METAMERIC REPRESENTATIONS IN EVOLUTIONARY ALGORITHMS

By

Matthew Lee Ryerkerk

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

Engineering Mechanics—Doctor of Philosophy

2018

ABSTRACT

METAMERIC REPRESENTATIONS IN EVOLUTIONARY ALGORITHMS

By

Matthew Lee Ryerkerk

Optimization problems traditionally involve fixed-length representations to specify a description of a solution. Every solution defines the same number of design variables, resulting in a search space of fixed dimensionality. However, there also exist a large number of variable-length problems, many of which share a common representation. A set of design variables is repeatedly defined, giving the solution vector a segmented structure. Each segment encodes a portion, frequently a single component, of the solution. For example, in a wind farm design problem each segment may encode the position of a single turbine.

We have proposed that such optimization problems, and their solution representation, be described as metameric. In biology this term describes anatomies that are composed of structurally similar, but not necessarily identical, segments. The term metavariable is used to describe a single segment of the encoded solution. Solution length, or the number of defined metavariables, can vary. Finding optimal solutions requires determining both the optimal length and metavariable composition.

Using standard approaches to solve these problems requires an assumption of a fixed number of metavariables, in fact, no theory exists to define optimality for metameric problems. If the optimal number of metavariables is not known *a priori*, this will lead to a sub-optimal solution. A better method is to allow the number of metavariables to vary. As the number varies, so does the dimensionality of the search space, making the use of gradient-based methods difficult. Evolutionary algorithms, using segmented variable-length genomes, are viable and frequently applied to such problems.

This dissertation contributes in the following ways. First, specific definitions for metameric representations and problems are proposed, followed by an extensive survey of metameric problems in literature. It is demonstrated that many practical optimization problems have already been approached using evolutionary algorithms with metameric representations. While there is little cross-referencing among the cited articles, it is demonstrated that there is already a strong overlap in their methodologies. We propose that by considering problems using a metameric representation as a single class, greater recognition of commonalities and differences among these works can be achieved. A greater level of knowledge dissemination would increase the overall quality of the studies.

This is followed by a study of the modifications required to adapt traditional evolutionary algorithms to metameric problems. A set of 6 benchmark metameric problems are created to assess the effectiveness of the new algorithms. First, several specific metameric representations observed in literature are explored. Variable-length genomes, where metavariables can be freely added or removed, are found to provide the

greatest level of flexibility to the other evolutionary operators. Second, several crossover operators, used to generate new candidate solutions, are compared. The best operators are those which minimize the amount of disruption that occurs. Finally, length niching selection is proposed to guarantee diversity of solution lengths in the population. This increases the ability of the algorithm to identify optimal solution lengths. A new selection operator is also developed to be used in conjunction with length niching. It significantly improves the overall algorithm performance and has also been found to be effective on non-metameric optimization problems.

The findings of these studies are used to create a general metameric evolutionary algorithm. This algorithm uses no problem-specific heuristics and can be applied to a broad range of metameric problems. Compared to traditional, fixed-length algorithms this is expected to provide a much better starting point for studies of metameric problems. It is applied to the design of an object with a lattice microstructure and found to improve on the results from the initial studies.

To Tsvetelina, for being so patient.

ACKNOWLEDGMENTS

I first want to extend my gratitude to my dissertation committee. Dr Ron Averill, Dr Kalyanmoy Deb, and Dr Erik Goodman have all offered a significant amount of advice and inspiration since the start of this work, and Dr Neil Wright has provided his guidance extending back to my days as an undergraduate researcher.

I would also like to acknowledge Dr Alejandro Diaz, Abhiroop Ghosh, Ming Liu, and all of our collaborators for their contributions to the TRADES project. I thank the entire Mechanical Engineering department for the opportunity to continue my education, and for all the wonderful colleagues I've met along the way.

My family and friends have made this possible through all of their loving support and encouragement over the years, and for making everything enjoyable.

This material is based in part upon work supported by the National Science Foundation under Cooperative Agreement No. DBI-0939454 to BEACON Center for the Study of Evolution in Action. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions and/or findings expressed are those of the author and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

This work was supported in part by Michigan State University through computational resources provided by the Institute for Cyber-Enabled Research.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

KEY TO SYMBOLS AND ABBREVIATIONS

B_G	Length bias factor at generation G
$D(\mathbf{x}^{(1)}, \mathbf{x}^{(2)})$	Normalized distance between two genotypes
$d(\mathbf{x}_1, \mathbf{x}_2)$	Normalized distance between two metavariables
F_G	Trend of best objective value at generation G
$f(\mathbf{x})$	Objective function
f_G^*	Objective function value of best solution at generation G
G	Current generation
G_{max}	Maximum number of generations
$L(\mathbf{x})$	Number of metavariables in, or length of, solution $\mathbf{x}$
L_G^*	Length of best solution at generation G
L_G^{LB}	Lower bound of length niching window at generation G
L_G^{UB}	Upper bound of length niching window at generation G
P_G	Parent population at generation G
P^{len}	Solutions selected from R^{len} to form the next parent population
Q_G	Child population at generation G
R_G	Combined parent and child population at generation G
R^{len}	Solutions in R with length len
$S(\mathbf{x})$	Total score of solution $\mathbf{x}$
$s_D(\mathbf{x})$	Distance score of solution $\mathbf{x}$
$s_f(\mathbf{x})$	Objective value score of solution $\mathbf{x}$
$s_\phi(\mathbf{x})$	Constraint violation score of solution $\mathbf{x}$
v	Number of design variables in each metavariable
$\mathbf{W}$	Solution lengths used to form niches for the current generation
w	Window width of certain window functions
$\mathbf{x}$	A metameric genotype, or solution
$\mathbf{x}_j$	The jth metavariable of $\mathbf{x}$
$x_{j,k}$	The kth design variable of the jth metavariable of $\mathbf{x}$
x_k^{LB}	Lower bounds for design variable k
x_k^{UB}	Upper bounds for design variable k
$\mathbf{x}^{(i)}$	Solution i of a particular population or set

z	Index value used in ordered problems.
δ_G	Dynamic constraint tolerance at generation G
γ	Balance term used in score selection
$\phi(\mathbf{x})$	Overall constraint violation for $\mathbf{x}$
CGP	Cartesian Genetic Programming
CPPN	Compositional Pattern-Producing Network
EA	Evolutionary Algorithm
GRN	Gene Regulatory Network
MEA	Metameric Evolutionary Algorithm
NEAT	NeuroEvolution of Augmenting Topologies

Chapter 1

Introduction

The task of determining the optimal solution to a particular system or model is known as an optimization problem and can be found in nearly every discipline. Structural design problems may seek to optimize the shape and size of a mechanical structure. Inverse problems might use optimization to determine underlying system information from observations. Businesses must decide how to allocate their resources, financial or otherwise, to maximize their utility.

Each optimization problem is defined by several parts. First, the goals or requirements of the optimization problem are detailed. This takes the form of one or more objectives and constraints that are used to assess the quality of a solution. Second, the aspects of the solution that can be controlled or altered are identified as the design variables. Finally, a model, or an evaluation function, is required to calculate the measures of performance (i.e., objective and constraint values) for a particular design. Once the optimization problem is defined an optimization algorithm can be applied to search the design space.

Solutions to optimization problems are often represented as vectors containing the design variables. In most optimization problems the number of design variables is fixed. As a result most optimization algorithms were designed to operate on a fixed-dimensional design space.

In variable-length optimization problems the number of design variables can vary among solutions. The representations used to encode the solutions and the algorithms applied to the resulting design space can differ radically. Some representations, such as the tree-based structures traditionally used by genetic programming [10], are specialized for that particular approach. Further, many lessons from specialized representations are difficult to generalize to a broader class of variable-length optimization problems.

There also exist a large number of variable-length optimization problems that share a common representation. These problems frequently have solutions that are formed by a number of analogous components.

Examples include turbines in a wind farm, plies in a composite laminate, or nodes in a coverage problem. Each of these components is defined using the same set of design variables (e.g., each turbine might require an x- and y-position). As a result, the vectors defining the solution take on a segmented structure, where each segment contains the design variables necessary to define a single component. The number of components can vary among solutions, resulting in a variable-length problem.

This work proposes the term *metameric* to describe such variable-length problems and their representations. In biology, the term *metameric* is used to describe anatomies that are composed of structurally similar, but not necessarily identical, segments. Examples include a vertebrate spine or the segments of an arthropod.

Most standard, fixed-length optimization algorithms may not be easily applied to metameric problems. For example, it is unclear how gradient-based approaches might be applied when the dimensionality of the design space is changing. Evolutionary algorithms have been successfully applied to a very broad range of optimization problems and are viable candidates for metameric problems. Efficient application of such algorithms to metameric problems requires modifications to the evolutionary operators.

1.1 Purpose

This work has two primary purposes. First, it seeks to bring recognition to the proposed class of metameric problems. It is shown that metameric representations have already been applied to a broad range of optimization problems. Many of the studies approach these problems by modifying traditional, fixed-length algorithms. Despite a frequent overlap in methodologies, there is little cross-referencing in the literature, particularly among studies of different problems. As a result, no commonly accepted algorithm or operators have emerged. Bringing recognition to the proposed class of metameric problems would allow for a greater level of knowledge dissemination.

The second purpose of this work is to develop an effective metameric evolutionary algorithm that can be applied to a wide range of metameric problems. Several aspects of the algorithm are explored in detail, including the solution encoding, variational operators, and selection operators. A set of metameric benchmark problems are used to demonstrate the effectiveness of the various encodings and operators.

We expect that the proposed algorithm will prove useful to studies of metameric problems. It would provide a much better starting point for these studies compared to standard, fixed-length algorithms. This would allow for a reduced focus on the adaptation of the algorithm, in favor of the other contributions of the paper. The proposed algorithm includes no problem-specific heuristics, however these could be incorporated into the algorithm if desired.

1.2 Dissertation Outline

Chapter 2 gives detailed definitions for the metameric representation, problems, algorithms, and related terms. Some key characteristics and challenges that must be considered when applying metameric representations are discussed.

Chapter 3 performs an extensive survey of literature that have applied representations that fit the proposed definition of metameric. Nearly all the cited literature applies evolutionary algorithms to these representations. General descriptions of the problems, representation, variational operators, and selection operator are provided. It is found that metameric representations have been applied to a wide range of problems. While no common methodology has emerged, many of the studies have used similar approaches to handling the metameric representation.

The remainder of the dissertation describes our efforts towards the development of a general and efficient metameric evolutionary algorithm. Chapter 4 gives an overview of the proposed algorithm. Several metameric benchmark problems for demonstrating the algorithm and its operators are also defined.

Chapter 5 investigates several types of metameric representations that were observed in the literature survey. The benchmark problems are used to demonstrate the effectiveness of the representations. Variable-length genomes are found to be the most flexible and result in better algorithm performance compared to representations using a fixed-length genome.

Chapter 6 discusses the variational operators that are used to produce child solutions, with a focus on the crossover operators. It is found that the most efficient operators are those which minimize disruption when forming child solutions.

Chapter 7 proposes several new selection operators for metameric problems. A length niching step partitions the parent solutions into a number of niches based on solution length. Local selection is then applied independently to each niche when forming the new parent population, ensuring diversity of solution lengths. Score selection is proposed as an improved local selection operator. It is not exclusive to metameric problems and its effectiveness is also demonstrated on non-metameric optimization problems.

Chapter 8 introduces recent work related to the design of objects formed by lattice microstructures. Results are improved by approaching this as a metameric problem using a variable-length genome.

Conclusions and suggestions for future work are given in Chapter 9.

Chapter 2

Metameric Definitions

This chapter gives detailed definitions of metameric representations, problems, algorithms, and other associated terms. Some of the challenges and considerations that must be made when approaching metameric problems are also discussed in each section. The following sections are adapted from the survey submitted to Genetic Programming and Evolvable Machines: *A Survey of Evolutionary Algorithms using Metameric Representations* [117].

2.1 Metameric Representations

A *metameric representation* is one in which the genome is at least partially segmented into a number of *metavariables* [118]. Each metavariable contains a set of design variables as defined by a *template*. Other names observed in literature to describe a metavariable include "gene" [40, 55, 149, 7, 81, 132, 13, 48, 26, 130, 131, 120, 135, 133, 32], "chromosome" [58], "substring" [134], or "subsolution" [129].

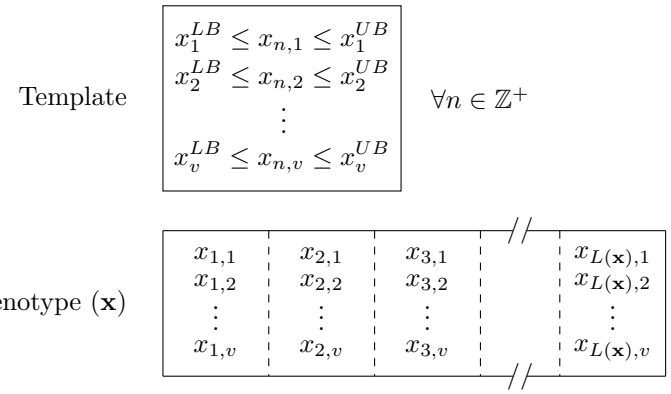


Figure 2.1: A generic template and genotype for metameric problems. This example uses one metavariable type and no global variables.

A generic template and genotype are shown in Figure 2.1. Each metavariable defines v design variables, where $x_{i,j}$ is the j^{th} design variable of the i^{th} metavariable in the genotype $\mathbf{x}$. The bounds for a particular design variable, x_j^{LB} and x_j^{UB}, apply to all metavariables. The length of the genotype $L(\mathbf{x})$ is equal to the number of metavariables it defines. Length is not fixed, metavariables can be added or removed from a genotype while still being interpretable by the evaluation function.

The template in Figure 2.1 uses only real-valued design variables, however a mix of variable types could be used (Section 4.1.1). It is possible that the genome is composed entirely of metavariables, or the problem may also require a fixed number of *global variables*—ones that are not part of the metavariable template. For example, studies investigating gene regulatory networks may include global variables that control protein dynamics [143, 119, 24]. Studies designing optical fibers may use an indirect representation that results in fibers with a radially symmetric cross-section; a global variable is used to control the degree of symmetry [87, 107]. When global variables are present the genome can be considered as two separate parts, a fixed-length portion of global variables and a variable-length portion of metavariables. Different search operators could be applied to each part of the genome.

Most metameric problems use a single metavariable template. Multiple templates can also be used, this is more common for problems with graph-based solutions such as neural networks or truss structures. For example, one template may be used by metavariables defining nodes, and another for the edges (or members, connections) between nodes. Alternatively, studies of such problems may use a metameric representation that only encodes nodes, with the edges then being inferred when mapping the genotype to the phenotype.

2.1.1 Variable-Length

Metameric representations are considered to be inherently variable-length. Their segmented nature allows metavariables to be easily added or removed from the genotype while remaining easily interpreted by the evaluation function. There are several reasons why it might seem to be desirable to use a fixed number of metavariables:

1. Standard algorithms are fixed-length and it is easier to adapt the problem to the algorithm than vice versa. It is possible to run the algorithm several times at different lengths to determine the optimum.
2. Previous experience or heuristics may be used to predict the optimum number of metavariables.
3. There is an equality constraint acting on solution length. For example, a wind farm may have an allocated budget that specifies how many turbines are to be used.

This work hopes to help alleviate the first point by promoting the development of efficient metameric

algorithms that can be applied to a broad range of problems. Such algorithms would be a better option than adapting existing fixed-length ones.

Regarding the second and third points: in previous work, we found that variable-length algorithms may outperform fixed-length ones even if the optimal length is known *a priori* [118]. This may be due to the additional diversity that a population of varying lengths provides, or because the expanded search space allows new and easier paths to optimal solutions. If necessary, a metameric algorithm could easily be tailored such that the final population includes solutions of the required length.

2.1.2 Solution Encoding

A direct encoding is one in which the genotypic space (i.e., the metavariables) is identical to the phenotypic space (i.e., the evaluated solution) [114]. An indirect encoding requires one or more additional steps when mapping the genotype to the phenotype.

Representations where each metavariable encodes multiple components in the phenotype are indirect encodings. For example, in composite laminate design problems, each metavariable may be expressed as a stack of several plies to ensure that certain constraints (e.g., symmetry) are satisfied. Studies to design the microstructure of an optical fiber repeat each encoded component (e.g., holes) in a radially symmetric fashion when forming the phenotype [87, 107]. Problems with graph-based solutions might only encode the nodes; the edges are then inferred during the genotype-phenotype mapping.

Most studies cited in this work use a one-to-one metavariable-to-component encoding; however, very few require many (>100) components in the optimal solutions. Problems that require hundreds or thousands of components may be extremely expensive to solve using a direct encoding. Indirect encodings can greatly reduce the search space size but require problem-specific heuristics. If the genotype of an indirect encoding fits the definition of metameric representations, as the above examples do, then the problem is metameric. Note that this work considers the length of a solution to be the number of metavariables in the genotype, regardless of the number of components in the solution.

Another form of indirect encoding occurs when the metameric representation uses a fixed-length genome. In this case only a subset of the genotype is used to form the phenotype, as discussed in more detail in Sections 3.2.2 and 5.2.

2.2 Metameric Problems

An *optimization problem* can be decomposed into two parts: a *problem* and an *approach*. The problem is the system to which a solution is being sought. For example, the design of a wind farm or a neural network. The

approach establishes the specifics of how a solution is defined (i.e., the design variables, the representation) and how solution quality is measured (i.e., objectives and constraints). Changing any aspect of the problem or the approach will alter the search space and be considered a different optimization problem.

A single problem can be solved by a number of different approaches. Some result in a metameric optimization problem while others do not. This distinction depends only on the representation used for the design variables, as shown in Figure 2.2. Problems whose solutions are formed by a varying number of structurally similar components are well suited for metameric representations. Examples of such problems are given in Table 2.1. However, no problem can be classified as metameric, only the problem-approach combination (i.e., the optimization problem) can.

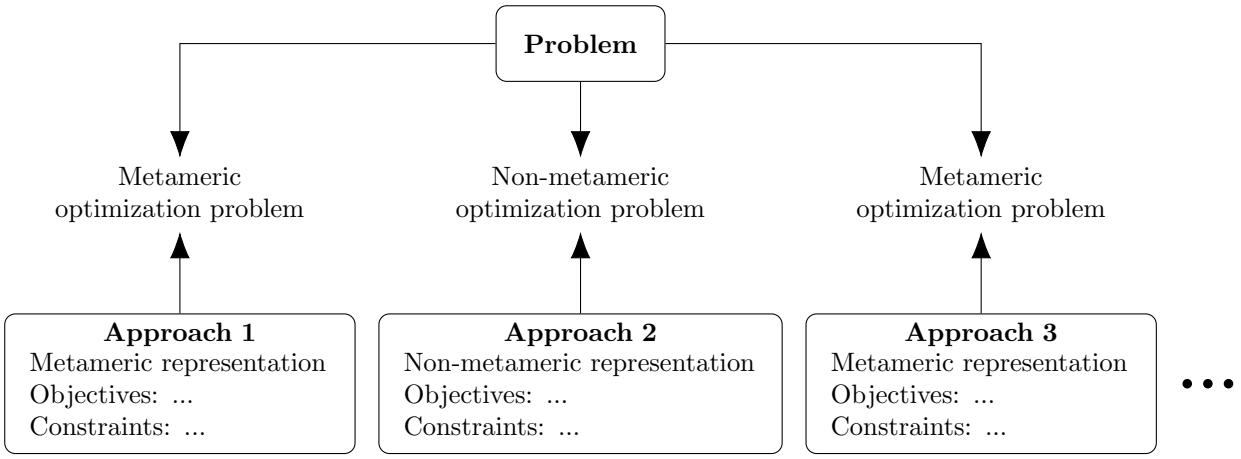


Figure 2.2: An optimization problem is formed by the combination of a problem and approach. Approaches that use a metameric representation result in metameric optimization problems.

For example, consider a wind farm problem seeking to position a variable number of turbines. One approach may use a metameric representation with each metavariable defining a turbine, resulting in a metameric optimization problem. An alternate, non-metameric approach may position the turbines in a geometric pattern controlled by a fixed number of design variables.

This distinction is similar to the one made for multi-objective optimization problems. Problems are not inherently single- or multi-objective. Depending on the approach, specifically the objectives, a problem can be cast as either a single- or multi-objective optimization problem.

In evolutionary terms, an optimization problem is metameric if the genotype uses a metameric representation. Frequently the phenotype will also be metameric, however this is not necessary. If a metameric genotype is mapped onto a non-metameric phenotypic space the optimization problem is still considered metameric. A non-metameric genotype mapped onto a metameric phenotype gives a non-metameric optimization problem.

Table 2.1: Examples of problems well suited for metameric representations. Note that some problems use multiple types of metavariables. Other metameric representations, where each metavariable encodes something other than what is listed here, are possible for each problem.

Problem:	Metavariables represent:	Template design variables:
Wind farm	Turbines	Position, turbine type, height
Coverage	Nodes	Position, type, radius
Composite laminate design	Plies	Material, thickness, orientation
Clustering	Cluster centers	Point in data space
Control	Rules	State-space conditions, resulting action
Microstructure optical fibers	Holes	Position, radius
Portfolio	Assets	Type, amount
Truss	Nodes Members	Position Connected nodes, area, shape
Neural network	Neurons Connections	Activation function, threshold Connected neurons, weight
Electronic circuit	Electronic component	Connected nodes, type, characteristic value

The generic optimization statement of a metameric optimization problem is given in (2.1). This example can also be modified to include multiple metavariable types, non-real-valued design variables, and global design variables while remaining metameric.

$$\begin{array}{lll} \text{Minimize} & f(\mathbf{x}) & \mathbf{x} \in \mathbb{R}^{L(\mathbf{x})\times v}, L(\mathbf{x}) \in \mathbb{Z}^+ \\ \text{Subject to} & g_i(\mathbf{x}) \leq 0 & i = 1, 2, ..., p \\ & h_j(\mathbf{x}) = 0 & j = 1, 2, ..., q \\ & x_k^{LB} \leq x_{n,k} \leq x_k^{UB} & k = 1, 2, ..., v \\ & & n = 1, 2, ..., L(\mathbf{x}) \end{array} \tag{2.1}$$

It is worth noting that non-metameric variable-length representations exist for many optimization problems. Examples include tree-based, grammatical, or other problem-specific encodings. Such representations are not considered by this survey. Generative, or developmental, encodings are only considered by this work if the genotype uses a metameric representation.

For brevity, outside of this section the term *metameric problem* is considered to be synonymous with *metameric optimization problem*.

2.2.1 Ordered Problems

An *ordered metameric problem* is one with a fitness that is sensitive to the ordering of metavariables in the genotype. This is common in composite laminate problems, wherein the layup is determined by the ordering of plies in the genotype. Classification problems that encode a rule in each metavariable may give priority to rules based on order of appearance [5, 4]. Some indirect representations also rely on the ordering of metavariables when forming the phenotype [84, 64, 107].

A permutation operator, capable of rearranging metavariables in the genotype, may be beneficial for ordered problems. When performing crossover the relative ordering of metavariables should be considered. Some operators, such as the cut and splice crossover (Section 3.2.1.1 and 6.1.2) or those used by representations with fixed-length genomes (Section 3.2.2 and 5.2), are also sensitive to the ordering metavariables in the genotype. This is true even if the problem is a non-ordered one.

2.2.2 Solution Validity

A *valid* solution is one to which the evaluation function can assign a meaningful fitness value; one which is directly pushing toward a goal, without artificially added penalty or regularization terms. Valid solutions may be feasible or infeasible, based on constraint satisfaction. An *invalid* solution, either created initially or by evolutionary operators, cannot be evaluated and the fitness values will be arbitrary. If the algorithm cannot distinguish the better of two invalid solutions based on their fitness there may be no pressure toward valid regions of the search space.

For many metameric problems the entire search space will be valid. Regardless of the number of metavariables, or their design variable values, the evaluation function will provide meaningful fitness values. Some problems, such as those with graph-based solutions, may have invalid regions of the search space. For example, a truss may be defined as a mechanism rather than a structure, or a neural network may have no connections to its output nodes. Such solutions are undesirable, and the resulting fitness values may contain no useful information.

Modifications to the evolutionary operators can help guide algorithms toward valid solutions, these are discussed in more detail in Section 3.2.5. These changes are not strictly necessary but may improve overall performance of the algorithm. Problem-specific heuristics are used to design these operators, making them difficult to generalize. A general metameric algorithm might be designed to take advantage of a user-defined repair operator or measure of validity, when available.

2.3 Metameric Algorithms

A *metameric algorithm* is one designed to take advantage of the metameric representation. Gradient-based algorithms are difficult to apply to metameric problems due to the changing dimensionality of the search space. The gradients are not defined as the dimensionality changes. Metaheuristics, such as evolutionary algorithms (EAs), are a better choice due to their lack of dependence on the existence of derivatives. The survey in Chapter 3 only considers the application of metaheuristics to metameric problems, with nearly all the cited studies using EAs.

Non-metameric algorithms can sometimes be applied to metameric problems. It is trivial to apply most EAs to the fixed-length genomes (Section 3.2.2 and 5.2) used by some metameric representations. However, the effectiveness of the search may be greatly improved by considering the representation when designing operators. We expect that the metameric representation shared by these problems will allow for efficient, general metameric algorithms. Such algorithms would not include problem-specific heuristics beyond the segmented, variable-length nature of the genome.

For example, the effectiveness of the crossover operators used in many EAs relies on their ability to preserve building blocks present in the parent solutions. In traditional EAs a building block is a short subsequence of the genome that has a positive effect on fitness. In metameric problems a building block may be a subset of metavariables. Disruptive crossovers are ones which fail to preserve the building blocks in child solutions. Traditional EAs applied to metameric problems may be restricted to exchanging metavariables based on their position in the genotype, this is likely to be disruptive. Specialized metameric crossovers might instead exchange metavariables based on their commonalities (Section 6.1).

Changing the length of a solution, either through crossover or mutation, is typically deleterious to solution fitness. As the length changes, so does the entire fitness landscape. Building blocks in the genotype may no longer be optimal for the new solution length. The deleterious effect can result in such solutions being quickly discarded by traditional selection operators. However, if given a chance to refine and improve, these solutions may emerge as more optimal than those found at previous lengths. Specialized metameric selection operators may facilitate this process by ensuring diversity among the selected solution lengths. Chapter 7 gives further discussion of metameric selection operators.

Chapter 3

Metameric Representations, Problems, and Algorithms in Literature

This chapter provides a survey of metameric representations in literature. A description of the problems solved and the general methodologies used to solve them are provided. It is observed that no commonly accepted metameric algorithms have emerged, despite the common representation. The following sections are adapted from the survey submitted to Genetic Programming and Evolvable Machines: *A Survey of Evolutionary Algorithms using Metameric Representations* [117]. Later chapters compare the performance of several of the representations and operators presented here. When detailing the implementation of these some of the background information from this chapter may be repeated.

3.1 Metameric Problems

This section will give a brief description of some problems that have been solved using EAs with a metameric representation. Of primary interest is how the representations employed fit the proposed metameric structure (i.e., what does each metavariable encode).

The representations used vary among the cited studies. Many employ a representation that matches the one given in Section 2.1. Some representations appear considerably different, however we believe that these could be mapped to the proposed format without significant changes to the study. It is also worth noting that the variable-length nature of these representations is not always a primary focus of these studies.

This list is certainly not exhaustive, however we believe it is sufficient to demonstrate that a large number of problems can be, and have already been, solved using metameric representations. Details on the algorithms used are given in Section 3.2.

This survey does not provide any of the detailed fitness models or nuances of the problem that each piece of literature introduces. Additionally, studies using a metameric representation constitute a small fraction of the literature available for these problems. Readers are referred to reviews or surveys focused on each of these problems, when available, if additional information is desired. The wind farm, coverage, and laminate composite problems described here are adapted as metameric benchmark problems in Section 4.3.

3.1.1 Wind Farm Design

Wind farm design focuses on the placement of wind turbines on a specified site with a defined wind profile. The power produced by an individual turbine is a function of the local wind speed. Each turbine creates a downstream wake which reduces the power produced by any turbine on which it impinges. The overall efficiency of the wind farm can be improved by limiting the downstream interactions between turbines. If the number of turbines is fixed, the objective might be simply to maximize the power produced. If the number of turbines is allowed to vary, then the objective is usually to minimize the cost per unit of power produced, either subject to a constraint on minimum power or with a second objective of maximizing power produced. Several reviews of the wind farm design problem are available [75, 61, 53].

Each metavariable might encode an x- and y-position for each turbine [98, 52, 118, 116]. Alternatively, the domain can be partitioned into a number of rectangular elements *a priori*; potential turbine locations are then limited to the center of each element [99, 54, 35, 126, 19]. This results in a static-metavariable representation, as described in Section 3.2.2.2. Some studies might also define a hub height [98, 52, 19] and/or generator type [98, 52] for each turbine.

3.1.2 Coverage

We use the term *coverage* to encompass a few different problems. Each focuses on the placement of a number of nodes that provide coverage to a domain. Examples include sensors used for detection or transmitters used for network coverage. Evaluation models vary, but each implementation typically seeks to minimize cost while maximizing the coverage and/or reliability of the system. Cost is primarily a function of the number and type of nodes used and may focus on the initial network cost or upkeep costs, such as energy consumption. Luna et al. provide a survey of evolutionary algorithms used in cellular system planning [85], and surveys on wireless sensor networks are available in [1, 37].

The studies discussed here focus on the deployment of such systems. Each metavariable represents one node, defined by a position and possibly a few auxiliary variables such as node type or range [58, 141, 70, 15, 96, 134, 118, 116]. Some studies consider nodes that are already deployed, with positions that cannot be changed. Only a subset of nodes are then activated to achieve the desired coverage for minimal energy usage [38, 69].

3.1.3 Clustering

Clustering is an example of unsupervised learning [65]. A set of items, each defined by a number of attributes, is partitioned into a number of clusters based on similarities among the items. For example, the pixels of an image may be clustered based on their intensities. The fitness measures vary, but are usually a measure of the trade-off between minimizing the intra-cluster and maximizing the inter-cluster distances. Clustering has been studied extensively in the literature; Hruschka et al. [65] and Nanda and Panda [100] perform surveys of evolutionary clustering algorithms while Jain et al. [66] provide a wide review of clustering techniques and applications. The survey by Hruschka et al. [65] contains a section focused on variable-length clustering algorithms.

The method of partitioning the data into clusters varies. Each metavariable might encode a cluster center in the attribute-space, with items then being assigned to clusters based on proximity to the centers [6, 7, 81, 27, 17]. Other studies might only allow for a limited number of potential cluster centers, either based on the location of items or a heuristic applied prior to optimization. A subset of these centers are then chosen to form clusters [137, 102, 104]. Ghozeil and Fogel [47] use an alternative representation in which each metavariable defines a hyperbox used to determine partitioning of the attribute-space.

3.1.4 Classification

Classification is a form of supervised learning. A set of potential classes and a set of items, each defined by a number of attributes, are provided. The correct class membership of each item is assumed to be known. The algorithm creates a classifier that acts as a mapping function from the attribute-space to the class-space [39]. For example, a dataset may describe several attributes for a number of tumors, and whether each tumor is benign or malignant. The classifier would be trained to identify the tumor type from a given set of attributes. Fitness is typically a function of the number of correctly classified items. Classification measures may be used as fitness functions for neural networks [147, 103, 148].

In some studies each metavariable defines a rule [28, 5, 4]. Each rule contains conditions acting on one or more attributes, and commonly the consequent class to which items are assigned if they satisfy the

conditions. Pulkkinen and Koivisto [109] use a decision tree heuristic to create an initial set of rules which are then refined. Alternatively, Fidelis et al. [40] represent each condition with a metavariable, where each individual represents a single rule determining whether or not an item belongs to a particular class. They repeat the optimization algorithm once for each possible class to form a complete set of rules.

Bandyopadhyay et al. [8] and Srikanth et al. [129] encode a hyperplane and ellipsoid, respectively, in each metavariable which are then used to partition the attribute space. Fernández et al. [39] performed a taxonomy of evolutionary algorithms applied to classification problems, including the representation of solutions.

3.1.5 Control Systems

Control system problems optimize a model that directs the behavior of one or more agents or systems in response to some input. Frequently control systems are used as evaluation functions for neural networks [86, 132, 33, 131, 74] and gene regulatory networks [101, 119, 24]. These studies explicitly define a different metameric problem (e.g., a neural network) and have been categorized as such. Fleming and Purshouse surveyed the applications of evolutionary algorithms to control systems [41].

Each metavariable might encode a rule, defining certain conditions for the state space (e.g., sensor data) and a resulting response [14, 144, 11, 133]. Typically the rule that best matches the current state space is used. Chang and Sim [16] optimize a control lookup table for a train, with each metavariable defining a position along the track, which is used to determine if the train should coast or motor.

3.1.6 Composite Laminate Design

Composite laminates are meta-materials formed by stacking and bonding a number of thin laminae, or plies. Each composite lamina is made of a matrix material (e.g., a polymer, such as epoxy) with embedded high strength fibers (e.g., graphite fiber). Through control of the fiber direction of each lamina, the mechanical properties of a composite laminate can be tailored as needed. As the number of laminae changes so does the thickness of the composite laminate. A typical goal of the optimization is to minimize mass (equivalently, thickness) while supporting a defined load. Analysis is usually performed using classical laminate theory [25], or in more complex cases, finite element analysis (FEA) may be employed. Ghiasi et al. [46, 45] performed reviews of the optimization techniques used for composite laminates.

There are frequently other constraints on the layup to ensure manufacturability or proper mechanical behavior. For example, symmetry about the mid-plane may be desirable, as it uncouples the membrane and bending response of the laminate [25]. In most studies, an indirect encoding is used (see Section 2.1.2). Each

metavariable is expressed as several plies in the phenotype; such an encoding helps ensure that the design constraints are met.

Each metavariable might encode, for example, a fiber orientation; some studies also encode material type, fiber volume fraction and/or lamina thickness. The layup is determined by the ordering of metavariables in the genotype. Frequently, studies only consider a single composite laminate panel [80, 127, 106, 105, 112, 78, 118]. Giger [48] simultaneously designs several laminate composite panels. Alternatively, each metavariable might represent a *patch* that covers one or more segments of the object [48, 73]. The layup of each segment is determined by the patches that cover it.

3.1.7 Analog Electronic Circuits

In analog electronic circuit design a number of components, such as resistors, inductors, and capacitors, are used to form a circuit that provides the desired output for a given input. The solutions to these problems are graph-based, and the connectivity of the components must be determined. Frequently, passive filters are designed [84, 55, 149, 3, 26, 77], with the fitness being measured as the difference between a desired and actual frequency response. Other analog applications include transistor amplifiers [84, 149] or computational circuits [120]. Design of digital circuits is also possible; examples are included in Section 3.1.11.

