A Genetic Algorithm for Automated Model Formulation

Obinna "Obi" Ezechukwu[1]
Department of Computing
Imperial College London
Exhibition Road
London SW7 2BZ
United Kingdom
mailto:oce@doc.ic.ac.uk

Istvan Maros
Department of Computing
Imperial College London
Exhibition Road
London SW7 2BZ
United Kingdom
mailto:im@doc.ic.ac.uk

Departmental Technical Report 2007/1

ISSN 1469-4174

March 2007

Abstract

The challenge of automating the formulation of optimization models is to produce, from a problem description, a well-formed model, which is a mathematically accurate representation of the real-world decision-making problem being considered, and which is suitable for computational purposes. This can be stated more formally as the automation problem, which is the problem of providing 'intelligent' (i.e. automated) assistance during the formulation stage of the mathematical programming process. In this paper, we explore the need to automate model formulation, thus providing a background on the automation problem. We also detail a solution to the problem which is based on evolutionary search techniques.

Keywords: Optimization, Mathematical Programming, Automation, Model Formulation, Evolutionary Search

[1] Corresponding author.

Notation Guide

Font Guide:

UPERCASE BOLD ITALIC	*denotes a subset e.g.* ***A***
UPPERCASE ITALIC	*denotes a set e.g. D*
lowercase bold	*denotes a vector e.g.* **v**
lowercase bold italic	*denotes a formulation genotype e.g.* ***g***
lowercase italic[equations and formulae]	*identifies a variable, set or vector element, also used for function names e.g. x, f*
lowercase subscript italic [equations and formuale]	*identifies an index e.g.* x_i
italic [body text]	*identifies a term or concept that will subsequently be explained in succeeding text/paragraphs.*
underline italic , underline	*emphasis e.g. any*, any

General Symbol and Alphabet Guide:

a, b, x, y	*general purpose letters for identifying variables*
c	*objective function co-efficients*
D	*the set of all the problem characteristics encapsulated in the model base—the problem domain.*
$\Subset$	*'part of' e.g.* $x \Subset y$ *means that x is a constituent of the tuple y*
$\in$	*'element of' a set or vector e.g.* $a \in \mathbf{b}$ *means a is an element of the vector* **b**
n	*denotes the dimension of a collection (vectors and sets), in terms of the number of components.*
$f(\mathbf{x})$	*real-valued function defined over the variable vector* **x**, *usually denotes an objective function*
$f(x)$	*function defined over a variable x*
$f(\boldsymbol{g})$	*the fitness function for a model formulation genotype* ***g***
$\bar{m}$	*a model in the formulation algorithm's model base*
m	*the model instance chromosome of a model formulation* ***g****; an instance is simply a model whose input-data structures have been asssigned values i.e. a model that relates to a specific*

	problem instance
$\boldsymbol{E}$	$\boldsymbol{E} \subset D$ *is the set representing the problem characteristics input to the formulation algorithm*
o	*specifically denotes the size of the problem characteristics set* $\boldsymbol{E}$*, in terms of its number of components.*
$\boldsymbol{W}$	*the set formed by taking the intersection of* $\boldsymbol{A}$ *and* $\boldsymbol{E}$
v	*the size of the set* $\boldsymbol{W}$
$\overline{\boldsymbol{A}}$	$\overline{\boldsymbol{A}} \subset D$ *is the characteristics set for a model in the formulation algorithm's model base*
$\boldsymbol{A}$	$\boldsymbol{A} = \overline{\boldsymbol{A}} \cap \boldsymbol{E}$ *, where* $\boldsymbol{A} \subset D$ *, is the characteristics chromosome of a candidate model formulation* $\boldsymbol{g}$ i.e. *the indexed set of problem characteristics encapsulated by the formulation* $\boldsymbol{g}$
$\boldsymbol{g}$	*a model formulation genotype i.e. the tuple* (m, $\boldsymbol{A}$)
$\overline{\boldsymbol{g}}$	*a prototype of a model formulation genotype i.e. one whose constituent model has not been instantiated; this is expressed formally as a tuple of the form* $\overline{\boldsymbol{g}} = (\overline{m}, \overline{\boldsymbol{A}})$
$Lv.(\boldsymbol{g}_i, \boldsymbol{g}_j)$	*The Levenshtein distance between two genotypes* $\boldsymbol{g}_i$ and $\boldsymbol{g}_j$
G, G', G''	*a generation (i.e. set) of genotypes (i.e. candidate model formulations)*
$h(\mathbf{x})$	*real-valued function defined over the variable vector* $\mathbf{x}$*, usually used in defining an inequality constraint on* $\mathbf{x}$
$h(x)$	*function defined over a variable* x
p	*the non-negative undershoot variable for a soft constraint*
q	*the non-negative overshoot variable for a soft constraint*
R	*the set of real numbers*
S	*the set of feasible solutions to a mathematical model*
$\overline{T}$	*the beginning of a time interval ending at the present time 0 i.e.* $[\overline{T}, 0]$
μ_i	*the average return on security* i *within the time interval* $[\overline{T}, 0]$

x, y, z	*general purpose letters for identifying variable vectors*
ϕ	*the empty set*
σ_{ij}	*the covariance between the security* i *and* j *within the time interval* $[\overline{T}, 0]$
φ	*load factor for the co-efficients in a goal-programming optimization model*
$\Omega(n)$	*order n*
z	*identifies an objective function*

Pseudo-code Notation:

{	start of segment
}	end of segment
*/**	etart of comment
***/**	end of comment
Start With	denotes initialisation sequence in operations
;	line termination
Input	input sequence, parameters, arguments
Begin	start of block of execution
End	end of block execution

Special Notes on Subscripts and Superscripts:

i, j, t, k	*indices (when used as subscripts) e.g.* r_i
$e_i(\mathbf{x}_i)$	*an equality constraint for the* i^{th} *model formulation*
$f_i(\mathbf{x}_i)$	*the objective function for the* i^{th} *model formulation*
$\boldsymbol{g}_i$	*the* i^{th} *genotype in a generation e.g.* $\boldsymbol{g}_i \in G$
G_j	*the* j^{th} *generation produced by a run of the formulation algorithm*
$h_i(\mathbf{x}_i)$	*an inequality constraint for the* i^{th} *model formulation*
m_i	*the m chromosome for the* i^{th} *genotype in a generation i.e.* $m_i \Subset \boldsymbol{g}_i$ *, where* $\boldsymbol{g}_i$ *is an element of some generation G*
$\overline{m}_i$	*the* i^{th} *model in the formulation algorithm's model base*
p_i	*the undershoot variable for the soft constraint created by re-*

	formulating the objective of the i^{th} model
q_i	*the overshoot variable for the soft constraint created by re-formulating the objective of the i^{th} model*
$\boldsymbol{A}_i$	*the **A** chromosome for the i^{th} genotype in a generation i.e. $A_i \Subset \boldsymbol{g}_i$, where $\boldsymbol{g}_i$ is an element of some generation G*
$\overline{\boldsymbol{A}_i}$	*the characteristics set for the i^{th} model in the formulation algorithm's model base*
S_i	*the set of feasible solutions for the i^{th} model formulation*
$\mathbf{x}_i$	*the vector of variables for the i^{th} model formulation*
x_i	*decision variable denoting the holding in the i^{th} security*
y_i	*the post-solution value of the decision variable x_i*

1 Introduction

As a result of vastly improved solution algorithms, modelling tools, and the proliferation of relatively inexpensive computing platforms, mathematical programmers are able to solve problems of much greater scale and complexity than previously possible. Also, relatively inexperienced programmers or novices have greater exposure to mathematical programming; most notably due to solvers embedded in popular desktop spreadsheet software. As opposed to the early practitioners of mathematical programming, current practitioners, from novice to experts, generally speaking, no longer face problems in the numerical computation of results; rather it is now possible to solve problems that are more complex or larger than the modeller can understand. Instead, the critical and often time consuming tasks involved in the creation of mathematical models are: formulating, communicating, managing, debugging, simplifying them and understanding their solutions.

The problems associated with building, using and maintaining mathematical programs have led to a concerted attempt within the research community to create an Intelligent Mathematical Programming System {IMPS} (Chinneck et al. 1999, Greenberg 1991) aimed at tackling these problems. There isn't, as yet, an individual IMPS which provides automated support for all stages of the mathematical programming process; however there are tools, such as MProbe (Chinneck 2001) and ANALYZE (Greenberg 1993)—this was actually born out of a consortium formed with the specific aim of creating an IMPS (see Greenberg 1991)—which provide automated support for individual stages of the process.

In this paper we describe the theoretical foundation—an evolutionary algorithm—for an IMPS that automates the formulation stage of the mathematical programming process. We begin by explaining the motivation for automating model formulation, and we proceed to describe, in detail, an evolutionary algorithm that can be used to automatically formulate mathematical programs using, as input, a set of problem characteristics.

