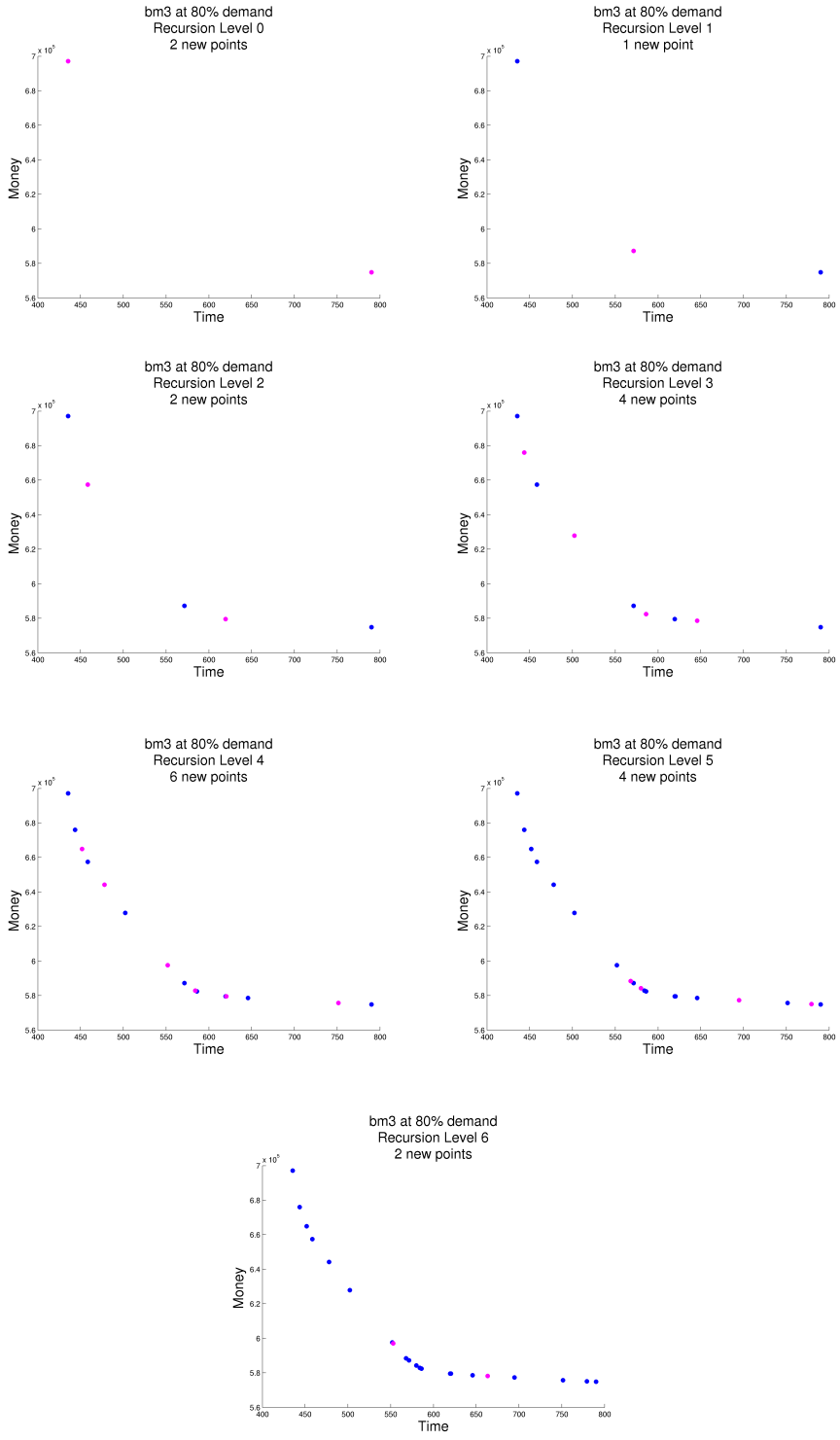

Figure 4.4: Benchmark 3 at 80% Demand Satisfaction

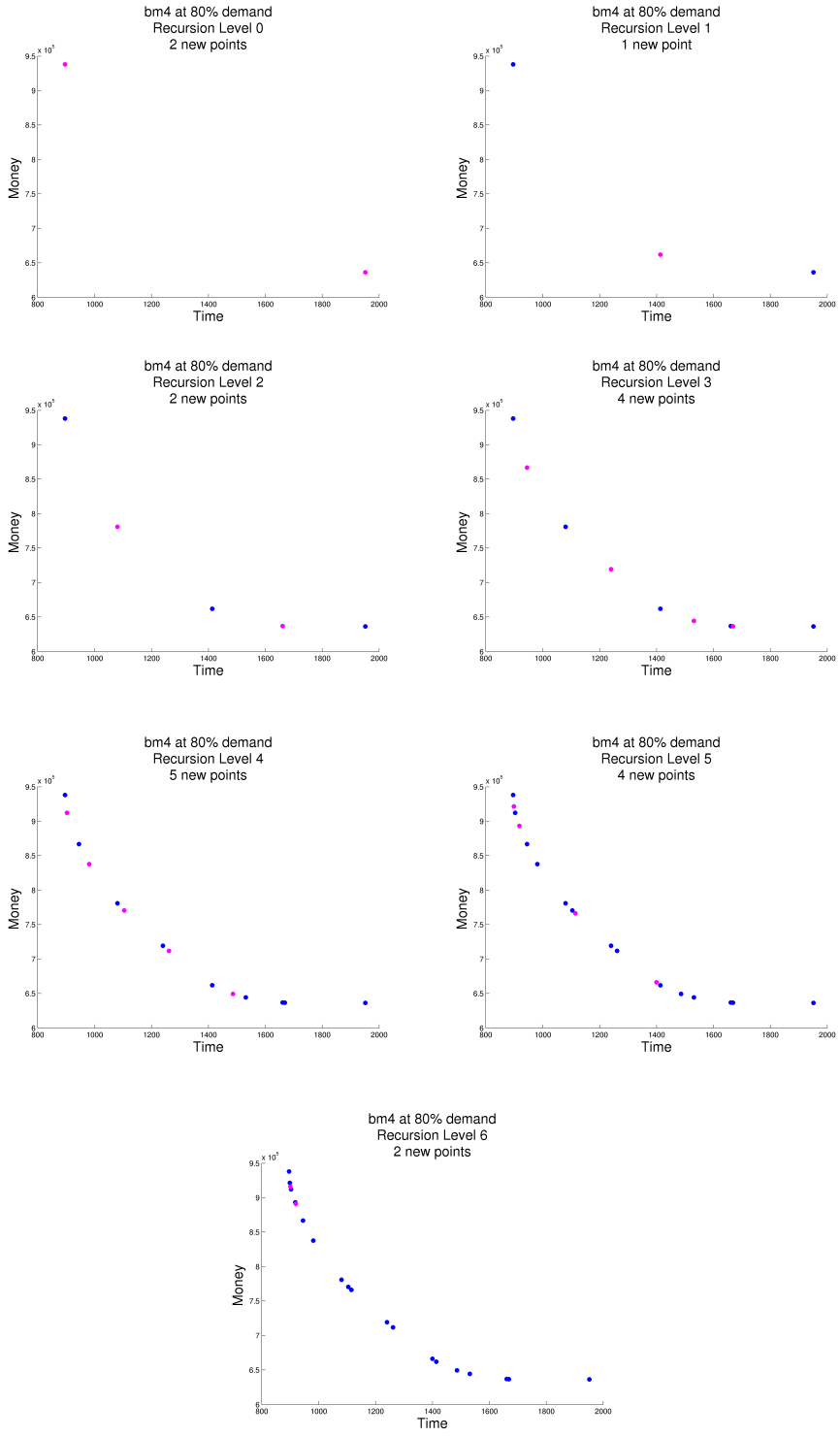

Figure 4.5: Benchmark 4 at 80% Demand Satisfaction

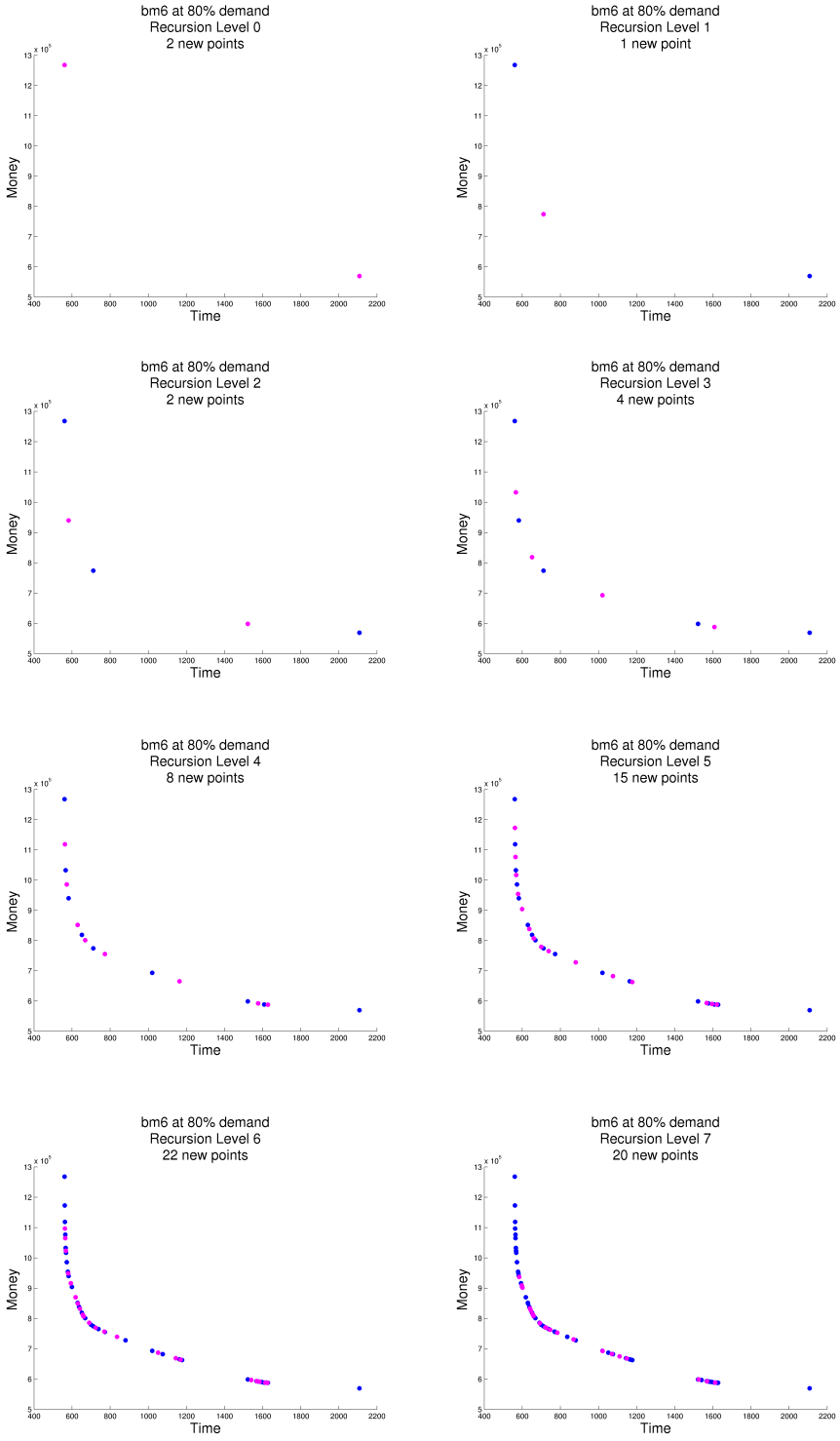

Figure 4.6: Benchmark 6 at 80% Demand Satisfaction – Levels 0-7

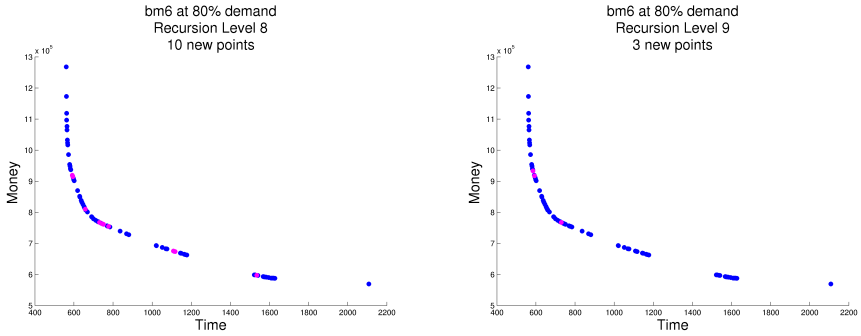

Figure 4.7: Benchmark 6 at 80% Demand Satisfaction – Levels 8-9

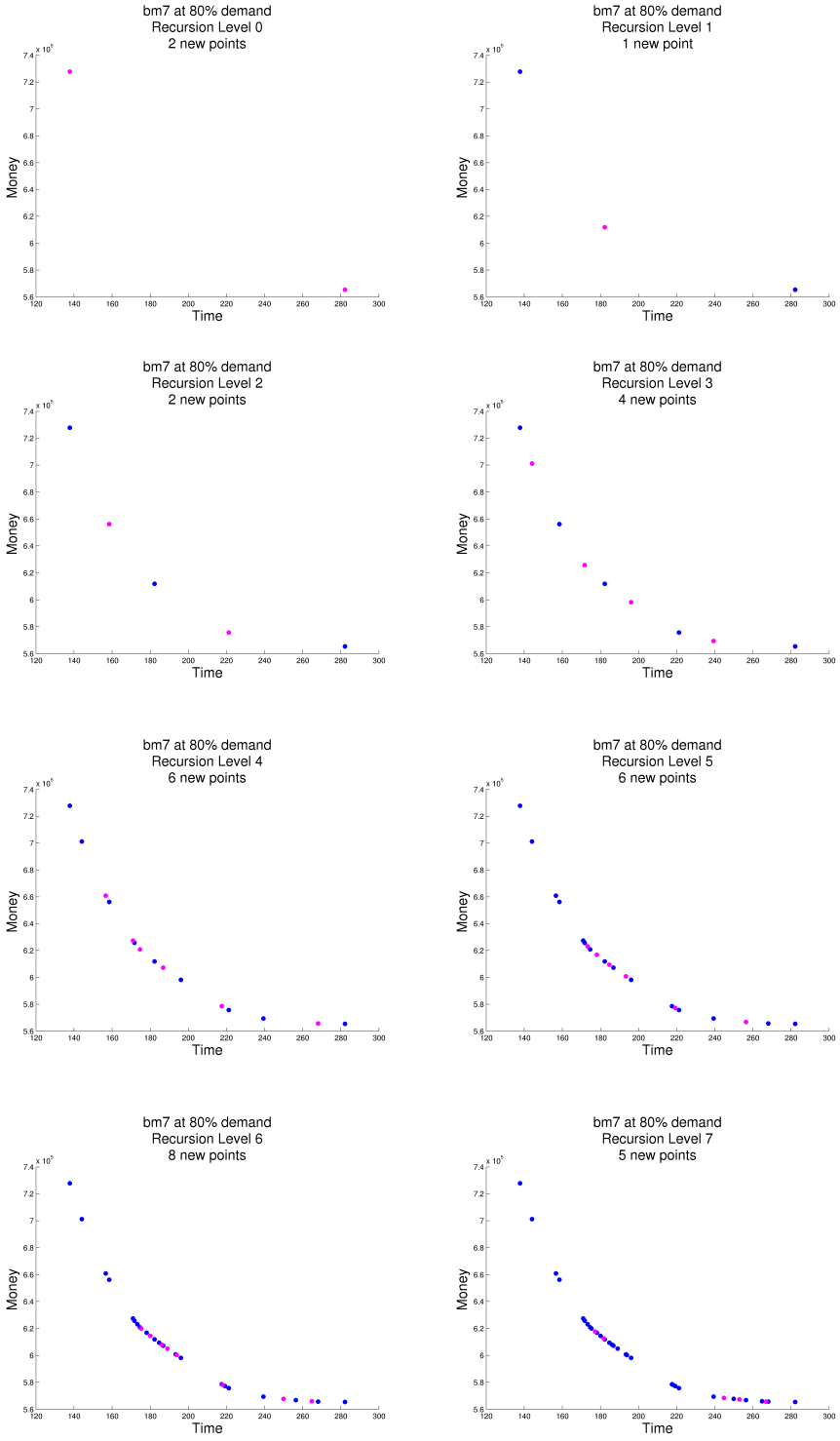

Figure 4.8: Benchmark 7 at 80% Demand Satisfaction – Levels 0-7

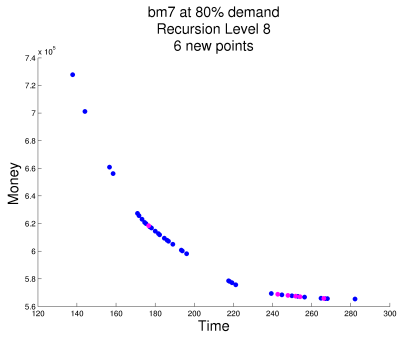