In most studies, each metavariable defines an electronic component type (e.g., resistor, inductor, etc.), a characteristic value (e.g., resistance, inductance, etc.), and the connectivity of that component. Node numbers might be encoded for each component's terminals to determine connectivity [55, 149, 3, 26, 77, 120]. Lohn and Colombano [84] proposed an alternative, ordered representation in which each metavariable encodes a single instruction for assembling the circuit. This representation was also adapted by Hollinger and Gwaltney [64].

3.1.8 Truss

A truss is a mechanical structure designed to support a given load. The solutions are graph-based, with a set of nodes and a set of members connecting them being defined. Optimization of trusses can typically be broken into three parts. *Size* optimization considers the cross-sectional areas of the members, *shape* optimization considers the positions of the nodes, and *topology* optimization allows members to be added or removed. Most frequently, the truss is optimized to minimize its mass while safely supporting the given load. A survey of evolutionary algorithms used for structural design, including design of trusses, is available in [76].

For this to be a variable-length problem, topology optimization needs to be considered. This is commonly

done by first defining a ground structure, as shown in Figure 3.1. This is an example of a static-metavariable representation (Section 3.2.2.2), with a metavariable encoding each member in the ground structure [113]. Other studies also include a metavariable to encode the position of each node, allowing for simultaneous optimization of truss topology, size, and shape [110, 30, 49, 48, 2]. Lee [81] does not use a ground structure; each metavariable encodes a node and includes additional design variables used to infer the members.

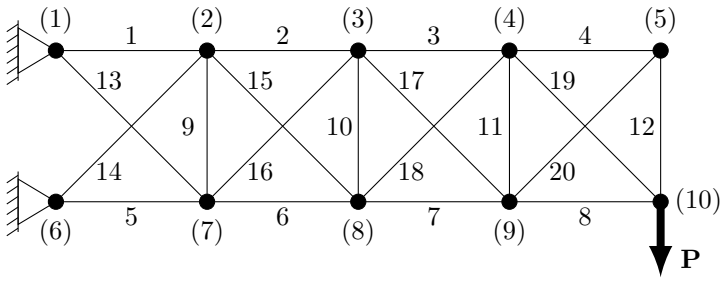


Figure 3.1: An example truss ground structure containing 10 nodes and 20 members.

3.1.9 Neural Networks

Artificial neural networks are directed graphs consisting of a set of *neurons*, or *nodes*, connected by a set of *synapses*, or *connections*. Each network has a number of input and output nodes, determined by the problem definition. The number of hidden nodes, which act as intermediaries between the input and output nodes, can vary. Each connection between nodes typically has an associated weight. Each hidden node typically applies an activation function to the weighted sum of its inputs to determine its output. These outputs are passed along to the subsequent hidden nodes and eventually to the output nodes.

Studies might define a base network topology, similar to the ground structure commonly used in truss problems. Each metavariable then encodes a connection in the base topology [142, 86, 147, 83, 103, 148].

Alternatively each metavariable might encode a node. Khan et al. [74] include a list of nodes in each metavariable to be used as inputs. Several studies encode additional information in each metavariable that can be used to infer the connections between nodes [81].

Stanley and Miikkulainen [132] developed the NEAT algorithm in which separate metavariables are used to represent nodes and connections. They proposed several methods to better handle the variable-length nature of the problem, discussed in Sections 3.2.1.3, 3.2.4.4, and 3.2.6. NEAT was extended to produce compositional pattern producing networks (CPPNs), as discussed in Section 3.1.11.

When large networks are required, a direct encoding may result in an intractable search space. In such cases indirect methods of encoding are a valid alternative. For example, Stanley adapted CPPNs as an indirect encoding for large neural networks in HyperNEAT [131]. Surveys of neuroevolution, including

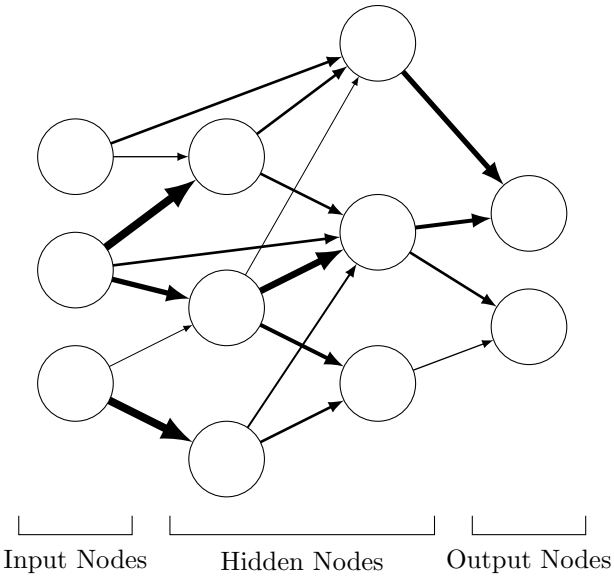


Figure 3.2: A sample artificial neural network using 3 input nodes, 6 hidden nodes, and 2 output nodes. Line width is proportional to connection weight.

indirect methods of encoding, are available in [146, 42].

3.1.10 Gene Regulatory Networks

Artificial Gene Regulatory Networks (GRNs) are an abstraction of the biological processes used by a cell to regulate the expression levels of gene products (e.g., proteins). The level of abstraction can vary significantly among studies of GRNs [128]. In the simplest models the expression levels of one protein may either signal, inhibit, or have no effect on the expression levels of a second protein. This can be represented as a graph, with each node representing a particular protein (or gene, metabolite) and each edge representing an interaction between proteins [59].

Each metavariable might encode for one protein, including information that can be used to infer its interactions with other proteins in the model. Several studies [143, 119, 24] use a protein interaction model similar to the one proposed by Banzhaf [9]. Each protein is defined using an identification, enhancer, and inhibitory tag. Protein interactions are determined based on the tags. Bentley [13] uses a fractal pattern defined by each protein to model interactions.

Trefzer et al. [135] investigate and compare several variable-length representations for GRNs, several of which fit our definition of metameric. Dinh et al. [32] adapt the NEAT algorithm, originally designed for neural networks, for GRNs. Two sets of metavariables are used, one defining proteins and one defining their interactions.

The studies cited here frequently evolve GRNs to match synthetic data or functions (e.g., low-pass filters,

oscillatory behavior, etc.) [13, 24, 32]. A GRN may also be used as a control system [101, 119, 24], or used to produce desired spatial patterns [135, 24]. Wilson et al. [143] use a GRN as an indirect encoding method for the wind farm design problem.

It is also possible to infer information or network information from biological data, however this was not an application of the cited metameric algorithms. Spirov and Holloway survey the models used to represent GRNs in evolutionary computation [128], Sîrbu et al. [125] compare several EAs for GRN inference, and Hecker et al. [59] give a more general review of GRN inference.

3.1.11 Other Metameric Problems

Optical fibers can be fabricated with an arrangement of wavelength-scale air holes through the length of the fiber. The size and arrangement of the holes can be adjusted to tailor the transmission characteristics. Each metavariable could encode the position and size of a hole in the fiber [87, 107]. Typically an indirect encoding is used where the holes encoded in the genotype are repeated in a radially symmetric pattern to form the phenotype.

Multilayer optical coatings are formed by layering a number of materials with varying optical properties. Each metavariable defines a material type and thickness; the coating is optimized to match a target transmission spectrum [56].

Lee and Antonsson [82, 81] use a 2-D shape-matching problem to demonstrate a variable-length algorithm. Each metavariable represents a segment angle and length, drawn from the end of the previous segment.

Chen et al. [20] use a metameric representation for reconfiguring satellite orbits in response to a new ground target for observation. Each metavariable represents the reconfiguration parameters for a particular satellite.

Metameric representations can also be used to create evolutionary art. Stanley augmented his NEAT algorithm, originally designed for neural networks, to produce compositional pattern producing networks (CPPNs) with outputs that could be used to form images [130]. Interactive evolution is employed, in which the user manually picks which images will form the next parent population.

Montemurro et al. [97] optimize the structure of a wing-box. Each metavariable represents a stiffener in the wing-box, defining its geometrical size as well as its desired mechanical properties. A second, non-metameric step is then used to determine a composite layup for each stiffener based on its thickness and desired mechanical properties as determined by the first step.

Gad and Abdelkhalik [44] optimize multi-gravity-assist space trajectories to minimize fuel expenditure. A variable number of planetary swing-bys are defined, as well as any deep space maneuvers that should

occur during the travel time between planets.

de Lucena et al. [29] design a submarine pipeline route, each metavariable representing a curved pipeline segment. The curved segments are then joined together by straight segments to form the complete route. Fitness is largely based on the total length of the pipeline, with some fitness penalties and constraints based on design criteria (e.g., avoiding obstacles, declivity, etc.).

Schoenauer [123] uses various representations for topology optimization, determining a structure's shape to optimally support a given load. In one representation, each metavariable defines a Voronoi point and a label. The resulting Voronoi diagram is used to partition the structure; the labels indicate the presence or absence of material in each cell. Two alternate representations use metavariables to define rectangular shapes. Depending on the representation used, each rectangle may represent the presence or absence of material. Schoenauer et al. [124] use these representations, along with an additional representation defining triangular shapes, to identify mechanical inclusions in a structure. Hamda et al. [57] propose a few alternative Voronoi-based representations for topology optimization.

Digital circuits can be approached as a metameric problem, but the representation can vary greatly among studies. Kajitana et al. [71] use a metameric representation to encode a programmable logic device. Zebelum et al. [149] define a boolean function, each metavariable encoding a single product term. The resulting function could then be used to design a digital circuit. Miller et al. [93, 94] encode in each metavariable a logic cell (e.g., AND, OR, XOR, etc.) and specification of which other cells are used as inputs.

3.2 Metameric Evolutionary Algorithms

The studies cited here typically adapt a traditional, fixed-length EA for the metameric representation. The particular type of EA used is not of particular importance for this survey. This section focuses on the changes to the representation, crossover, and selection operators.

Metameric algorithms can be broadly sorted into two categories, based on whether they use a variable-length or fixed-length genome. Section 3.2.1 discusses algorithms using variable-length genomes, which are then grouped based on the type of crossover operator used. Algorithms using a fixed-length genome, described in Section 3.2.2, are grouped based on the specifics of their representations. Table 3.1 shows which methodologies have been observed for each metameric problem. Some studies fall into multiple categories, because they either compared multiple methods or implemented a blend of operators into a single algorithm.

Changes to the mutation operator, discussed in Section 3.2.3, are relatively straightforward for most metameric algorithms. The selection operator, an important but often neglected part of metameric algo-

Table 3.1: A list of methodologies that have been used to solve each metameric problem.

	Wind Farm	Coverage	Clustering	Classification	Control System	Composite Laminate	Electronic Circuit	Truss	Neural Network	Gene Regulatory Network	Other
Variable-Length Genomes											
Cut and Splice	98, 52, 118	134, 118	6, 17	28, 129, 5	144, 11	48, 73, 118	84			143, 119	71, 87, 107
Spatial Crossover	118, 116	141, 118, 116	81		14	48, 118		81	81		123, 124, 57
Similarity Crossover	118	118				118			132, 131	13, 24, 32	130
Mutation Only	118	96, 118	47		16, 133	118	77, 120			13, 24	
Other				8, 4			3, 26				82, 81
Fixed-Length Genomes											
Hidden-Metavariable	118	58, 70, 15, 118	7, 27			80, 127, 48, 106, 105, 112, 78, 118	55, 149, 64			135	149, 56, 44, 97
Static-Metavariable	99, 54, 126, 35, 19	38, 69	137, 104	40, 109		48		110, 30, 48, 49, 113, 2	142, 86, 147, 83, 103, 148, 74	135	93, 94, 20, 29
Other			102								
Surveys and Reviews	75, 61, 53	85, 1, 37	66, 65, 100	39	41	46, 45		76	146, 42	59, 125, 128	

rithms, is discussed in Section 3.2.4. Section 3.2.5 discusses the adaptations made for handling invalid solutions (see also Section 2.2.2), including repair operators, and Section 3.2.6 discusses population initialization.

It is worth noting that not all algorithms cited here may be metameric. Section 2.3 defined a metameric algorithm as one that gives special consideration to the metameric representation. Most of the cited studies make at least one modification to the algorithm to better handle the representation, resulting in a metameric algorithm. When using a fixed-length genome it is also possible that traditional, non-metameric algorithms can be blindly applied. We have not made special note of such studies here.

3.2.1 Variable-Length Genomes

A variable-length genome changes its length to accommodate more, or fewer, metavariables. In most cases the length of a genotype is equal to the number of metavariables in the evaluated solution. The mutation and crossover operators may both be used to alter solution length. Modifications to the mutation operator

are typically straightforward, as discussed in Section 3.2.3. Several types of crossover operators capable of handling parents of differing lengths are described in the following subsections.

3.2.1.1 Cut and Splice Crossover

A cut and splice crossover is the simplest and most frequently used crossover operator in the studies using a variable-length genome. It is an adaptation of the n-point (most commonly, one- or two-point) crossover often used in EAs, the difference being that the crossover points in the parents do not have to match. As a result, the length of the children may differ from that of either parent. Kajitani et al. [71] use an alternative operator where either a single parent is cut into two shorter children, or two parents are spliced into a single, longer child solution.

Most implementations only exchange metavariables in their entirety, restricting the crossover points to the metavariable boundaries. Some studies allow for intra-metavariable crossover points [28, 129, 11, 5, 87, 17]. In this case the crossover points in both parents must be chosen such that the partially exchanged metavariables will form a complete metavariable in the child.

While simple, the cut and splice crossover is also very disruptive to the solution. The exchanged metavariables are effectively random, depending largely on their positions in the genotype, making it a relatively poor choice of operator for many metameric problems.

3.2.1.2 Spatial Crossover

Spatial crossover operators partition the parent genomes based on one or more chosen design variables. For example, suppose each metavariable defines a location on a plane. A random line could be used to divide that plane into two parts; each parent would then be partitioned into two sets of metavariables that lie on either side of the line. Children are formed by inheriting one set of metavariables from each parent. Other terms used to describe this operator include "geometric(al)" [123, 124, 57, 48], "geographical" [85], and "ordered" [14] crossover.

Since the resulting partitions may not be the same length in both parents this is a length-varying operator. The user is required to identify the design variables used to partition the genomes. Typically, spatial coordinates are used; other variables could be considered but may not result in an effective crossover. For example, a coverage problem may allow each node to have either a small or large coverage radius. Two parent solutions may each have a node at the same location but with different radii. It would be redundant for a child to inherit both of these nodes, while a child inheriting neither node would leave a gap in coverage. Such a result is a possibility if node radius is considered when partitioning the parent solutions.

The advantage of the spatial crossover is that, when properly employed, there is a meaningful relationship between the sets of metavariables exchanged. Building blocks present in the parents are likely to be preserved in the children. Disruption of building blocks can still occur, but the region of disruption is limited to the line or boundary used to partition the solution [21]. Our example divides the solution using a straight line, however any line or shape could be used. One disadvantage of this method is its inability to easily generalize to problems with metavariables that do not define spatial locations or other design variables that allow for a meaningful partitioning.

3.2.1.3 Similarity Crossover

Frequently both parent solutions will share some common, or very similar, metavariables between them. It is expected that these metavariables are functionally similar. It would be redundant for a child solution to receive a copy of a common metavariable from both parents, and a child that doesn't receive either copy will suffer a loss of function. This is a possibility with some crossover operators—the random nature of the cut and splice operator could easily result in a child with multiple copies of the same metavariable. It is also possible with a spatial crossover if two very similar metavariables lie on either side of the dividing line.

Similarity crossovers attempt to avoid this problem by identifying pairs, or groups, of similar metavariables between the parents. Doing so allows the operator to preserve similar structures, or building blocks, in the parents while exchanging dissimilar structures to form new, unique children. Similarity among metavariables is typically a measure of the differences in each design variable. The NEAT algorithm [132], and several studies based on NEAT [130, 131, 32], assign a unique identifying number to each metavariable as it is created. These numbers are used to pair similar metavariables when performing crossover.

Typically, only one-to-one metavariable pairs are allowed, with each child randomly inheriting one metavariable of each pair. Depending on the implementation, some metavariables may remain *disjoint* if a sufficiently similar metavariable doesn't exist in the other parent. Whether or not this crossover can vary solution length frequently depends on how the disjoint metavariables are handled. If one child inherits all the disjoint metavariables from one parent, then the solution length will not change [13, 24]. In several of the NEAT-based algorithms, all the disjoint metavariables in the fitter parent will be inherited by the child. However, the length can vary if both parents have the same fitness.

We have previously proposed an operator that forms a linkage between each metavariable and its most similar counterpart in the other parent [118]. Crossover is then performed in such a way that ensures that no child inherits a pair of linked metavariables. The pairings are not always one-to-one—a single metavariable may be linked to multiple metavariables in the other parent. As a result, changes in length are possible during crossover.

3.2.1.4 Mutation Only

Metameric algorithms may rely entirely on mutation as the length-varying operator. Such algorithms may not use any form of crossover [47, 77, 120, 118], or the implemented crossover is not length-varying. For example, studies might use an n-point crossover where the crossover points match in each parent [16, 96, 133], or a similarity crossover that handles the disjoint metavariables in a way that does not allow for changes in length [13, 24].

3.2.1.5 Other Variable-Length Genome Methods

Das and Vemuri [26] use a specialized crossover for electronic circuit design that ensures that the exchanged subcircuits of two parents will produce valid children.

Ang et al. [4] use a crossover that pools all metavariables from the two parents and randomly distributes them to the child solutions.

When solving a 2-D shape matching problem, Lee and Antonsson [82, 81] assign monotonically increasing index values to each line segment based on length. Crossover is then performed by exchanging metavariables with an index value within a randomly determined range.

Bandyopadhyay et al. [8] use variable-length strings to encode solutions but allow for some bits to be non-coding. Prior to crossover and mutation, solutions are padded using non-coding bits such that they are all the same length. Fixed-length operators are then employed.

Ando et al. [3] use a variable-length genome but do not provide a description of the crossover or mutation operators.

3.2.2 Fixed-Length Genomes

Fixed-length genomes are often used to represent metameric problems. Solution length can be varied by allowing metavariables in the genotype to remain unexpressed in the phenotype. Table 3.2 provides a description of the mechanisms used to accomplish this, along with a list of studies that have employed them. The non-coding portions of the genotype are often referred to as *introns* in the literature.

Despite their frequent use, there is no commonly agreed upon terminology or methodology for fixed-length genomes in metameric problems. Observed terms used to describe this representation include "hidden gene" [44], "active/inactive gene or flag" [149, 135], "control gene" [15], "filtering mask" [104], "activation threshold" [27, 29], "activation mask" [149], "boolean representation" [48], and "don't care symbol" [7].

Each of the methods described in this section requires that the length of the genome be defined *a priori*, creating an upper bound on effective solution length. Care should be taken to ensure the genome is long

Table 3.2: Observed mechanisms for controlling metavariable expression in algorithms using a fixed-length genome.

Methodology	Citations
A binary flag is included in each metavariable. The value of the flag controls whether or not that metavariable is expressed.	[142, 86, 99, 110, 147, 149, 137, 54, 48, 102, 15, 104, 148, 69, 35, 126, 113, 19, 74, 135, 20, 2, 118]
A real-valued flag is included in each metavariable, the value of the flag relative to a defined threshold determines whether or not a metavariable is expressed.	[40, 83, 27, 29]
A single integer mask controls which metavariables are expressed. This may be as simple as using the first N metavariables.	[44, 97]
An existing design variable is adapted such that certain values signal that the metavariable should not be expressed. Integer, discrete, or enumerated variables might include an additional "do not express" value. Some real-valued variables, such as connection weights in a neural network, could be set to 0 to effectively remove them from the solution.	[80, 55, 30, 127, 56, 70, 103, 49, 48, 106, 38, 105, 109, 112, 78]
The inclusion of a *do not care* or *null* symbol that replaces the entire metavariable at that position in the genome.	[58, 7]
A stop metavariable or sequence is defined. Once this is encountered the decoding of the genotype terminates and the remaining metavariables are unexpressed.	[64]
Some graph-based problems use a representation based on Cartesian Genetic Programming [92]. Each metavariable encodes a node, including which other nodes are used as inputs. Nodes that are not used as inputs by any other nodes will remain unexpressed.	[93, 94, 135, 74]

enough to represent solutions of the optimal length. An algorithm could be designed to increase the genome length of the entire population if the upper bound is approached, however no such methodology has been observed.

One of the primary benefits of using a fixed-length genome is the ease of implementation. It is possible that traditional, fixed-length operators might be applied, such as n-point crossover. Despite this, many studies will modify one or more operators to increase their effectiveness for metameric representations.

This survey proposes that metameric algorithms using a fixed-length genome can be divided into two classes: the *hidden-metavariable representation* and the *static-metavariable representation*. The difference between the two is that the static-metavariable representation incorporates some problem-specific, underlying structural information into the representation, while the hidden-metavariable representation does not.

3.2.2.1 Hidden-Metavariable Representation

The hidden-metavariable representation is the more general form of using fixed-length genomes. While the genotype is of a fixed-length, the phenotype remains variable-length. Whether or not a metavariable is expressed in the phenotype is determined by one of the mechanisms described in Table 3.2. An example of a hidden-metavariable genotype and resulting phenotype is shown in Figure 3.3.

Genotype

$x = 6$ $y = 2$ flag $= 0$	$x = 2$ $y = 5$ flag $= 1$	$x = 8$ $y = 1$ flag $= 1$	$x = 1$ $y = 7$ flag $= 0$	$x = 9$ $y = 2$ flag $= 0$	$x = 3$ $y = 2$ flag $= 0$	$x = 7$ $y = 8$ flag $= 1$

Phenotype

$x = 2$ $y = 5$	$x = 8$ $y = 1$	$x = 7$ $y = 8$

Figure 3.3: An example of a hidden-metavariable representation using a binary flag to control metavariable expression.

Studies apply traditional crossover and mutation operators. The performance of the crossover operator is highly dependent on the distribution of the expressed metavariables in the genome. Metavariables can usually only be exchanged if they occupy the same position in the genotype of each parent, however there may be no meaningful relationship between the two. If the distributions of metavariables in two parent genomes are drastically different, then the exchange of metavariables to form children is effectively random. Over time, it is likely that the population will converge toward a common distribution of metavariables in the genome, allowing for a less destructive crossover.

The interaction of the mutation operator and the mechanism for controlling metavariable expression should be considered. It might be desirable that different mutation rules be applied to these design variables. If the algorithm requires a measure of distance between solutions, perhaps for a niching operator (Section 3.2.4.4), it is unclear if it should be based on the phenotypic or genotypic distance.

3.2.2.2 Static-Metavariable Representation

The static-metavariable representation includes some underlying structural information of the solution in the representation. Each genotype has two parts: a *static-genotype* and a *free-genotype*. Each is defined using the same number of metavariables, but contain different sets of design variables. The static-genotype contains design variables describing the underlying structure that are defined *a priori*, either by the user or by an initial computational step. The free-genotype contains the design variables that are subject to optimization, including those that control metavariable expression as described in Table 3.2. To form the

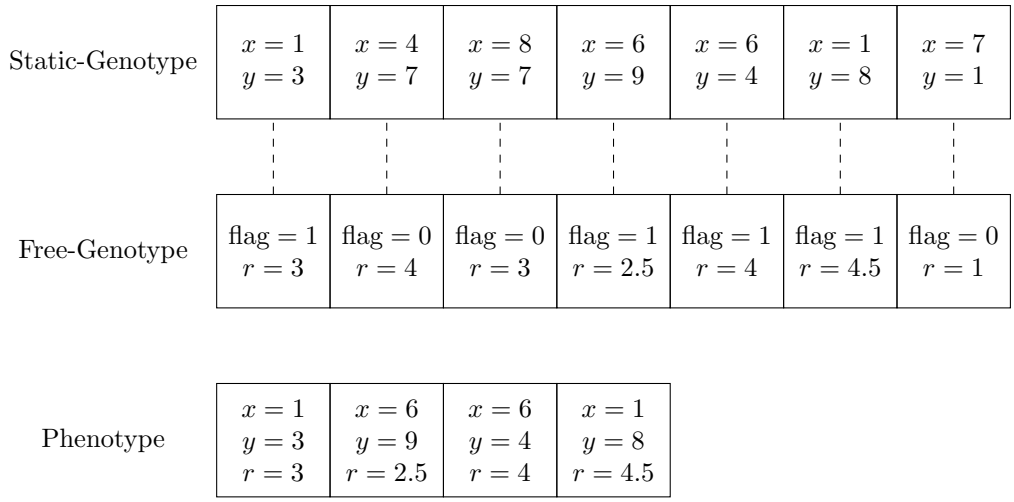


Figure 3.4: An example of a static-metavariable representation using a binary flag to control metavariable expression.

phenotype, the free-genotype is combined with the static-genotype, as demonstrated in Figure 3.4.

Due to their similarities, it can sometimes be difficult to distinguish between the hidden- and static-metavariable representations. The primary difference is that in the static-metavariable representation the phenotype is dependent on the position of each expressed metavariable in the free-genotype. Changing the position of a metavariable in the free genotype will also alter the static design variables associated with that metavariable, resulting in a different phenotype. In the hidden-metavariable representation the positioning of the expressed metavariables does not affect the phenotype. If the problem is an ordered one (see Section 2.2.1), then the relative ordering of metavariables is important for a hidden-metavariable representation, but not their particular positions.

An efficient metameric algorithm may function differently for hidden- or static-metavariable representations. It is possible that the static-genotype is visible to the optimization algorithm. In this case certain operators may be improved by considering both the free- and static-genotypes. Even if the problem is considered black box, where the static genotype is not visible, knowing that there is an underlying solution structure can be beneficial when designing operators.

The static- and free-genotypes are terms proposed by this survey. Most studies cited here propose a particular solution structure *a priori*; the genotype is then mapped onto this structure to form the phenotype. The information required to define this structure is what we consider the static-genotype. This does require some problem-specific heuristics—the static-genotype used for one problem is unlikely to be applicable to other metameric problems.

This representation might be used when there is a limited number of permissible components in the solution—for example, there may be a limited number of acceptable sites to place a cell phone tower. The

acceptable sites could be encoded into the static-genotype and the free-genotype would then control which sites contain a tower. Alternatively, this representation may be used to reduce the size of the search space if the general structure of an optimal solution is known *a priori*. This information could be included in the static-genotype, rather than requiring the algorithm to discover this structure.

For the remainder of this section we will describe the various implementations of a static-metavariable representation in the cited literature.

Wind farm problems commonly limit the potential positions of turbines to points on a grid; these points are then used to define the static-genotype [99, 54, 126, 35, 19]. A similar representation has been applied to several coverage problems where sensors are considered to be pre-deployed. The location of each sensor is contained in the static-genotype; the optimization then determines a subset of sensors to activate [38, 69].

Clustering problems may define a set of potential cluster centroids, determined by a heuristic method, in the static-genotype [137, 104]. Pulkkinen and Koivisto [109] use a heuristic to construct an initial set of rules in a classification problem. These rules are then refined using a metameric algorithm; certain portions of each rule are not subject to optimization and form the static-genotype. Fidelis et al. [40] solve a classification problem by having each metavariable represent a condition; the location of the metavariable in the genome determines which attribute the condition acts on.

Giger [48] designs several composite laminates in a part simultaneously. Each metavariable defines a ply; the laminate that the ply belongs to is dependent on the position of the metavariable in the genome. Chen et al. [20] solve a satellite orbit reconfiguration problem; each metavariable position in the free-genotype is associated with a particular satellite. In de Lucena et al. [29], they define each curved segment of a submarine pipeline route as a metavariable; each curve is positioned relative to base points that are encoded in the static-genotype.

Problems with a graph-based solution may define a highly connected ground structure or network *a priori*—for example, the truss shown in Figure 3.1. The static-genotype defines the potential edges between nodes, and the free-genotype controls which edges are present in the phenotype. Such a representation is common when designing neural networks [142, 86, 147, 83, 103, 148] and trusses [110, 30, 49, 48, 113, 2]. The free-genotype may also define problem-specific design variables for particular edges. For example, each metavariable might also define the cross-sectional area of each member in a truss, or connection weights in a neural network. Studies may also include additional metavariables for each node; these might define positions of nodes in a truss or the biases of nodes in a neural network.

Cartesian genetic programming (CGP) [92] may also be used to represent graph-based solutions. Each metavariable encodes a node, including which other nodes are connected as inputs. This is considered a static-metavariable representation since the connections are encoded using node labels, and the node labels

are assigned based on the position of that node's metavariable in the genotype. Studies in this survey have applied a CGP representation to neural networks [74], gene regulatory networks [135], and digital circuit design [93, 94].

3.2.2.3 Other Fixed-Length Genomes

Omran et al. [102] use a representation that closely resembles static-metavariables for a clustering problem. The static-genotype initially contains a random subset of data points that can be used as cluster centroids. Occasionally this static-genotype is updated using the best solution in the population and a random subset of the data points.

3.2.3 Mutation Operators

Modifications to the mutation operator should consider the representation. When a variable-length genome is used, most studies include a small chance to add or remove a metavariable. If a fixed representation is used then the interaction of the mutation operator with the design variables that control metavariable expression, described in Table 3.2, is important.

It may also be beneficial to adapt the rate of mutation to solution length, which may vary considerably during a single trial. A fixed rate of mutation may result in too many or too few mutations, depending on solution length.

Some studies also introduce a permutation operator to alter the ordering of metavariables in a genome [80, 127, 15, 87, 105]. This may be done if the problem is an ordered one, as described in Section 2.2.1. In unordered problems, the permutation operator does not directly affect solution fitness but may improve the effectiveness of certain crossover operators.

More specialized mutation operators have been observed, but these are generally problem-specific and considered to be outside the scope of this survey.

3.2.4 Selection Operator

The selection operator is an important, and oftentimes neglected, aspect of metameric algorithms. Even though the number of metavariables can vary, the number of constraints and objectives remains fixed. This leads many studies to employ the standard selection operators that have been developed for fixed-length EAs. However, there are two reasons why more specialized selection operators may be necessary: bloat and premature convergence.

Bloat, a common issue in genetic programming, is the uncontrolled growth of solution length without

Table 3.3: A list of adaptations to the selection operator used in metameric problems.

	Wind Farm	Coverage	Clustering	Classification	Control System	Composite Laminate	Electronic Circuit	Truss	Neural Network	Gene Regulatory Network	Other
Multi-Objective	126, 118	141, 70, 15, 96, 69, 134, 118		109		106, 112, 78, 118	149	113			149, 87, 107, 20
Parsimony Pressure				8, 5	144, 11		55, 3		142, 81, 103		
Length Constraints	98, 126, 52		47	8	144, 11	105, 73	84			143, 119	
Niching	116	116			16				81, 132, 131	32, 24	130, 20

a significant return in fitness [108]. Bloat lowers the comprehensibility of the solution while increasing the computational resources required to evaluate it. This is possible, in part, because solution fitness is not generally considered to be a function of length. In genetic programming, it is possible that larger programs are favored since they tend to produce children more similar to the parents than do smaller programs [79, 108].

Some metameric problems discussed in this survey may also be susceptible to bloat. This is more likely to be an issue in classification, control system, electronic circuit, neural network, and gene regulatory network problems. In each of these, fitness is frequently not treated as a function of length; some minimal level of complexity may be required, but there is not an inherit penalty for very large solutions.

Whether or not bloat can occur also depends heavily on the crossover and mutation operators used. A cut-and-splice operator can produce children significantly longer than either parent, this may make the algorithm more susceptible to bloat. Other operators, such as a spatial or similarity crossover, generally produce children closer to the length of either parent. Although bloat may still occur, this can help limit the rate at which it occurs.

In other metameric problems, fitness is already a function of solution length. Longer solutions tend to have a higher costs in terms of mass, finances, power consumption, or thickness, for example. Typically this cost is weighed against some other constraint or objective that produces a pressure toward longer solutions. For example, a coverage problem may seek to minimize cost while requiring a certain level of coverage, or a laminate composite problem will minimize mass or thickness while safely supporting a given load. These competing objectives and constraints will result in an optimal solution length and limit the risk of bloat.

Premature convergence describes the tendency for some metameric algorithms to get stuck at a sub-

optimal solution length. As discussed in Section 2.3, length-varying operators tend to become more destructive to solution fitness as an algorithm progresses. Eventually the algorithm may converge to a single solution length. Solutions of other lengths may still be produced, but if they are not given a chance to survive and refine themselves the algorithm may never reach more optimal lengths. Increasing the diversity of the population—in particular, the diversity of solution lengths—can help to avoid premature convergence.

The following subsections describe several of the modifications to the selection operator that have been observed in the surveyed literature. An overview of the types of modifications used for each metameric problem is given in Table 3.3.

3.2.4.1 Multi-Objective Selection

Multi-objective EAs, such as NSGA-II [31], have been extensively used for optimization. One goal of these is to maintain a level of diversity among the solutions of the Pareto-optimal set [31].

A level of length-based diversity can be maintained in metameric problems through multi-objective optimization. This can be a result of using solution length as an objective, or of using several objectives that apply competing pressures on length. Suppose one objective creates a pressure toward longer solutions and a second objective creates a pressure toward shorter solutions; the resulting Pareto-optimal set will likely contain solutions of different lengths. An example of this is shown in Figure 3.5.

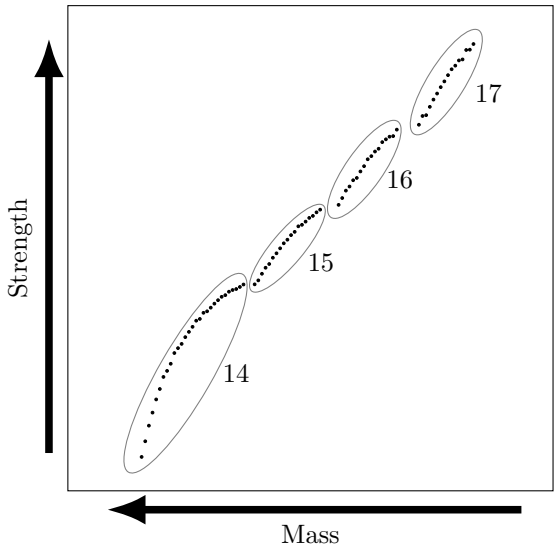


Figure 3.5: An example Pareto-optimal set for a multi-objective composite laminate problem. Arrows indicate direction of optimization, numbers indicate the number of metavariables in each solution.

Care should be taken such that the size of the Pareto-optimal set does not become so large that the computational resources are stretched too thin. In our previous work we proposed limiting the selection of solutions to only those that are near the best-so-far length [118].

3.2.4.2 Parsimony Pressure

Parsimony pressure is applied by modifying the fitness function to penalize for solution length. This can prevent bloat from occurring—any increase in length must be accompanied by an increase in fitness to offset the parsimony penalty. Including this penalty distorts the entire fitness landscape, so care should be taken when determining the weight of the penalty term.