2 Why Automate Model Formulation?

In order to solve any given real-world problem using optimization techniques, it is necessary to formulate (i.e. create) a model, which sufficiently captures its details

and is suitable for computational purposes. Consequently, formulation can be considered as one of, if not the focal point of the mathematical programming process. It is generally carried out after the analysis phase i.e. after a sufficient understanding of the real-world problem characteristics has been gained by the modeller(s). In some cases, the modeller possesses a priori knowledge of the particular problem domain thus removing the need for the analysis phase. Irrelevant of how the problem knowledge is gained, a model of the problem ultimately has to be created in order to enable a solution to be obtained. Formulation is performed by inferring relationships between the problem characteristics (i.e. requirements), decision variables, objectives, and constraints limiting the values of the decision variables—thus constraining the solution space. *Therefore, formulation can be viewed as the process of converting the identifiable characteristics of a real-world problem into a coherent and accurate mathematical representation of it.*

Whereas this is relatively simple for mathematical programming practitioners, it constitutes a barrier to novices and less experienced modellers. Greenberg (2003), states clearly that the role of the formulation component (also known as the model assistant, see Figure 1) in an IMPS is to assist novices in formulating mathematical models. Even if a user possesses detailed knowledge of the problem being solved, a lack of knowledge in mathematical programming techniques can hinder the adoption of such techniques in tackling decision-making problems. The situation is further complicated by the fact that the knowledge requirement is generally not limited to the mathematics of optimization, but also to computing knowledge, specifically proficiency in either a modelling language or a high-level programming language; with the exception of trivial cases, the solutions to mathematical programs can only be obtained in a timely fashion by the use of computing power.

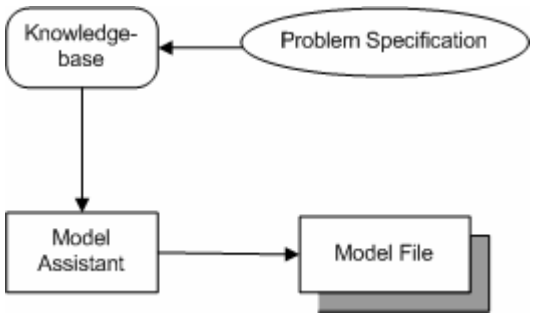

Figure 1: Formulation Component of an IMPS

Solving the IMPS formulation problem essentially involves automating the model formulation process i.e. automatically interpreting the problem characteristics in order to produce an optimization model which is an adequate representation of it. The main advantages of doing this are: novice and non-specialist users are shielded from the mathematical intricacies of optimization; and the need for programming or modelling language proficiency is removed. Automation effectively provides access to optimization techniques to novices and non-specialists whilst shielding them from the complexities of model formulation and implementation.

An example of a mathematical programming system, which provides support for automated model formulation, is MIMI (Chesapeake Decision Sciences). It provides a full set of capabilities for graphical representation of models and data for scheduling problems. In addition to windows and pull-down menus, the user has a collection of icons representing process units or machines, which can be used to change the model by simply repositioning them on the screen. These graphical features provide an interactive and precise picture of plant operations and scheduling options, thus enabling users to specify the problem characteristics utilising natural language. The specified characteristics are processed by the system to create an optimization model matching the user-supplied input.

3 A Genetic Algorithm for Model Formulation

This section presents an algorithm, which automates the formulation of optimization models by adapting techniques from artificial intelligence. It utilises encodings of the domain-specific knowledge of experts, and applies evolutionary search strategies to map a given set of problem characteristics to one or more output models. In simple terminology, the solution is an evolutionary algorithm which employs heuristics or rules to map a set of real-world problem characteristics to a set of candidate models, and by performing the *selection*, *mutation* and *crossover* genetic operations, it evolves new model formulations which provide "*fitter"* matches to an instance of the real-world problem being considered. The algorithm terminates when it is no longer possible to evolve new model formulations from the pool of candidates.

Evolutionary algorithms rely on the principles of natural selection in order to determine, evolve or find the solution to a given problem—such algorithms are frequently categorized under the general heading of evolutionary search techniques.

Evolutionary search techniques, specifically genetic algorithms have been used with great success in the past to generate computer programs spurning a new field of endeavour called genetic programming (Koza 1992) or GP for short. Genetic programs are a special class of genetic algorithms which are concerned with automating the generation of whole computer programs, relying on tree based structures (see Figure 2 for an example) for encoding complex logic structures (e.g. operators) within a chromosome i.e. computer program formulation. In general, GP implementations require: a set of input parameters called terminals e.g. a set of real numbers; a set of functions required to solve a problem (e.g. adding two numbers), where the functions are also normally supplied by the user as input or generated randomly; a fitness measure; and a criterion for terminating a run. The GP works by combining a terminal and a function set in a population of programs (parse trees as in Figure 2), which are then individually evaluated against some fitness criteria, and the operators of selection, crossover, and mutation, applied to produce the next generation. For example, consider a genetic program which generates LISP expressions, if the set of numbers {9, 4, and 7} are specified as the input terminals, the fitness measure *x* specified as the result of evaluating the candidate LISP expression, and the termination condition given as $x = \sqrt{9*4*7}$, then given the right set of input functions, the algorithm may generate an output parse tree similar to that shown in Figure 2—note that in LISP operators precede their arguments, for example, a+b is expressed as (+ a b).

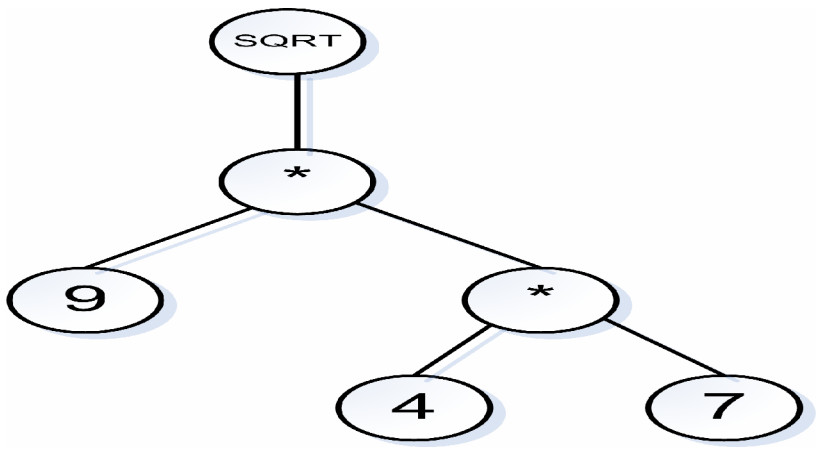

Figure 2: Parse tree for the LISP expression (SQRT (*9(*4 7)))

Genetic programming as a field proves that it is possible to generate computer programs utilising evolutionary search strategies, in particular genetic algorithms. If optimization models are viewed in a declarative programming context (i.e. as a

collection of algebraic modelling language constructs), then it is possible to utilise similar evolution-based techniques to generate them. However, it is not possible to utilise a straightforward or generic genetic programming approach to achieve this because of three principal reasons.

The first of which is that genetic programming either relies on the user to specify a set of input functions for creating an initial generation, or generates them randomly. The major motivation for automating model formulation is to shield the users from the complexities of optimization modelling; clearly, forcing the user to specify functions or operators defeats this purpose. Secondly optimization models have a semantic as well as syntactic meaning, therefore cannot be viewed purely as computer programs—*Semantic meaning in the context of optimization models refers to the real-world problem characteristics embodied by the model i.e. the real-world meaning of model structures, including parameters, variables, objective and constraints*.

Thirdly, it is also not possible to specify termination conditions or fitness functions in the genetic programming sense, because these are dependent on the semantic meaning of the model. As such, a means is required to ensure that the generated models or instances have the semantic characteristics intended by the user i.e. to ensure that the generated model matches the problem's characteristics.

The evolutionary algorithm described in this paper builds on the wealth of research results in genetic algorithms and genetic programming, and utilises heuristics to ensure the semantic integrity of the generated models. The following schematic (Figure 3) provides a visual illustration of the algorithm. As is shown in the figure, the algorithm entails that the problem characteristics are supplied at the onset, and it relies on a 'model base' and a set of encoded rules to map these characteristics to a pre-formulated set of models. This rule-based mapping process effectively constitutes the first phase of the algorithm, and is responsible purely for generating an initial population of models as input to the evolutionary phase of the algorithm.

The pre-formulated models (the 'model base') and the rule store essentially constitute a "knowledge base" of expert knowledge. This is because the models in the model base are formulated by experts in response to previous problems, and the rules in the rule store dictate how to map similar real-world problems as described by a set of characteristics to these models using inference techniques and basic fuzzy operators. The knowledge base therefore ensures that the real-world problem

descriptions as provided by the user, are mapped to models with the right semantic meaning i.e. encapsulate some aspect of the problem's characteristics.

The evolutionary aspect of the algorithm constitutes its core, and, as already mentioned, relies on the mapping aspect simply for supplying the initial generation of candidate formulations. It uses the genetic operators of *selection*, *mutation* and *crossover* to produce composite models, which provide closer fits to the specified set of problem characteristics. Semantic integrity is preserved by the fact that an offspring's semantic meaning is a combination of those of the parents i.e. an offspring represents the same characteristics as both parents combined, and nothing more.