Figure 4.9: Benchmark 7 at 80% Demand Satisfaction – Level 8

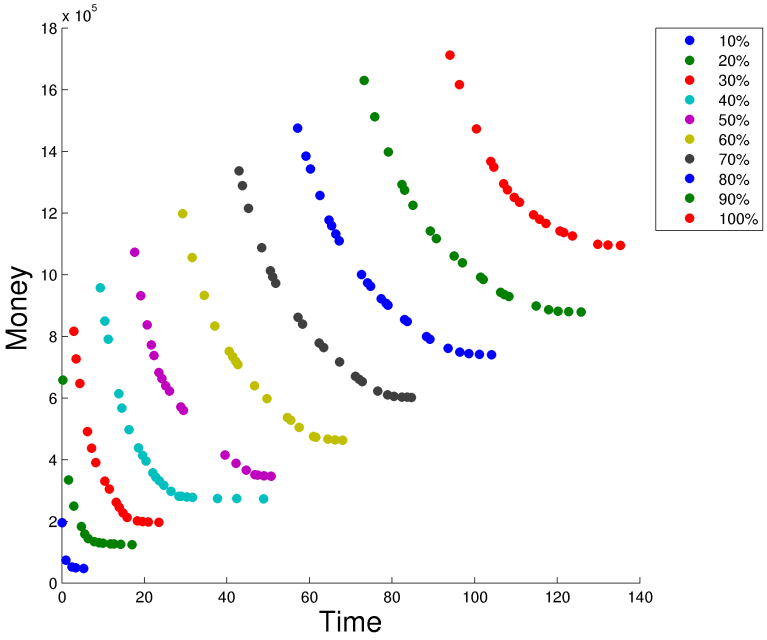

Figure 4.10: Benchmark 1 From 10% to 100% Demand Satisfaction

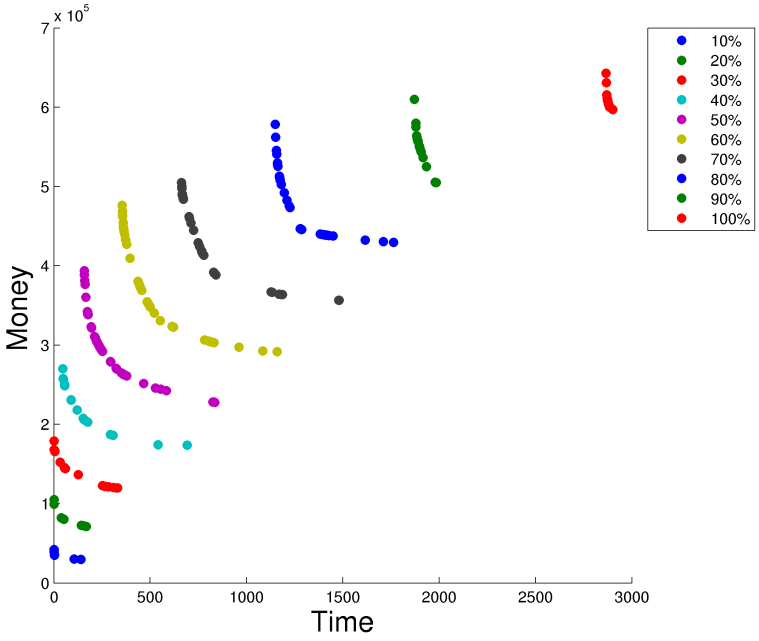

Figure 4.11: Benchmark 2 From 10% to 100% Demand Satisfaction

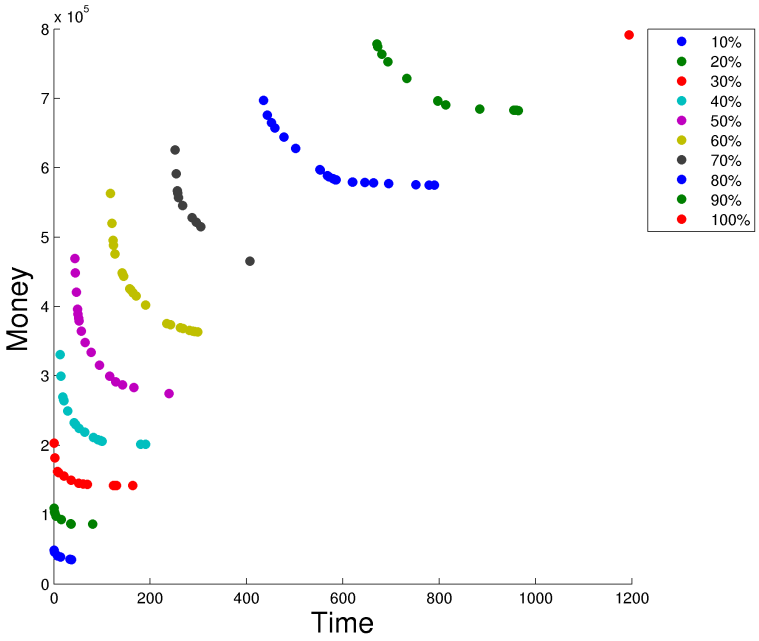

Figure 4.12: Benchmark 3 From 10% to 100% Demand Satisfaction

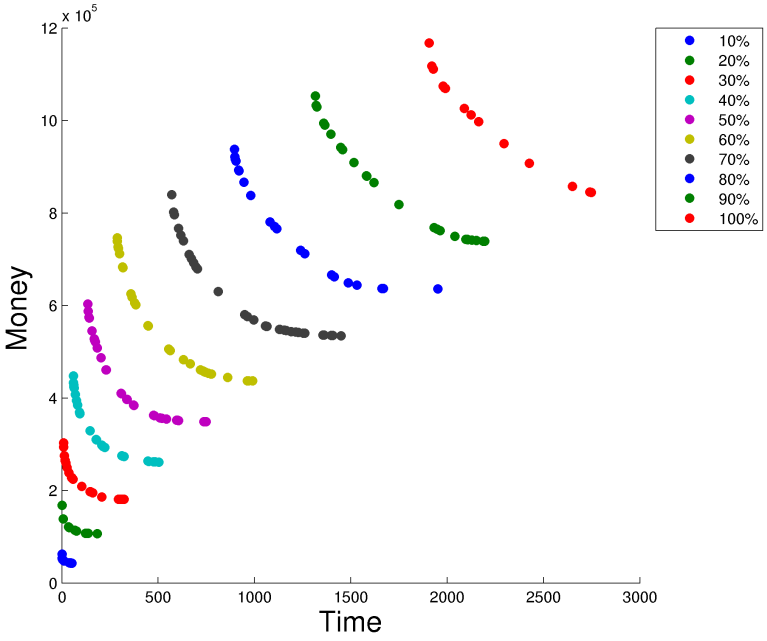

Figure 4.13: Benchmark 4 From 10% to 100% Demand Satisfaction

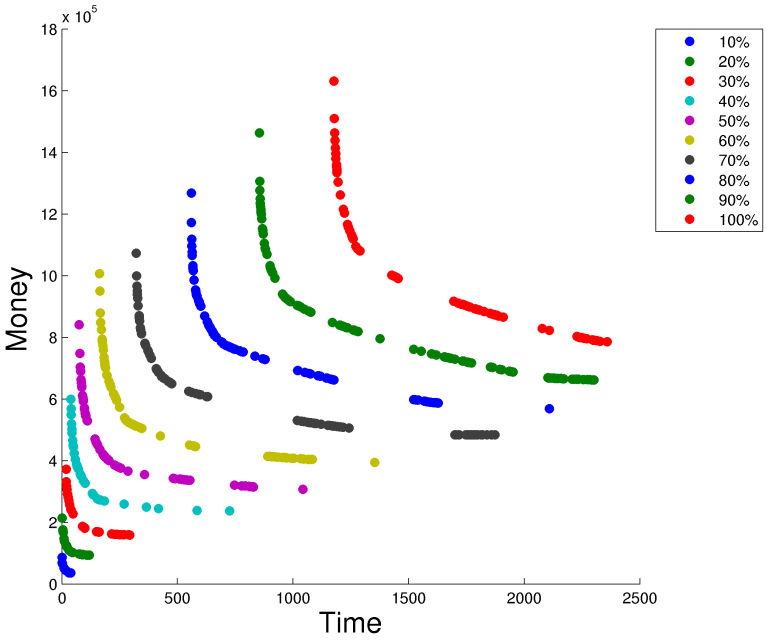

Figure 4.14: Benchmark 6 From 10% to 100% Demand Satisfaction

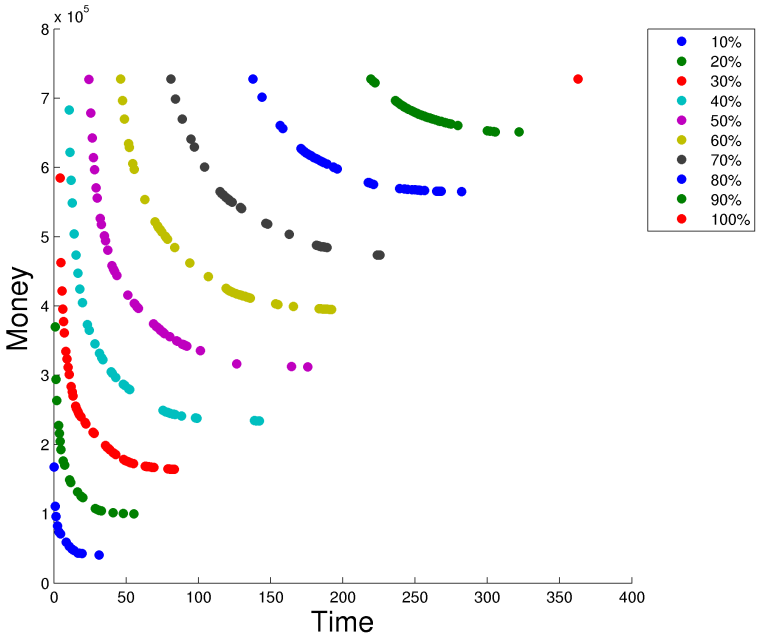