As mentioned in Section 3.2.4, many metameric problems have objectives or constraints that are already a function of length. This is generally true for the coverage, clustering, composite laminate, wind farm, and truss problems. While the fitness functions used might be considered similar to a parsimony pressure, we have not included these studies in Table 3.3. The fitness functions for gene regulatory networks are not generally a function of length; however, no such studies in this survey were observed to apply a parsimony pressure.

Whitley et al. [142] apply an indirect method of parsimony pressure when evolving neural networks. The connection weights of each network are determined by backpropagation, less complex solutions are afforded a greater number of cycles in this process.

3.2.4.3 Constraints on Solution Length

Applying constraints to solution length is a simple method of preventing uncontrolled bloat. However, bloat may still result in populations that are at or near the maximum allowed length. Length constraints might also alleviate some issues caused by premature convergence. If the optimal length is known *a priori*, constraints can be used to restrict the population to this region of the search space.

Any metameric algorithm that uses a fixed-length genome will have an upper bound on solution length. These studies are not included in the Table 3.3 unless there is an additional constraint that further limits allowable lengths.

3.2.4.4 Niching

We use the term niching to describe selection methods that consider additional criteria, besides fitness, when forming the parent population. This is typically done to maintain a greater level of diversity in the population.

To avoid premature convergence in metameric problems, it is important to maintain a population that has solutions of several different lengths. However, very few studies use a niching step that explicitly considers length. Chang and Sim [16] ensure that the fittest solution with a length different from the best-so-far solution is selected. Chen et al. [20] select several of the fittest individuals at each solution length; a

fixed-length genome of only several metavariables is used.

Studies that use NEAT, or an adaptation of NEAT, divide the population into species based on genetic similarity [132, 130, 131, 32, 24]. The selection operator can then help ensure that one species does not come to dominate the population, allowing for diversity to be maintained. These studies do not explicitly include solution length in the measures of similarity; however, a change in solution length most likely makes it more distinct from its current species. Lee [81] uses speciation to restrict potential mating pairs.

We have recently proposed a length niching selection operator for metameric algorithms. Each niche is composed of solutions of the same length; this operator selects solutions from several niches with lengths near to the current-best when forming the parent population. An improved local selection operator has also been proposed to maintain a higher level of diversity within each niche. These operators are discussed in more detail in Chapter 7.

It should be noted that other methods exist for increasing diversity in populations. Small tournament sizes can help maintain diversity when a tournament selection operator is used. Multi-objective EAs may introduce mechanisms to achieve a more uniform distribution of solutions on the Pareto-optimal front, such as crowding in NSGA-II [31]. Larger population sizes, or using multiple subpopulations, generally results in greater diversity. Examples of these have not been included in Table 3.3.

3.2.5 Solution Validity and Repair

As discussed in Section 2.2.2, some graph-based solutions may be invalid and cannot be meaningfully evaluated. In such cases the algorithm may have difficulty reaching the valid regions of the search space. Several methods for handling invalid solutions have been observed.

A measure of validity may be used as a constraint [30] to apply a pressure toward valid solutions. Lohn and Colombano [84] propose a representation for analog electronic circuits that ensures virtually all solutions remain valid; this representation is also used by Hollinger and Gwaltney [64]. Alternatively, the population might simply be initialized with valid solutions [55, 26, 77]. NEAT-based algorithms start with minimally complex, but valid, solutions [132, 130, 131, 32, 24]. Studies using a static-metavariable representation may start with fully connected, and valid, solutions [142, 148].

Crossover and mutation operators that frequently produce invalid solutions may also be detrimental to algorithm performance. It is expected that some operators, such as the similarity or spatial crossovers, will produce fewer invalid solutions compared to more destructive operators. More specialized crossover or mutation operators might be employed to help maintain solution validity [147, 132, 26, 130, 131, 77, 24, 32]. Studies may use a repair operator for invalid solutions [49, 48, 113] or simply discard such solutions [2] prior

to evaluation. However, these specialized operators utilize problem-specific heuristics.

All of the repair operator literature cited above investigates problems with graph-based solutions, however some studies of other metameric problems also employ repair operators [80, 141, 15, 87, 98, 104, 27, 52, 19]. Validity may not be a concern for these studies; these repair operators typically attempt to correct constraint violations, addressing feasibility rather than representation validity.

Several studies propose a solution simplification step [56, 5, 77]. Metavariables are combined or removed such that the new, shorter solution is functionally equivalent to the original.

3.2.6 Population Initialization

The initialization step, used to form the initial parent population, can help control bloat (see Section 3.2.4). NEAT, proposed by Stanley and Miikkulainen [132], begins with a minimally complex population of neural networks. Initially, no hidden nodes are used, minimizing the size of the initial search space. Nodes and connections can be added by the genetic operators, leading to more complex solutions. The authors of NEAT note that a minimally complex initial population will have very little topological diversity [132]. Niching, described in Section 3.2.4.4, is applied to counteract this.

Several other studies have adapted the NEAT algorithm, including its minimally complex initial solutions, for other problems. Stanley [130] uses it to produce compositional pattern producing networks which are then used as an indirect encoding for large scale neural networks [131]. Dinh et al. [32] and Cussat-Blanc et al. [24] adapt NEAT for gene regulatory networks.

Non-NEAT studies include Zebulum et al. [149], which uses a hidden-metavariable representation for electronic circuits. Only a few genes are set to be initially active. Palmes et al. [103] use a static-metavariable representation for a neural network, with all values initialized at zero, effectively removing all connections.

The initial population may also be generated in a way that ensures solution validity, as discussed in Section 3.2.5.

3.3 Summary

Metameric representations are ones in which the genome is composed of a number of metavariables, resulting in a segmented structure. This survey has identified a large number of studies, investigating a variety of optimization problems, that have been solved using such a representation. Evolutionary algorithms are frequently adapted to this representation, and despite little cross-referencing among the studies, there is a large overlap in the methodologies employed.

A fixed-length or variable-length genome can be used in a metameric representation. Fixed-length genomes require a mechanism that is able to control metavariable expression when forming the phenotype. This allows traditional crossover and mutation operators to be employed, however these may not be ideal for metameric problems. Many studies use a static-metavariable representation which incorporates a problem-specific underlying structure into the solution.

Variable-length genomes require new crossover operators; most commonly a basic cut-and-splice operator is implemented. However, this is a destructive operator due to the effectively random distribution of metavariables when forming children. Better options include spatial and similarity crossovers, which are more likely to preserve building blocks present in the parents. Spatial crossover is not applicable to all metameric problems, and similarity crossovers are relatively rare in the literature.

Most algorithms use traditional selection operators that do not consider solution length when comparing individuals. Due to the destructiveness of the length-varying operators, this may cause an algorithm to stagnate at suboptimal lengths. New selection operators need to be developed that maintain a range of solution lengths in the population in a sensible manner.

The existing overlap in methodologies suggests that more efficient, general metameric EAs could be developed. Such algorithms would require further work, primarily on the crossover and selection operators. The application of metameric EAs to problems with graph-based solutions may require additional considerations, such as allowing for the inclusion of a user-defined measure of validity or a repair operator.

The development of a general metameric EA would be a great resource for studies focused on metameric problems. This would provide a much better starting point compared to adapting a traditional fixed-length EA. More focus could then be given to the other contributions of the study.

Chapter 4

Metameric Evolutionary Algorithm and Benchmark Problems

This chapter describes the proposed metameric evolutionary algorithm (MEA), including the framework and set of default operators. The choice of these operators is justified in subsequent chapters which investigate the performance of various metameric representations and operators. Section 4.2 proposes a measure of distance between pairs of metavariables or metameric genotypes, this measure is used by certain crossover and selection operators. Section 4.3 presents several benchmark metameric problems that are used to evaluate the effectiveness of the explored representations and operators. Section 4.4 gives a normalized measure of algorithm performance.

4.1 Metameric Evolutionary Algorithm

A flowchart of the proposed MEA is shown in Figure 4.1. An initial parent population P_1 is randomly generated, the length of each solution is randomly determined from bounds provided by the user. Each generation G crossover and mutation are applied to the parent population to produce a child population Q_G, which is then evaluated. Next generation's parent population P_{G+1} is then selected from the combined child and parent population R_G. This process then repeats iteratively for a fixed number of generations G_{max}.

The proposed MEA and all benchmark evaluation functions in this study are coded in MATLAB[1]. For each study the user provides a *setup* file defining the algorithm parameters to be used. Included in this file are the function handles of the desired genetic operators. Substituting operators is accomplished by altering

[1]Source code, along with some documentation, is available at https://github.com/ryerkerk/metameric. The provided code only handles variable-length representations.

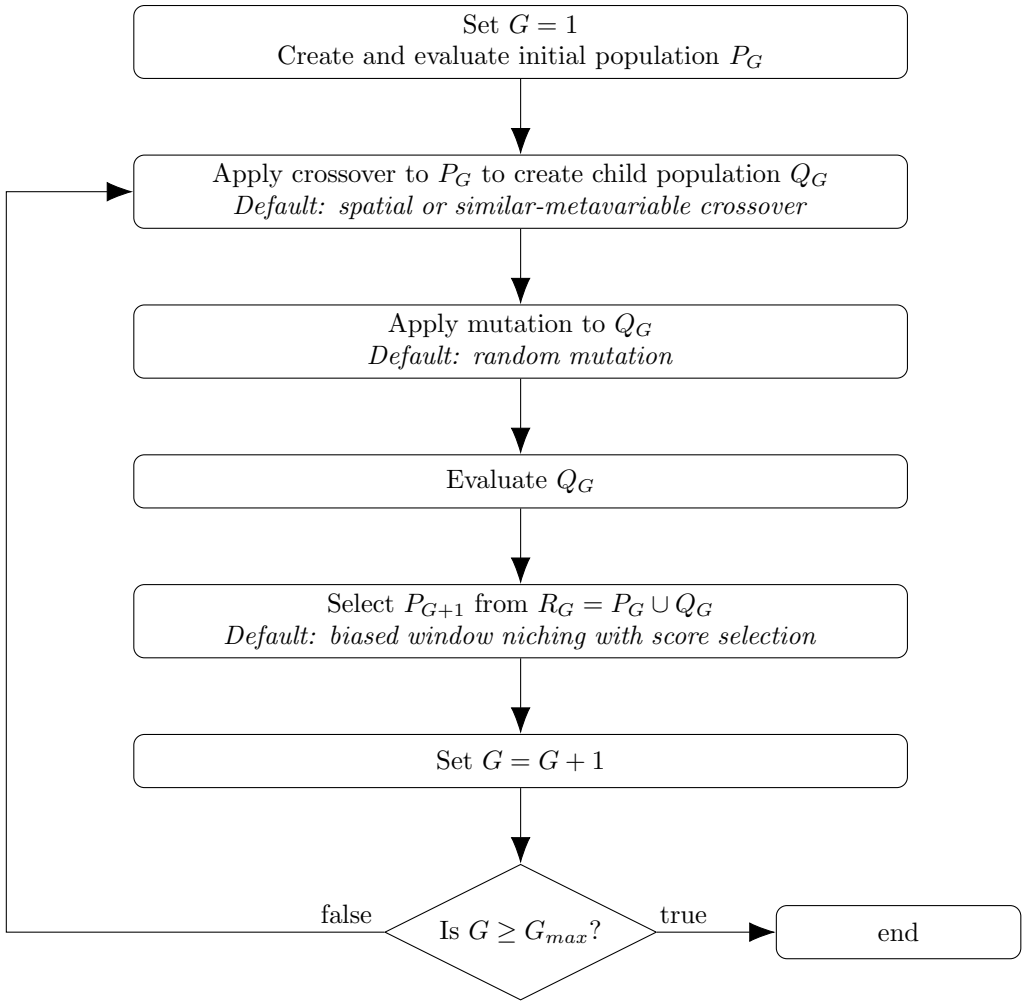


Figure 4.1: A flowchart for the proposed metameric evolutionary algorithm with the default operators.

the function handles in the setup file and requires no further changes to the remaining code.

A number of different metameric representations and operators are explored in the subsequent chapters. Each chapter compares the performance of these using the benchmark problems given in Section 4.3. It is not practical to perform a study for every possible combination of representations and operators that are proposed. Instead, a default set of operators for the MEA is defined here and listed below. The justifications for using each operator can be found in the appropriate chapters.

Representation: A variable-length genome is used. Chapter 5 compares the use of variable-length and fixed-length genomes.

Crossover: Spatial crossover is used for all problems with the exception of the portfolio problem (Section 4.3.4). Instead, the portfolio problem employs a similar-metavariable crossover. Chapter 6 compares the performance of several crossover operators.

Mutation: Only one mutation operator is demonstrated in this work, detailed in Section 6.2.

Table 4.1: Default parameters used by the metameric evolutionary algorithm.

Population size	20
Total evaluations	100,000
Minimum initial length	10
Maximum initial length	50
Crossover rate	0.80
Mutation rate	$(L(\mathbf{x}) \times v)^{-1}$
Mutation magnitude	$0.05(x_k^{UB} - x_k^{LB})$
Add metavariable rate	0.05
Remove metavariable rate	0.05
Metavariable permutation rate	0.05

Selection: Length niching is employed along with a biased window function. Local selection is performed using the score selection operator. Chapter 7 details the methodology for this operator and compares its performance to several alternatives.

Unless mentioned otherwise these operators will be used by the MEA. It is certainly possible that these operators will not always give the best results for all benchmark problems. However, compared to the other operators proposed, they are expected to give the best average performance over a range of metameric problems.

4.1.1 Design Variable Types

Sections 2.1 and 2.2 gave generic definitions for metameric representations and problems respectively. In each of these it was assumed that all design variables were real values, however the MEA is not restricted to only real variable types. Design variables can also be integer, discrete, or enumerated.

Only a few modifications are required to handle each design variable type. The mutation operator handles each type differently, details are given in Section 6.2. The metavariable distance measure functions the same for real, integer, and discrete types. Enumerated variables must be handled differently since the difference between two unequal values cannot be easily quantified, this is discussed in Section 4.2.

4.1.2 Ordered Problems

Ordered problems are ones in which a solution's fitness is affected by the relative ordering of metavariables in the genotype (Section 2.2.1). Among the benchmark metameric problems considered here, only the composite laminate problem is ordered. The layup order of the plies is determined by the sequence of ply stacks in the genotype.

The proposed MEA operators generally do not consider order. Rather than modifying the operators it is possible to modify the genotype such that their order is accounted for. This is done by adding an index

value z to each metavariable. A set of linearly spaced index values between 0 and 1 is assigned to each metavariable, as shown in Figure 4.2.

Genotype	$x_{1,1} = 2$ $x_{1,2} = 9$	$x_{2,1} = 6$ $x_{2,2} = 3$	$x_{3,1} = 1$ $x_{3,2} = 6$	$x_{4,1} = 7$ $x_{4,2} = 1$	$x_{5,1} = 7$ $x_{5,2} = 8$	$x_{6,1} = 2$ $x_{6,2} = 1$	$x_{7,1} = 3$ $x_{7,2} = 2$
Modified genotype	$x_{1,1} = 2$ $x_{1,1} = 9$ $z = 0$	$x_{2,1} = 6$ $x_{2,2} = 3$ $z = 0.17$	$x_{3,1} = 1$ $x_{3,2} = 6$ $z = 0.33$	$x_{4,1} = 7$ $x_{4,2} = 1$ $z = 0.50$	$x_{5,1} = 7$ $x_{5,2} = 8$ $z = 0.67$	$x_{6,1} = 2$ $x_{6,2} = 1$ $z = 0.83$	$x_{7,1} = 3$ $x_{7,2} = 2$ $z = 1$

Figure 4.2: An index value z is added to each metavariable for an ordered problem. This allows the proposed MEA operators to consider metavariable ordering without further modification.

By applying the MEA operators to the modified genotype the ordering of metavariables will be considered. The spatial crossover operator (6.1.3) will partition solutions based on z for the composite laminate problem. Measures of solution distance (Section 4.2) will now take order into account, affecting the functionality of certain crossover and selection operators. After crossover is performed, the child genotypes will be reordered based on the z values of their inherited metavariables. The index values are then removed from the genotypes prior to calling the next operator.

Among ordered problems the relative importance of the genotype composition and its ordering will vary. For example, if order was thought to be of critical importance then the index value could be given more weight when measuring solution distance. However, quantifying the relative importance is a difficult question and analogous to determining the relative importance of each design variable. This is an area of future work, discussed in Section 9.2.

4.2 Metavariable and Genotypic Distance

Similar-metavariable crossover (Section 6.1.4) and score selection (Section 7.3.2) rely on measures of distance between metavariables and solutions. Equation (4.1) calculates the distance between two metavariables as the average absolute difference between design variables, normalized by the range of each variable. This equation can be applied to real, integer, or discrete valued variables. Enumerated variables can only be compared to determine whether or not they have the same value. If two enumerated variables contain the same value, their normalized difference is set as 0, otherwise it is set to 1. The distance between any two metavariables will be in the range [0,1], where a distance of 0 indicates identical metavariables.

$$d(\mathbf{x}_1, \mathbf{x}_2) = \frac{1}{v} \sum_{k=1}^{v} \frac{|x_{1,k} - x_{2,k}|}{x_k^{UB} - x_k^{LB}} \tag{4.1}$$

The distance between two genotypes is given in (4.2). First, the distance between every pair of metavariables in the two solutions is calculated. For each metavariable in both parents, its most similar (i.e., lowest distance) pair is identified in the opposite parent. The distance between two parents is then calculated as the average distance between every metavariable and its most similar counterpart. The distance between two solutions will be in the range of [0,1].

$$D(\mathbf{x}^{(1)}, \mathbf{x}^{(2)}) = \frac{1}{L(\mathbf{x}^{(1)}) + L(\mathbf{x}^{(2)})} \left(\sum_{i=1}^{L(\mathbf{x}^{(1)})} \min_{1 \leq j \leq L(\mathbf{x}^{(2)})} d\left(\mathbf{x}_i^{(1)}, \mathbf{x}_j^{(2)}\right) + \sum_{j=1}^{L(\mathbf{x}^{(2)})} \min_{1 \leq i \leq L(\mathbf{x}^{(1)})} d\left(\mathbf{x}_i^{(1)}, \mathbf{x}_j^{(2)}\right) \right) \tag{4.2}$$

When hidden-metavariable representations (Section 5.2.1) are used, these measures of distance are applied to the set of expressed metavariables. The introns, or unexpressed metavariables, will not affect the calculated distance. The static-metavariable representations demonstrated in this work are considered to be black-box (Section 5.2.2). That is, the static-genotype is not considered visible to the algorithm. For these problems an alternate measure of solution distance is used where only metavariables occupying the same locus of the parent genotypes are compared. This is given in (4.3), where L is the number of metavariables in the fixed-length genome.

$$D(\mathbf{x}^{(1)}, \mathbf{x}^{(2)}) = \frac{1}{L} \sum_{i=1}^{L} \begin{cases} 0 & \text{if both } \mathbf{x}_i^{(1)} \text{ and } \mathbf{x}_i^{(2)} \text{ are unexpressed} \\ d\left(\mathbf{x}_i^{(1)}, \mathbf{x}_i^{(2)}\right) & \text{if both } \mathbf{x}_i^{(1)} \text{ and } \mathbf{x}_i^{(2)} \text{ are expressed} \\ 1 & \text{otherwise} \end{cases} \tag{4.3}$$

4.3 Benchmark Problems

Several benchmark metameric problems are used to demonstrate the effectiveness of the proposed metameric evolutionary algorithm and associated operators. Many variations of these problems exist, but the nuances of each are considered outside the scope of this paper. Only basic implementations of each problem are considered here, however the search spaces remain considerably complex.

Many of these problems are subject to a constraint. Equation (4.4) gives a measure of the total constraint violation. A solution is considered feasible if $\phi(\mathbf{x}) = 0$, otherwise it is infeasible. Note that the equality constraint has been transformed into an inequality constraint and relaxed by σ, this is a common method for handling such constraints. For the metameric benchmark problems $\sigma = 0$.

$$\phi(\mathbf{x}) = \sum_{i=1}^{p} \max(0, g_i(\mathbf{x})) + \sum_{j=1}^{q} \max(0, |h_j(\mathbf{x})| - \sigma) \tag{4.4}$$

4.3.1 Constrained and Unconstrained Coverage

The term coverage can be used to encompass several different problems. The common characteristic of these problems is their attempt to position a number of nodes such that they are able to cover a specified domain as efficiently as possible. Common examples include sensor coverage or wireless transmitter networks, such as radio or cellular networks.

Metameric representations have frequently been applied to coverage problems in literature [58, 141, 70, 15, 38, 96, 69, 134, 118]. The fitness models for these problems vary considerably, readers are referred to surveys on cellular planning [85] and wireless sensor networks [1, 37] for additional information.

Two coverage problems are used as metameric benchmarks: a constrained and an unconstrained formulation. Both problems use relatively simple fitness models where each node covers a circular area. A 2x2 square domain is to be covered. Each metavariable represents a single node defined by an x-position, y-position, and radius. Cost is measured as the sum of all node radii in the solution. Large sensors are more expensive but also more cost efficient. Coverage is approximated using a square point lattice covering the domain, with a spacing of 0.01 between points. The calculated value will be in the range of $[0, 1]$, where 1 would indicate a fully covered domain.

The constrained version of this problem seeks to cover at least 98% of the domain while minimizing cost. The coverage requirement is enforced through a constraint. Equation (4.5) gives the optimization statement for this problem. Figure 4.3 shows a sample solution to this problem. The best known solutions use 25 nodes with a total cost of approximately 6.03.

$$\begin{array}{lll} \text{Minimize} & f(\mathbf{x}) = \sum_{n=1}^{L(\mathbf{x})} x_{n,3} & \mathbf{x} \in \mathbb{R}^{L(\mathbf{x})\times v}, L(\mathbf{x}) \in \mathbb{Z}^+ \\ \text{Subject to} & \text{coverage}(\mathbf{x}) \geq 0.98 & \\ \text{(x-position)} & 0 \leq x_{n,1} \leq 2 & \\ \text{(y-position)} & 0 \leq x_{n,2} \leq 2 & n = 1, 2, ..., L(\mathbf{x}) \\ \text{(radius)} & 0.10 \leq x_{n,3} \leq 0.25 & \end{array} \tag{4.5}$$

The unconstrained coverage uses a weighted sum of cost and uncovered area as the objective function. The weights used here result in a strong pressure toward solutions that cover most ($> 99\%$) of the domain. The best obtained solution uses 27 nodes to cover 99.65% of the domain, achieving an objective function value of 6.811. Equation (4.6) gives the optimization statement.

$$\begin{array}{lll} \text{Minimize} & f(\mathbf{x}) = \sum_{n=1}^{L(\mathbf{x})} x_{n,3} + 50 * (1 - \text{coverage}(\mathbf{x})) & \mathbf{x} \in \mathbb{R}^{L(\mathbf{x})\times v}, L(\mathbf{x}) \in \mathbb{Z}^+ \\ \text{Subject to} & \text{(unconstrained)} & \\ \text{(x-position)} & 0 \leq x_{n,1} \leq 2 & \\ \text{(y-position)} & 0 \leq x_{n,2} \leq 2 & n = 1, 2, ..., L(\mathbf{x}) \\ \text{(radius)} & 0.10 \leq x_{n,3} \leq 0.25 & \end{array} \tag{4.6}$$

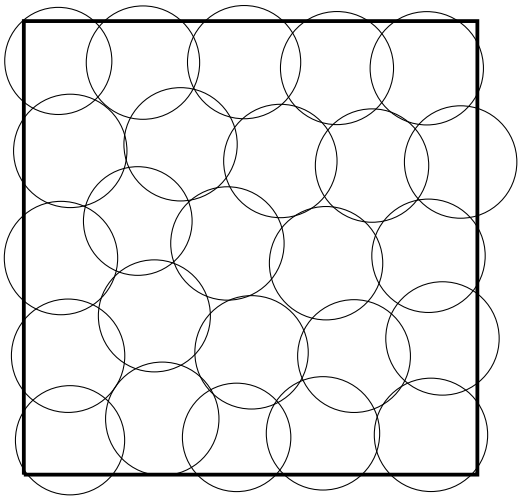


Figure 4.3: Sample solution to the constrained coverage problem. Solution uses 25 nodes, with a total cost of 6.198, to cover 98% of the domain.

4.3.2 Packing

There are many practical applications for packing problems. For example, minimizing unused space in storage or shipping, or minimizing waste when cutting stock material into smaller pieces. Optimization algorithms for such problems may employ a constructive heuristic to position the objects. When dissimilar objects are packed, the order in which they are placed by the constructive heuristic will affect the result. In this case, the best packing may be found by optimizing the order of objects in the genome [62].

This benchmark problem seeks to pack circular objects into a 2x2 square domain without the use of any constructive or other problem-specific heuristics. Each metavariable defines the position and size of a circle, either large with a radius of 0.25 or small with a radius of 0.15. The objective function scores solutions based on the number of large and small circles used. Large circles are considered worth 3 points, and small circles worth 1. The total overlap of a solution is the sum of all overlaps between two circles and the overlap between a circle and the region outside the square domain. The optimization statement is given in (4.7).

The best solutions found in this study had a score of 46, an example of which is shown in Figure 4.4. This score was achieved by solutions using 16, 18, and 20 circles. Scores of 45 were also observed at solution lengths of 15, 17, 19, and 21. For the given domain, a score of 48 could be achieved by arranging 16 large circles in a 4x4 array. However, the position of each circle would have to be exact such that no overlap exists. It is extremely unlikely that such a solution could be achieved through the Gaussian mutation operator used in this work. The inclusion of a constructive heuristic in the evaluation function could allow for such solutions.

$$
\begin{array}{lll}
\text{Minimize} & f(\mathbf{x}) = \text{score}(\mathbf{x}) & \mathbf{x} \in \mathbb{R}^{L(\mathbf{x}) \times v}, L(\mathbf{x}) \in \mathbb{Z}^{+} \\
\text{Subject to} & \text{total overlap}(\mathbf{x}) = 0 & \\
\text{(x-position)} & 0 \leq x_{n,1} \leq 2 & \\
\text{(y-position)} & 0 \leq x_{n,2} \leq 2 & n = 1, 2, ..., L(\mathbf{x}) \\
\text{(radius)} & x_{n,3} \in \{0.15, 0.25\} &
\end{array}
\tag{4.7}
$$

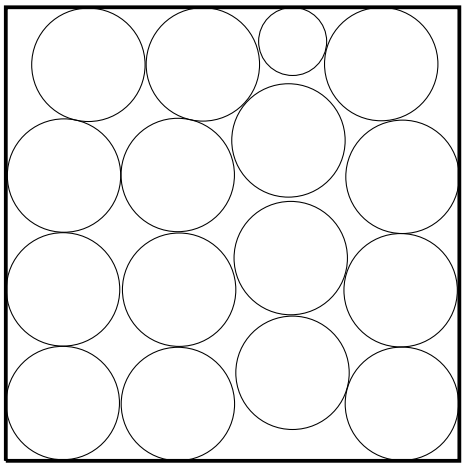


Figure 4.4: Sample solution to the packing problem. The domain contains 16 circles, achieving a score of 46, with no overlap.

4.3.3 Wind farm

Wind farm design focuses on the placement of wind turbines on a specified site with a defined wind profile. The power produced by an individual turbine is a function of the local wind speed. Each turbine creates a downstream wake which reduces the power produced by any turbine on which it impinges. The overall efficiency of the wind farm can be improved by limiting the downstream interactions between turbines. A minimum spacing distance between turbines must be considered. Equation (4.8) calculates the sum of all spacing violations, a value of 0 indicates a feasible solution.

$$
\text{spacing violations}(\mathbf{x}) = \sum_{i=1}^{L(\mathbf{x})} \sum_{j=i+1}^{L(\mathbf{x})} \max\left(0, 200 - \sqrt{(x_{i,1} - x_{j,1})^2 + (x_{i,2} - x_{j,2})^2}\right) \tag{4.8}
$$

Metameric representations have been used for wind farm design [99, 54, 98, 35, 126, 52, 19, 118]. These studies frequently partition the wind farm domain into a number of rectangular elements *a priori*; potential turbine locations are then limited to the center of each element. Chapter 5 demonstrates this type of encoding. Several reviews of the wind farm design problem are available [75, 61, 53].

This benchmark uses continuous design variables, allowing turbines to be placed at any point in the domain. The turbine data and wake model used in this paper are the same as those used by Mosetti et al. [99], and Grady et al. [54]. These are based on a turbine wake decay model developed by Jensen [67, 43]. Readers are referred to these papers for the methodology of calculating the power produced by a wind farm.

This study uses the environmental wind data from the second case considered by Grady et al. [54], which assumes a uniform wind speed of 12 m/s occurring at an equal rate from all directions. Turbines are placed on a 2000 m∗2000 m domain, however turbines must be located at least 100 m from the domain boundaries.

The cost model of the wind farm, which considers economies of scale, is given in (4.9). This model gives a non-dimensionalized cost/year value for the wind farm and was originally proposed by Mosetti et al. [99]. Solution fitness is then measured as the cost per unit power produced (cost/(kW×year)). The optimization problem statement is given in (4.10).

$$\text{cost}(\mathbf{x}) = L(\mathbf{x}) \times \left(\frac{2}{3} + \frac{1}{3} e^{-0.00174 L^2(\mathbf{x})} \right) \tag{4.9}$$

$$\begin{array}{lll} \text{Minimize} & f(\mathbf{x}) = \frac{\text{power produced}(\mathbf{x})}{\text{cost}(\mathbf{x})} & \mathbf{x} \in \mathbb{R}^{L(\mathbf{x}) \times v}, L(\mathbf{x}) \in \mathbb{Z}^+ \\ \text{Subject to} & \text{spacing violations}(\mathbf{x}) = 0 & \\ \text{(x-position)} & 100 \leq x_{n,1} \leq 1900 & \\ \text{(y-position)} & 100 \leq x_{n,2} \leq 1900 & n = 1, 2, ..., L(\mathbf{x}) \end{array} \tag{4.10}$$

A sample wind farm solution is shown in Figure 4.5. The best solution found in this work uses 40 turbines to achieve an objective function value of 1.469. It has been observed that the optimal solutions found by variable-length studies typically use between 39 and 41 turbines with little difference in performance.

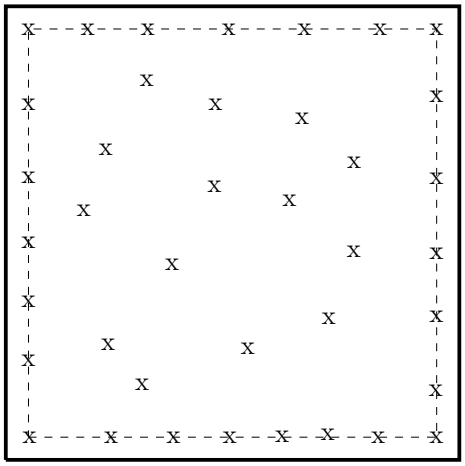


Figure 4.5: Sample solution found to the wind farm problem, using 40 turbines to achieve a fitness value of 1.484. Each turbine location is denoted by an x, the dashed box represents the closest point that turbines can be placed to the boundary.

4.3.4 Portfolio

Portfolio optimization seeks to find a set of assets that balance the expected returns and risk as desired. This benchmark was run using data available in the OR-library [12][2]. Historical weekly price data from the S&P 100 was used to create the dataset [18]. There are $N = 98$ assets (i.e., stocks) available, the expected return of each asset i is denoted by μ_i and the covariance between two assets by σ_{ij}.

[2]The data can be found in port4.txt downloaded from http://people.brunel.ac.uk/~mastjjb/jeb/orlib/portinfo.html

Each metavariable $\mathbf{x}_j$ defines an asset type $x_{j,1}$ and amount $x_{j,2}$. Asset type is an enumerated variable (see Section 4.1.1) and amount is real-valued. Let $w_i(\mathbf{x})$ represent the proportion of asset i in the portfolio, calculated by (4.11) such that $\sum w_i(\mathbf{x}) = 1$. The set of metavariables $\mathbf{y}^{(i)}$ is formed by all metavariables in $\mathbf{x}$ that define asset i. If this asset is not represented by any metavariables in $\mathbf{x}$, then $w_i(\mathbf{x}) = 0$.

$$w_i(\mathbf{x}) = \frac{\sum\limits_{\mathbf{y}_j \in \mathbf{y}^{(i)}} y_{j,2}}{\sum\limits_{\mathbf{x}_j \in \mathbf{x}} x_{j,2}} \qquad \text{where } \mathbf{y}^{(i)} = \{\mathbf{x}_j \in \mathbf{x} : x_{j,1} = i\} \tag{4.11}$$

The expected return and risk (i.e., variance) of a particular portfolio can be calculated by (4.12) and (4.13) respectively [18]. This work assumes a transaction cost of T = 2e-5 for each asset. It is possible for two metavariables in a solution to define the same asset type, in this case both metavariables are considered as separate transactions. Efficient solutions will only use one metavariable for each asset type.

$$\text{return}(\mathbf{x}) = -TL(\mathbf{x}) + \sum_{i=1}^{N} w_i(\mathbf{x})\mu_i \tag{4.12}$$

$$\text{risk}(\mathbf{x}) = \sum_{i=1}^{N}\sum_{j=1}^{N} w_i(\mathbf{x})w_j(\mathbf{x})\sigma_{ij} \tag{4.13}$$

The full optimization statement is given in (4.14). The best portfolio obtained contains 9 assets and has an expected weekly return of 6.533e-3.

$$\begin{array}{lll} \text{Maximize} & f(\mathbf{x}) = \text{return}(\mathbf{x}) & \mathbf{x} \in \mathbb{Z}^{L(\mathbf{x})} \times \mathbb{R}^{L(\mathbf{x})},\ L(\mathbf{x}) \in \mathbb{Z}^+ \\ \text{Subject to} & \text{risk}(\mathbf{x}) \leq \text{5e-4} & \\ \text{(asset type)} & x_{n,1} \in \{1, 2, ..., N\} & \text{n = 1, 2, ..., L}(\mathbf{x}) \\ \text{(asset amount)} & 0 \leq x_{n,2} \leq 1 & \end{array} \tag{4.14}$$

4.3.5 Composite Laminate

The benchmark composite laminate problem considered here optimizes a single panel made up of a number of plies. The objective is to minimize the mass of the laminate with a load factor λ of at least 5 for the given geometry and load conditions. The optimization problem statement is given in (4.15). Each metavariable encodes a fiber orientation and fiber volume fraction (i.e., the amount of fiber in each ply). This is an ordered problem, the layup order of the plies is determined by the order of metavariables in the genotype. The best obtained solution uses 17 ply stacks (i.e., metavariables) to define a laminate with a mass of 0.5258 kg.

$$\begin{array}{lll} \text{Minimize} & f(\mathbf{x}) = \text{mass}(\mathbf{x}) & \mathbf{x} \in \mathbb{Z}^{L(\mathbf{x})} \times \mathbb{R}^{L(\mathbf{x})},\ L(\mathbf{x}) \in \mathbb{Z}^+ \\ \text{Subject to} & \lambda(\mathbf{x}) \geq 5 & \\ \text{(fiber orientation)} & x_{n,1} \in \{1, 2, 3, 4, 5, 6, 7\} & \text{n = 1, 2, ..., L}(\mathbf{x}) \\ \text{(fiber volume fraction)} & x_{n,2} \in \{0.3, 0.4, 0.5, 0.6, 0.7\} & \end{array} \tag{4.15}$$

Fiber orientations are restricted to angles corresponding to 15° increments, starting with 0°. To ensure a balanced design, each integer is expressed as a $\pm\theta$ pair of plies. Specifically, the values {1, 2, 3, 4, 5, 6, 7} are interpreted as $\{0^\circ_2, \pm 15^\circ, \pm 30^\circ, \pm 45^\circ, \pm 60^\circ, \pm 75^\circ, 90^\circ_2\}$ ply stacks. To ensure symmetry, only the top

half of the laminate is encoded in the genotype; the bottom half is formed by mirroring the top half over the mid-plane. For example, the laminate $[\pm 75°, \pm 30°, 0°]_s$ is encoded by the fiber orientation values [6 3 1]. As a result, each metavariable encodes a stack of 4 plies in the phenotype.