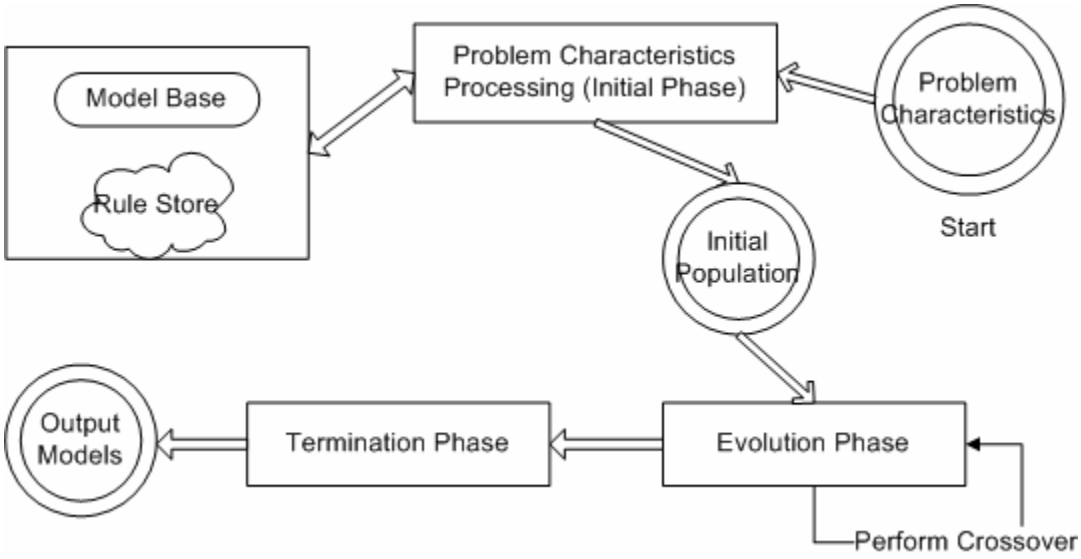

Figure 3: Formulation algorithm schematic

In keeping with the typical structure of genetic algorithms, the algorithm is split functionally into three stages: the creation of an initial generation; evolution; and termination.

The initial phase of the algorithm, also referred to as the mapping phase, creates an initial generation of models based on problem input (incl. problem data); whilst the evolution phase evolves fitter formulations based on the output of the mapping phase. The algorithm enters the termination phase when it is no longer possible or neccessary to evolve formulations from the current generation. **Note:** problem data refers to data describing the specific instance of the problem being solved, for example, the candidate portfolio of securities for a mean-variance optimization problem. It is collected as part of model instantiation, however the means by which this is done and the manner in which it (the data) is represented are implementation specific.

The following subsections describe the phases of the algorithm in greater detail, however, as a precursor, it is necessary to first of all present some background information on the foundations of the algorithm, specifically the genetic encoding of, and the fitness function for, candidate formulations.

The *genotype*—also referred to in this paper as a mate—for a given formulation refers to its complete set of chromosomes, where a chromosome represents a collection of genes. A gene in this context serves the same function as a gene in a living organism, i.e. it encodes a particular trait of a candidate formulation in the same manner as a human gene can encode eye colour.

Formally, the *genotype* $\boldsymbol{g}$ for a candidate formulation consists of two chromosomes namely $\boldsymbol{A}$ and *m* where: $\boldsymbol{A}$ encapsulates the real-world problem characteristics (unique characteristics) embodied by the candidate formulation (expressed as a set); and *m* is the candidate model formulation, and its associated instance data. Thus, the genotype can be expressed mathematically as the tuple $(m, \boldsymbol{A})$. The *allele* of the genes in $\boldsymbol{A}$ is limited to the elements of the set *D* which is a finite set that defines the scope of the application domain to which the formulation algorithm is being applied i.e. the set of all known problem characteristics that can occur in the given domain. A gene's allele is its set of permissible values; in this context, it is the range of values for each element $a_i \in \boldsymbol{A}$.

The authors appreciate that the meaning of the set *D* may not be entirely clear at this juncture; however we believe that its meaning will become clearer with subsequent examples. In the meantime, it is worth mentioning that *D* is defined along the lines of applications of optimization techniques. Consequently product mix problems, for example, belong to a separate domain from financial optimization problems, which in turn also belong to a separate domain from transportation problems.

The **fitness function** measures the closeness of the model formulation encapsulated by a given genotype i.e. the extent to which it represents the real-world problem being modelled, and it is expressed as $f(\boldsymbol{g})$. Its value is given by fraction v/o where: $\boldsymbol{E}$ is the set which holds problem characteristics provided by the user such that $\boldsymbol{E} \subset D$; $\boldsymbol{W} = \boldsymbol{A} \cap \boldsymbol{E}$; $v = |\boldsymbol{W}|$; $o = |\boldsymbol{E}|$.

The fitness function essentially captures "to what extent" a given model formulation resembles the real-world problem i.e. the number of characteristics of the

real-world problem that is embodied by the formulation. As is clear from its definition, the value of the fitness function has a range [0, 1], where 1 represents maximum/full fitness i.e. a full match.

It is worth emphasizing that the contents of D are implementation specific (determined at the time of implementation) and effectively define the scope of the algorithm i.e. limits the problems that it is able to cater for.

The explanations in the following subsections frequently refer to Markowitz's seminal mean-variance optimization model (Markowitz 1952), and as such, it is necessary for reference purposes to present the model in its classic form. This is done in the following set of equations:

$$\min \sum_{i=1}^{n} \sum_{\substack{j=1 \\ j \neq i}}^{n} x_i x_j \sigma_{ij} \tag{1}$$

$$\text{subject to } \sum_i x_i \mu_i \geq b \tag{2}$$

$$x_i \geq 0 \tag{3}$$

$$\sum_i x_i = 1 \tag{4}$$

where: $i, j \in Securities$ i.e. the candidate set of securities (financial assets) from which the optimized portfolio is built; $x_i \in R$ is the holding in security *i* expressed as a fraction of the total holdings i.e. 1; μ_i is the average return on security *i* within a time interval $[\overline{T}, 0]$; σ_{ij} is the co-variance of the security *i* and *j* within the interval $[\overline{T}, 0]$; and $b \in R$ is a desired level of return or dividend. The objective given by equation (1) minimizes the portfolio variance i.e. risk, and the constraint (2) places a minimum limit on the expected return of the portfolio. The additional constraints (3) and (4) restrict short sales, and limit the maximum holdings to 100% respectively.

3.1.1 Creating an Initial Generation

An optimization model can be viewed as encapsulating one or more unique characteristics of a real-world decision-making problem. Consider for example Markowitz's classic mean-variance optimization model, this can be said to encapsulate four core characteristics of the portfolio optimization problem, namely: (a) minimizing the exposure to residual asset risk; (b) a minimum return/dividend requirement or target; (c) a restriction on short sales; and (d) a restriction on the total

holdings. As such, given the set $\boldsymbol{E}$ describing a real-world portfolio optimization problem, where $\boldsymbol{E} = \{a, b, c, d\}$, it is possible to use rules to match it to the classic Markowitz model. NOTE: *The contents of the set* $\boldsymbol{E}$ *refer to the problem characteristics described above i.e. the characteristics (a), (b), (c) and (d) encapsulated by the classic formulation of the Markowitz mean-variance optimization model. Similar notation is used from this point on in the paper.*

The initial phase of the algorithm (listed in pseudo-code in Figure 4) matches a given problem description $\boldsymbol{E}$ to one or more model formulations. It maintains a set of model formulations *M* where each $\bar{m}_i \in M$ is associated with (i.e. logically encapsulates) a set of problem characteristics $\overline{\boldsymbol{A}}_i$. Given a problem description $\boldsymbol{E}$, it creates a set of model formulation genotypes *G* which match the given characteristics to an extent—each model in the set (i.e. each formulation *m* contained in a genotype $\boldsymbol{g} \in G$) encapsulates at least one of the real-world problem's characteristics; so that for each genotype in *G*, its fitness as given by the value of the fitness function $f(\boldsymbol{g})$ is greater than zero i.e. $f(\boldsymbol{g}) > 0$.

The objective of this stage of the algorithm is not to find a single model $\bar{m}$ which encapsulates all the characteristics $\boldsymbol{E}$ of the given problem, although it is possible that an $\bar{m}$ resulting in a genotype $\boldsymbol{g}$ with $f(\boldsymbol{g}) = 1$ may be found and included in the set *G*. It is not always possible to find a single pre-formulated model, which embodies all the characteristics of a given real-world problem. This is because although a real-world problem can reoccur in different scenarios or environments, the reoccurrence may be a modification or variation of the original problem with characteristics that are not fully encapsulated by a single element in the model base. To elaborate, consider the example portfolio optimization model presented earlier; if the real-world portfolio optimization problem being considered requires a positive holding in each security, this would essentially introduce an added characteristic (e) to the problem. As, such the set $\boldsymbol{E}$ describing the problem becomes $\{a, b, c, d, e\}$ where the classic Markowitz model only encapsulates *a*, *b* and *d* giving it a fitness of $\frac{3}{4}$. The added requirement (e) essentially creates a slight variation of the same problem, which is not captured fully by the classic version of the model. Based on the

principles of fuzzy logic, it is possible to say that the classic version of the model captures $3/4$ of the characteristics of the modified problem.