Figure 4.15: Benchmark 7 From 10% to 100% Demand Satisfaction

Chapter 5

Conclusions

This thesis studied an algorithm to find efficient supported solutions of two-objective optimization problems. I modified an existing algorithm (the two-phase method) for totally unimodular (convex) two-objective optimization problems such that it can also be used for non-convex problems. I proved theoretical results concerning the correctness of the modified algorithm, and applied my implementation of it to a particular optimization problem: a modified version of the Single Commodity Allocation Problem (SCAP). I additionally proposed two modifications of my implementation of the procedure $solveRecursion(\cdot)$ that would allow me to find non-unique and collinear solutions. Finally, I discussed two related areas that are open to further study: finding non-supported efficient solutions of two-dimensional optimization problems, and finding efficient supported solutions of three-dimensional optimization problems. These two areas are of particular interest for the SCAP, in which policy makers may want to find non-supported efficient solutions due to specific budget constraints and time limitations, and whose original formulation was as a three-dimensional model. Because they could be of use in solving real-world humanitarian logistics problems, algorithms to find non-supported efficient solutions of two-dimensional problems, as well as algorithms to find efficient supported solutions of three-dimensional problems, are interesting from a practical standpoint as well as from a theoretical one.

Bibliography

[1] Dynadec Decision Technologies, Inc., Dnyadec website: http://:dynadec.com/. *Comet 2.1 User Manual*, 2010.