Fiber volume fraction is defined using a discrete variable, its permissible values are {0.3, 0.4, 0.5, 0.6, 0.7}. Increasing fiber volume fraction results in a stiffer, but denser, ply. All 4 plies encoded by a single metavariable will have the same volume fraction.

Analysis of each laminate design is performed using Classical Laminate Theory [25]. The laminate is considered to be simply supported on all four edges. The global coordinate system is denoted by (x,y,z), and the material coordinates by (1,2,3), as shown in Figure 4.6. The relationship between the force/moment resultants and laminate strain/curvature responses are given in (4.16).

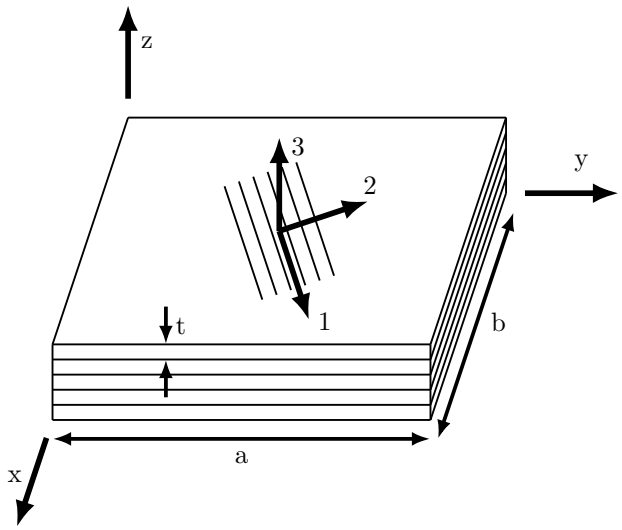


Figure 4.6: Simply supported laminate. (x,y,z) denote global coordinates, (1,2,3) denote material coordinates.

$$\begin{bmatrix} \mathbf{N} \\ \mathbf{M} \end{bmatrix} = \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{B} & \mathbf{D} \end{bmatrix} \begin{bmatrix} \epsilon^0 \\ \kappa \end{bmatrix} \tag{4.16}$$

The vector $\mathbf{N}$ contains the normal and shear forces per unit length (N_x, N_y, N_{xy}), and the vector $\mathbf{M}$ contains the bending and twisting moments per unit length (M_x, M_y, M_{xy}). Only in-plane normal and shear force loads are considered in this work. $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{D}$ are each 3×3 matrices containing the extensional, coupling, and bending stiffnesses, respectively. As our laminates in this paper are symmetric, the coupling stiffness matrix $\mathbf{B}$ will always be zero. The vector ϵ^0 contains the in-plane mid-surface strains $(\epsilon_x^0, \epsilon_y^0, \gamma_{xy}^0)$ and the vector κ contains the curvatures of the laminate $(\kappa_x, \kappa_y, \kappa_{xy})$.

The material properties, obtained from [25], and loading conditions used in this paper are shown in Table 4.2. The stiffness of each ply will depend on its fiber volume fraction.

The ability of a laminate to support a load is measured by its load factor λ. For a baseline compressive

Table 4.2: Composite laminate plate geometry, material properties, and loading conditions.

Carbon fibers (AS-4)		Epoxy (3501-6)		Maximum allowed strain	
E_{1f}	235 GPa	E_{1m}	4.3 GPa	ϵ_1^a	0.0117
E_{2f}	15 GPa	E_{2m}	4.3 GPa	ϵ_2^a	0.0221
G_f	27 GPa	G_m	1.6 GPa	γ_{12}^a	0.0109
ν_f	0.20	ν_m	0.35		
ρ_f	1.810 g/m^3	ρ_m	1.270 g/m^3		
Plate dimensions		Loads			
a	500 mm	N_x	900 kN/m		
b	100 mm	N_y	450 kN/m		
t (per ply)	0.1 mm	N_{xy}, M_x, M_y, M_{xy}	0		

in-plane loading (N_x, N_y, N_{xy}), the load factor is the largest scaling factor that can be applied to the loads ($\lambda N_x, \lambda N_y, \lambda N_{xy}$) without the laminate failing. Therefore, a load factor greater than 1 indicates the laminate can support the baseline load. A laminate can fail either through compressive material failure or buckling. Each mode of failure has its own load factor, λ_b for failure by buckling and λ_m for material failure. The load factor for the laminate is the minimum of these two values.

To determine λ_m, the strain response calculated by (4.16) is used to find the in-plane material strains ($\epsilon_x^0, \epsilon_y^0, \gamma_{xy}^0$) for each ply. The material load factor λ_m is the smallest value that results in at least one ply failing in at least one direction. The allowable strains ($\epsilon_1^a, \epsilon_2^a, \gamma_{12}^a$) are listed in Table 4.2. All plies assume the same maximum allowed strains regardless of fiber volume fraction.

The buckling load factor λ_b is approximated by (4.17) [111, 139]. When the laminate buckles, it will do so in m and n half-waves in the x and y directions, respectively. The buckling load factor is determined by the combination of m and n half-waves that produces the minimum factor.

$$\lambda_b = \min_{m,n} \left(\pi^2 \frac{D_{xx} \left(\frac{m}{a}\right)^4 + 2\left(D_{xy} + 2D_{ss}\right)\left(\frac{m}{a}\right)^2 \left(\frac{n}{b}\right)^2 + D_{yy}\left(\frac{n}{b}\right)^4}{N_x \left(\frac{m}{a}\right)^2 + N_y \left(\frac{n}{b}\right)^2 + N_{xy}\left(\frac{mn}{ab}\right)} \right) \tag{4.17}$$

4.4 Normalized Performance

When presenting results in subsequent chapters a normalized measure of performance is used. Each algorithm demonstrated is run for 200 independent trials. The best solution from each trial is identified as the feasible solution with the best objective function value, or if no feasible solutions exist then the solution with the smallest total constraint violation. The average objective value of the best solution from each trial is denoted as f_{avg}.

The normalized performance f_{norm} of each algorithm is calculated using (4.18) at several points throughout the study. The baseline objective value $f_{baseline}$ is equal to the final f_{avg} of the metameric evolutionary algorithm using the default operators described in Section 4.1. Table 4.3 gives $f_{baseline}$ for each problem.

Table 4.3: Baseline objective values $f_{baseline}$ used to normalize algorithm performance for each problem.

Constrained coverage	6.1802
Unconstrained coverage	7.0143
Wind farm	1.4783 $\frac{\text{cost}}{\text{kW}\times\text{year}}$
Packing	42.870
Portfolio	6.5311e-3
Composite laminate	0.5259 kg

Normalized performance is calculated differently depending on whether the problem is minimizing or maximizing the objective value. This is done such that higher f_{norm} values always indicate better performance, regardless of the optimization problem statement. The metameric evolutionary algorithm with the default operators will always have $f_{norm} = 1$ at the final generation. Note that, due to faster rates of convergence, only results for the first 50k function evaluations are shown for the composite laminate and portfolio problems. However, $f_{baseline}$ is still based on the results after 100k evaluations. As a result the final f_{norm} values shown for these problems may be below 1.

$$f_{norm} = \begin{cases} 1 - \dfrac{f_{avg} - f_{baseline}}{|f_{baseline}|} & \text{for minimization problems} \\ 1 + \dfrac{f_{avg} - f_{baseline}}{|f_{baseline}|} & \text{for maximizaton problems} \end{cases} \tag{4.18}$$

Note that the normalized performance does not consider any constraints. The default operators were able to find feasible solutions in every trial, but some trials of other algorithms fail to find feasible solutions. When this occurs, the portion of trials with feasible solutions is provided alongside f_{norm}. Algorithms that fail to find a feasible solution in every trial are considered to have a worse performance compared to the baseline algorithm, regardless of the f_{norm} values.

Chapter 5

Genome Encoding

Section 2.1 defines a metameric representation as one where the genome is at least partially segmented into a number of metavariables. The representation must also allow for changes to solution length during optimization. Both *variable-length* and *fixed-length* genomes are able to satisfy this definition of metameric. Chapter 3 found that both genome types were frequently used to solve metameric problems in literature.

This chapter explores the differences between these representations. Variable-length genomes are discussed in Section 5.1. Section 5.2 introduces fixed-length genomes, these can be categorized as either using a hidden-metavariable or static-metavariable representation, discussed in Sections 5.2.1 and 5.2.2 respectively. The performance of these representations is then compared in Section 5.3. Variable-length genomes are generally found to give best performance due to the flexibility they afford the variational operators.

Earlier work comparing variable-length genomes to the hidden-metavariable representation appears in an article published in Genetic Programming and Evolvable Machines: *Solving metameric variable-length optimization problems using genetic algorithms* [118].

5.1 Variable-Length Genomes

If the representation allows for metavariables to be wholly added or removed from the genotype, it is said to be a variable-length genome. Typically all metavariables in the genotype will be used to form the phenotype. Several examples of variable-length genotypes are given in Figure 5.1.

Variable-length genomes are preferred in this work due to the flexibility provided when implementing variational operators. Children can be formed from any combination of the metavariables in the parent solutions. This allows the crossover operator to be unrestricted in how it partitions parent solutions. Various crossover operators are detailed and compared in Chapter 6. Fixed-length genomes allow much less flexibility

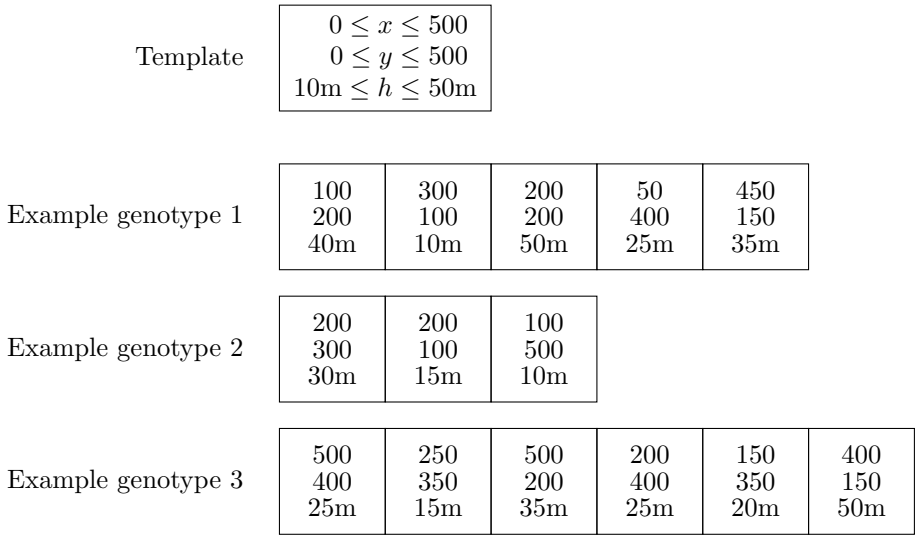


Figure 5.1: Examples of variable-length genotypes for a given template.

in exchanging metavariables, as discussed in the following sections.

For ordered problems, the arrangement of metavariables in the phenotype is typically the same as in the genotype. In this case, the variational operators should take the relative ordering of metavariables into account. One method for doing so is discussed in Section 4.1.2. For non-ordered problems, the fitness of the solution will remain the same for any permutation of the genotype.

5.2 Fixed-Length Genomes

A fixed-length genome is one that contains a fixed number of metavariables, but allows for certain metavariables to remain unexpressed in the phenotype. This results in an indirect encoding since the genotype does not map directly to the phenotype. The genotype-phenotype mapping must determine which metavariables in the genotype will be expressed. A number of mechanisms have been used in literature to control metavariable expression, examples of these are given in Section 3.2.2. Oftentimes the expressed metavariables will be copied unaltered from the genotype to the phenotype, a static-metavariable representation will also insert additional underlying structural information. More complex mappings are also possible (Section 2.1.2).

Solution length is determined by the number of expressed metavariables in the genotype. The term *introns* is frequently used, when applicable, in evolutionary computing to describe the parts of the genotype that are unexpressed in the phenotype. While the introns do not affect the phenotype, it is possible that they contain useful information. Introns in a parent solution may become expressed in child solutions through the mutation operator (Section 6.2).

Fixed-length genomes allow for the application of traditional, fixed-length optimization algorithms to

metameric representations. However, the effectiveness of the algorithm may be negatively impacted if it does not account for the representation. For example, it is possible that traditional variational operators may produce children whose only differences from their parents are found in the introns. Since the phenotypes of the children are identical to the parents, it may be wasteful to evaluate them.

There are two types of metameric representations that use a fixed-length genome: hidden-metavariable and static-metavariable representations. The hidden-metavariable representation is a generic implementation, while the static-metavariable representation utilizes problem-specific information when mapping from the genotype to phenotype.

5.2.1 Hidden-Metavariable Representation

A hidden-metavariable representation is the more generic implementation of a fixed-length genome. It requires no problem-specific information to implement, only a mechanism through which metavariables may remain unexpressed. In this chapter this is accomplished by adding a binary flag to every metavariable in the genome. Only metavariables with a flag set to 1, or *true*, will be expressed in the phenotype. An example of a hidden-metavariable representation is given in Figure 5.2.

Genotype

$x = 6$ $y = 2$ flag $= 0$	$x = 2$ $y = 5$ flag $= 1$	$x = 8$ $y = 1$ flag $= 1$	$x = 1$ $y = 7$ flag $= 0$	$x = 9$ $y = 2$ flag $= 0$	$x = 3$ $y = 2$ flag $= 0$	$x = 7$ $y = 8$ flag $= 1$

Phenotype

$x = 2$ $y = 5$	$x = 8$ $y = 1$	$x = 7$ $y = 8$

Figure 5.2: A fixed-length genome with a hidden-metavariable representation. The number of metavariables in the phenotype can be varied by changing the values of the flags in the genotype.

When using a hidden-metavariable representation, the crossover operator is generally restricted to only exchanging metavariables that occupy the same locus (e.g., position) in the genome. The n-point crossover (Section 6.1.1) can be, and frequently is, used. Other crossover operators discussed in Section 6.1 may partition solutions such that an unequal number of metavariables may be exchanged, or the exchanged metavariables may not occupy the same loci. Rather than modifying the crossover operator, allowing it to function on a hidden-metavariable representation, it is recommended to use a variable-length genome.

As with the variable-length genome, if the metameric problem is not ordered then the permutation of metavariables in the genotype will not affect the fitness. However, the arrangement of metavariables in the genotype may affect certain operators, such as n-point crossover.

5.2.2 Static-Metavariable Representation

In the static-metavariable representation, each genotype has two parts: a *static-genotype* and a *free-genotype*. Each is defined using the same number of metavariables, but contain different sets of design variables. The free-genotype contains the design variables that are subject to optimization, including those that control metavariable expression. The static-genotype encodes some underlying structural information used to form the phenotype. The design variables it contains are defined *a priori*, either by the user or by an initial computational step, and are not subject to optimization. To form the phenotype, the free-genotype is combined with the static-genotype, as demonstrated in Figure 5.3.

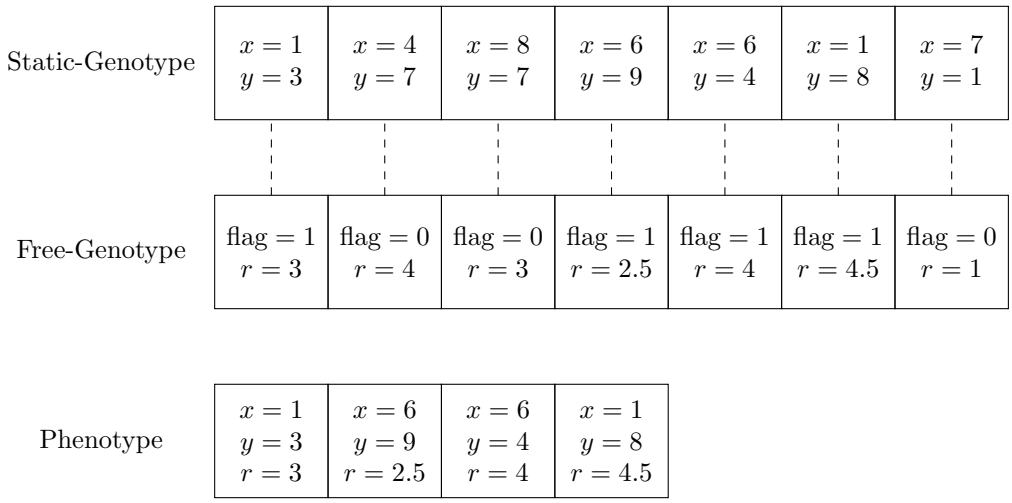


Figure 5.3: An example of the static-metavariable representation. When forming the phenotype, the free-genotype is combined with the static-genotype. In this example, metavariable expression is controlled by the binary flag in the free-genotype.

The contents of the static-genotype may or may not be visible to the optimization algorithm. If it is available, this information might be used to improve the variational and selection operators. Crossover operators that could not be used with the hidden-metavariable representation could be used with a static-metavariable representation. For example, spatial crossover could be used to partition solutions based on the static-genotype. Since all solutions share the same static-genotype, the exchanged metavariables in the free-genotype will share the same loci.

If the static-genotype is not available to the algorithm, then the representation will resemble that of the hidden-metavariable. However, despite the similarities, the two representations are not functionally equivalent. Static-metavariable is sensitive to the locus of each expressed metavariable, while hidden-metavariable is not. This difference should be accounted for by the genetic operators.

Even if the metameric problem is considered black-box (i.e., the static-genotype is not visible to the algorithm), it is usually possible to determine the type of representation in only three function evaluations.

Genotype $\mathbf{x}^{(1)}$	$x = 6$ $y = 2$ $flag = 0$	$x = 2$ $y = 5$ $flag = 1$	$x = 8$ $y = 1$ $flag = 1$	$x = 1$ $y = 7$ $flag = 0$	$x = 9$ $y = 2$ $flag = 1$	$x = 3$ $y = 2$ $flag = 0$
Genotype $\mathbf{x}^{(2)}$	$x = 2$ $y = 5$ $flag = 1$	$x = 3$ $y = 2$ $flag = 0$	$x = 8$ $y = 1$ $flag = 1$	$x = 1$ $y = 7$ $flag = 0$	$x = 6$ $y = 2$ $flag = 0$	$x = 9$ $y = 2$ $flag = 1$
Genotype $\mathbf{x}^{(3)}$	$x = 1$ $y = 7$ $flag = 0$	$x = 8$ $y = 1$ $flag = 1$	$x = 6$ $y = 2$ $flag = 0$	$x = 2$ $y = 5$ $flag = 1$	$x = 9$ $y = 2$ $flag = 1$	$x = 3$ $y = 2$ $flag = 0$

Figure 5.4: Three fixed-length genotypes that all contain different permutations of the same metavariables. This is an example of a set of genotypes that, once evaluated, can usually determine the type of fixed-length representation used.

Consider the three genotypes that have the same set of metavariables, but in different permutations. Genotypes $\mathbf{x}^{(1)}$ and $\mathbf{x}^{(2)}$ maintain the same relative ordering of expressed metavariables, while $\mathbf{x}^{(3)}$ does not. An example of this is in Figure 5.4. The following cases may be used to determine the type of representation being used.

1. If any objective or constraint values are different for $\mathbf{x}^{(1)}$ and $\mathbf{x}^{(2)}$ then the problem uses a static-metavariable representation.

2. If the above case is not true, but any differences in performance exist for $\mathbf{x}^{(1)}$ and $\mathbf{x}^{(3)}$, then the problem is ordered and uses a hidden-metavariable representation.

3. If both of the above cases are not true then the problem is non-ordered and uses a hidden-metavariable representation.

It should be noted that it is possible two different solutions map to the same objective and constraint values. For example, there is a chance $\mathbf{x}^{(1)}$ and $\mathbf{x}^{(2)}$ map to the same performance values even for a static-metavariable representation. If the first case is false that doesn't necessarily mean the problem does not use a static-metavariable representation. However, any example where the first case is true is sufficient to determine that a static-metavariable representation is used. Similar examples may exist for the second case above. As a result, the three function evaluations may not always be sufficient.

Static-metavariable representations might be used when there are a limited number of permissible components in the solution—for example, there may be a limited number of acceptable sites to place a cell phone tower. The acceptable sites could be encoded into the static-genotype and the free-genotype would then control which sites contain a tower. Alternatively, this representation may be used to reduce the size

of the search space if the general structure of an optimal solution is known *a priori*. This information could be included in the static-genotype, rather than requiring the algorithm to discover this structure. Section 3.2.2.2 discusses a number of examples of a static-metavariable representation found in literature. Forming the static-genotype does require problem-specific heuristics—the static-genotype used for one problem is most likely not applicable to most other metameric problems.

5.2.2.1 Static-Metavariable Benchmark Implementations

A static-metavariable representation is demonstrated for several of the benchmark problems. For each problem some, or all, of the design variables normally subject to optimization will be placed into the static-genotype. Whether or not each metavariable is expressed is controlled by a binary flag. The objectives and constraints are the same as those presented in Section 4.3. Due to the reduced number of design variables in these problems, the default mutation operator parameters were found to be insufficient to explore the design space. To improve performance, the rates of metavariable addition, deletion, and permutation mutations (Section 6.2) were increased to 0.20 when using a static-metavariable representation.

For certain coverage problems the nodes may be randomly deployed on the domain (e.g., from an airdrop). The optimization problem may then be to determine the subset of nodes that can be activated to provide efficient coverage while minimizing energy usage of the system [68]. Only the unconstrained coverage problem is considered here. The static-genotype defines the x- and y-position of each node, these are randomly determined at the start of the algorithm. Several studies are performed using 50, 100, or 200 randomly placed nodes. The distribution of nodes will be the same for all studies using a particular number of nodes. The distribution when considering 100 nodes is shown in Figure 5.5. Each metavariable in the free-genotype contains an expression flag and a node radius.

Literature solving the wind farm problem using a metameric representation frequently use a grid to partition the domain. Turbine locations are then restricted to the center of each grid. This is a static-metavariable representation, where the static-genotype contains the allowed turbine locations. In this study, the free-genotype contains only the binary expression flag, a *true* value indicates the presence of a turbine at the associated position. Several partitions are demonstrated, the domain is divided into a 10x10, 20x20, or 40x40 grid. This results in genotypes of lengths 100, 400, and 1600 respectively. The grid is formed such that the minimum and maximum x- and y-positions are 100 and 1900 respectively, matching the bounds defined in Section 4.3.3. The 10x10 grid and its permissible turbine locations are shown in Figure 5.6

In the portfolio problem each metavariable defines an asset type and the amount of that asset present in the portfolio. There are 98 potential assets in the data used for the benchmark problem. In the static-metavariable representation of this problem the static-genotype defines the asset type. A genome of 98

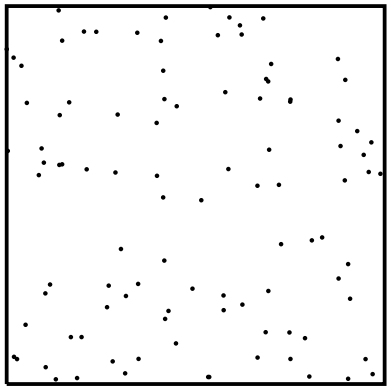


Figure 5.5: Each '.' represents a potential node location for the unconstrained coverage problem when using the proposed static-metavariable representation.

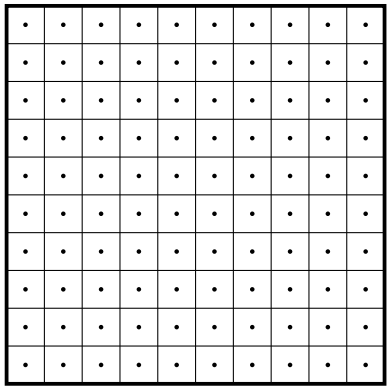


Figure 5.6: Each '.' represents a potential turbine location for the wind farm problem when using the proposed static-metavariable representation.

metavariables is used, with one metavariable for each asset. Each metavariable in the free-genotype contains a binary expression flag and the asset amount.

Static-metavariable representation is not applied to the constrained coverage, packing, or laminate composite problems. The first two problems could potentially use the same representation as the unconstrained coverage example. It is unclear how a static-metavariable representation could be applied to the laminate composite problem.

5.3 Results

Representations using a variable-length and a fixed-length genome are applied to each benchmark problem. Hidden-metavariable representation is demonstrated on each problem using several different genome lengths L. Static-metavariable representations are only applied to the unconstrained coverage, wind farm, and portfolio problems. Several different static-genotypes are tested for the unconstrained coverage and wind farm problems; these are distinguished by their genotype length L. This length is equal to the number of

permissible node or turbine locations.

When using a hidden-metavariable or static-metavariable representation, an n-point crossover is used (Section 6.1.1). The mutation operator takes the fixed-length genome into account (Section 6.2). Additionally, certain mutation rates are increased for the static-metavariable representation, as mentioned in Section 5.2.2.1. Length niching with the biased window function and score selection is applied to all representations. Section 4.2 discusses how the measures of distance between solutions handle the fixed-length representations. In this work, static-metavariable representations are considered to be black-box (i.e., the static-genotype is not visible to the crossover or selection operators).

The results for each problem are given in Tables 5.1-5.6. The best results, and those not significantly different ($p > 0.05$) from the best, for each generation are shown in bold. Algorithms using a variable-length genome outperform those using a hidden-metavariable representation for all problems. The difference in performance is most notable during the earlier generations. Hidden-metavariable representation using longer genomes perform slightly better than those using shorter genomes. The static-metavariable representation performs poorly for the coverage and wind farm problems; the representations using longer genomes (i.e., a greater number of permissible node or turbine locations) give better results. The static-metavariable representation gives slightly better results than the variable-length genome for the portfolio problem.

5.4 Discussion

The difference in performance between using a variable-length genome and hidden-metavariable representation is primarily due to the crossovers applied. All fixed-length genomes in this study use an n-point crossover (Section 6.1.1). Metavariables can only be exchanged if they occupy the same locus in both parents. This is very restrictive and the exchanged metavariables may have little to no commonalities. Variable-length genomes allow much more flexibility to the crossover operator. Spatial crossover (Section 6.1.3) is used by the variable-length genomes in this chapter for all except the portfolio problem.

The poor performance of the static-metavariable representations is unsurprising for the unconstrained coverage and wind farm problems. Restricting the locations of the nodes and turbines dramatically decreases the size of the search space. The new search spaces are unlikely to contain the optimal solutions that are found using the variable-length or hidden-metavariable representations. When applied to the portfolio problem, the static-metavariable representation performs slightly better than the variable-length genome. For this problem, the static-metavariable representation does not restrict the search space: any optimal solution can be represented by both the variable-length genome or static-metavariable representation.

Table 5.1: Normalized performance for the constrained coverage problem using different representations.

	Fitness function evaluations		
	20k	50k	100k
Variable-length genome	**0.7850 ± 0.0429**	**0.9749 ± 0.0157**	**1.0000 ± 0.0129**
Hidden-metavariable (L=30)	(0.99) 0.9092 ± 0.0186	0.9622 ± 0.0190	0.9847 ± 0.0191
Hidden-metavariable (L=60)	0.6982 ± 0.0484	0.9107 ± 0.0271	0.9850 ± 0.0167
Hidden-metavariable (L=100)	0.6516 ± 0.0728	0.9010 ± 0.0407	0.9842 ± 0.0189
Hidden-metavariable (L=200)	0.6549 ± 0.0670	0.9176 ± 0.0304	0.9893 ± 0.0167

Table 5.2: Normalized performance for the unconstrained coverage problem using different representations.

	Fitness function evaluations		
	20k	50k	100k
Variable-length genome	**0.9138 ± 0.0231**	**0.9883 ± 0.0135**	**1.0000 ± 0.0114**
Hidden-metavariable (L=30)	0.7582 ± 0.0639	0.9561 ± 0.0206	0.9866 ± 0.0152
Hidden-metavariable (L=60)	0.8056 ± 0.0581	0.9580 ± 0.0216	0.9845 ± 0.0159
Hidden-metavariable (L=100)	0.8197 ± 0.0450	0.9660 ± 0.0213	0.9895 ± 0.0176
Hidden-metavariable (L=200)	0.8359 ± 0.0424	0.9688 ± 0.0181	0.9894 ± 0.0140
Static-metavariable (L=50)	0.1088 ± 0.0074	0.1172 ± 0.0021	0.1178 ± 0.0018
Static-metavariable (L=100)	0.5954 ± 0.0369	0.6478 ± 0.0250	0.6605 ± 0.0183
Static-metavariable (L=200)	0.6923 ± 0.0333	0.7523 ± 0.0251	0.7706 ± 0.0239

Table 5.3: Normalized performance for the packing problem using different representations.

	Fitness function evaluations		
	20k	50k	100k
Variable-length genome	**0.8792 ± 0.0311**	**0.9607 ± 0.0256**	**1.0000 ± 0.0260**
Hidden-metavariable (L=20)	0.8101 ± 0.0343	0.9086 ± 0.0298	0.9584 ± 0.0259
Hidden-metavariable (L=40)	0.8246 ± 0.0342	0.9250 ± 0.0324	0.9684 ± 0.0282
Hidden-metavariable (L=60)	0.8380 ± 0.0336	0.9328 ± 0.0302	0.9795 ± 0.0271
Hidden-metavariable (L=100)	0.8366 ± 0.0348	0.9350 ± 0.0289	0.9819 ± 0.0256

Table 5.4: Normalized performance for the wind farm problem using different representations.

	Fitness function evaluations		
	20k	50k	100k
Variable-length genome	**0.9890 ± 0.0025**	**0.9976 ± 0.0018**	**1.0000 ± 0.0015**
Hidden-metavariable (L=50)	0.9768 ± 0.0041	0.9939 ± 0.0022	0.9976 ± 0.0018
Hidden-metavariable (L=70)	0.9785 ± 0.0040	0.9942 ± 0.0022	0.9979 ± 0.0018
Hidden-metavariable (L=100)	0.9791 ± 0.0037	0.9948 ± 0.0023	0.9985 ± 0.0018
Hidden-metavariable (L=200)	0.9805 ± 0.0035	0.9954 ± 0.0020	0.9988 ± 0.0018
Static-metavariable (L=100)	0.9640 ± 0.0009	0.9642 ± 0.0008	0.9642 ± 0.0008
Static-metavariable (L=400)	0.9740 ± 0.0030	0.9822 ± 0.0039	0.9846 ± 0.0041
Static-metavariable (L=1600)	0.9742 ± 0.0041	0.9893 ± 0.0048	0.9957 ± 0.0049

Table 5.5: Normalized performance for the portfolio problem using different representations.

	Fitness function evaluations		
	10k	25k	50k
Variable-length genome	**0.9846 ± 0.0063**	0.9973 ± 0.0015	0.9996 ± 0.0006
Hidden-metavariable (L=10)	0.9823 ± 0.0068	0.9960 ± 0.0022	0.9983 ± 0.0011
Hidden-metavariable (L=20)	0.9794 ± 0.0080	0.9957 ± 0.0022	0.9984 ± 0.0013
Hidden-metavariable (L=40)	0.9744 ± 0.0143	0.9955 ± 0.0029	0.9986 ± 0.0014
Hidden-metavariable (L=100)	0.9765 ± 0.0115	0.9948 ± 0.0033	0.9986 ± 0.0014
Static-metavariable (L=98)	**0.9856 ± 0.0078**	**0.9978 ± 0.0018**	**0.9997 ± 0.0008**

Table 5.6: Normalized performance for the laminate problem using different representations.

	Fitness function evaluations		
	10k	25k	50k
Variable-length genome	**0.9886 ± 0.0042**	**0.9975 ± 0.0014**	**0.9998 ± 0.0009**
Hidden-metavariable (L=20)	**0.9889 ± 0.0032**	0.9973 ± 0.0014	0.9994 ± 0.0011
Hidden-metavariable (L=40)	0.9858 ± 0.0039	0.9955 ± 0.0018	0.9989 ± 0.0012
Hidden-metavariable (L=60)	0.9855 ± 0.0042	0.9953 ± 0.0022	0.9989 ± 0.0011
Hidden-metavariable (L=100)	0.9850 ± 0.0042	0.9950 ± 0.0025	0.9989 ± 0.0012

Chapter 6

Variational Operators

Variational operators are used to produce child solutions from parent solutions. The use of a metameric representation requires modifications to the traditional, fixed-length variational operators. This chapter presents several crossover operators, as well as a modified mutation operator, and compares their performance. The best crossover operators are found to be the spatial and similar-metavariable crossover, which minimize disruption. Earlier work on this topic appears in the article published in Genetic Programming and Evolvable Machines: *Solving metameric variable-length optimization problems using genetic algorithms* [118].

6.1 Crossover

Crossover operators create child solutions by exchanging genotypic information between parent solutions. Each operator discussed here uses two parents for each crossover, however it is possible to use more. The simplex crossover, a non-metameric operator used in Section 7.6, can be applied to two or more parents.

Effective crossover operators for metameric problems differ considerably from those typically used by traditional evolutionary algorithms. In standard, fixed-length problems, each solution is defined by the same set of design variables. For example, each genotype might define a length, height, and radius. The representation of these variables in the genotype remains fixed (e.g., length is defined at the same locus in all solutions). It's clear that crossover must result in children that follow the same representation as the parents. Frequently the crossover exchanges information at a randomly determined set of loci (e.g., n-point crossover).

In metameric problems it isn't clear how parent solutions should be partitioned. Each metavariable must define the same set of design variables, but there are generally no restrictions on the overall composition of

metavariables in a solution. Two parents may contain the same metavariable but at different loci, or they may contain functionally dissimilar metavariables at the same locus. With this in mind, it doesn't make sense to exchange metavariables based on their location in the parent genotypes[1]. The crossover operators proposed in this chapter can only exchange whole metavariables, the contents of which are not modified. While such an operator is possible for metameric problems, it is not considered here.