To illustrate with an example: consider that the model base *M* is initialized with the models $\bar{m}_1$, $\bar{m}_2$ and $\bar{m}_3$, all derived from the classic Markowitz portfolio optimization model. Where the example model $\bar{m}_1$ is given by:

$$\max \sum_i \mu_i x_i \tag{5}$$

$$\text{subject to } x_i \geq 0 \tag{6}$$

$$\sum_i x_i = 1 \tag{7}$$

The associated problem characteristics set for $\bar{m}_1$ is $\overline{\boldsymbol{A}_1} = \{a, b, c\}$ where: (*a*) is a target to maximize the portfolio return; (*b*) is a restriction on short sales i.e. no negative holdings; (*c*) is the requirement that all funds are invested, and that the holdings do not exceed 100% of the investment funds available i.e. 1. Note: x, μ_i, σ_{ij}, *i* and *j* retain the same meaning in $\bar{m}_1$, $\bar{m}_2$ and $\bar{m}_3$ as for the model expressed in equations (1) to (4). The second model base element $\bar{m}_2$ is as follows:

$$\min \sum_{i=1}^{n} \sum_{\substack{j=1 \\ j \neq i}}^{n} x_i x_j \sigma_{ij} \tag{8}$$

$$\text{subject to } x_i \geq 0 \tag{9}$$

$$\sum_i x_i = 1 \tag{10}$$

and its associated characteristics set is $\overline{\boldsymbol{A}_2} = \{b, c, d\}$ where (*b*) and (*c*) retain their meaning, and (*d*) represents the target to minimize the portfolio risk. The final element of the model base $\bar{m}_3$ is as follows:

$$\min \sum_{i=1}^{n} \sum_{\substack{j=1 \\ j \neq i}}^{n} x_i x_j \sigma_{ij} \tag{11}$$

$$\sum_i x_i = 1 \tag{12}$$

and its corresponding characteristic set is $\overline{\boldsymbol{A}_3} = \{c, d\}$ where (*c*) and (*d*) retain the same meaning as for $\bar{m}_1$ and $\bar{m}_2$.

Having defined the characteristics sets of $\overline{m}_1$, $\overline{m}_2$ and $\overline{m}_3$, then $D = \{a, b, c, d\}$ i.e. $\overline{\boldsymbol{A}_1} \cup \overline{\boldsymbol{A}_2} \cup \overline{\boldsymbol{A}_3}$. Given an input set $\boldsymbol{E} = \{a, b, c, d\}$ the heuristics phase of the algorithm will perform the following sequence of operations:

1. Evaluate the fitness of candidate genotype $\overline{\boldsymbol{g}_1} = (\overline{m}_1, \{a, b, c\})$, with result $f(\overline{\boldsymbol{g}_1}) = 3/4 = 0.75$.
2. Instantiate[1] $\overline{m}_1$ as m_1 and add new genotype $\boldsymbol{g}_1 = (m_1, \boldsymbol{A}_1)$ to the output set *G*, where $\boldsymbol{A}_1 = \overline{\boldsymbol{A}_1} \cap \boldsymbol{E} = \{a, b, c\}$.
3. Evaluate the fitness of candidate genotype $\overline{\boldsymbol{g}_2} = (\overline{m}_2, \{b, c, d\})$, which will produce the result $f(\overline{\boldsymbol{g}_2}) = 3/4 = 0.75$.
4. Instantiate $\overline{m}_2$ as m_2 and add new genotype $\boldsymbol{g}_2 = (m_2, \boldsymbol{A}_2)$ to the output set *G*, where $\boldsymbol{A}_2 = \overline{\boldsymbol{A}_2} \cap \boldsymbol{E} = \{b, c, d\}$.
5. Evaluate the fitness of candidate genotype $\overline{\boldsymbol{g}_3} = (\overline{m}_3, \{c, d\})$, with result $f(\overline{\boldsymbol{g}_3}) = 2/4 = 0.5$.
6. Instantiate $\overline{m}_3$ as m_3 and create a new genotype $\boldsymbol{g}_3 = (m_3, \boldsymbol{A}_3)$ and add it to the output set *G*, where $\boldsymbol{A}_3 = \overline{\boldsymbol{A}_3} \cap \boldsymbol{E} = \{c, d\}$.

As such, for the given example, the end result or output *G* for this phase of the algorithm will contain the genotypes $\boldsymbol{g}_1$, $\boldsymbol{g}_2$ and $\boldsymbol{g}_3$. The loop within the main body of the algorithm performs three iterations demonstrating that its order can be expressed in terms of the size of the model base. If the model base contains *n* models, then this phase of the algorithm will perform *n* steps, thus its speed can be given as $\Omega(n)$ i.e. order n.

[1] Instantiation in this context refers to the assignment of values to the input data structures of the model, thus turning the model into a 'model instance' i.e. tying it to a specific problem instance. Post instantiation, the generic model formulation $\overline{m}_i$ is transformed into the instance m_i.

The output set *G* serves as input for the evolution phase of the algorithm, which is described in the following section. Note that if the cardinality of *G* is equal to zero, the algorithm moves to the termination phase.

NOTE: Instantiation involves collecting the problem instance data relevant to the model, for example the average return of the securities in the example mean-variance optimization problem. The manner in which this is done and also the way in which the data is represented are implementation specific. In order to simplify the explanation of the algorithm, the chromosome $m \Subset \boldsymbol{g}$ by definition encapsulates both the model and the data needed to describe it.

Start With:
*M /*A set of pre-formulated models*/*
*D /*The problem domain to which the algorithm is being applied, where D is formed by the union of all* $\overline{\boldsymbol{A}_i}$ *, and* $\overline{\boldsymbol{A}_i}$ *are the problem characteristics encapsulated by* $\overline{m}_i \in M$ **/*
*G /*An initially empty output set*/*
Input:
$\boldsymbol{E}$ */*The problem characteristics vector such that* $\boldsymbol{E} \subset D$ **/*
Begin:
loop for all $\overline{m}_i \in M$
{
Create a candidate genotype $\overline{\boldsymbol{g}_i} = (\overline{m_i}, E)$
if ($f(\overline{\boldsymbol{g}_i}) > 0$)
{
Instantiate $\overline{m_i}$ as m_i ; */*this is done by populating the data structures of the model which in a software environment would require prompting the user for input or querying an external system*/*
Create $\boldsymbol{A}_i = \overline{\boldsymbol{A}_i} \cap \boldsymbol{E}$
Create a new genotype $g_i = (m_i, \boldsymbol{A}_i)$
Add the genotype g_i to the output set *G;*
}
}
End:

Figure 4: Pseudo-code for phase 1 of the formulation algorithm

3.1.2 Evolution

The evolution phase constitutes the core of the algorithm. It accepts as input the set of genotypes *G* output by the initial phase, and by applying the genetic operators of *selection*, *mutation* and *crossover* evolves new generations of model

formulations that bear a closer resemblance to the real-world decision-making problem being solved—note that the terms ‘crossover’ and ‘combination’ refer to the same concept and are used interchangeably, the same applies to ‘cross’ and ‘combine’. The resemblance to the real-world problem is obviously measured by the fitness (as given by the fitness function) of the resulting formulations. This phase of the algorithm is essentially the second stage of a genetic algorithm, which given an initial generation *G* of candidate genomes representing candidate model formulations, *selects*, *mutates* and *combines* the genotypes to provide semantically richer offspring.

Crucial to this phase of the algorithm are the *selection, mutation,* and *crossover* operators, and the concepts of *dominance* and *distance.* These are explained in the following subsections; however in order to ensure clarity, it is necessary to make an initial and cursory pass through them.

The *selection* operator is used to select two genotypes for the purpose of *crossover* i.e. mating to produce a new offspring (composite model formulation), and it relies on the feasibility of the parent’s constituent model formulations, and their *distance*. The feasibility restriction is placed on the *selected* genotypes for two main reasons: the first of which is to enable *mutation*; and secondly to enable *crossover*. The *distance* between two genotypes is measured as a ‘*Levenshtein distance’* and indicates how ‘semantically different’ two genotypes (specifically the model formulations which they encapsulate) are. It is crucial for the simple reason that it is better to *cross* dissimilar genotypes than it is to *cross* similar ones. The *‘Levenshtein distance’* is calculated as $Lv.(\boldsymbol{g}_i, \boldsymbol{g}_j)$ and can result in either a zero, positive or negative value. A value of zero indicates equivalence, whilst the sign for a non-zero value indicates the *dominant* genotype, which in turn is the genotype that provides the closer[1] match to the real-world problem being solved. *Mutation* is used to transform the dominant genotype into a form that is more suitable for multi-objective optimization. The subsequent subsections delve into the operators and the concepts mentioned above in greater detail and will enable the reader to gain a fuller appreciation of them.