[2] Matthias Ehrgott. A discussion of scalarization techniques for multiple objective integer programming. *Annals OR*, 147(1):343–360, 2006.

[3] Matthias Ehrgott. *Multicriteria Optimization (2. ed.)*. Springer, 2005.

[4] Pascal Van Hentenryck and Russell Bent and Carleton Coffrin. Strategic Planning for Disaster Recovery with Stochastic Last Mile Distribution. In Andrea Lodi, Michela Milano, and Paolo Toth, editors, *CPAIOR*, volume 6140 of *Lecture Notes in Computer Science*, pages 318–333. Springer, 2010.

[5] Anthony Przybylski, Xavier Gandibleux, and Matthias Ehrgott. Two phase algorithms for the bi-objective assignment problem. *European Journal of Operational Research*, 185(2):509–533, 2008.

[6] Anthony Przybylski, Xavier Gandibleux, and Matthias Ehrgott. A recursive algorithm for finding all nondominated extreme points in the outcome set of a multiobjective integer programme. *INFORMS Journal on Computing*, 22(3):371–386, 2010.

[7] E.L. Ulungu and J. Teghem. The two phases method: An efficient procedure to solve bi-objective combinatorial optimization problems. *Foundations of Computing and Decision Sciences*, (20):149–165, 1995.