A key to the success of a genetic algorithm is its ability to form building blocks, short subsequences (or *schemata*) of the genome that have a positive influence on the fitness of an individual. The crossover operator must be able to perform *respectful* recombination [50] and minimize the risk of *disruption* [95]. A respectful crossover operator is one that produces children containing any schemata shared by both parents. Disruption occurs when a building block is not fully inherited from a parent by a child. Metameric crossover operators should be designed with these properties in mind.

The following subsections define several different crossovers that are used with metameric representations. The n-point crossover (Section 6.1.1) can only be applied to metameric representations using a fixed-length genome. Cut and splice is a modified n-point crossover that is frequently used for variable-length genomes in literature. The spatial and similar-metavariable crossovers (Sections 6.1.3 and 6.1.4) are specialized for metameric representations and try to minimize the chance of disruption. Details on the application of these operators to ordered problems is discussed in Section 4.1.2.

6.1.1 N-point

The n-point crossover is one of the most common operators found in traditional, fixed-length genetic algorithms. Parent genotypes are divided at n randomly placed crossover points, typically 1 or 2 points are used. Genotypic information is then swapped between the parents based on the crossover points to create the child genotypes. Since this is applied to fixed-length problems the location of the crossover points must match between the parents. This ensures that the representation of variables in the child genotypes matches those of the parent genotypes.

In this work the n-point crossover is only applied to metameric problems using a fixed-length genome, this includes the hidden-metavariable representation (Section 5.2.1) and static-metavariable representation (Section 5.2.2). The locations of the crossover points are limited to the boundaries between metavariables in the genotype, individual metavariables are considered to be indivisible. An example of an n-point crossover applied to a fixed-length metameric genome is given in Figure 6.1.

[1]This is not true for the static-metavariable representation. Also, if the problem is ordered then the relative positioning of metavariables should be taken into considering, but not necessarily their precise loci.

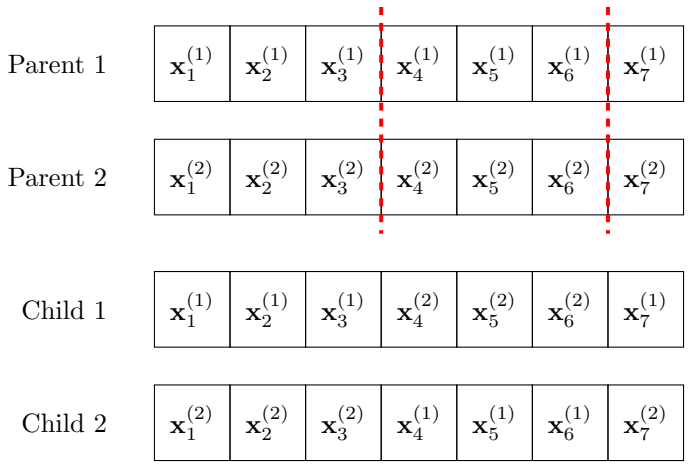


Figure 6.1: An example of n-point crossover applied to a fixed-length metameric genome. Dashed red lines represent crossover points.

6.1.2 Cut and Splice

Cut and splice crossover, demonstrated in Figure 6.2, is an n-point crossover where the crossover points do not have to match between the two parents. Parents may exchange an unequal number of metavariables during crossover, resulting in children of lengths different from either parent.

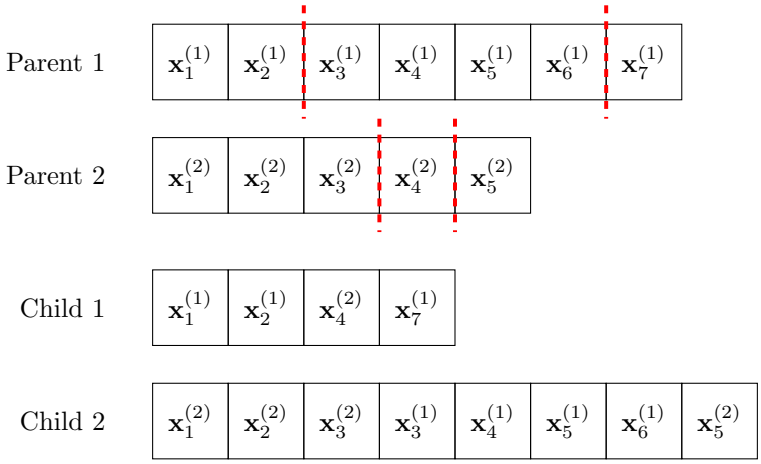


Figure 6.2: An example of cut and splice crossover. Dashed red lines represent crossover points.

Metavariables are exchanged only based on their locus and the chosen crossover points, no consideration is given to the design variables contained within. As a result, the cut and splice crossover is extremely disruptive. For non-ordered metameric problems the permutation of metavariables in the genotype is inconsequential. The distribution of metavariables from the parents to the children is almost random. Most frequently, the child solutions will have a length in the range bounded by the two parent lengths. However, it is also possible that one child will inherit almost all metavariables from both parents, while the other child inherits only a few. In ordered problems this operator maintains the relative order of metavariables in

the parent solutions. This is desirable for such problems, but the crossover remains considerably disruptive. Despite these drawbacks, the cut and splice operator has been applied frequently in literature, as shown by Table 3.1 in Chapter 3.

6.1.3 Spatial

Spatial crossover operators partition the parent genomes based on one or more chosen design variables. For example, suppose each metavariable defines a location on a plane. A random line could be used to divide that plane into two parts; each parent would then be partitioned into two sets of metavariables that lie on either side of the line. Children are formed by inheriting one set of metavariables from each parent. An example of this is shown in Figure 6.3.

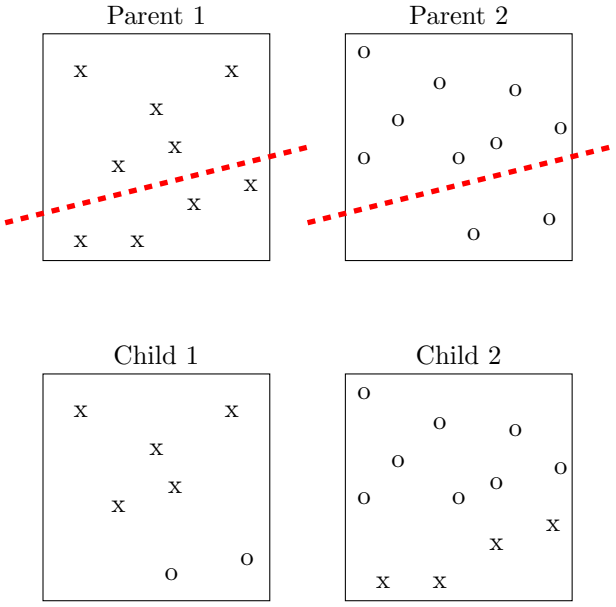


Figure 6.3: An example of spatial crossover. Each metavariable in this example is assumed to include an x- and y-position.

The effectiveness of spatial crossover depends partly on which design variables are considered when partitioning solutions. For example, partitioning solutions to the coverage problem based primarily on node radius is more likely to be disruptive compared to partitioning based on node location. The user must identify which design variables are used for spatial crossover. In this work, the x- and y-location variables are used for the coverage, packing, and wind farm problems. The order index variable (Section 4.1.2) is used for the laminate composite problem. Spatial crossover is not applied to the portfolio problem.

Any shape could be used to partition solutions. This work uses randomly generated hyperplanes defined in the normalized design space of a single metavariable. A vector $\mathbf{n}$ normal to the hyperplane is generated

from a normal distribution $\mathcal{N}(0,1)$. The vector values for variables not considered by the spatial crossover are set to 0, as given in (6.1). A point $\mathbf{y}$ is randomly generated in the normalized metavariable design space, given by (6.2).

$$n_i = \begin{cases} \mathcal{N}(0,1) & \text{if design variable } i \text{ is used in spatial crossover} \\ 0 & \text{otherwise} \end{cases} \qquad \text{for } i = 1, ..., v \tag{6.1}$$

$$y_i = \mathcal{U}(0,1) \qquad \text{for } i = 1, ..., v \tag{6.2}$$

A value $s(\mathbf{x}_i)$ can be calculated for each metavariable $\mathbf{x}_i \in \mathbf{x}$ using (6.3). If $\mathbf{n}$ is a unit vector then this value would be the signed distance to the plane in the normalized space, however for the purposes of this crossover only the sign matters. A positive value of $s(\mathbf{x}_i)$ indicates that $\mathbf{x}_i$ lies on the same side of the plane as vector $\mathbf{n}$. The parent solutions are partitioned into two sets of metavariables, those with $s(\mathbf{x}_i) \geq 0$ and those with $s(\mathbf{x}_i) < 0$. The partitions are then exchanged to form child solutions.

$$s(\mathbf{x}_i) = \sum_{j=1}^{v} n_j * \left(\frac{x_{i,j}}{x_j^{UB} - x_j^{UL}} - y_j \right) \tag{6.3}$$

When this operator is properly employed, there is a meaningful relationship between the sets of metavariables exchanged. Building blocks present in the parents are likely to be preserved in the children. Disruption of building blocks can still occur, but the region of disruption is limited to the hyperplane used to partition the solution [21]. The drawback of spatial crossover is its inability to easily generalize to all metameric problems. It may not be applicable to certain problems, such as the portfolio problem, and when it can be applied the user must determine which design variables are used.

6.1.4 Similar-Metavariable

Similar-metavariable crossover operates by identifying common subsets of metavariables in each parent. The distance of every metavariable pair between the two parents is calculated using the measures proposed in Section 4.2. Each metavariable is then linked to its most similar (i.e., smallest distance) counterpart in the other parent. Oftentimes, two metavariables will form a mutual linkage, however this is not always the case. A metavariable in parent 1 may link itself to a metavariable in parent 2, but there may exist a better match for that metavariable. When this happens, all metavariables in a parent that are connected through a pathway of linkages will form a subset, or building block, of metavariables. Figure 6.4 gives an example of forming the linkages and subsequent subsets of metavariables.

The parent genomes will segment themselves into paired subsets of one or more metavariables based on the formed linkages. The number of subsets N in each parent will match, but the number of metavariables in each subset do not have to match. A random number of paired subsets, chosen between 1 and N, are exchanged between the parents to form child solutions.

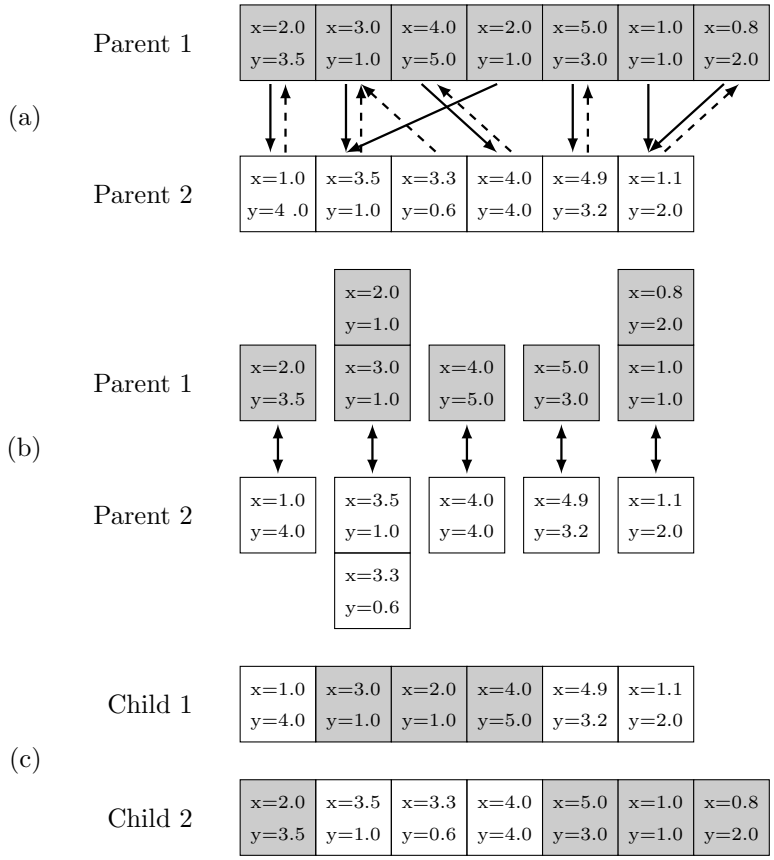


Figure 6.4: Similar-metavariable crossover. (a) Each metavariable identifies its most similar counterpart in the other parent and forms a link. Solid lines show linkages formed by the metavariables in parent 1, and dashed lines for those in parent 2. (b) All metavariables connected by these linkages are grouped together to form a subset. (c) A random number of these subsets are exchanged to form child solutions.

Compared to spatial crossover, similar-metavariable is a more generic operator. It can be applied to all of the benchmark problems, and the user is not required to identify the set of design variables considered during crossover. However, there is also a greater risk of disruption occurring when using this operator. Spatial crossover partitions the parent solutions into two large sets, while similar-metavariable crossover partitions the parent solutions into many smaller sets. Section 6.4 shows how exchanging many small sets of metavariables can result in a disruptive crossover.

6.2 Mutation

Several types of mutation have a chance to be applied to each solution: design variable mutation, metavariable addition, metavariable deletion, and permutation. These types of mutation are not mutually exclusive, each type has an independent chance of being applied during a single mutation operator call. Every type of mutation might be applied to some solutions, or no mutation may occur during a single call. It is possible

that some child solutions will remain identical to their parent solution after both crossover and mutation. When this occurs the mutation operator is repeatedly applied to the solution until a change occurs.

Design variable mutation is applied to the design variables within each metavariable. The mutation rate, given by (6.4), is set such that an average of 1 design variable is mutated per call. When applied to fixed-length genomes (Section 5.2), the mutation rate is calculated based on the number of expressed metavariables. The binary flag that controls metavariable expression is not subject to design variable mutation.

$$\text{design variable mutation rate } (\mathbf{x}) = \frac{1}{vL(\mathbf{x})} \tag{6.4}$$

Gaussian mutation, given in (6.5), is applied to the design variables marked for mutation. In this study the mutation magnitude is set to 0.05 of the design variable's range. When applied to integer or discrete design variables the new value is rounded to the nearest permissible value, other than the previous value, in the direction of mutation. When enumerated variables are mutated a random value from the set of permissible values is chosen.

$$x_{i,j}^{new} = \max\left(\min\left(x_{i,j}^{old} + \mathcal{N}(0, 0.05) * (x_j^{UB} - x_j^{LB}),\ x_j^{UB}\right),\ x_j^{LB}\right) \tag{6.5}$$

Metavariable addition and *metavariable deletion* each have a 5% chance of occurring during mutation. During metavariable addition, a randomly generated metavariable is inserted at a random location in the genotype. Metavariable deletion will remove a randomly chosen metavariable from the genotype. When applied to fixed-length genomes these types of mutations act on the binary expression flag.

Permutation also has a 5% chance of occurring. Two metavariables are chosen at random and their positions swapped in the genotype. Note that if a problem is unordered, and a static-metavariable representation is not used, then this operator will have no effect on the solution's objective or constraint values.

6.3 Results

Each crossover operator was applied to each benchmark problem, with the exception that spatial crossover is not applied to the portfolio problem. Additionally, a *mutation only* algorithm is tested for each problem. The remaining operators are the default ones described in Section 4.1. Tables 6.1-6.6 contain the normalized performance (Section 4.4) for each algorithm at several different points. The best results, and those not significantly different ($p > 0.05$) from the best, for each generation are shown in bold. Figures 6.5-6.10 give plots of normalized performance f_{norm} and the average best solution length for each algorithm.

Spatial crossover gave the best results across all problems where it was applied. Similar-metavariable crossover had a slightly worse performance on all problems except the wind farm problem. Algorithms using only mutation had weaker performances compared to the spatial and similar-metavariable crossovers. Cut

and splice crossover consistently gave the worst performance of all tested algorithms. The differences in performance are less notable on the portfolio and laminate problems.

6.4 Discussion

There are two potential explanations for the performance difference between the spatial and similar-metavariable crossovers. First, spatial crossover only considers a subset of the design variables for most problems, while similar-metavariable crossover considers all design variables. It may be that using only the defined subset of variables allows for a more reliable crossover operator. Alternatively, it may be that the spatial crossover is generally less disruptive. Similar-metavariable crossover partitions each parent into many small subsets of metavariables, which are then randomly exchanged to form children. As a result, the metavariables from each parent might be highly interspersed, providing many opportunities for disruption to occur. This is demonstrated in Figure 6.11.

The cut and splice crossover is extremely disruptive and its poor performance is unsurprising. The effectively random exchange of metavariables between parents is unlikely to produce well-formed child solutions. Its performance on the portfolio and laminate composite problems is closer to, but still worse than, that of the other operators. These problems have relatively small search spaces, it is possible that the algorithm remains able to sufficiently explore such spaces without an efficient crossover.

Despite their destructive tendencies, crossover operators do improve overall algorithm performance. This is demonstrated by the superior performance of algorithms using a spatial or similar-metavariable crossover compared to the mutation only algorithms.

Table 6.1: Normalized performance of crossovers on the constrained coverage problem.

	Fitness function evaluations		
	20k	50k	100k
Mutation only	0.5135 ± 0.0702	0.8557 ± 0.0323	0.9753 ± 0.0131
Cut and splice	-0.7591 ± 0.2175	-0.0455 ± 0.1404	0.4519 ± 0.0889
Spatial	**0.7850 ± 0.0429**	**0.9749 ± 0.0157**	**1.0000 ± 0.0129**
Similar-metavariable	0.7575 ± 0.0522	0.9667 ± 0.0159	0.9955 ± 0.0144

Table 6.2: Normalized performance of crossovers on the unconstrained coverage problem.

	Fitness function evaluations		
	20k	50k	100k
Mutation only	0.6573 ± 0.0531	0.8917 ± 0.0324	0.9688 ± 0.0178
Cut and splice	0.2469 ± 0.0733	0.5964 ± 0.0576	0.7949 ± 0.0406
Spatial	**0.9138 ± 0.0231**	**0.9883 ± 0.0135**	**1.0000 ± 0.0114**
Similar-metavariable	0.8903 ± 0.0285	0.9779 ± 0.0173	0.9933 ± 0.0129

Table 6.3: Normalized performance of crossovers on the packing problem.

	Fitness function evaluations		
	20k	50k	100k
Mutation only	0.7638 ± 0.0332	0.8795 ± 0.0241	0.9412 ± 0.0184
Cut and splice	0.6621 ± 0.0333	0.7531 ± 0.0314	0.8231 ± 0.0297
Spatial	**0.8792 ± 0.0311**	**0.9607 ± 0.0256**	**1.0000 ± 0.0260**
Similar-metavariable	0.8308 ± 0.0371	0.9293 ± 0.0280	0.9803 ± 0.0280

Table 6.4: Normalized performance of crossovers on the wind farm problem.

	Fitness function evaluations		
	20k	50k	100k
Mutation only	0.9719 ± 0.0029	0.9889 ± 0.0020	0.9960 ± 0.0018
Cut and splice	0.9320 ± 0.0071	0.9643 ± 0.0036	0.9806 ± 0.0027
Spatial	0.9890 ± 0.0025	**0.9976 ± 0.0018**	**1.0000 ± 0.0015**
Similar-metavariable	**0.9898 ± 0.0025**	**0.9977 ± 0.0019**	**1.0000 ± 0.0017**

Table 6.5: Normalized performance of crossovers on the portfolio problem.

	Fitness function evaluations		
	10k	25k	50k
Mutation only	0.9608 ± 0.0157	0.9860 ± 0.0050	0.9952 ± 0.0031
Cut and splice	0.9760 ± 0.0074	0.9886 ± 0.0040	0.9953 ± 0.0022
Similar-metavariable	**0.9846 ± 0.0063**	**0.9973 ± 0.0015**	**0.9996 ± 0.0006**

Table 6.6: Normalized performance of crossovers on the laminate problem.

	Fitness function evaluations		
	10k	25k	50k
Mutation only	0.9816 ± 0.0049	0.9926 ± 0.0026	0.9978 ± 0.0011
Cut and splice	0.9786 ± 0.0046	0.9900 ± 0.0032	0.9960 ± 0.0017
Spatial	**0.9886 ± 0.0042**	**0.9975 ± 0.0014**	**0.9998 ± 0.0009**
Similar-metavariable	0.9840 ± 0.0040	0.9948 ± 0.0019	0.9984 ± 0.0011

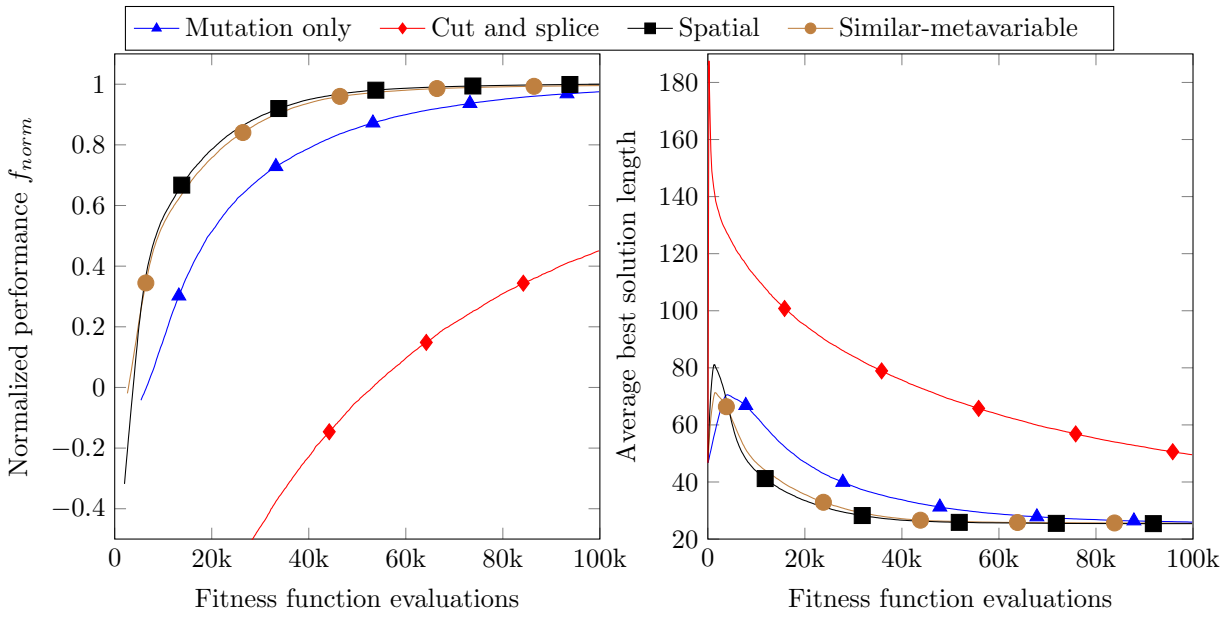


Figure 6.5: Normalized performance and average best solution length for the constrained coverage problem using different crossover operators. Performance lines are not shown if any infeasible solution exist.

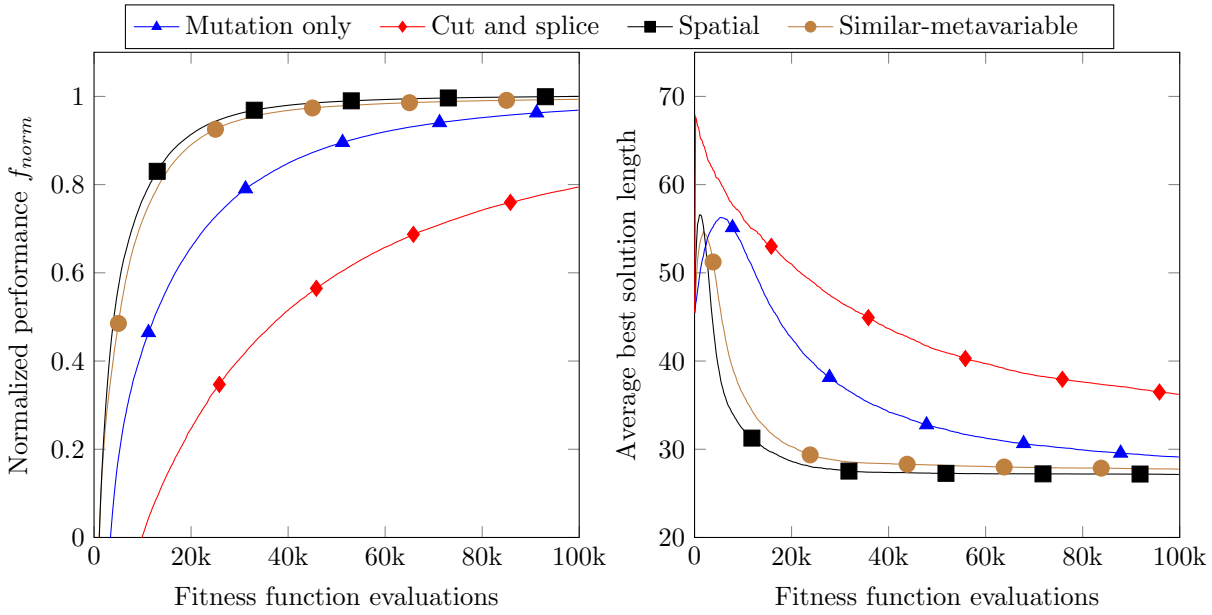


Figure 6.6: Normalized performance and average best solution length for the unconstrained coverage problem using different crossover operators.

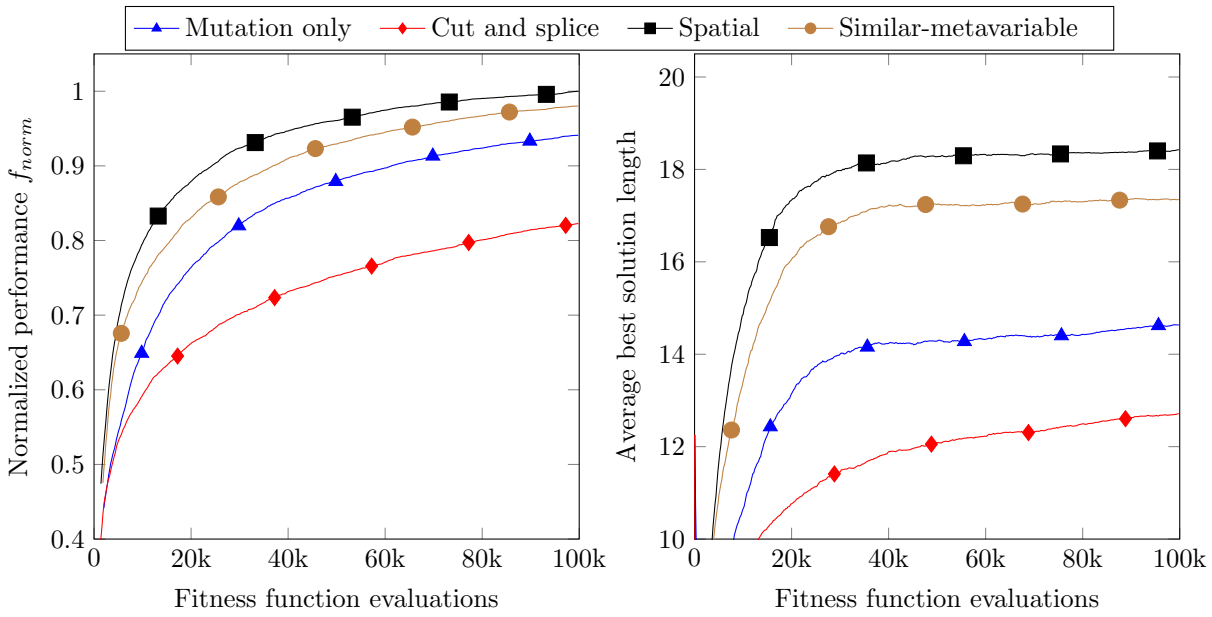


Figure 6.7: Normalized performance and average best solution length for the packing problem using different crossover operators. Performance lines are not shown if any infeasible solution exist.

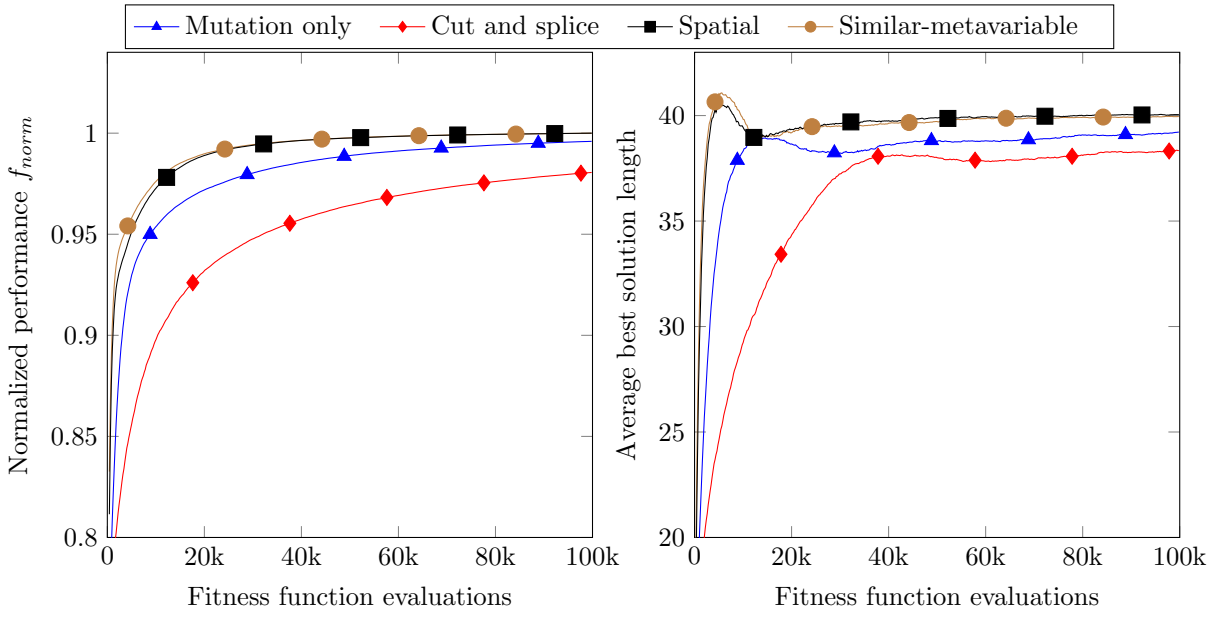


Figure 6.8: Normalized performance and average best solution length for the wind farm problem using different crossover operators.

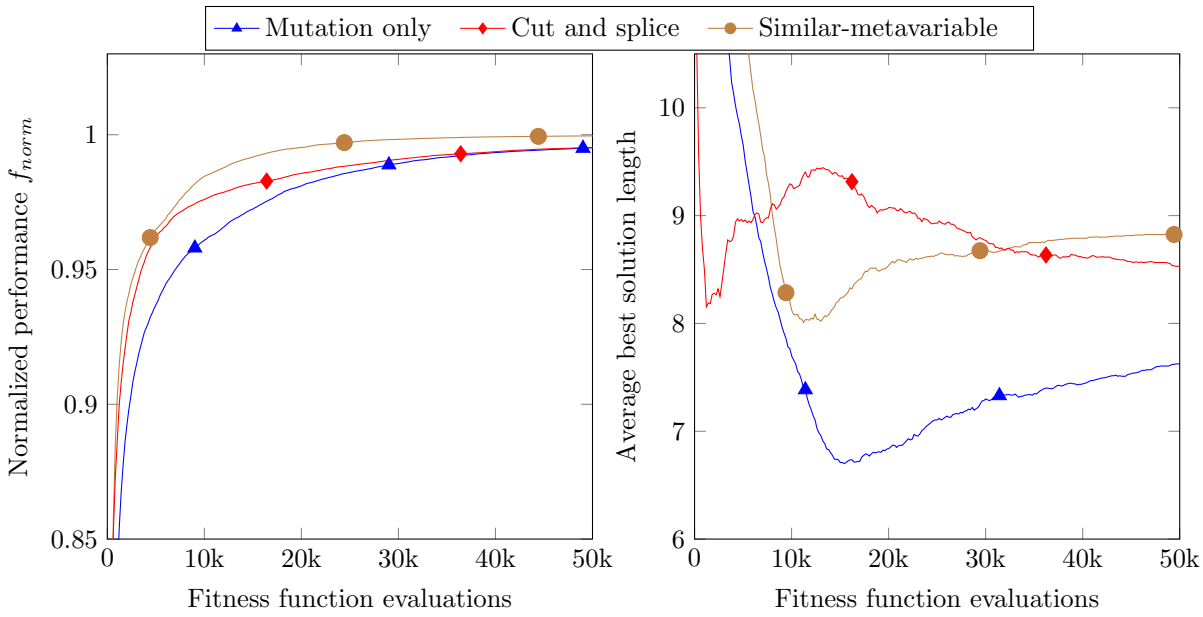


Figure 6.9: Normalized performance and average best solution length for the portfolio using different crossover operators.

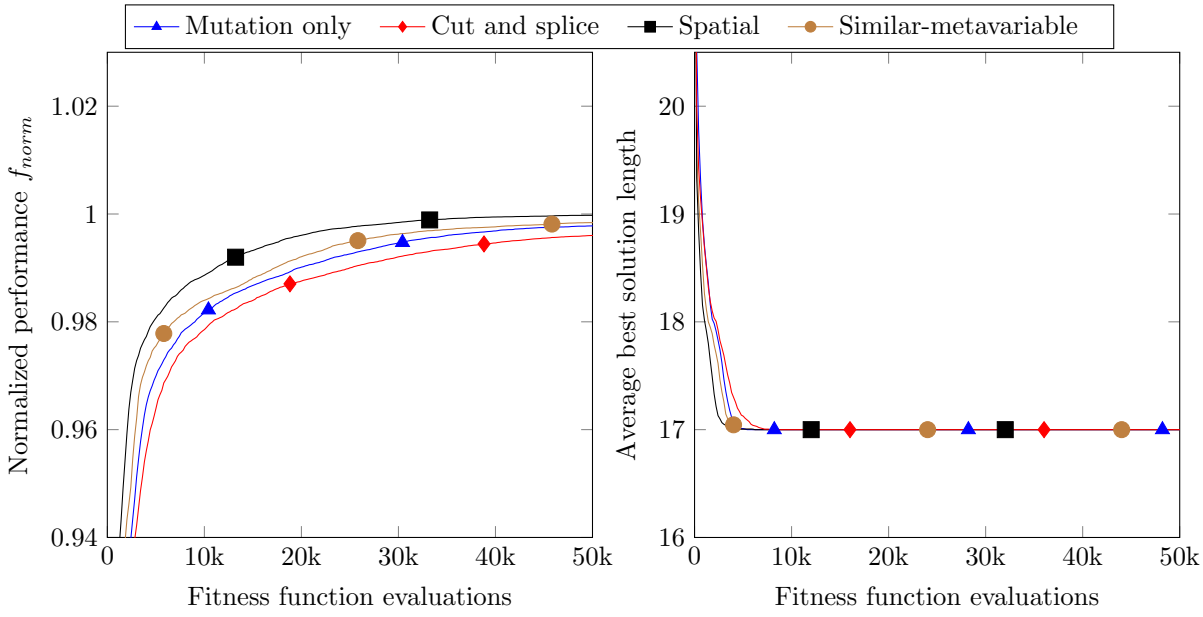


Figure 6.10: Normalized performance and average best solution length for the laminate problem using various crossover operators.