The evolution phase of the algorithm can be summarised as the following sequence of iterative steps (Figure 5 presents an alternative graphical illustration):

[1] The crossover operator relies on at least one genotype being designated as the dominant one, as such, even in the case where both genotypes have the same fitness i.e. provide equally good representations of the problem being modelled, one of them is still designated as dominant. This will become clearer in the subsection describing the Levenshtein distance.

1. Create an initially empty output set G'.
2. Sort G s.t. $f(\boldsymbol{g}_i) \geq f(\boldsymbol{g}_j)\ \forall \boldsymbol{g} \in G$ iff. $i < j$
3. Outer Loop[1] (for $i=1...n$): *Select* a genotype $\boldsymbol{g}_i \in G$ where $m_i \Subset \boldsymbol{g}_i$ is feasible, and $i \neq n$ where n is the cardinality of G. If $i = n$ then go to step 5.
4. Inner Loop (for $j=i+1 \ldots n$): *Select* genotype $\boldsymbol{g}_j \in G$ where $Lv.(\boldsymbol{g}_i, \boldsymbol{g}_j) \neq 0$ and $m_j \Subset \boldsymbol{g}_j$ is feasible.
    a. *Mutate* $\boldsymbol{g}_i$
    b. *Crossover* $\boldsymbol{g}_i$ and $\boldsymbol{g}_j$ to create a new offspring $\boldsymbol{g}_t = (m_t, \boldsymbol{A}_t)$, where m_t is formed by combining m_i and m_j into a multi-objective optimization model, and $\boldsymbol{A}_t = \boldsymbol{A}_i \cup \boldsymbol{A}_j$.
    c. Add all $\boldsymbol{g}_t$ to G'.

    If there is no such genotype $\boldsymbol{g}_j \in G$ and $\boldsymbol{g}_i$ has the highest fitness value for all genotypes *selected so far* in this generation, and $\boldsymbol{g}_i$ has not mated at all, then move to the termination phase, else increment i and return to the outer loop.
5. If $G' = \varnothing$, then output G and move to the termination phase. Else replace G with G' and return to step 1.

Note: The authors recommend that the algorithm implementation maintain the generation history i.e. that all the generations G should be stored in some sort of collection. This is so as to provide an audit trail which can be used to trace the origins of the output model formulations.

The steps (1-5) above iterate through the set G in a left to right fashion, in an attempt to mate each genotype in G with a semantically *distant* genotype. Clearly each element i selected from the set G (in the outer loop) potentially can mate up to $n-i$ times with the exception of the genotype $\boldsymbol{g}_n$ which does not have any element to its right i.e. it is the last element in the genotype list. The reason for traversing the set in a left to right fashion is to prevent duplicate mating i.e. a case where $\boldsymbol{g}_i$ mates with $\boldsymbol{g}_j$, and on a separate iteration $\boldsymbol{g}_j$ mates with $\boldsymbol{g}_i$. This essentially results in two

[1] Please note that in this paper, set indices are 1 based i.e. the first index is always 1 and not 0.

different copies of the same offspring $\boldsymbol{g}_t$ in G', with the obvious consequence of degrading algorithm performance and introducing unnecessary duplication in the gene pool. In the case where the models $m_i \Subset \boldsymbol{g}_i$ and $m_j \Subset \boldsymbol{g}_j$ model contradictory problem characteristics e.g. in the form of contradictory constraints, the offspring model $m_t \Subset \boldsymbol{g}_t$ will become infeasible and as such $\boldsymbol{g}_t$ will not be selected in the next evolutionary cycle i.e. its genes will not be propagated any further. For example if m_i contains the constraint $x > 0$ for some decision variable $x \in R$, and m_j imposes the constraint $x < 0$ on the same variable, then the resulting model is bound to be infeasible as both constraints cannot be satisfied. However in the interest of performance, it is recommended that some form of conflict detection is built into the algorithm implementation.

Note that step 2 sorts the set *G* in order of fitness, so that the genotype with the highest fitness is at the beginning of the set. This ensures that in each iteration: $f(\boldsymbol{g}_i) \geq f(\boldsymbol{g}_j)$, which simplifies the termination conditions in step 4; and that $Lv.(\boldsymbol{g}_i, \boldsymbol{g}_j) \geq 0$, which implies that $\boldsymbol{g}_i$ can be treated as the dominant genotype and, as such, can be mutated in step 4a. It is possible to forgo the sort in step 2 but, as explained, it greatly simplifies step 4. To provide a degree of flexibility to the implementer, the operators—*selection*, *distance*, *mutation* and *crossover*—described in this paper, are so done in a generic manner so that they are not reliant on the genotypes in a given generation being sorted.

This phase of the algorithm can also be expressed in pseudo-code as in Figure 6. It ends if the set G' is empty or if it is unable to find any mates for a *selected* individual $g_i \in G$ which has the highest fitness value for all selected genotypes in the current generation *G*. In essence, it terminates when it is not possible to evolve any further offspring from *G* (the current generation). It is difficult to measure its performance as the speed is dictated by how quickly the algorithm runs out of distant genetic material i.e. how many runs before it is unable to create further generations from the current generation *G*. In a software environment, the speed is obviously also dictated by the computing architecture, for example the use of parallel computing techniques can be used to speed up the reproduction process so that for a generation of size *n*, *n-1* processes can be used for the selection, mutation, and crossover operators.

3.1.2.1 Selection

The purpose of the *selection* operator is the identification of two genotypes $\boldsymbol{g}_i$ and $\boldsymbol{g}_j$ for *mutation* (*dominant* genotype only) and *reproduction* i.e. *crossover*— where one genotype is the *dominant* genotype (mate), and the other a *subservient* mate. The terms '*dominant*' and '*subservient*' are used for lack of better terminology to indicate that the model formulation encapsulated in the *m* chromosome of the dominant genotype will carry a heavier (or equal) *weight* in the offspring model, than the subservient genotype's. This is because the dominant genotype's model chromosome *m* is considered to be more important than, or equally as important as, the subservient genotype's in the offspring. The importance of the distinction between the two genotypes will be made evident in the explanation of the *mutation* and *crossover* operators. The decision as to which of the genotypes $\boldsymbol{g}_i$ and $\boldsymbol{g}_j$ is designated as dominant or subservient is based on the sign (+/-) of the *Levenshtein distance* which is explained in subsection 3.1.2.1.1.

It is also possible that the dominant and subservient genotypes make equal contributions to the fitness of their offspring, however due to the fact that the mutation operator is performed on the dominant genotype alone, it is still necessary to designate one of them as dominant and the other as subservient. Even in such a case, the *Levenshtein distance* between the two genotypes would still be positive as will become evident later on in this paper.

In a given iteration of the evolution phase, the selection operator is used initially for selecting the genotype $\boldsymbol{g}_i$. In this operation the only restriction imposed by the operator is that $m_i \Subset \boldsymbol{g}_i$ is feasible. Once $\boldsymbol{g}_i$ is selected, a sub- loop is initiated, and in each of its iterations, a *distant* and feasible $\boldsymbol{g}_j$ is selected. Thus, in addition to feasibility, $\boldsymbol{g}_j$ has to satisfy the added restriction of *distance*.

3.1.2.1.1 Levenshtein Distance

The *"Levenshtein distance"* between two genomes $\boldsymbol{g}_i$ and $\boldsymbol{g}_j$ is defined as follows:

$$Lv.(\boldsymbol{g}_i, \boldsymbol{g}_j) = \begin{cases} \left|\boldsymbol{A}_j - \boldsymbol{A}_i\right| & if \ \left|\boldsymbol{A}_i\right| \geq \left|\boldsymbol{A}_j\right| \\ -\left|\boldsymbol{A}_j - \boldsymbol{A}_i\right| & if \ \left|\boldsymbol{A}_i\right| < \left|\boldsymbol{A}_j\right| \end{cases}$$

In words, the Levenshtein distance is the number of elements in $\boldsymbol{A}_j \in \boldsymbol{g}_j$ not in $\boldsymbol{A}_i \in \boldsymbol{g}_i$ when $\boldsymbol{A}_i$ is larger (i.e. has more elements in it) than, or of the same size as $\boldsymbol{A}_j$; or the negation of the number of elements in $\boldsymbol{A}_j$ not in $\boldsymbol{A}_i$ when $\boldsymbol{A}_j$ is larger than $\boldsymbol{A}_i$. In essence, it measures the size of the difference between the $\boldsymbol{A}$ chromosomes of the two genotypes. The Levenshtein distance can either be zero, or a negative or positive (i.e. -/+) integer: it is zero if and only if the $\boldsymbol{A}$ chromosomes of $\boldsymbol{g}_i$ and $\boldsymbol{g}_j$ are equivalent i.e. if the model formulations encapsulated by both genotypes model the same characteristics of the real-world problem being solved; it is negative if and only if the $\boldsymbol{A}$ chromosome of $\boldsymbol{g}_j$ is longer than the $\boldsymbol{A}$ chromosome of $\boldsymbol{g}_i$ (i.e. has more elements in it); else it is positive.

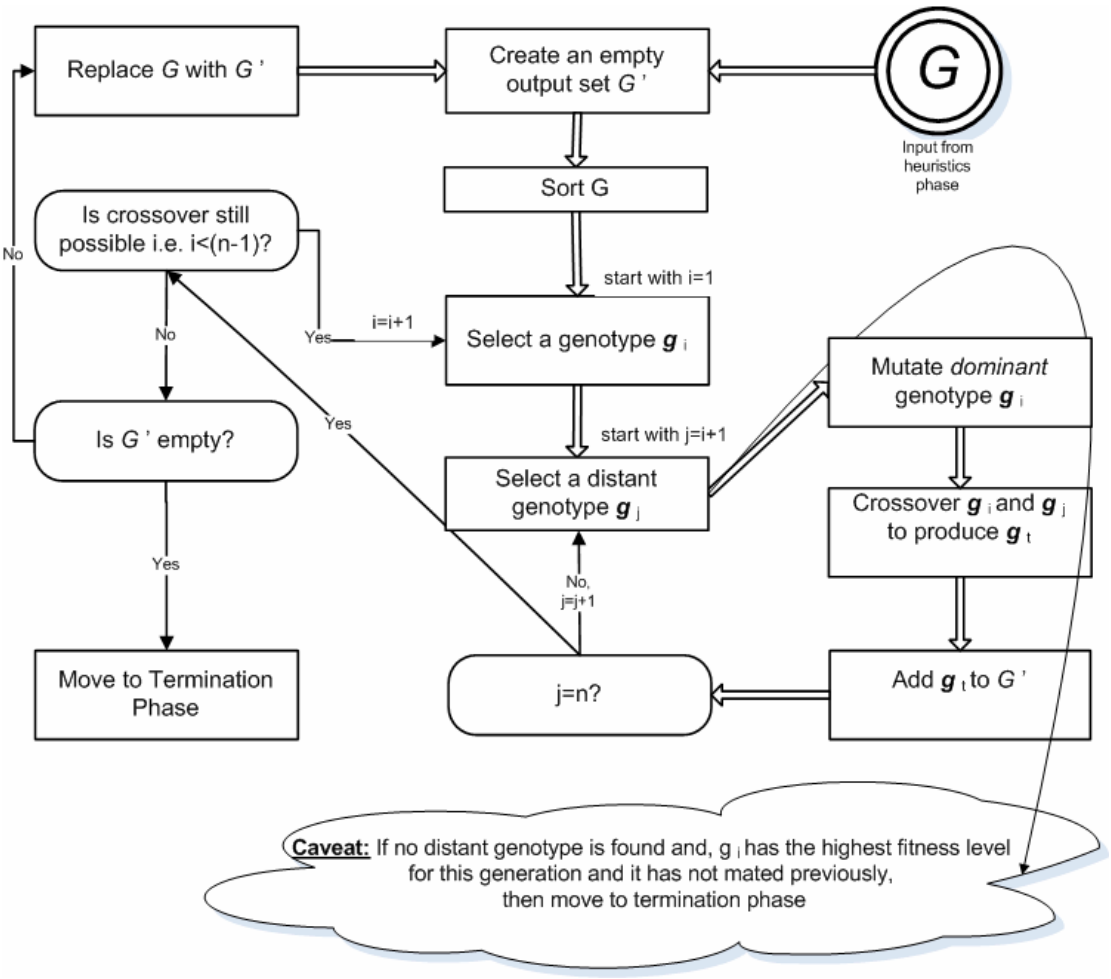

Figure 5: Schematic for the evolution phase of the formulation algorithm

Thus it is possible that $Lv.(\boldsymbol{g}_i, \boldsymbol{g}_j)$ may result in a different signed value from $Lv.(\boldsymbol{g}_j, \boldsymbol{g}_i)$. The reason behind the use of signed values is that in the case of $Lv.(\boldsymbol{g}_i, \boldsymbol{g}_j)$, a positive value dictates that $\boldsymbol{g}_i$ is the dominant genotype, whereas a negative value means that $\boldsymbol{g}_j$ is the dominant genotype. The sign essentially denotes which of the genotypes is more important or should carry more weight in the creation of an offspring $\boldsymbol{g}_t$. This will be explained further in the section covering the *crossover* operator.

Working from the example model-base and problem presented in section 3.1.1, the Levenshtein distance between the genomes $\boldsymbol{g}_1$ and $\boldsymbol{g}_2$ expressed as $Lv.(\boldsymbol{g}_1, \boldsymbol{g}_2)$ is 1. This is because the relative complement of $\boldsymbol{A}_2 = \{b, c, d\}$ in $\boldsymbol{A}_1 = \{a, b, c\}$ is $\{d\}$ which is of length 1, and both sets are the same size, thus the result has a positive sign.

The Levenshtein distance between $\boldsymbol{g}_3$ and $\boldsymbol{g}_1$ (expressed as $Lv.(\boldsymbol{g}_3, \boldsymbol{g}_1)$) on the other hand is -2. This is because the relative complement of $\boldsymbol{A}_1 = \{a, b, c\}$ in $\boldsymbol{A}_3 = \{c, d\}$ is $\{a, b\}$ which is of length 2; and because $\boldsymbol{A}_1$ is the longer set, the result is negative. This example provides a more intuitive reason why one genotype should be considered dominant in the *crossover* process: $\boldsymbol{g}_1$ obviously provides a closer fit to the real world problem, and, as such, the objective of its model instance chromosome (i.e. $m_1 \Subset \boldsymbol{g}_1$) carries a heavier weight in the multi-objective model m_t formed by combining m_1 and m_2. Note that the result of $Lv.(\boldsymbol{g}_1, \boldsymbol{g}_3)$ in this case would be 1, and g_1 would still be the dominant genotype.

A Levenshtein distance of zero between two genotypes $\boldsymbol{g}_i$ and $\boldsymbol{g}_j$ indicates that both genotypes are semantically equal, and consequently, that there would be no benefit in combining them via *crossover.* In essence, because both genotypes model the same characteristics of the real-world problem being formulated, there is no point in combining them as no new characteristics will be modelled by the offspring, rather it will introduce redundancy into the gene pool. To paraphrase, both genotypes combined will not result in an evolutionary step.

3.1.2.2 Mutation

The mutation operator is applied to the dominant genotype only. It re-arranges its *m* chromosome so that the objective function of the model it encapsulates is transformed into a soft constraint—as described in the following paragraphs. The reason for mutation is to ensure that the *'objective function coefficients'* are weighted correctly in the model formulation created by combining the *m* chromosomes of the dominant and subservient genotypes during *crossover*.

The function of the *weights* will become clearer in the description of the *crossover* operator. For the sake of clarity, we state that the 'objective function coefficients' refers to the coefficients of the variables that occur in the objective function; for example, assume that the objective function is $z = c_1x_1 + c_2x_2 + ... + c_nx_n$, where *x* represents the decision variables, the coefficients are denoted by *c*. The mutation process is described as follows. *[NOTE: The procedure/operation described remains the same regardless of whether or not the original formulation models a minimization or maximization problem.]*

If a given *m* chromosome of a dominant genotype ***g*** encapsulates the following optimization model formulation,

$$\max \ f(\mathbf{x}) \tag{13}$$

$$\text{subject to } \mathbf{x} \in S \tag{14}$$

where: f is the objective function; $\mathbf{x}$ is the n-dimensional vector of decision variables; $S \subseteq R^n$ is the set of feasible solutions determined by equality $e(\mathbf{x}) = 0$, and/or inequality $h(\mathbf{x}) \leq 0$ constraints.