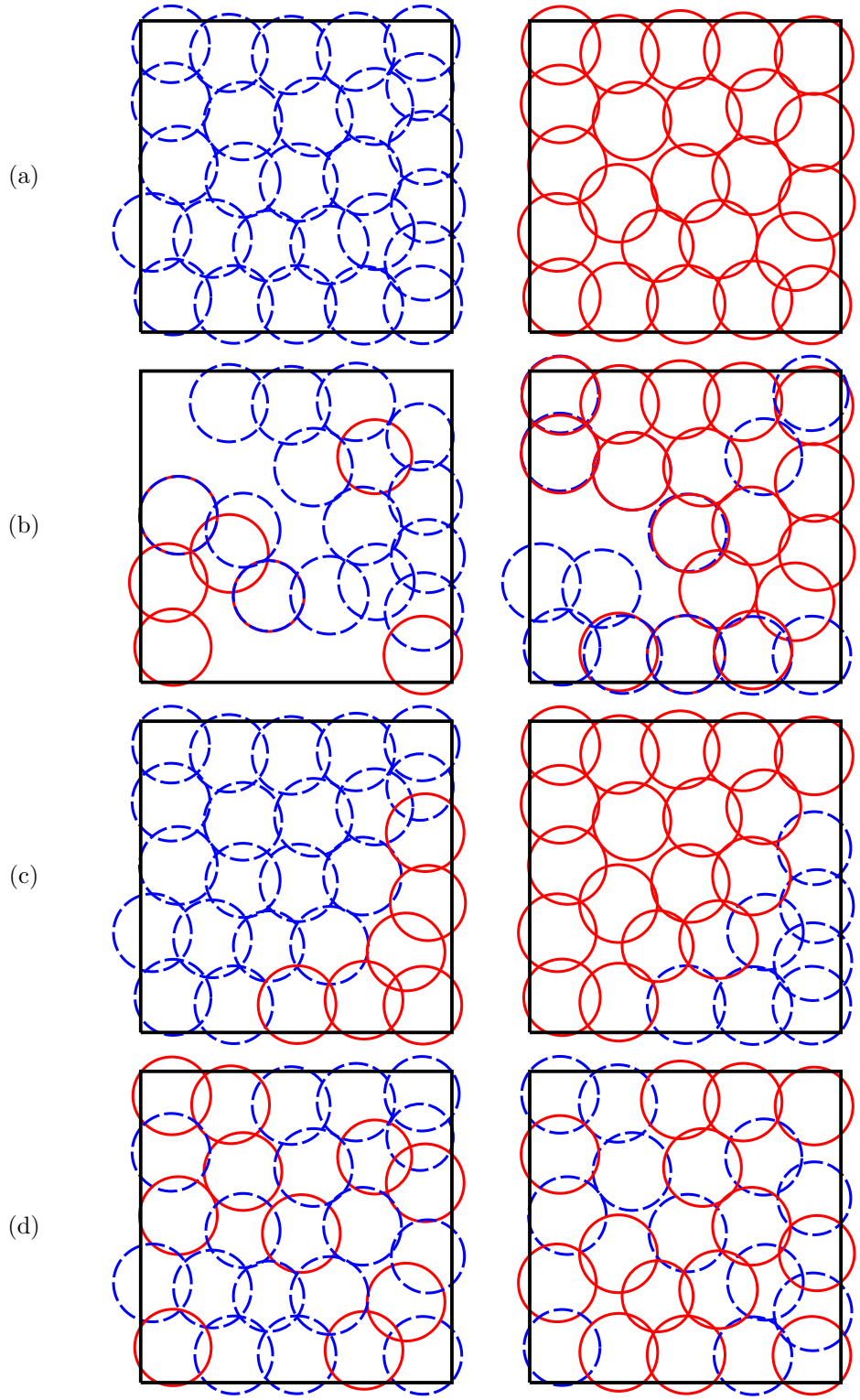


Figure 6.11: Comparison of children produced by different crossover operators on the constrained coverage problem. (a) Parent solutions used to produce children through (b) cut and splice, (c) spatial, or (d) similar-metavariable crossover. The line style of each node, either dashed blue or solid red, indicates from which parent each metavariable was inherited.

Chapter 7

Selection Operators

Among the literature cited in Chapter 3, the selection operator is commonly overlooked. While the dimensionality of the genotypic space can vary, the dimensionality of the fitness landscape remains fixed, allowing for traditional operators to be employed. However, such operators may be insufficient when used with metameric representations. Notably, they may preserve little to no diversity of solution length, which may result in convergence to sub-optimal lengths.

This chapter proposes and compares the performance of new selection operators for metameric problems. Length niching is proposed, in which the population is partitioned based on length. Local selection is then applied to each niche independently, guaranteeing a new population of diverse solution lengths. A new local selection operator, score selection, is found to significantly improve results compared to tournament selection.

This chapter is largely adapted from two articles in preparation: *A length niching operator for metameric problems* [116], and *Improved local selection for metameric problems* [115]. Note that these titles may change prior to publication.

7.1 Background

In Chapter 3 it was found that studies of metameric problems frequently do not consider solution length when performing selection. Instead, traditional selection operators were applied. Depending on the problem this may lead to bloat or premature convergence. Bloat is the uncontrolled growth of solution length without a significant return in fitness [108]. The benchmark problems considered here are not susceptible to bloat since their fitnesses are, at least in part, functions of solution length. Unnecessary or uncontrolled increases in length will be detrimental to fitness. Parsimony pressure (Section 3.2.4.2) can be applied to alleviate the effects of bloat when needed.

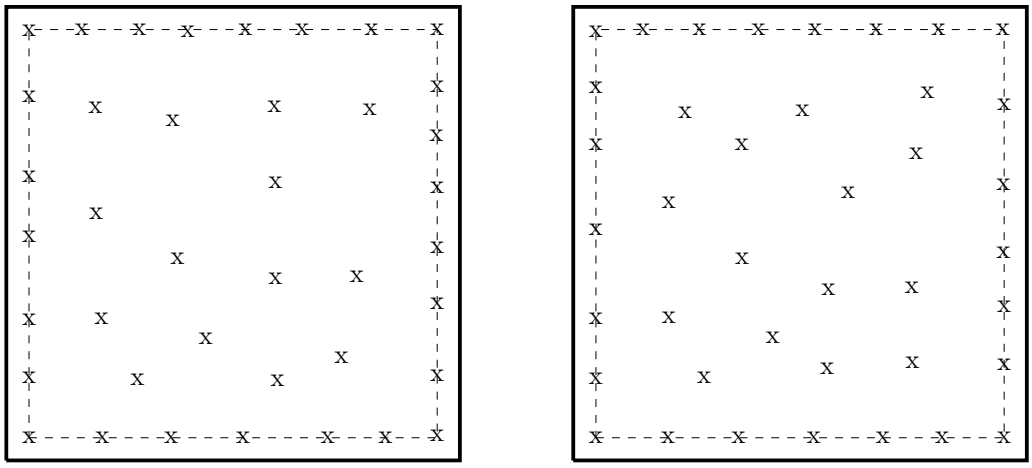


Figure 7.1: Two most similar solutions from 5000 independent trials of the wind farm problem.

Premature convergence occurs when the metameric algorithm converges to a suboptimal length. Once converged, the deleterious effect of the length-varying operators may prevent any diversity of solution lengths from persisting in the population. Maintaining diversity of solution lengths can help avoid premature convergence. Some literature of metameric problems have used niching (Section 3.2.4.4) or multi-objective selection operators (Section 3.2.4.1) that are expected to result in more diverse populations.

The search spaces of metameric problems are extremely multimodal, as evidenced by the variety of solutions obtained for each problem. As a demonstration, the wind farm problem was solved 5000 times using the selection operators proposed here. A fixed-length selection window (Section 7.2.1) was used to ensure that all trials found a solution using 40 turbines. All the best solutions from each trial were compared to one another using the distance measure given in Section 4.2. No two trials produced the same solution, the two most similar solutions are shown in Figure 7.1. While these solutions share certain commonalities they are clearly two different solutions.

We have previously proposed a helper objective, acting on solution length, for metameric problems [118]. While the results were promising, there are two drawbacks to this approach. First, the user must decide whether the helper objective minimizes or maximizes solution length. The correct choice is not always apparent without testing both options. Second, premature convergence could occur if the length-varying operators fail to produce feasible solutions at new lengths. Feasible solutions may exist at these lengths, but the algorithm may only find them if given a chance to refine the initially infeasible solutions. This may never occur since feasible solutions are considered to dominate infeasible ones.

Section 7.2 proposes length niching selection to be used with metameric representations. First, the population is partitioned into a number of niches based on solution length. It is inefficient to create a niche for every possible length, the computational resources available to the algorithm would be spread too

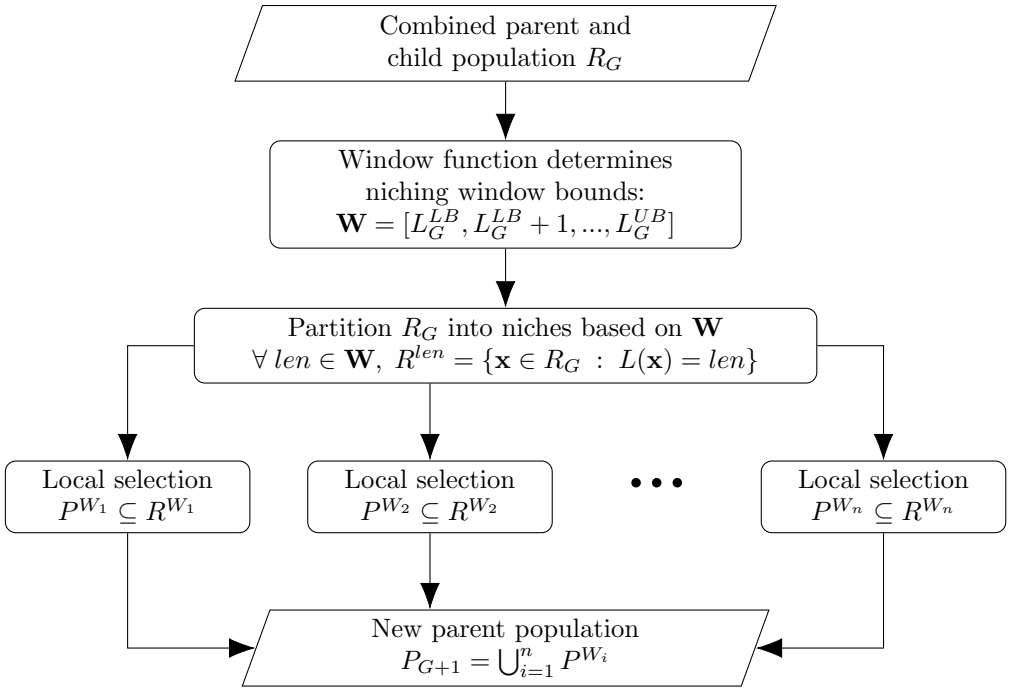


Figure 7.2: A flowchart for the length niching selection operator.

thin. To prevent this, a window function determines the set of lengths at which niches will be formed. Several window functions are proposed and demonstrated in this study. A local selection operator is then applied independently to each niche, selecting the solutions that will constitute the next generation's parent population. Section 7.3 discusses the local selection operators used in this work.

7.2 Length Niching Selection

This section presents the procedure of the length niching selection operator. A flowchart and pseudocode for this operator are available in Figure 7.2 and Algorithm 7.1 respectively.

At each generation G, the parent population P_G is used to generate a child population Q_G through crossover and mutation. These populations are combined to form $R_G = P_G \cup Q_G$. The objective function value and length of the best solution in R_G are denoted as f_G^* and L_G^* respectively. R_G is then partitioned into a number of niches such that each niche contains all solutions of a particular length: $R^{len} = \{\mathbf{x} \in R_G : L(\mathbf{x}) = len\}$.

Niches are not created at every length present in R_G. The subset of lengths used to form niches is determined by a window function. This function determines the lower and upper bounds, L_G^{LB} and L_G^{UB}, of a selection window W such that $W = [L_G^{LB}, L_G^{LB}+1, ..., L_G^{UB}]$. A niche will be formed for every solution length in W. Several window functions are proposed in Sections 7.2.1-7.2.4.

Algorithm 7.1: Pseudocode for length niching selection.

Input: G current generation
R_G combined parent and child population
N population size
Output: P_{G+1} new parent population
$f_G^* \leftarrow$ Objective function value of best solution in R_G
$L_G^* \leftarrow$ Length of best solution in R_G
$L_G^{LB}, L_G^{UB} \leftarrow$ Bounds from window function
$\mathbf{W} \leftarrow [L_G^{LB}, L_G^{LB}+1, ..., L_G^{UB}]$
$r \leftarrow N \bmod \text{size}(\mathbf{W})$
$P_{G+1} \leftarrow \emptyset$
for $i := 1$ to size($\boldsymbol{W}$) **do**
  $N_i \leftarrow \text{floor}(\frac{N}{\text{size}(\mathbf{W})})$
  if W_i is among r closest lengths to $L_G^* - 0.1$ **then**
    $N_i \leftarrow N_i + 1$
  $R^{W_i} \leftarrow \{\mathbf{x} \in R_G \ : \ L(\mathbf{x}) = W_i\}$
  $P^{W_i} \leftarrow$ Local Selection(R^{W_i}, N_i)
  $P_{G+1} \leftarrow P_{G+1} \cup P^{W_i}$
if size$(P_{G+1}) < N$ **then**
  $N_{filler} \leftarrow N - \text{size}(P_{G+1})$
  $R^{rem} \leftarrow R_G \cap P_{G+1}$
  $P^{filler} \leftarrow$ Local Selection(R^{rem}, N_{filler})
  $P_{G+1} \leftarrow P_{G+1} \cup P^{filler}$

Once formed, each niche is subject to a local selection operator, choosing a subset of solutions to include in the next parent population: $P^{len} \subseteq R^{len}$. The selected solutions from each niche are combined to form the new parent population: $P_{G+1} = \bigcup_{i=1}^{n} P^{W_i}$. By performing local selection on each niche independently, P_G is ensured to remain diverse in terms of solution length.

Length niching selection will try to select the same number of solutions from each niche. This is not always possible since the number of niches may not divide cleanly into the population size. In such cases, the niches nearest to L_G^* are allocated an extra selection until the population is full. If two niches are equidistant from L_G^* then the niche with shorter solution is preferred.

Some niches may not contain enough solutions to fill their allocation, in which case the entire niche is selected: $P^{len} = R^{len}$. Once local selection has been performed on each niche, a set of the remaining, unselected solutions is created: $R^{rem} = R_G \cap P_{G+1}$. This includes the solutions with lengths not in the selection window. Local selection is applied to R^{rem} in order to fill P_{G+1}.

7.2.1 Fixed-Length Window

The fixed-length window function assumes that the optimum solution length is known *a priori*. The window bounds are both equal to this length (i.e., $L_G^{LB} = L_G^{UB}$). Children produced by crossover or mutation with lengths other than the window length are discarded and replaced prior to evaluation. No length niching occurs when using this window since only a single niche is formed during each generation.

7.2.2 Static Window

The static window function also assumes some knowledge of the optimal length *a priori*. Window bounds are chosen such that the window is believed to contain this length, along with one or more adjacent lengths (i.e., $L_G^{LB} < L_G^{UB}$).

This window allows for the simultaneous exploration of several lengths. Even if the optimal length is known for certain it may be beneficial to use a static window rather than a fixed-length window. Selecting solutions from several length niches will maintain a greater level of diversity in the population compared to selecting from only a single niche. As with the fixed-length window, child solutions are discarded prior to evaluation if their length is not within the static window.

7.2.3 Moving Window

Rather than defining the window bounds *a priori* this function uses a window that is centered at each generation's best solution length L_G^*. No knowledge of the optimal length is required, but the user must define a window width w. Equation (7.1) defines the window bounds for the moving window function. Note that, due to the floor and ceiling functions, a range of w values will result in the same window bounds. For example, all values in the interval (2,4) are functionally equivalent. Unlike the fixed-length and static windows, child solutions are not discarded should they fall outside of the current window bounds.

In the study which proposed the helper objective, a selection window was also used to limit the size of the resulting Pareto front [118]. This window was functionally equivalent to the moving window.

$$\begin{aligned} L_G^{LB} &= \left\lceil L_G^* - \frac{w}{2} \right\rceil \\ L_G^{UB} &= \left\lfloor L_G^* + \frac{w}{2} \right\rfloor \end{aligned} \tag{7.1}$$

7.2.4 Biased Window

The biased window also follows L_G^*, however unlike the moving window it is not simply centered on the best solution length. Instead, the window bounds are shifted based on the history of the best known solution

length. A bias factor B_G is initially set to 0 and updated each generation using (7.2).

$$B_G = L^*_G - L^*_{G-1} + B_{G-1}e^{-\lambda\sqrt{|B_{G-1}|}} \tag{7.2}$$

The bias factor is increased or decreased whenever a change in the best solution length is detected, represented by $L^*_G - L^*_{G-1}$. An increase in the best length will result in a positive change in the bias factor value. The exponential term in (7.2) will decay the bias factor toward 0 over time. Note that the rate of decay is itself a function of the current bias factor. This work uses a decay factor $\lambda = 0.004$.

The bias factor is an approximation of the current trend in best solution length. Positive values indicate a general trend toward longer solutions, while negative values show a trend toward shorter solutions. B_G is used to adjust the bounds of the niching window using (7.3). Note that the window bounds cannot be shifted past L^*_G.

$$\begin{aligned} L^{LB}_G &= min\left(\left\lceil L^*_G - \frac{w}{2} + B_G \right\rceil, L^*_G\right) \\ L^{UB}_G &= max\left(\left\lfloor L^*_G + \frac{w}{2} + B_G \right\rfloor, L^*_G\right) \end{aligned} \tag{7.3}$$

Small B_G magnitudes will simply shift the window toward shorter or longer solutions, while large magnitudes will also stretch the total window width. Examples of this are given in Figure 7.3.

Odd-valued window widths w are used in this study. As a result, the window will only shift if $|B_G| \geq 0.5$. For smaller $|B_G|$ values this window will be functionally equivalent to the moving window until a change in best solution length occurs.

A demonstration of how the bias factor reacts to changes in the best solution length is shown in Figure 7.4. Rapid increases and decreases in L^*_G result in large positive and negative values of B_G respectively. This demonstration is performed using the constrained coverage problem (Section 4.3.1). In early generations, the best solutions use many nodes in order to satisfy the constraint on minimum coverage. As the algorithm progresses, the excess nodes are removed and solution length decreases. Eventually, best solution length settles into its optimal value and the bias factor decays toward 0. At small B_G values the niching window will center back onto L^*_G, allowing the algorithm to simultaneously search solutions shorter and longer than the current best.

7.3 Local Selection

In the following sections there is no mention of solution length, the operators are applied to a combined parent and child population R. Neither of the described local selection operators are specific to metameric problems. When applied along with length niching, the population R is assumed to be equivalent to one of the length niches R^{len} described in Section 7.2.

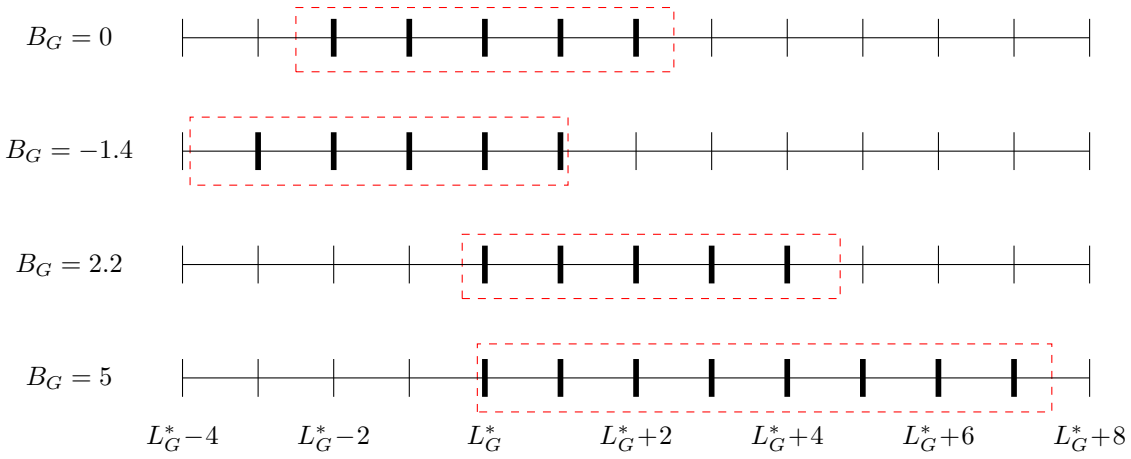


Figure 7.3: Biased windows, visualized using the dashed box, for various B_G values assuming a window width of $w = 5$. The thicker tick marks indicate solution lengths at which a niche will be formed.

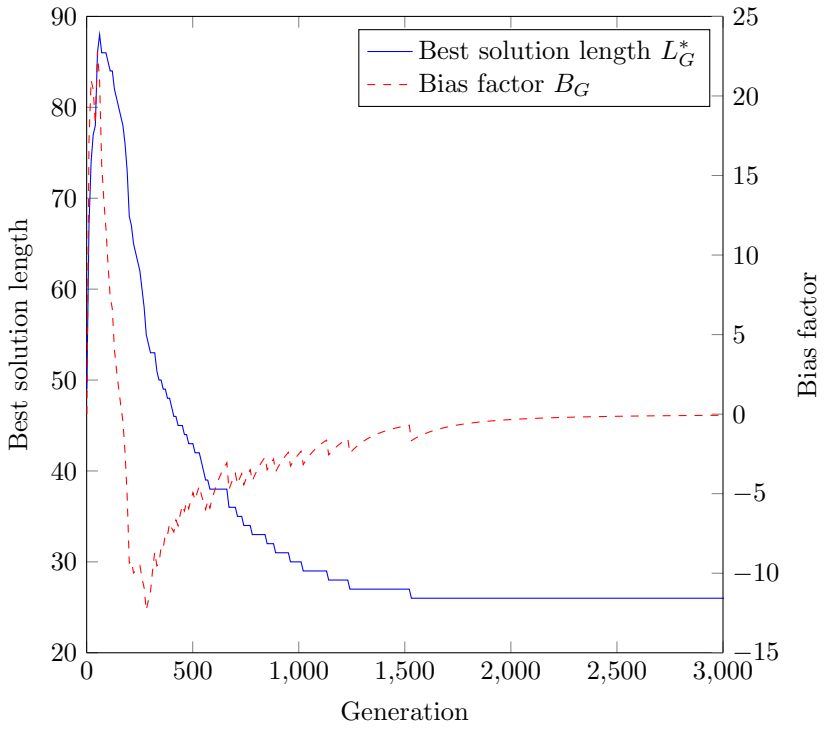


Figure 7.4: An example of how the bias factor reacts in response to changes in the best solution length.

7.3.1 Tournament Selection

Deterministic tournament selection [34] is demonstrated as one option for local selection in metameric algorithms. In standard implementations a tournament size k needs to be defined. Smaller tournament sizes will result in a smaller selection pressure acting on the population. This can help the population remain diverse for a longer period compared to larger tournament sizes.

When using length niching selection, the number of solutions in each niche can vary. For this reason, an adaptive tournament size is used. k is chosen as the smallest size that allows every solution in the niche to participate in at least one tournament. For example, if 5 solutions are to be selected from a niche of 12 solutions then $k = 3$. In this case, several randomly selected solutions will participate in two tournaments.

7.3.2 Score Selection

Score selection is proposed as an improved local selection operator for metameric problems. Scores are calculated for each solution based on their objective value, constraint violation, and distance to other solutions. A weighted sum of these scores is used to compare candidate solutions.

Standard selection schemes, such as tournament selection, apply a strong pressure toward feasible solutions over infeasible ones. Even solutions with minimal constraint violations are unlikely to survive more than a few generations, if any, when feasible solutions are also present. This can impede exploration of the search space. Better feasible solutions may exist, but the variational operators may require several intermediate steps through the search space to reach them. If the intermediate solutions are infeasible then the algorithm may stagnate.

Performance might be improved by handling constraints in such a way that a mix of feasible and infeasible solutions are selected each generation. Several comprehensive reviews of constraint handling in evolutionary algorithms are available [91, 22, 90]. These reviews classify constraint handling techniques into a number of categories. The technique used by score selection is best described as an adaptive penalty function.

Frequently, the term exploration is used to describe the act of visiting new regions of the search space, while exploitation describes locating new solutions in the vicinity of already visited regions [23]. Successful optimization algorithms should seek a good balance between exploration and exploitation. Algorithms that do not perform sufficient exploration are more likely to get stuck in local optima. This is a concern in many metameric problems since the fitness landscapes can be extremely multimodal (Section 7.1).

Maintaining diversity in the population can help increase the level of exploration. Score selection calculates a score for each solution based in part on their genotypic distance to other solutions. Those that are more similar (i.e., smaller distance) to others are less likely to be selected. This is comparable to fitness sharing [121, 23], although the implementation is notably different than the traditional fitness sharing formulations [51, 63].

It is worth noting that studies of multimodal problems frequently seek to identify more than one optimal solution. This is not the case in this work, the proposed operator only explores the search space in order to find a more optimal solution.

7.3.2.1 Methodology

This section gives a detailed explanation of score selection. Pseudocode is given in Algorithm 7.2.

Let R be a set of candidate solutions, formed from the combined parent and child populations: $R = P \cup Q$. The selection operator is to select N solutions from R to form the next generation's parent population. Score selection is an iterative process, requiring one iteration for each solution in R.

Let X be the candidate set of selected solutions. This set is initially empty. Each iteration a solution $\mathbf{r}$ is removed from R and added to X, forming a new set $X' = X \cup \mathbf{r}$. Any parent solutions that exist in R will be chosen as $\mathbf{r}$ prior to any child solutions. If $|X'| \leq N$, then the new set becomes the candidate set $X = X'$, and the next iteration begins.

Eventually, enough solutions will be added to the new candidate set such that $|X'| = N + 1$. Once this occurs, the operator must determine one solution in X' that will not be included in X. Each solution in X' is given a score for its objective value, constraint violation, and distance to other solutions in X'. A final score is calculated as a weighted sum of the individual scores, the score functions are given in Section 7.3.2.2. The solution with the worst (i.e., highest) score in X' is identified as $\mathbf{x}^{worst}$. This solution is removed from X' to form the new candidate set of selected solutions: $X = X' \cap \mathbf{x}^{worst}$. If any solutions remain in R then the next iteration begins, otherwise X becomes next generation's parent population.

It is important to discuss the reason this is performed as an iterative process, rather than scoring all solutions in R simultaneously and selecting the N best scores. The objective, constraint, and distance scores must all be normalized values such that a final score can be calculated as a weighted sum. This study uses adaptive normalization factors that are determined using the solutions in X. Every time X is updated the normalization factors are recalculated.

Suppose instead the normalization factors are determined using the entire population R. A child with significantly worse objective values or constraint violations, compared to the parent solutions, is unlikely to be selected. However, this child could still affect the normalization factors. Certain methods of calculating these factors may be very sensitive to outliers, resulting in nearly meaningless values. This impedes the operator's ability to select solutions in a worthwhile manner. Instead, the selected set of solutions will be heavily influenced by the worst performing solutions in R.

A similar concern arises when calculating scores based on distance to other solutions. A child solution may be nearly identical to a parent solution, both having good objective and constraint values. Calculating distance scores by comparing all solutions in R simultaneously will result in a poor score for both solutions. This might lead to neither solution being selected.

The described iterative process helps alleviate these concerns. The normalization factors are calculated

Algorithm 7.2: Pseudocode for the score selection operator.

Input: R candidate solutions
N number of solutions to select
F_G trend in fitness values at current generation
Output: X selected solutions
$R \leftarrow$ Rearrange R such that parent solutions appear first
$X \leftarrow \emptyset$
for each $\boldsymbol{r}$ in R **do**
  $X' \leftarrow X \cup \mathbf{r}$
  if size$(X') \leq N$ **then**
    $X \leftarrow X'$
    continue
  $f^*_{X'} \leftarrow$ Fitness of best solution in X'
  Calculate D_{mean} using (7.7)
  Calculate γ using (7.11)
  for each $\boldsymbol{x}$ in X' **do**
    Calculate $s_f(\mathbf{x})$ using (7.5)
    Calculate $s_\phi(\mathbf{x})$ using (7.6)
    Calculate $s_D(\mathbf{x})$ using (7.8)
    Calculate $S(\mathbf{x})$ using (7.9)
  $\mathbf{x}^{worst} \leftarrow \arg\min_{\mathbf{x} \in X'} S(\mathbf{x})$
  $X \leftarrow X' \cap \mathbf{x}^{worst}$

using only solutions in X. Recall that the parent solutions in R are the first ones added to X', and subsequently X. This helps ensure some continuity in the normalization factors from generation to generation. If X is initially set to a random set of solutions in R, then the normalization factors may be meaningless, as described above. X' is not used to calculate the normalization factors as it includes the newly added candidate solution $\mathbf{r}$. Doing so would make it possible for certain solutions $\mathbf{r}$ to skew the normalization factors for their own benefit.

7.3.2.2 Score Calculation

Let $\mathbf{x} \in X'$, the objective value and constraint violation scores of $\mathbf{x}$ are denoted as $s_f(\mathbf{x})$ and $s_\phi(\mathbf{x})$ respectively. These scores are normalized by the minimum and maximum objective and constraint violation values in X, as given in (7.4). Equations (7.5) and (7.6) are used to calculate $s_f(\mathbf{x})$ and $s_\phi(\mathbf{x})$. Note that all solutions in X will have scores in the range [0, 1]. It is possible that the newly added candidate solution $\mathbf{r}$ has a value outside this range.

$$
\begin{aligned}
f_{min} &= \min_{\mathbf{x} \in X} f(\mathbf{x}) \\
f_{max} &= \max_{\mathbf{x} \in X} f(\mathbf{x}) \\
\phi_{min} &= \min_{\mathbf{x} \in X} \phi(\mathbf{x}) \\
\phi_{max} &= \max_{\mathbf{x} \in X} \phi(\mathbf{x})
\end{aligned}
\tag{7.4}
$$

$$
s_f(\mathbf{x}) = \begin{cases} \dfrac{f(\mathbf{x}) - f_{min}}{f_{max} - f_{min}} & \text{if } f_{max} \neq f_{min} \\ 0 & \text{otherwise} \end{cases}
\tag{7.5}
$$

$$
s_\phi(\mathbf{x}) = \begin{cases} \dfrac{\phi(\mathbf{x}) - \phi_{min}}{\phi_{max} - \phi_{min}} & \text{if } \phi_{max} \neq \phi_{min} \\ 0 & \text{otherwise} \end{cases}
\tag{7.6}
$$

The function $D(\mathbf{x}^{(1)}, \mathbf{x}^{(2)})$, defined in Section 4.2, is a measure of genotypic distance between two solutions. Equation (7.7) calculates the average distance D_{mean} between all solutions in X. The distance score, $s_D(\mathbf{x})$, is calculated by (7.8). Calculating this score requires a pairwise calculation of distance between $\mathbf{x}$ and all other solutions in X'. Each calculated distance is normalized by D_{mean}.

$$
D_{mean} = \frac{1}{|X| * |X - 1|} \sum_{\mathbf{x} \in X} \sum_{\mathbf{p} \in X}^{\mathbf{p} \neq \mathbf{x}} D(\mathbf{x}, \mathbf{p})
\tag{7.7}
$$

$$
s_D(\mathbf{x}) = \frac{1}{N} \sum_{\mathbf{p} \in X}^{\mathbf{p} \neq \mathbf{x}} \frac{1.6875}{\left(\frac{D(\mathbf{x}, \mathbf{p})}{D_{mean}} + 0.5 \right)^2}
\tag{7.8}
$$

It is clear that the distance score is nonlinear, it increases rapidly as the normalized distance between two solutions approaches zero. This nonlinearity affects the potential trade-off between distance and objective value, or constraint violation, when selecting solutions. Consider a subset of solutions in X' that have a very small normalized distance between them, resulting in high $s_D(\mathbf{x})$ scores. There will be a greater pressure to improve $s_D(\mathbf{x})$ compared to their $s_f(\mathbf{x})$ or $s_\phi(\mathbf{x})$ scores. In a subset of solutions that have large normalized distance between them the opposite is true.

Given the above, there must also be a subset of solutions where the trade-offs between distance, objective, and constraint violation scores are equal. This should occur when the distance between all solutions in that subset is equal to D_{mean}. For all solutions, the rate of change between normalized objective values and $s_f(\mathbf{x})$ is simply 1, the two values are the same. This is also true for the constraint violation score. To achieve an even trade-off, $s_D(\mathbf{x})$ can be scaled such that its rate of change is equal to -1 when the distance between solutions is equal to D_{mean}. The sign is negative since higher normalized distances result in smaller $s_D(\mathbf{x})$ values.

Suppose we have an equation $t(y) = (y + 0.5)^{-2}$, this function is similar to the summed function in (7.8) where $y = D(\mathbf{x}, \mathbf{p})/D_{mean}$. If the distance between solutions is equal to D_{mean} then $y = 1$. The rate of

change at this point is calculated as $t'(y=1) = -2 * (1+0.5)^{-3} = 16/27$. Scaling $t(y)$ by 27/16, or 1.6875, results in $t'(y=1) = -1$. By including this coefficient in the distance score, an even trade-off between distance, objective value, and constraint violation is achieved for a set of solutions that are all spaced by D_{mean} from one another.

Final scores are calculated as a weighted sum of the individual scores using (7.9). Three different weighted sums can be used. If all solutions in X' are infeasible, then the final score applies only a small weight to $s_f(\mathbf{x})$. This results in a strong pressure toward solutions with small constraint violations.

$$S(\mathbf{x}) = s_D(\mathbf{x}) + \begin{cases} \gamma * (\frac{1}{8} s_f(\mathbf{x}) + s_\phi(\mathbf{x})) & \text{if } \phi_{min} > 0 \\ \gamma * (\frac{1}{2} s_f(\mathbf{x}) + s_\phi(\mathbf{x}) - \frac{1}{2}) & \text{if } f(\mathbf{x}) < f^*_{X'} \\ \gamma * (s_f(\mathbf{x}) + s_\phi(\mathbf{x})) & \text{otherwise} \end{cases} \tag{7.9}$$

If any feasible solutions exist in X', then the second two cases are used. The objective value of the best feasible solution in X' is identified as $f^*_{X'}$. Any solution with an objective function value better than $f^*_{X'}$ uses a smaller weighting for s_f and has its final score reduced by 0.5γ. Note that any solutions that fit this criteria must be infeasible. Any other solutions, feasible or infeasible, use the third case. A balancing term γ is used to help maintain a reasonable level of diversity in X, more details are given in the following section.