The mutation operator first of all solves the model (using the decision-making problem instance data which is also encapsulated in *m*) to obtain the best possible solution. Secondly, if the post-solution values of the decision variable vector **x** are denoted by the vector **y**, it reformulates the model so that it becomes:

$$\min p \tag{15}$$

$$\text{subject to } f(\mathbf{x}) + p - q = f(\mathbf{y}) \tag{16}$$

$$\mathbf{x} \in S \tag{17}$$

where: $p \in R$ is a non-negative undershoot variable, and $q \in R$ is a non-negative overshoot variable.

```
Input:
    G  /*The non-empty output from the heuristics phase*/
Begin:
    loop /*a control loop for initializing G', but not part of the core logic!*/
    {
        t=1; /*output indexing variable*/
        Create an empty set G' to hold the new generation;
        Sort G
        loop for all g_i ∈ G where i ≠ n
        {
                if (m_i ∈ g_i is feasible)
                {
                        loop for all g_j ∈ G where j > i
                        {
                                if (Lv.(g_i, g_j) > 0 and m_j ∈ g_j is feasible)
                                {
                                        Mutate g_i;
                                        g_t =Crossover g_i and g_j;
                                        Add g_t to G';
                                        t=t+1;
                                }
                        } /*end j loop*/
                        if (no distant g_j ∈ G is found and g_i is fittest selection in G)
                        then terminate;
                }
        }/*end i loop*/
        if (G' = ∅) then terminate;
        else replace G with G';
    }/*end control loop*/
End:
```

Figure 6: Pseudo-code for the evolution phase of the formulation algorithm

The objective of the pre-mutated model is essentially turned into a soft constraint (i.e. a goal) with a penalty *p* for under-achievement. It is easy to deduce that the re-formulation does not bias the model in any way, as the minimum value of *p* is zero, and as such, its value after the solution of the mutated instance is forced to be as close to zero as possible. As a consequence of the soft-constraint (16), the value of *q* should

also be zero—this facilitates the desired end-result, which is that $f(\mathbf{x}) = f(\mathbf{y})$. To present a concrete example, consider the example model $\bar{m}_2$ presented in equations (8) to (10) which together with the problem instance data constitutes the *m* chromosome of the genotype $\boldsymbol{g}_2$ (i.e. $m_2 \Subset \boldsymbol{g}_2$): if the post-solution value of each x_i is denoted by y_i the mutated genotype's model chromosome m_2 would be as follows:

$$\min \; p \tag{18}$$

$$\text{subject to } (\sum_{i=1}^{n}\sum_{\substack{j=1 \\ j\neq i}}^{n} x_i x_j \sigma_{ij}) + p - q = \sum_{i=1}^{n}\sum_{\substack{j=1 \\ j\neq i}}^{n} y_i y_j \sigma_{ij} \tag{19}$$

$$x_i \geq 0 \tag{20}$$

$$\sum_i x_i = 1 \tag{21}$$

where: *p* and *q* retain the same meaning as for (15) to (17).

It can be deduced from the above paragraphs that the mutation operator consists of two steps, the first of which is to solve the model with the given problem data, and the second is to reformulate it using goal programming techniques i.e. by turning the objective into a soft constraint. After the mutation of the model formulation, the new objective function consists of a single variable *p* with a coefficient of 1.

3.1.2.3 Crossover

The purpose of the crossover operation is to create a new genotype $\boldsymbol{g}_t$ by combining the genetic material of two parents $\boldsymbol{g}_i$ and $\boldsymbol{g}_j$, where one is the dominant mate, and the other is the subservient mate. As already explained, the dominant mate is chosen based on the value of the Levenshtein distance. In order to simplify the crossover explanation, we will assume that for a crossover operation between $\boldsymbol{g}_i$ and $\boldsymbol{g}_j$: $\boldsymbol{g}_i$ is the dominant mate and $\boldsymbol{g}_j$ is the subservient mate i.e. $Lv.(\boldsymbol{g}_i, \boldsymbol{g}_j)$ is positive; and the mutation operator has been applied to $\boldsymbol{g}_i$.

The crossover operator consists of two core steps, the first of which is to create the $\boldsymbol{A}$ chromosome of $\boldsymbol{g}_t$ i.e. $\boldsymbol{A}_t$. The chromosome is formed by taking a union of the $\boldsymbol{A}$ chromosomes of $\boldsymbol{g}_i$ and $\boldsymbol{g}_j$ i.e. $\boldsymbol{A}_t = \boldsymbol{A}_i \cup \boldsymbol{A}_j$. The new characteristics set $\boldsymbol{A}_t$ reflects the fact that the new genotype encapsulates the problem characteristics

modelled by the two parent genotypes i.e. offers the same expressive power as both parents combined.

After initialising the set $\boldsymbol{A}_t$, the operator merges the model, *m*, chromosomes (i.e. m_i and m_j) of the parent genotypes so that the model chromosome m_t of $\boldsymbol{g}_t$ reflects the new characteristics set $\boldsymbol{A}_t$. This is achieved by combining m_i and m_j into a multi-objective optimization model i.e. (m_t) using goal programming techniques.

If the mutated dominant model m_i is:

$$\min \; p_i \tag{22}$$

$$\text{subject to } \; f_i(\mathbf{x}_i) + p_i - q_i = f_i(\mathbf{y}_i) \tag{23}$$

$$\mathbf{x}_i \in S_i \tag{24}$$

where: p_i and q_i are the non-negative undershoot and overshoot variables respectively, which were introduced during mutation; $f_i(\mathbf{x}_i)$ is its objective function; $\mathbf{x}_i$ is the n-dimensional vector of decision variables; $S_i \subseteq R^n$ is the set of feasible solutions determined by a set of equality $e_i(\mathbf{x}_i) = 0$, and/or inequality $h_i(\mathbf{x}_i) \le 0$ constraints; and $\mathbf{y}_i$ represents the pre-mutation solution values of the variables $\mathbf{x}_i$ for the given problem data.

And the subservient genotype's model formulation m_j is,

$$\max^{1} \; f_j(\mathbf{x}_j) \tag{25}$$

$$\text{subject to } \; \mathbf{x}_j \in S_j \tag{26}$$

where: $f_j(\mathbf{x}_j)$ is the subservient model's objective function; $\mathbf{x}_j$ is its vector of decision variables; $S_j \subseteq R^n$ is the set of feasible solutions for the given problem data, as determined by a mixture of equality $e_j(\mathbf{x}_j) = 0$, and inequality $h_j(\mathbf{x}_j) \le 0$ constraints.

Then, the offspring model produced as a result of the crossover operation will be as follows:

$$\min \; p_i + \lambda p_j + \lambda q_j \tag{27}$$

$$\text{subject to } \; f_i(\mathbf{x}_t) + p_i - q_i = f_i(\mathbf{y}_i) \tag{28}$$

[1] The same approach applies for minimization and maximization problems.

$$f_j(\mathbf{x}_t) + p_j - q_j = f_j(\mathbf{y}_j) \quad (29)$$

$$\mathbf{x}_t \in S_t \quad (30)$$

where: $\mathbf{x}_t$ is formed by taking the union of $\mathbf{x}_i$ and $\mathbf{x}_j$; $S_t = S_i \cup S_j$; $p_j \in R, p_j \geq 0$ and $q_j \in R,\ q_j \geq 0$ are the undershoot and overshoot variables respectively for soft constraint (29) obtained by reformulating the objective of the subservient model m_j; $f_j(\mathbf{y}_j)$ is the pre-crossover solution[1] of the model m_j for the given problem data; λ is a positive weight reflecting the importance of achieving the solution $f_j(\mathbf{y}_j)$ of the subservient model m_j, relative to that of achieving the solution $f_i(\mathbf{y}_i)$ of the model m_i.

In order to create a union of $\mathbf{x}_i$ and $\mathbf{x}_j$, a means is required to prevent the duplication of variables i.e. the double inclusion of variables that occur in both vectors into the new vector $\mathbf{x}_t$. Variables by definition are assumed to have a meaning i.e. they model some real-world quantity, as such, equivalence tests can be performed based on the meaning (probably indicated by the name), type and range of variables. Equality testing or comparison however is within the realms of implementation, and the means by which it is achieved should have no effect on the logic of the algorithm. To present an example means of comparing variables, the AML grammar (Ezechukwu and Maros 2003) uses generic structure to group model elements into logical units, where each unit represents some aspect of the real-world problem being solved. As such, if the *m* chromosome is implemented using such a grammar, it can be assumed that two variable genera which share the same identifier (i.e. name), range, type and calling sequence, model the same real-world phenomenon.

The feasible values of the variables $\mathbf{x}_t$, are dictated by the set of feasible solutions S_t for the new model, where S_t is the union of the solution space of m_i and m_j.

The objective of the new model m_t is created by augmenting the objective of m_i with the undershoot p_j and overshoot q_j variables, where *the positive coefficient* λ *indicates the importance of meeting the objective of* m_j i.e. satisfying the

[1] Assumed to be the best possible solution i.e. may be a local or global optimum depending on the problem and the instance data.

constraint (29). From a genetic programming perspective, it can be viewed as the ratio of the contributions made by the subservient genotype to the new offspring, compared to that made by the dominant genotype. λ is calculated as:

$$\lambda = \frac{\left|\boldsymbol{A}_j - \boldsymbol{A}_i\right|}{\left|\boldsymbol{A}_i\right|}$$

In essence, λ is the ratio of real-world problem characteristics in the set $\boldsymbol{A}_j$, which are not in $\boldsymbol{A}_i$, to the size of $\boldsymbol{A}_i$. So that if there are two characteristics in $\boldsymbol{A}_j$ not in $\boldsymbol{A}_i$, where $\boldsymbol{A}_i$ has a size of four, then $\lambda = 2/4$, meaning that $\boldsymbol{A}_i$ contributes four parts to the fitness of the new offspring, whilst $\boldsymbol{A}_j$ contributes just two parts. Consequently achieving the best possible solution for m_j is only half as important as for m_i. In effect, meeting the requirements embodied by the subservient model is only half as important as meeting those represented in the dominant model, because the dominant model is making twice as many contributions to the fitness of the new offspring.

The reason for mutation is to ensure that the value of λ is effective by fixing the objective coefficient of p_i to 1. In other words, prevent unknown coefficients from appearing in the new objective function. This is because very large coefficients for the objective function variables can render λ insignificant and consequently p_j and q_j. Small or insignificant coefficients can also have the opposite effect i.e. over-emphasize p_j and q_j. In essence, the size of the other coefficients can skew the solution if an effective value of λ is not used. In conventional goal programming, the objective coefficients of the undershoot and overshoot variable are sometimes set by trial and error. A common approach is to use a second load factor φ which is the average of the other coefficients, so that if an objective function is given as $z = c_1x_1 + c_2x_2 + ... + c_nx_n$ where *c* is a vector of coefficients, and *x* is a vector of variables: $\varphi = \frac{\left|c_1\right| + \left|c_2\right| + ... + \left|c_n\right|}{n}$. The objective *z* can then be reformulated as $z = c_1x_1 + c_2x_2 + ... + c_nx_n + \lambda\varphi p + \lambda\varphi q$ where *p* and *q* are the under and overshoot variables respectively for the sub-objective or goal i.e. soft constraint.

To illustrate with an example, consider the genotypes $\boldsymbol{g}_1$ and $\boldsymbol{g}_3$ presented in section 3.1.1, where $\boldsymbol{g}_1$ is the dominant genotype (from section 3.1.2.1.1we know that

$Lv.(\boldsymbol{g}_1, \boldsymbol{g}_3) = 1$). The resulting genotype's ($\boldsymbol{g}_t$) $\boldsymbol{A}$ chromosome is $\boldsymbol{A}_t = \{a, b, c, d\}$ which is a union of $\boldsymbol{A}_1 = \{a, b, c\}$ and $\boldsymbol{A}_3 = \{c, d\}$, consequently giving it ($\boldsymbol{g}_t$) a fitness of 1. Its *m* chromosome (m_t) encapsulates the model formulation:

$$\min \; p_1 + \frac{1}{3} p_3 + \frac{1}{3} q_3 \tag{31}$$

$$\text{subject to } \sum\nolimits_i x_i \mu_i + p_1 - q_1 = \sum_i l_i \mu_i \tag{32}$$

$$\sum_{i=1}^{n} \sum_{\substack{j=1 \\ j \neq i}}^{n} x_i x_j \sigma_{ij} + p_3 - q_3 = \sum_{i=1}^{n} \sum_{\substack{j=1 \\ j \neq i}}^{n} y_i y_j \sigma_{ij} \tag{33}$$

$$x_i \geq 0 \tag{34}$$

$$\sum\nolimits_i x_i = 1 \tag{35}$$

where: p_1 and q_1 are the undershoot and overshoot variables introduced during the mutation of $\boldsymbol{g}_1$; l denotes the post-solution value of x_i obtained during the mutation of $\boldsymbol{g}_1$, by solving the model m_1 for the given problem data; p_3 and q_3 are the undershoot and overshoot variables for the soft constraint (33) which corresponds to the objective function of the model m_3 i.e. the *m* chromosome of $\boldsymbol{g}_3$; and y is introduced during the crossover process and denotes the value of x_i obtained after solving the model m_3 for the given problem instance. In this example: λ indicates that m_1 contributes thrice as much as m_3 to the fitness of the new formulation; and the set of feasible solutions S_t for the offspring formulation (m_t) is defined by constraints (34) and (35).

In summary, the model encapsulated by an offspring's *m* chromosome (m_t) is formed by combining a mutation of the model m_i with m_j, representing the objective function of the latter as a soft constraint with penalty costs for under and over achievement. *Although not stated explicitly, the crossover operator should ensure that the model data (data describing the instance of the decision-making problem) encapsulated in both m_i and m_j is carried forward (or is available) to the offspring m_t, however the manner in which the data is represented and collected is very much implementation specific, as such, this paper does not delve into the data structures*

associated with the model formulations for a given instance of the decision-making problem being solved.

3.1.3 Termination

The algorithm enters the termination phase when it is no longer able to evolve genetically fitter individuals from the current generation *G*. In the evolution phase, this manifests itself in one of two ways: an empty output set G' from an evolution iteration; or an inability to select/find a mate for a given genotype $\boldsymbol{g}_i$ where $\boldsymbol{g}_i$ has the highest fitness value for all genotypes selected in the current generation i.e. its $\boldsymbol{A}$ chromosome contains the highest number of problem characteristics as recorded in the current generation (note that this covers the scenario where its fitness is 1). It is also possible that the initial phase of the algorithm failed to create at least one genotype in response to the problem description; in which case the evolution phase is skipped entirely.

If the output set G' of an evolution iteration is empty, it implies that either: selecting a starting genotype $\boldsymbol{g}_i$ failed i.e. there is no $\boldsymbol{g}_i \in G$ such that $\boldsymbol{g}_i$ is feasible; or for every given starting genotype $\boldsymbol{g}_i$ it is not possible to find a mate $\boldsymbol{g}_j$ where $Lv.(\boldsymbol{g}_i, \boldsymbol{g}_j) \neq 0$ i.e. $\boldsymbol{g}_j$ is distant from $\boldsymbol{g}_i$, $\boldsymbol{g}_j$ is feasible, and $j > i$.

In the case where it is not possible to find a mate for the genotype that has the highest fitness value, then the *m* chromosomes of all the feasible genotypes must model the same problem characteristics i.e. $\forall i, j \;\; Lv.(\boldsymbol{g}_i, \boldsymbol{g}_j) = 0$, or all genotypes $\boldsymbol{g}_j$ which are far from $\boldsymbol{g}_i$ are not feasible. Clearly if the *m* chromosomes of all the feasible genotypes model the same problem characteristics, the genotypes must all have the same fitness value.

In either case, the generation cannot evolve any further and it would be pointless for the algorithm to continue. The objective then becomes to find the subset of fittest individuals from the current generation. The termination phase of the algorithm can be summarised in the following steps:

1. Create an output set G''.
2. Establish the highest fitness value *a* by iterating through *G*.
3. Add all $\boldsymbol{g} \in G$ to G'' where the fitness value of $\boldsymbol{g}$ is equal to *a* i.e. $f(\boldsymbol{g}) = a$.
4. Terminate and output G''.

The algorithm run is considered successful if G'' is not empty. If it is successful its output is the set of genotypes G''. The *m* chromosomes (specifically the model formulations they encapsulate) of the genotypes $\boldsymbol{g} \in G''$ capture the most characteristics of the real-world decision problem being considered, when compared to the other genotypes in *G* which were not included in G''. This is manifested by the fact that they have the longest ***A*** chromosomes.

4 Conclusion

This paper is focused on the problem of automating the formulation of models for optimization problems from specification; a problem whose solution forms a crucial step in the quest to provide automated assistance to modellers for the mathematical programming process. This paper has provided a brief background, thus illuminating the reason for the drive to create an Intelligent Mathematical Programming System (IMPS). It has also explored the specific motivation behind the quest to automate the formulation stage of the mathematical programming process, and presented an evolutionary algorithm that can be used to achieve this.

The authors have thus laid the theoretical foundation for a software solution to the automated formulation problem; a foundation, which is based on basic artificial intelligence techniques and discrete mathematics and which is capable of generating an optimization model given a description of the problem to be solved. The examples provided in this paper show that, in theory, this solution can be applied to real world problem domains.

5 Bibliography

Chinneck, J.W. 2001. Analyzing Mathematical Programs using Mprobe. *Ann. of Oper. Res.* **104** 33-48.

Chinneck, J.W., and Greenberg, H.J. 1999. Intelligent Mathematical Programming Software: Past, Present, and Future. *INFORMS Newsletter.*

Ezechukwu, O.C., and Maros, I. 2003. AML: Algebraic Markup Language, *DoC Departmental Technical Reports 2003/12, Imperial College, London.*

Greenberg, H. J. 1991. An Industrial Consortium to Sponsor the Development of An Intelligent Mathematical Programming System. *Interfaces*. **20(6)** 88-93.

Greenberg, H. J. 1993. A Computer-Assisted Analysis System for Mathematical

Programming Models and Solutions: A User's Guide for ANALYZE. *Kluwer Academic Press, Boston, MA.*

Greenberg, H.J. 2003. The Role of Software in Optimization and Operations Research. Ulrich Derigs, ed. *Optimization and Operations Research. Encyclopedia of Life Support Systems. [http://www.eolss.net],* **E6** 05

Holland, J. 1975. Adaptation in Natural and Artificial System. *Ann Arbour, Univ. of Michigan Press.*

Jones, C. V. 1990. An introduction to graph-based modeling systems, part I: Overview. *ORSA J. on Comput.* **2(2)** 136-151.

Jones, C. V. 1991. An introduction to graph-based modeling systems, part II: Graph-grammars and the implementation. *ORSA J. on Comp.* **3(3)** 180-206.

Koza, J. 1992. Genetic Programming: On the Programming of Computers by Means of Natural Selection. *MIT Press Cambridge MA.*

Markowitz, H. 1952. Portfolio selection. *J. of Finance*. **7** 77-91.

MIMI/LP User Manual, *Chesapeake Decision Sciences, New Providence, NJ,* 1988.