7.3.2.3 Balancing Diversity

Each of the scores use adaptive normalization factors, calculated from the solutions in X. There is a general pressure toward solutions with better objective values, constraint violations, or distance to other solutions. However, since they are adaptive, the scores provide no information regarding the level of diversity present in the population. There is a possibility the population becomes highly converged, containing almost no diversity, or the population diverges to the point where local refinement of solutions (i.e., exploitation) is hindered.

It would be possible to apply pressure toward a specific level of diversity by setting a fixed value for D_{mean}. It is unclear what an optimal value would be, a value that works well for one problem may not for others. The optimal level of diversity may also vary through the course of a trial. A high level of diversity in early generations may help to fully explore the design space, while a lower level in later generations may help to further refine the obtained solutions.

To maintain the correct level of diversity this work proposes a measure of general objective value trend F_G, calculated using (7.10). This factor is similar to the bias factor used for the biased window function (Section 7.2.4). Rapid changes in f^*_G will lead to larger F_G magnitudes. If no feasible solutions exist[1] then F_G is set to 0. As f^*_G converges the value of F_G will decay toward 0.

[1]If length niching is used, then this includes all solutions in the parent population, not just the niche population. The same F_G value is used by all niches at any given generation.

$$F_G = \begin{cases} \left| f_G^* - f_{G-1}^* \right| + F_{G-1} * e^{-\lambda} & \text{if any feasible solutions have been found} \\ 0 & \text{otherwise} \end{cases} \tag{7.10}$$

The range of objective values in X can be compared to F_G to approximate the current level of diversity. This is quantified as the balance term γ, given in (7.11). If F_G is much larger than the range of objective values, then the relative level of diversity is considered too low, resulting in $\gamma < 1$. If F_G is much smaller, then the relatively level of diversity is considered too high, resulting in $\gamma > 1$. The weighting of $s_D(\mathbf{x})$, relative to $s_f(\mathbf{x})$ and $s_\phi(\mathbf{x})$, is affected by γ. Larger or smaller values of γ will respectively decrease or increase the pressure toward a more diverse population.

$$\gamma = \begin{cases} \left| (f_{max} - f_{min}) \frac{2}{F_G + (f_{max} - f_{min})} \right| & \text{if } (F_G \neq 0) \vee (f_{max} \neq f_{min}) \\ 1 & \text{otherwise} \end{cases} \tag{7.11}$$

It is expected that the greatest changes in f_G^* will occur during the early stages of the algorithm. This results in larger F_G values and more pressure towards diverse populations. As the algorithm progresses, and F_G approaches zero, the algorithm is more likely to converge, allowing for a greater level of exploitation to occur.

7.4 Results

A number of different selection schemes were tested for each benchmark problem. Length niching selection was applied using each of the window functions, several different window bounds or widths were used for each function. A *non-niching* selection operator was tested in which the entire population is considered a single niche, with no consideration given to length. Each of these setups was tested using tournament and score selection as the local selection operator. The helper objective [118] was also tested for each problem.

Tables 7.1-7.6 contain the normalized performance (Section 4.4) for each algorithm at several different points. The best results, and those not significantly different ($p > 0.05$) from the best, for each generation are shown in bold. Figures 7.5-7.10 give plots of normalized performance and the average best solution length for a subset of the algorithms.

When using tournament selection, the length niching selection operator with a biased window or a static window were always among the top performing operators. The moving window function performed well but had worse performances on both coverage problems. The fixed-length window performed poorly on all problems except the portfolio and laminate problems. Non-niching selection performed very poorly on the laminate and both coverage problems.

All algorithms performed significantly better when using the score selection operator, and frequently resulted in similar performances. The biased window performs best on the constrained coverage problem,

but is outperformed by several other algorithms on the packing problem. The packing problem performance may be explained by its tendency to converge to longer solutions, as shown in Figure 7.7. The moving window function and non-niching selection algorithms give poor results on both coverage problems, Figures 7.5 and 7.6 show them slow, or failing, to converge to optimal lengths. Non-niching performs well on the remaining problems when score selection is used. Generally, the biased and moving windows perform better when using a narrower window width w.

The fixed-length and static-window functions generally perform well, but they require that the optimal solution length is known *a priori*. Even when the optimal length is known the performance can vary based on the exact bounds used. These functions tend to outperform the other algorithms early on since they are not required to locate optimal solution lengths.

The results using the helper objective were generally poor. On some problems its performance was comparable to algorithms using tournament selection. However, it was not competitive with algorithms using the score selection.

7.5 Discussion

The results demonstrate the effectiveness of both length niching and score selection. Both result in a greater level of diversity in the populations, resulting in a more efficient exploration of the search space.

Length niching selection preserves diversity based on solution length. Tournament selection will eventually, and sometimes quickly, converge to a single solution. Forming niches at several lengths and applying tournament selection to each niche independently results in much greater population diversity. Each niche will converge toward solutions that are optimal for that particular length. An example of solutions from several niches in a single population is given in Figure 7.11. Note that when applying tournament selection, there may be very little diversity within each niche.

Score selection maintains diversity by considering each solution's objective value, constraint violations, and distance to other solutions. When used with length niching this results in increased diversity within each niche. This is demonstrated in Figure 7.12. Compared to tournament selection, algorithms using score selection are less likely to prematurely converge to less optimal solutions.

In the early stages of the algorithm, tournament selection may outperform score selection. While score selection will maintain a diverse population, tournament selection will quickly converge toward the best-so-far solution. Since tournament selection primarily searches the space near to a single solution it is able to quickly improve the fitness of that solution. However, once converged the algorithm may fail to escape local optima.

Table 7.1: Normalized performance for the constrained coverage problem using various selection operators. Values in parentheses indicate the fraction of trials with feasible solutions, fractions are not shown if all trials have found feasible solutions.

	Fitness function evaluations		
	20k	50k	100k
Tournament Selection			
Fixed-length (L=24)	(0.00) 1.0362 ± 0.0071	(0.00) 1.0296 ± 0.0019	(0.00) 1.0293 ± 0.0011
Fixed-length (L=25)	(0.00) 0.9998 ± 0.0092	(0.17) 0.9912 ± 0.0049	(0.23) 0.9919 ± 0.0065
Fixed-length (L=26)	(0.10) 0.9656 ± 0.0100	(0.74) 0.9655 ± 0.0136	(0.91) 0.9728 ± 0.0145
Fixed-length (L=27)	(0.39) 0.9383 ± 0.0119	(0.93) 0.9529 ± 0.0171	(0.98) 0.9613 ± 0.0157
Fixed-length (L=28)	(0.68) 0.9186 ± 0.0160	(0.99) 0.9443 ± 0.0150	0.9520 ± 0.0137
Static window (L=23-25)	(0.04) 0.9922 ± 0.0043	(0.40) 0.9921 ± 0.0055	(0.57) 0.9945 ± 0.0072
Static window (L=24-25)	(0.08) 0.9913 ± 0.0038	(0.44) 0.9923 ± 0.0057	(0.56) 0.9950 ± 0.0076
Static window (L=24-26)	(0.64) 0.9613 ± 0.0115	(0.96) 0.9802 ± 0.0168	(0.99) 0.9892 ± 0.0152
Static window (L=24-27)	(0.98) 0.9546 ± 0.0190	0.9804 ± 0.0159	0.9903 ± 0.0138
Static window (L=25-26)	(0.68) 0.9609 ± 0.0105	(0.94) 0.9762 ± 0.0162	(0.97) 0.9834 ± 0.0161
Static window (L=25-27)	(0.96) 0.9516 ± 0.0175	0.9748 ± 0.0146	0.9840 ± 0.0132
Moving window (w=3)	0.7552 ± 0.0433	0.8890 ± 0.0414	0.9409 ± 0.0384
Moving window (w=5)	0.7916 ± 0.0413	0.9339 ± 0.0301	0.9734 ± 0.0227
Moving window (w=7)	0.8091 ± 0.0388	0.9455 ± 0.0194	0.9757 ± 0.0170
Biased window (w=3)	**0.8746 ± 0.0319**	0.9696 ± 0.0181	0.9841 ± 0.0172
Biased window (w=5)	**0.8700 ± 0.0343**	0.9675 ± 0.0162	0.9840 ± 0.0155
Biased window (w=7)	0.8651 ± 0.0341	0.9624 ± 0.0174	0.9813 ± 0.0155
Non-niching	0.6040 ± 0.0463	0.6641 ± 0.0404	0.6861 ± 0.0392
Score Selection			
Fixed-length (L=24)	(0.00) 1.0299 ± 0.0027	(0.00) 1.0294 ± 0.0001	(0.00) 1.0295 ± 0.0002
Fixed-length (L=25)	(0.12) 0.9915 ± 0.0047	(0.33) 0.9936 ± 0.0081	(0.39) 0.9959 ± 0.0100
Fixed-length (L=26)	(0.54) 0.9624 ± 0.0116	(0.89) 0.9769 ± 0.0153	(0.94) 0.9820 ± 0.0153
Fixed-length (L=27)	(0.87) 0.9475 ± 0.0170	(0.99) 0.9670 ± 0.0151	(0.99) 0.9728 ± 0.0142
Fixed-length (L=28)	(0.99) 0.9374 ± 0.0150	0.9568 ± 0.0122	0.9622 ± 0.0118
Static window (L=23-25)	(0.11) 0.9922 ± 0.0040	(0.53) 0.9954 ± 0.0074	(0.76) 1.0013 ± 0.0093
Static window (L=24-25)	(0.20) 0.9915 ± 0.0046	(0.56) 0.9973 ± 0.0092	(0.71) 1.0022 ± 0.0110
Static window (L=24-26)	(0.74) 0.9673 ± 0.0144	(0.97) 0.9897 ± 0.0162	(0.98) 0.9990 ± 0.0145
Static window (L=24-27)	(0.99) 0.9616 ± 0.0166	**0.9899 ± 0.0147**	**0.9986 ± 0.0125**
Static window (L=25-26)	(0.85) 0.9689 ± 0.0136	(0.98) 0.9889 ± 0.0147	(0.99) 0.9963 ± 0.0133
Static window (L=25-27)	(0.98) 0.9644 ± 0.0154	**0.9877 ± 0.0148**	0.9956 ± 0.0133
Moving window (w=3)	0.7185 ± 0.0500	0.9200 ± 0.0298	0.9776 ± 0.0209
Moving window (w=5)	0.7493 ± 0.0468	0.9371 ± 0.0234	0.9867 ± 0.0159
Moving window (w=7)	0.7593 ± 0.0458	0.9436 ± 0.0210	0.9886 ± 0.0146
Biased window (w=3)	0.7850 ± 0.0429	0.9749 ± 0.0157	**1.0000 ± 0.0129**
Biased window (w=5)	0.7919 ± 0.0430	0.9688 ± 0.0175	0.9975 ± 0.0122
Biased window (w=7)	0.7973 ± 0.0415	0.9629 ± 0.0155	0.9926 ± 0.0126
Non-niching	0.8544 ± 0.0292	0.9398 ± 0.0211	0.9520 ± 0.0196
Helper Objective			
Helper objective (w=3)	0.6109 ± 0.0450	0.6586 ± 0.0384	0.6743 ± 0.0377
Helper objective (w=5)	0.5995 ± 0.0504	0.6488 ± 0.0458	0.6649 ± 0.0446
Helper objective (w=7)	0.6016 ± 0.0479	0.6488 ± 0.0431	0.6646 ± 0.0419

Table 7.2: Normalized performance for the unconstrained coverage problem using various selection operators.

	Fitness function evaluations		
	20k	50k	100k
Tournament Selection			
Fixed-length (L=24)	0.7835 ± 0.0675	0.8690 ± 0.0487	0.8865 ± 0.0477
Fixed-length (L=25)	0.8280 ± 0.0589	0.9024 ± 0.0481	0.9177 ± 0.0458
Fixed-length (L=26)	0.8589 ± 0.0542	0.9252 ± 0.0420	0.9402 ± 0.0383
Fixed-length (L=27)	0.8722 ± 0.0538	0.9346 ± 0.0400	0.9485 ± 0.0358
Fixed-length (L=28)	0.8928 ± 0.0431	0.9440 ± 0.0314	0.9548 ± 0.0285
Static window (L=24-26)	0.9411 ± 0.0253	0.9757 ± 0.0192	0.9853 ± 0.0187
Static window (L=24-27)	0.9479 ± 0.0199	0.9825 ± 0.0145	0.9922 ± 0.0125
Static window (L=25-26)	0.9390 ± 0.0284	0.9709 ± 0.0243	0.9800 ± 0.0217
Static window (L=25-27)	**0.9529 ± 0.0214**	0.9820 ± 0.0170	0.9910 ± 0.0151
Static window (L=25-28)	**0.9555 ± 0.0167**	0.9854 ± 0.0126	0.9936 ± 0.0113
Static window (L=26-28)	**0.9560 ± 0.0193**	0.9845 ± 0.0130	0.9910 ± 0.0115
Moving window (w=3)	0.8495 ± 0.0352	0.9122 ± 0.0332	0.9476 ± 0.0324
Moving window (w=5)	0.8610 ± 0.0349	0.9358 ± 0.0259	0.9728 ± 0.0194
Moving window (w=7)	0.8689 ± 0.0356	0.9500 ± 0.0247	0.9803 ± 0.0152
Biased window (w=3)	0.9157 ± 0.0302	0.9767 ± 0.0181	0.9869 ± 0.0171
Biased window (w=5)	0.9139 ± 0.0272	0.9784 ± 0.0135	0.9896 ± 0.0117
Biased window (w=7)	0.9107 ± 0.0295	0.9744 ± 0.0130	0.9882 ± 0.0109
Non-niching	0.7971 ± 0.0351	0.8239 ± 0.0323	0.8322 ± 0.0309
Score Selection			
Fixed-length (L=24)	0.8786 ± 0.0218	0.9463 ± 0.0131	0.9540 ± 0.0122
Fixed-length (L=25)	0.9251 ± 0.0255	0.9933 ± 0.0233	1.0005 ± 0.0231
Fixed-length (L=26)	0.9435 ± 0.0196	**1.0010 ± 0.0134**	**1.0078 ± 0.0128**
Fixed-length (L=27)	0.9458 ± 0.0148	**0.9993 ± 0.0098**	**1.0062 ± 0.0092**
Fixed-length (L=28)	0.9463 ± 0.0128	0.9960 ± 0.0071	1.0028 ± 0.0068
Static window (L=24-26)	0.9113 ± 0.0223	0.9901 ± 0.0149	1.0030 ± 0.0136
Static window (L=24-27)	0.9143 ± 0.0235	0.9860 ± 0.0133	1.0009 ± 0.0120
Static window (L=25-26)	0.9278 ± 0.0224	0.9973 ± 0.0146	**1.0078 ± 0.0135**
Static window (L=25-27)	0.9251 ± 0.0219	0.9945 ± 0.0117	**1.0057 ± 0.0105**
Static window (L=25-28)	0.9286 ± 0.0198	0.9908 ± 0.0113	1.0024 ± 0.0101
Static window (L=26-28)	0.9343 ± 0.0181	0.9955 ± 0.0100	1.0050 ± 0.0088
Moving window (w=3)	0.8807 ± 0.0319	0.9628 ± 0.0212	0.9823 ± 0.0184
Moving window (w=5)	0.8777 ± 0.0279	0.9644 ± 0.0181	0.9883 ± 0.0132
Moving window (w=7)	0.8782 ± 0.0304	0.9659 ± 0.0162	0.9884 ± 0.0113
Biased window (w=3)	0.9138 ± 0.0231	0.9883 ± 0.0135	1.0000 ± 0.0114
Biased window (w=5)	0.9080 ± 0.0246	0.9826 ± 0.0125	0.9956 ± 0.0106
Biased window (w=7)	0.9126 ± 0.0233	0.9804 ± 0.0123	0.9950 ± 0.0104
Non-niching	0.8853 ± 0.0226	0.9460 ± 0.0183	0.9550 ± 0.0173
Helper Objective			
Helper objective (w=3)	0.8912 ± 0.0330	0.9507 ± 0.0266	0.9728 ± 0.0264
Helper objective (w=5)	0.9216 ± 0.0262	0.9777 ± 0.0153	0.9883 ± 0.0155
Helper objective (w=7)	0.9346 ± 0.0215	0.9812 ± 0.0116	0.9877 ± 0.0181

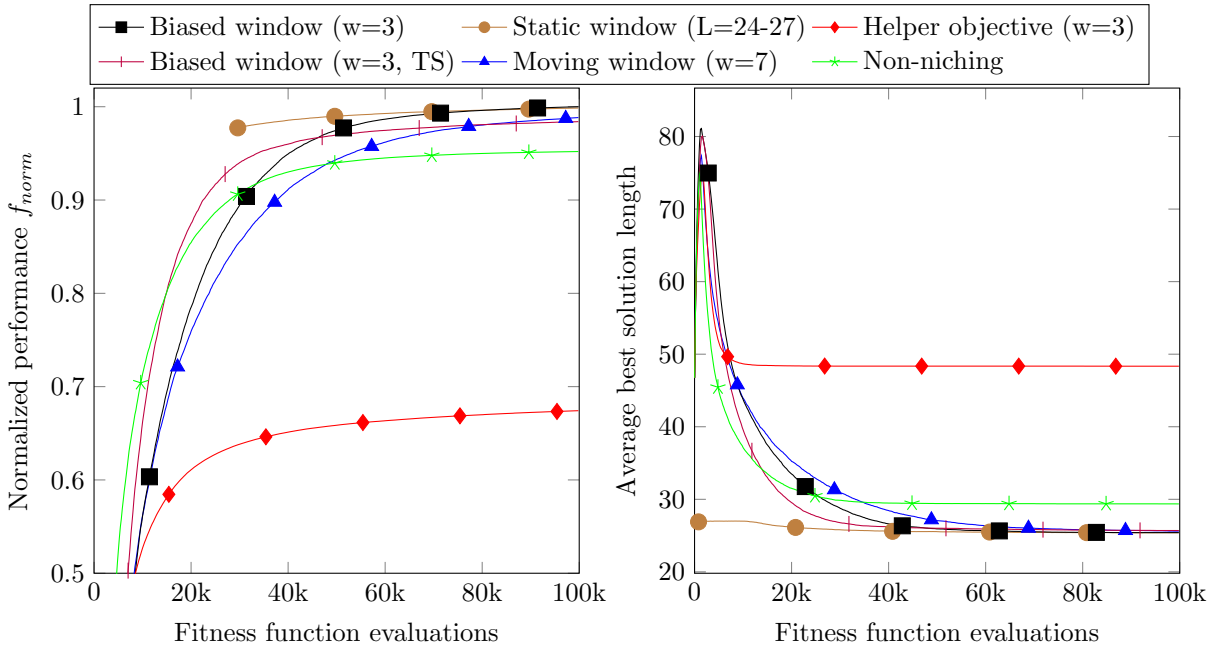


Figure 7.5: Normalized performance and average best solution length for the constrained coverage problem using various selection operators. Studies marked 'TS' use tournament selection, the remaining studies use score selection with the exception of the helper objective. Performance lines are not shown if any infeasible solution exist.

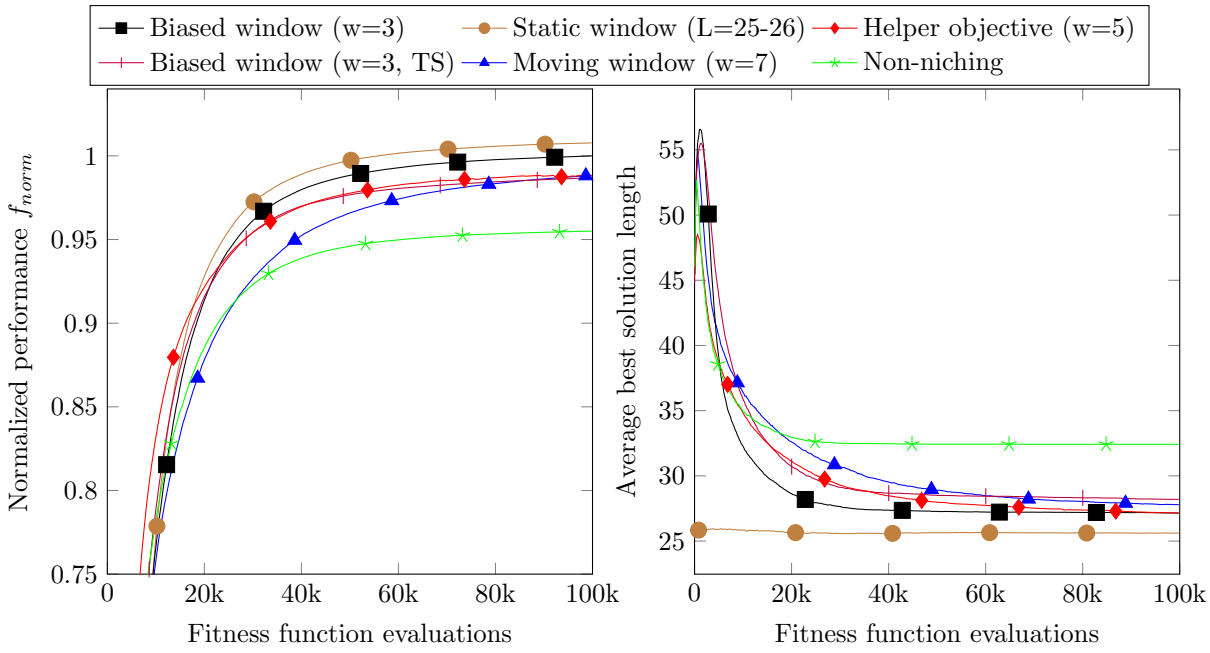


Figure 7.6: Normalized performance and average best solution length for the unconstrained coverage problem using various selection operators. Studies marked 'TS' use tournament selection, the remaining studies use score selection with the exception of the helper objective.

Table 7.3: Normalized performance for the packing problem using various selection operators. Values in parentheses indicate the fraction of trials with feasible solutions, fractions are not shown if all trials have found feasible solutions.

	Fitness function evaluations		
	20k	50k	100k
Tournament Selection			
Fixed-length (L=15)	0.7301 ± 0.0479	0.7915 ± 0.0415	0.8314 ± 0.0379
Fixed-length (L=16)	0.7210 ± 0.0451	0.7814 ± 0.0443	0.8258 ± 0.0439
Fixed-length (L=17)	0.7187 ± 0.0455	0.7805 ± 0.0438	0.8220 ± 0.0376
Fixed-length (L=18)	0.7110 ± 0.0432	0.7770 ± 0.0393	0.8195 ± 0.0379
Fixed-length (L=19)	0.6986 ± 0.0428	0.7644 ± 0.0386	0.8045 ± 0.0359
Fixed-length (L=20)	(0.99) 0.6947 ± 0.0514	0.7609 ± 0.0403	0.8001 ± 0.0368
Fixed-length (L=21)	(0.99) 0.6874 ± 0.0461	0.7506 ± 0.0400	0.7933 ± 0.0356
Static window (L=15-17)	0.7523 ± 0.0376	0.8130 ± 0.0360	0.8508 ± 0.0367
Static window (L=15-18)	0.7582 ± 0.0393	0.8211 ± 0.0368	0.8591 ± 0.0333
Static window (L=17-19)	0.7375 ± 0.0371	0.8020 ± 0.0341	0.8434 ± 0.0332
Static window (L=17-20)	0.7481 ± 0.0366	0.8101 ± 0.0351	0.8479 ± 0.0331
Static window (L=18-21)	0.7301 ± 0.0401	0.7974 ± 0.0342	0.8358 ± 0.0319
Moving window (w=3)	0.7748 ± 0.0377	0.8304 ± 0.0369	0.8610 ± 0.0338
Moving window (w=5)	0.7815 ± 0.0365	0.8331 ± 0.0320	0.8694 ± 0.0306
Moving window (w=7)	0.7826 ± 0.0362	0.8380 ± 0.0286	0.8751 ± 0.0276
Biased window (w=3)	0.7722 ± 0.0362	0.8235 ± 0.0323	0.8525 ± 0.0320
Biased window (w=5)	0.7774 ± 0.0352	0.8265 ± 0.0335	0.8548 ± 0.0348
Biased window (w=7)	0.7683 ± 0.0325	0.8237 ± 0.0322	0.8616 ± 0.0332
Non-niching	0.7655 ± 0.0361	0.8255 ± 0.0344	0.8625 ± 0.0309
Score Selection			
Fixed-length (L=15)	0.9128 ± 0.0328	0.9788 ± 0.0288	1.0191 ± 0.0291
Fixed-length (L=16)	0.9160 ± 0.0341	0.9855 ± 0.0310	1.0142 ± 0.0297
Fixed-length (L=17)	0.9125 ± 0.0323	0.9781 ± 0.0288	1.0084 ± 0.0261
Fixed-length (L=18)	0.9046 ± 0.0365	0.9746 ± 0.0306	1.0054 ± 0.0259
Fixed-length (L=19)	0.8971 ± 0.0376	0.9662 ± 0.0298	0.9974 ± 0.0283
Fixed-length (L=20)	0.8873 ± 0.0368	0.9650 ± 0.0294	0.9977 ± 0.0287
Fixed-length (L=21)	0.8761 ± 0.0372	0.9603 ± 0.0306	0.9890 ± 0.0277
Static window (L=15-17)	0.9089 ± 0.0312	0.9809 ± 0.0286	1.0140 ± 0.0222
Static window (L=15-18)	0.8983 ± 0.0326	0.9722 ± 0.0266	1.0066 ± 0.0214
Static window (L=17-19)	0.8955 ± 0.0278	0.9745 ± 0.0261	1.0078 ± 0.0216
Static window (L=17-20)	0.8799 ± 0.0322	0.9656 ± 0.0270	1.0001 ± 0.0237
Static window (L=18-21)	0.8744 ± 0.0298	0.9596 ± 0.0293	0.9990 ± 0.0214
Moving window (w=3)	0.9014 ± 0.0302	0.9768 ± 0.0261	1.0134 ± 0.0221
Moving window (w=5)	0.8883 ± 0.0306	0.9659 ± 0.0257	1.0016 ± 0.0226
Moving window (w=7)	0.8793 ± 0.0320	0.9532 ± 0.0257	0.9889 ± 0.0234
Biased window (w=3)	0.8792 ± 0.0311	0.9607 ± 0.0256	1.0000 ± 0.0260
Biased window (w=5)	0.8681 ± 0.0329	0.9579 ± 0.0250	0.9944 ± 0.0246
Biased window (w=7)	0.8575 ± 0.0325	0.9462 ± 0.0268	0.9857 ± 0.0242
Non-niching	**0.9265 ± 0.0293**	**0.9925 ± 0.0245**	**1.0253 ± 0.0222**
Helper Objective			
Helper objective (w=3)	0.7735 ± 0.0399	0.8256 ± 0.0364	0.8591 ± 0.0330
Helper objective (w=5)	0.7747 ± 0.0357	0.8294 ± 0.0357	0.8640 ± 0.0332
Helper objective (w=7)	0.7765 ± 0.0384	0.8260 ± 0.0360	0.8620 ± 0.0302

Table 7.4: Normalized performance for the wind farm problem using various selection operators.

	Fitness function evaluations		
	20k	50k	100k
Tournament Selection			
Fixed-length (L=38)	0.9801 ± 0.0041	0.9867 ± 0.0032	0.9896 ± 0.0031
Fixed-length (L=39)	0.9807 ± 0.0035	0.9873 ± 0.0029	0.9904 ± 0.0027
Fixed-length (L=40)	0.9802 ± 0.0044	0.9878 ± 0.0035	0.9911 ± 0.0031
Fixed-length (L=41)	0.9794 ± 0.0038	0.9873 ± 0.0031	0.9905 ± 0.0029
Fixed-length (L=42)	0.9786 ± 0.0041	0.9863 ± 0.0033	0.9899 ± 0.0031
Static window (L=38-40)	0.9880 ± 0.0026	0.9930 ± 0.0023	0.9955 ± 0.0021
Static window (L=38-41)	0.9881 ± 0.0029	0.9935 ± 0.0023	0.9960 ± 0.0021
Static window (L=39-40)	0.9876 ± 0.0033	0.9931 ± 0.0028	0.9955 ± 0.0022
Static window (L=39-41)	0.9882 ± 0.0028	0.9935 ± 0.0025	0.9958 ± 0.0023
Static window (L=40-41)	0.9873 ± 0.0030	0.9929 ± 0.0026	0.9955 ± 0.0025
Static window (L=40-42)	0.9881 ± 0.0033	0.9936 ± 0.0027	0.9961 ± 0.0024
Moving window (w=3)	0.9878 ± 0.0028	0.9936 ± 0.0023	0.9961 ± 0.0022
Moving window (w=5)	0.9878 ± 0.0029	0.9933 ± 0.0025	0.9957 ± 0.0023
Moving window (w=7)	0.9874 ± 0.0030	0.9932 ± 0.0027	0.9958 ± 0.0025
Biased window (w=3)	0.9877 ± 0.0029	0.9938 ± 0.0026	0.9964 ± 0.0023
Biased window (w=5)	0.9877 ± 0.0029	0.9937 ± 0.0024	0.9963 ± 0.0022
Biased window (w=7)	0.9876 ± 0.0027	0.9937 ± 0.0024	0.9963 ± 0.0022
Non-niching	0.9870 ± 0.0030	0.9927 ± 0.0026	0.9951 ± 0.0025
Score Selection			
Fixed-length (L=38)	0.9933 ± 0.0019	0.9981 ± 0.0015	0.9995 ± 0.0015
Fixed-length (L=39)	**0.9940 ± 0.0018**	**0.9989 ± 0.0016**	1.0005 ± 0.0015
Fixed-length (L=40)	0.9935 ± 0.0020	**0.9991 ± 0.0017**	**1.0008 ± 0.0016**
Fixed-length (L=41)	0.9933 ± 0.0020	**0.9989 ± 0.0017**	**1.0007 ± 0.0016**
Fixed-length (L=42)	0.9920 ± 0.0019	0.9980 ± 0.0016	1.0000 ± 0.0016
Static window (L=38-40)	0.9918 ± 0.0022	0.9976 ± 0.0018	0.9997 ± 0.0016
Static window (L=38-41)	0.9912 ± 0.0022	0.9972 ± 0.0018	0.9995 ± 0.0017
Static window (L=39-40)	0.9929 ± 0.0019	0.9984 ± 0.0017	1.0004 ± 0.0016
Static window (L=39-41)	0.9920 ± 0.0022	0.9978 ± 0.0018	1.0001 ± 0.0017
Static window (L=40-41)	0.9926 ± 0.0021	0.9983 ± 0.0019	1.0002 ± 0.0017
Static window (L=40-42)	0.9920 ± 0.0025	0.9980 ± 0.0019	1.0000 ± 0.0017
Moving window (w=3)	0.9894 ± 0.0023	0.9975 ± 0.0018	1.0001 ± 0.0016
Moving window (w=5)	0.9891 ± 0.0023	0.9970 ± 0.0019	0.9995 ± 0.0017
Moving window (w=7)	0.9888 ± 0.0024	0.9968 ± 0.0020	0.9992 ± 0.0017
Biased window (w=3)	0.9890 ± 0.0025	0.9976 ± 0.0018	1.0000 ± 0.0015
Biased window (w=5)	0.9885 ± 0.0025	0.9969 ± 0.0019	0.9993 ± 0.0017
Biased window (w=7)	0.9881 ± 0.0027	0.9966 ± 0.0021	0.9992 ± 0.0017
Non-niching	0.9916 ± 0.0025	**0.9988 ± 0.0019**	**1.0010 ± 0.0016**
Helper Objective			
Helper objective (w=3)	0.9888 ± 0.0029	0.9936 ± 0.0027	0.9958 ± 0.0025
Helper objective (w=5)	0.9886 ± 0.0029	0.9939 ± 0.0025	0.9962 ± 0.0023
Helper objective (w=7)	0.9891 ± 0.0027	0.9941 ± 0.0023	0.9963 ± 0.0022

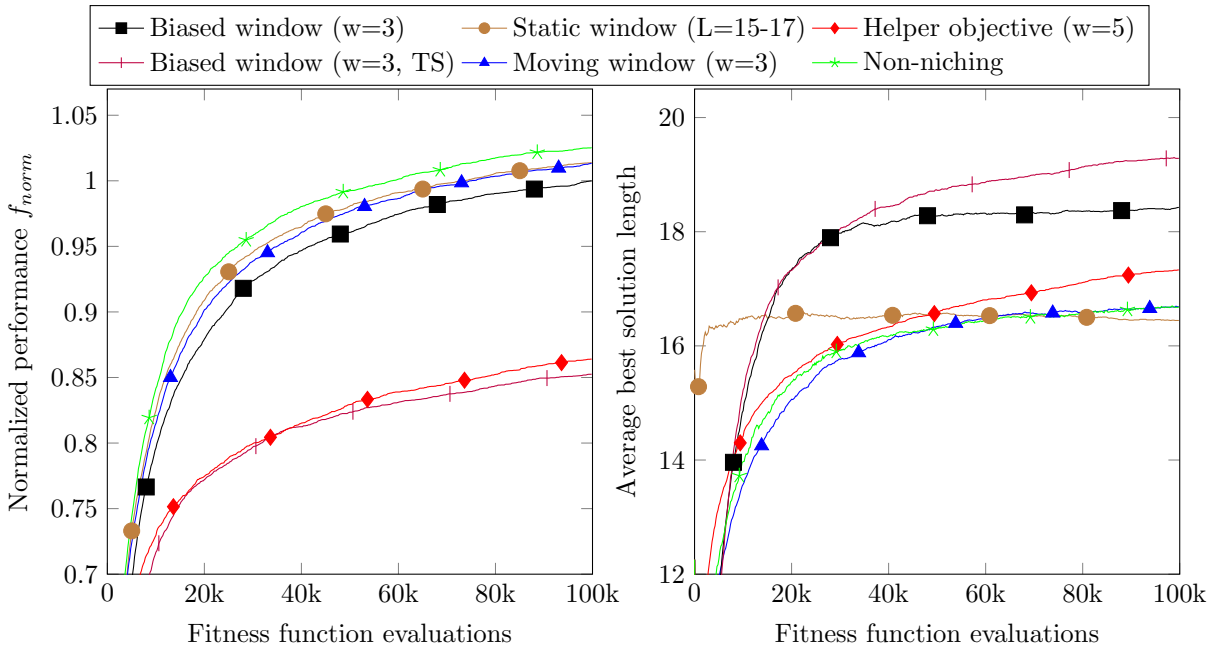


Figure 7.7: Normalized performance and average best solution length for the packing problem using various selection operators. Studies marked 'TS' use tournament selection, the remaining studies use score selection with the exception of the helper objective.

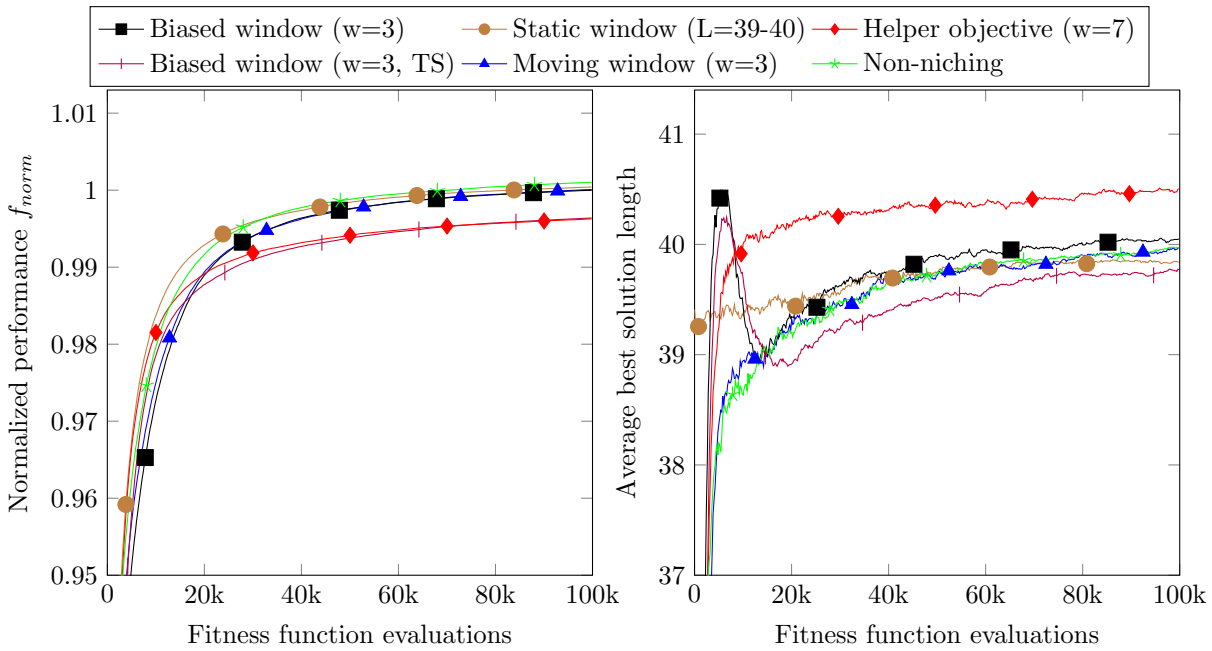


Figure 7.8: Normalized performance and average best solution length for the wind farm problem using various selection operators. Studies marked 'TS' use tournament selection, the remaining studies use score selection with the exception of the helper objective.

Table 7.5: Normalized performance for the portfolio problem using various selection operators.

	Fitness function evaluations		
	10k	25k	50k
Tournament Selection			
Fixed-length (L=7)	0.9754 ± 0.0139	0.9826 ± 0.0107	0.9852 ± 0.0090
Fixed-length (L=8)	0.9810 ± 0.0110	0.9886 ± 0.0069	0.9913 ± 0.0041
Fixed-length (L=9)	0.9822 ± 0.0096	0.9900 ± 0.0060	0.9932 ± 0.0055
Fixed-length (L=10)	0.9827 ± 0.0070	0.9903 ± 0.0054	0.9935 ± 0.0046
Static window (L=6-9)	0.9838 ± 0.0067	0.9903 ± 0.0046	0.9935 ± 0.0038
Static window (L=7-9)	0.9837 ± 0.0068	0.9901 ± 0.0048	0.9942 ± 0.0040
Static window (L=8-9)	0.9838 ± 0.0070	0.9905 ± 0.0052	0.9938 ± 0.0044
Static window (L=8-10)	0.9847 ± 0.0066	0.9918 ± 0.0046	0.9954 ± 0.0041
Static window (L=9-10)	0.9842 ± 0.0067	0.9916 ± 0.0051	0.9950 ± 0.0045
Static window (L=9-11)	0.9844 ± 0.0061	0.9911 ± 0.0053	0.9944 ± 0.0048
Moving window (w=3)	0.9832 ± 0.0069	0.9898 ± 0.0046	0.9931 ± 0.0037
Moving window (w=5)	0.9827 ± 0.0064	0.9898 ± 0.0043	0.9930 ± 0.0037
Moving window (w=7)	0.9834 ± 0.0064	0.9903 ± 0.0041	0.9934 ± 0.0040
Biased window (w=3)	0.9783 ± 0.0099	0.9880 ± 0.0060	0.9923 ± 0.0043
Biased window (w=5)	0.9780 ± 0.0101	0.9877 ± 0.0055	0.9922 ± 0.0049
Biased window (w=7)	0.9789 ± 0.0075	0.9876 ± 0.0056	0.9924 ± 0.0047
Non-niching	0.9833 ± 0.0067	0.9897 ± 0.0046	0.9930 ± 0.0037
Score Selection			
Fixed-length (L=7)	0.9854 ± 0.0027	0.9919 ± 0.0016	0.9933 ± 0.0011
Fixed-length (L=8)	0.9862 ± 0.0031	0.9954 ± 0.0032	0.9977 ± 0.0022
Fixed-length (L=9)	0.9868 ± 0.0035	0.9957 ± 0.0027	0.9980 ± 0.0026
Fixed-length (L=10)	0.9876 ± 0.0030	0.9949 ± 0.0022	0.9962 ± 0.0017
Static window (L=6-9)	**0.9906 ± 0.0036**	0.9978 ± 0.0010	0.9995 ± 0.0006
Static window (L=7-9)	**0.9906 ± 0.0033**	0.9983 ± 0.0008	0.9998 ± 0.0003
Static window (L=8-9)	0.9898 ± 0.0033	0.9984 ± 0.0008	1.0000 ± 0.0003
Static window (L=8-10)	**0.9908 ± 0.0032**	**0.9987 ± 0.0008**	**1.0000 ± 0.0002**
Static window (L=9-10)	0.9899 ± 0.0029	**0.9987 ± 0.0011**	**1.0000 ± 0.0005**
Static window (L=9-11)	**0.9904 ± 0.0030**	**0.9989 ± 0.0011**	**1.0000 ± 0.0003**
Moving window (w=3)	0.9869 ± 0.0038	0.9979 ± 0.0011	0.9998 ± 0.0006
Moving window (w=5)	0.9882 ± 0.0040	0.9977 ± 0.0012	0.9994 ± 0.0007
Moving window (w=7)	0.9880 ± 0.0040	0.9974 ± 0.0011	0.9990 ± 0.0009
Biased window (w=3)	0.9846 ± 0.0063	0.9973 ± 0.0015	0.9996 ± 0.0006
Biased window (w=5)	0.9843 ± 0.0064	0.9973 ± 0.0011	0.9992 ± 0.0008
Biased window (w=7)	0.9863 ± 0.0051	0.9970 ± 0.0013	0.9990 ± 0.0008
Non-niching	0.9869 ± 0.0032	0.9985 ± 0.0009	**1.0000 ± 0.0004**
Helper Objective			
Helper objective (w=3)	0.9793 ± 0.0082	0.9851 ± 0.0063	0.9880 ± 0.0053
Helper objective (w=5)	0.9776 ± 0.0121	0.9830 ± 0.0091	0.9862 ± 0.0073
Helper objective (w=7)	0.9660 ± 0.0779	0.9665 ± 0.0981	0.9628 ± 0.1187

Table 7.6: Normalized performance for the laminate composite problem using various selection operators. Values in parentheses indicate the fraction of trials with feasible solutions, fractions are not shown if all trials have found feasible solutions.

	Fitness function evaluations		
	10k	25k	50k
Tournament Selection			
Fixed-length (L=16)	(0.00) 0.9972 ± 0.0003	(0.00) 0.9972 ± 0.0000	(0.00) 0.9972 ± 0.0000
Fixed-length (L=17)	0.9956 ± 0.0022	0.9981 ± 0.0013	0.9988 ± 0.0012
Fixed-length (L=18)	0.9723 ± 0.0022	0.9748 ± 0.0014	0.9758 ± 0.0011
Fixed-length (L=19)	0.9354 ± 0.0017	0.9373 ± 0.0011	0.9381 ± 0.0008
Fixed-length (L=20)	0.8930 ± 0.0014	0.8942 ± 0.0005	0.8943 ± 0.0001
Static window (L=15-17)	0.9885 ± 0.0041	0.9929 ± 0.0030	0.9952 ± 0.0023
Static window (L=16-17)	0.9908 ± 0.0039	0.9947 ± 0.0025	0.9963 ± 0.0017
Static window (L=16-18)	0.9921 ± 0.0040	0.9959 ± 0.0027	0.9974 ± 0.0018
Static window (L=16-19)	0.9917 ± 0.0045	0.9959 ± 0.0028	0.9976 ± 0.0018
Static window (L=17-18)	0.9929 ± 0.0035	0.9967 ± 0.0021	0.9983 ± 0.0015
Static window (L=17-19)	0.9902 ± 0.0046	0.9953 ± 0.0027	0.9975 ± 0.0017
Moving window (w=3)	0.9925 ± 0.0040	0.9962 ± 0.0028	0.9978 ± 0.0017
Moving window (w=5)	0.9903 ± 0.0047	0.9950 ± 0.0031	0.9967 ± 0.0022
Moving window (w=7)	0.9883 ± 0.0057	0.9932 ± 0.0041	0.9958 ± 0.0026
Biased window (w=3)	0.9874 ± 0.0050	0.9950 ± 0.0028	0.9972 ± 0.0017
Biased window (w=5)	0.9872 ± 0.0057	0.9944 ± 0.0034	0.9967 ± 0.0021
Biased window (w=7)	0.9875 ± 0.0052	0.9929 ± 0.0039	0.9955 ± 0.0030
Non-niching	0.8915 ± 0.0304	0.8934 ± 0.0305	0.8937 ± 0.0307
Score Selection			
Fixed-length (L=16)	(0.00) 0.9972 ± 0.0000	(0.00) 0.9972 ± 0.0001	(0.00) 0.9972 ± 0.0000
Fixed-length (L=17)	**0.9978 ± 0.0010**	**0.9997 ± 0.0009**	**1.0000 ± 0.0006**
Fixed-length (L=18)	0.9739 ± 0.0011	0.9765 ± 0.0003	0.9765 ± 0.0000
Fixed-length (L=19)	0.9359 ± 0.0011	0.9385 ± 0.0002	0.9385 ± 0.0000
Fixed-length (L=20)	0.8932 ± 0.0010	0.8952 ± 0.0010	0.8961 ± 0.0007
Static window (L=15-17)	0.9935 ± 0.0028	0.9974 ± 0.0014	0.9988 ± 0.0012
Static window (L=16-17)	0.9956 ± 0.0018	0.9984 ± 0.0012	0.9993 ± 0.0010
Static window (L=16-18)	0.9949 ± 0.0022	0.9988 ± 0.0012	0.9999 ± 0.0008
Static window (L=16-19)	0.9931 ± 0.0030	0.9978 ± 0.0014	0.9997 ± 0.0010
Static window (L=17-18)	0.9957 ± 0.0018	0.9994 ± 0.0010	**1.0000 ± 0.0006**
Static window (L=17-19)	0.9925 ± 0.0029	0.9981 ± 0.0013	0.9998 ± 0.0008
Moving window (w=3)	0.9894 ± 0.0049	0.9972 ± 0.0015	0.9997 ± 0.0009
Moving window (w=5)	0.9880 ± 0.0055	0.9960 ± 0.0023	0.9989 ± 0.0012
Moving window (w=7)	0.9878 ± 0.0053	0.9949 ± 0.0027	0.9981 ± 0.0015
Biased window (w=3)	0.9886 ± 0.0042	0.9975 ± 0.0014	0.9998 ± 0.0009
Biased window (w=5)	0.9883 ± 0.0047	0.9959 ± 0.0021	0.9990 ± 0.0014
Biased window (w=7)	0.9884 ± 0.0050	0.9951 ± 0.0025	0.9981 ± 0.0015
Non-niching	0.9919 ± 0.0034	0.9981 ± 0.0016	**0.9999 ± 0.0011**
Helper Objective			
Helper objective (w=3)	0.8906 ± 0.0362	0.8923 ± 0.0365	0.8926 ± 0.0367
Helper objective (w=5)	0.8912 ± 0.0346	0.8932 ± 0.0345	0.8936 ± 0.0348
Helper objective (w=7)	0.8886 ± 0.0344	0.8902 ± 0.0347	0.8906 ± 0.0350

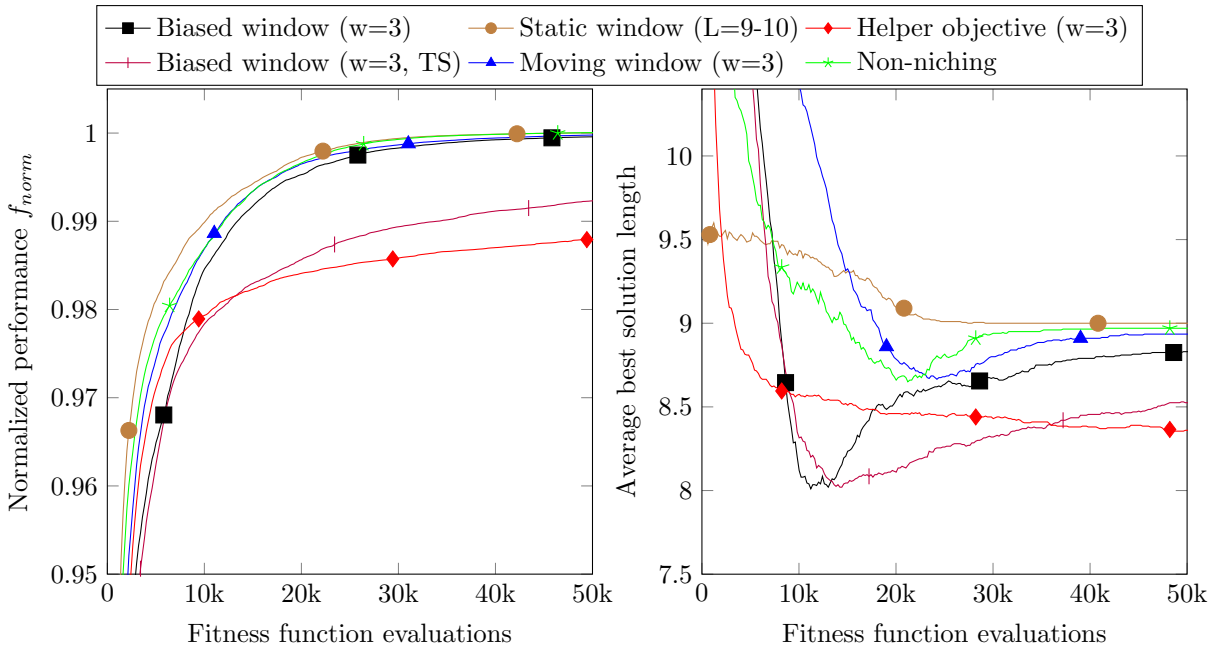


Figure 7.9: Normalized performance and average best solution length for the portfolio problem using various selection operators. Studies marked 'TS' use tournament selection, the remaining studies use score selection with the exception of the helper objective.

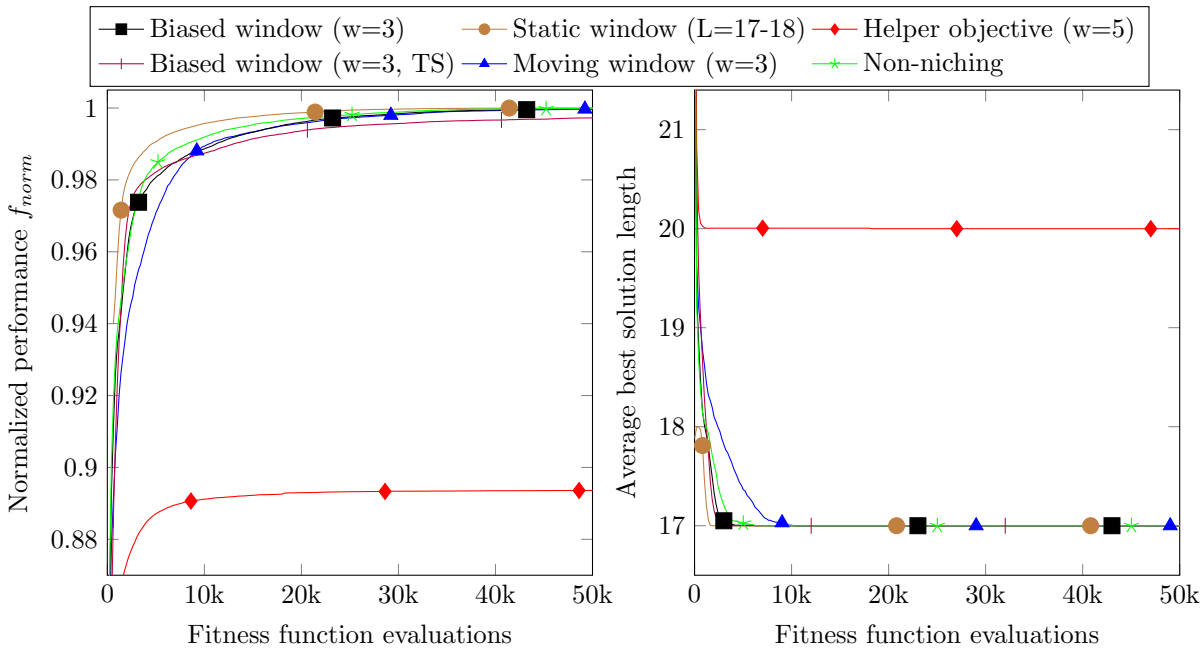


Figure 7.10: Normalized performance and average best solution length for the laminate composite problem using various selection operators. Studies marked 'TS' use tournament selection, the remaining studies use score selection with the exception of the helper objective.

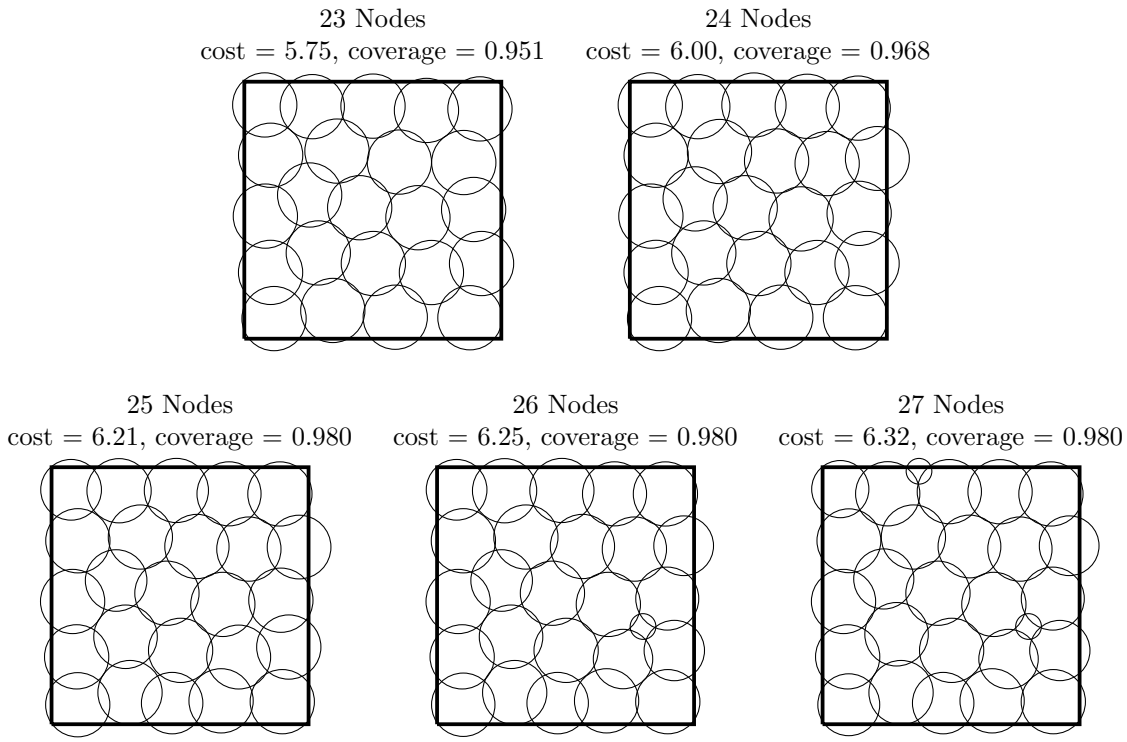


Figure 7.11: Sample solutions from all lengths present in the final population of a trial solving the coverage problem using length niching with tournament selection.

The deleterious effect of length-varying operators differs between problems. For several problems length niching is not necessary to reach optimal solution lengths. The non-niching, score selection algorithm performs very well for the packing, wind farm, portfolio, and laminate composite problems. However it fails to consistently reach optimal lengths for the coverage problem. While score selection does not explicitly consider solution length it is possible that solutions of varying lengths are selected. However, it is apparent that for some problems this is not sufficient and length niching is required.

When non-niching, score selection reaches optimal solution lengths it generally outperforms length niching selection. This may be explained by different balances between exploration and exploitation among the algorithms. Length niching selection is generally more exploratory, maintaining a roughly equal number of solutions in several different niches. The non-niching score selection algorithm may maintain some diversity in solution length, however it is likely to be less than that of length niching. By focusing primarily on solutions of a particular length it is able to perform slightly better on some problems.

A similar observation can be made when comparing the performance of biased window functions using various widths. The smaller widths tend to give the best performance. The wider widths are likely spreading the computational effort too thinly across many niches, failing to fully refine solutions of the optimal length. On the coverage problems, the moving window function performs better with wider window widths, this is

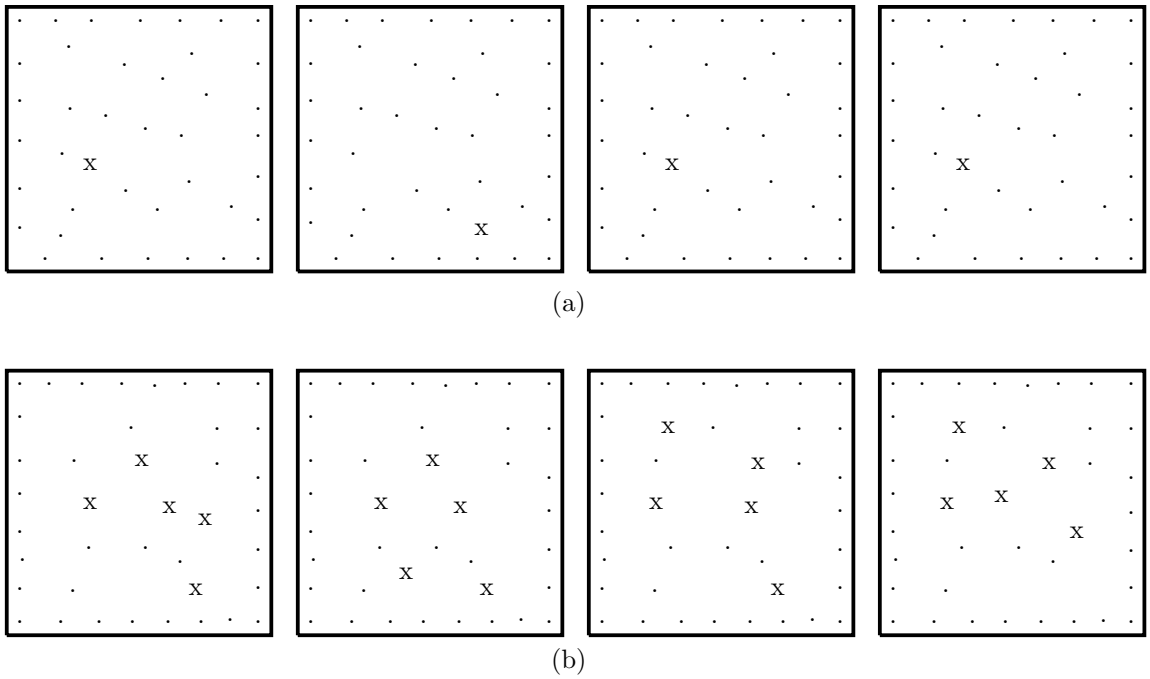


Figure 7.12: Sample solutions from within a single length niche using (a) tournament selection and (b) score selection. Solutions are for the wind farm problem and sampled after 40k evaluations. Turbines that are common to all solutions in that niche, within a small tolerance, are denoted by a '.', and the remaining turbines denoted by an 'x'.

a result of the shorter window widths failing to consistently reach optimal lengths.

The fixed-length operator performed very poorly when using tournament selection. This is likely due to the lack of diversity in the population, performance is significantly improved when using score selection. The static window function had similar overall performance to the biased window function. With the exception of the coverage problems the static window and fixed-length algorithms generally outperformed the biased window function during the early stages of the algorithm. Since these algorithms assume the optimal length is known *a priori* they do not have to spend the early generations searching solutions at non-optimal lengths.

The poor performance of the helper objective is due to two reasons. First, as discussed in Section 7.1, when difficult to satisfy constraints are present this operator has a strong tendency to converge to suboptimal lengths. This is very apparent on the constrained coverage and laminate composite problems. On problems where it does reach optimal lengths its performance is comparable to length niching selection using tournament selection. Score selection gave significantly better results, but it is unclear how the score selection methodology could be incorporated into the helper objective operator.

7.6 Score Selection Applied to Non-Metameric Problems

Score selection was developed for, but not limited to, metameric problems. This section demonstrates score selection on a set of non-metameric problems. A set of single-objective, constrained optimization benchmark problems were used in a competition of evolutionary algorithms at the 2017 and 2018 IEEE Congress on Evolutionary Computation [145]. There are 28 problems in this set, each using one or more equality constraints, inequality constraints, or both. All problems are scalable, the competition was performed for each problem using 10, 30, 50, and 100 design variables.

The non-metameric algorithm used for this demonstration is detailed in Section 7.6.1. Section 7.6.2 discusses a few small modifications made to score selection for these problems. The results, and a comparison to the best performing algorithms from the previous competition, are given in Section 7.6.3.

7.6.1 Non-Metameric Evolutionary Algorithm

The framework of the non-metameric evolutionary algorithm is the same as the one given in Section 4.1. A population size of 20 is used. Each study is run for $DIM * 20,000$ evaluations, where DIM is the dimensionality of the problem. This is the number of evaluations allowed by the competition.

Simplex crossover is used [138] with a crossover rate of 0.5. Three parents are used here for each crossover, and the expanding rate control parameter is set to 3. Random Gaussian mutation is used. When the problem dimensionality is 10, a mutation rate of 0.1 is used, for all other dimensionalities a rate of 0.05 is used. Mutation magnitude is selected from one of several possibilities, as shown in (7.12) where r is a randomly chosen value in the interval $[0, 1]$. A new magnitude is selected for each solution undergoing mutation.

$$\text{mutation magnitude} = \begin{cases} D_{mean} & \text{if } 0 \leq r \leq 0.45 \\ D_{mean} * 10^{1-4(G/G_{max})^2} & \text{if } 0.45 < r \leq 0.90 \\ 0.05 & \text{if } 0.90 < r \leq 1 \end{cases} \tag{7.12}$$

The first possible magnitude is based on D_{mean}, the average distance between all solutions in the parent population. The second magnitude scales D_{mean} by values greater than 1 in early generations, promoting exploration. In later generations it scales D_{mean} by values less than 1, promoting exploitation. The final possible mutation magnitude is a fixed value, helping to promote exploration at all phases of the algorithm. The design variables are then mutated as described in Section 6.2. It is possible that some mutations may be small enough that they cannot be captured by a double-precision number. When this happens, the design variables are mutated to the next double-precision number in the direction of mutation.

The feasible regions of some of the benchmark problems are extremely small and difficult to locate. To assist the EA in locating these regions, a dynamic constraint tolerance δ_G is calculated at each generation

G. This is adapted from the dynamic mechanism used in [89, 140].

$$\delta_G = \begin{cases} \delta_1 * \left(\frac{\text{1e-4}}{\delta_1}\right)^{G/0.7G_{max}} & \text{if } G < 0.7G_{max} \\ 0 & \text{otherwise} \end{cases} \tag{7.13}$$

Equation (7.14) gives the initial value δ_1, calculated using the average constraint violation in the initial population P_1.

$$\delta_1 = 10 * \frac{1}{N} \sum_{\mathbf{x} \in P_1} \phi(\mathbf{x}) \tag{7.14}$$

Equation (7.15) gives a modified constraint violation, $\phi'(\mathbf{x})$, used by score selection. This is recalculated every generation for each solution using the updated δ_G. Note that $\phi'(\mathbf{x}) = \phi(\mathbf{x})$ in later generations, when δ_G is set to 0. The relaxation of the equality constraints σ is set to 1e-4, as used in the competition.

$$\phi'(\mathbf{x}) = \max\left(0, \sum_{i=1}^{p} \max(0, g_i(\mathbf{x})) + \sum_{j=1}^{q} \max(0, |h_j(\mathbf{x})| - \sigma) - \delta_G\right) \tag{7.15}$$

7.6.2 Modifications to Score Selection

A few modifications are made to score selection. The constraint violation score is calculated using $\phi'(\mathbf{x})$. This section also proposes modified values for the measure of objective value trend and the balance term. These values are used in place of their standard values proposed in Section 7.3.2.3.

In metameric problems the best solution objective value f^*_G will monotonically improve once a feasible solution is found. When using the dynamic constraint tolerance this is no longer the case, feasible solutions may become infeasible as the value of δ_G decreases. This invalidates the measure of objective value trend F_G proposed in Section 7.3.2.3. Equation (7.16) proposes an alternative measure F'_G, which is the range of f^*_G values observed over the previous τ generations. Here we set $\tau = 0.025G_{max}$.

$$F'_G = \max_{g \in [G-\tau, G]} f^*_g - \min_{g \in [G-\tau, G]} f^*_g \tag{7.16}$$

Second, a modified balance term γ' is defined in (7.17). For most of the study this is simply equal to γ. In later generations γ' will begin to increase in value, decreasing the pressure toward a diverse population. This results in a greater level of exploitation in later generations, encouraging the algorithm to converge and refine the best-so-far solution.

$$\gamma' = \gamma * \begin{cases} 1 & \text{if } G \leq 0.7G_{max} \\ 4 * (G - 0.7G_{max})/(0.3G_{max}) & \text{otherwise} \end{cases} \tag{7.17}$$

7.6.3 Results and Discussion

The non-metameric evolutionary algorithm was run 25 times for each benchmark problem at each dimensionality. These results are compared to the three algorithms that were submitted to the 2018 competition: matrix adaptation evolutionary strategy (MA-ES) [60], improved unified differential evolutionary algorithm

(IUDE) [136], and LSHADE44 with an improved ϵ constraint handling method (LSHADE44-IEpsilon) [36]. The following criteria are used, in the presented order, to compare algorithms. All objective and constraint violation values are rounded to 6 significant digits when comparing.

1. The fraction of trials with a feasible solution.
2. The average constraint violation of the best solution from each trial.
3. The average objective function value of the best solution from each trial.

Table 7.7 gives the results for problems using 10 or 30 variables, and Table 7.8 for 50 or 100 variables. The proposed algorithm using score selection was found to produce the best results, including ties, on 45 of the problems. LSHADE44-IEpsilon, MA-ES, and IUDE respectively produced the best results for 24, 43, and 32 problems.

On a number of problems there is not a large difference between the proposed algorithm and the best observed results. This is the case for some dimensionalities of problems 1, 2, 5, 8, 9, 10, 19, and 23. It is possible that improving these results may require improved crossover and mutation operators. Regardless, the results demonstrate that score selection may have applications beyond metameric problems.

Table 7.7: Results on non-metameric benchmarks for problem dimensonalities of 10 and 30. If all trials produced a feasible solution then mean objective values are shown. Otherwise, the proportion of trials with a feasible solution is shown in parenthesis. If all algorithms produced no feasible solutions then the mean constraint violation is also shown. The best performing algorthm(s) are highlighted for each problem.

	DIM = 10				DIM = 30			
#	Score Selection	LSHADE44-IEpsilon	MA-ES	IUDE	Score Selection	LSHADE44-IEpsilon	MA-ES	IUDE
1	**0.00**	**0.00**	1.65e-30	**0.00**	3.07e-11	**5.70e-30**	3.75e-28	4.13e-29
2	**0.00**	**0.00**	**0.00**	**0.00**	2.56e-11	**6.06e-30**	3.76e-28	4.42e-29
3	**0.00**	59.3	4.73e-31	35.4	6.03e-11	7192	**6.73e-28**	129
4	13.7	13.6	29.8	**2.90**	14.3	**13.6**	70.3	**13.6**
5	1.32e-11	**0.00**	**0.00**	1.74e-30	1.10e-20	6.56e-27	**0.00**	5.71e-29
6	**0.00**	(0.60)	35.8	**0.00**	(0.84)	(0)	**180**	(0.12)
7	**-421**	-121	-317	(0.44)	**-1242**	-277	-701	(0.76)
8	**-1.35e-3**	**-1.35e-3**	**-1.35e-3**	**-1.35e-3**	-2.03e-4	-2.84e-4	**-2.84e-4**	**-2.84e-4**
9	-4.97e-3	**-4.98e-3**	**-4.98e-3**	**-4.98e-3**	-2.65e-3	**-2.67e-3**	**-2.67e-3**	**-2.67e-3**
10	-5.10e-4	**-5.10e-4**	**-5.10e-4**	**-5.10e-4**	-5.98e-5	**-1.03e-4**	**-1.03e-4**	**-1.03e-4**
11	-1.69e-1	-1.69e-1	-1.68e-1	**-8.01e-1**	(0)	(0)	**-9.25e-1**	(0.96)
12	3.99	**3.99**	7.00	3.99	**3.98**	3.99	46.1	3.98
13	5.10e-28	2.82	1.59e-1	**0.00**	1.44e-6	51.6	**2.89e-27**	3.54
14	**2.38**	2.39	(0.72)	**2.38**	**1.41**	1.41	1.63	**1.41**
15	**4.62**	(0.80)	(0.28)	6.38	6.00	**5.00**	(0.28)	5.87
16	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	6.28	**0.00**	1.57
17	**(0) 4.50**	(0) 4.50	(0) 5.85	(0) 4.54	**(0) 15.1**	(0) 15.5	(0) 15.5	(0) 15.3
18	36.6	36.7	**36.6**	(0)	**36.5**	36.5	36.5	(0)
19	(0) 6634	**(0) 6634**	(0) 6635	**(0) 6634**	(0) 2.14e4	**(0) 2.14e4**	(0) 2.14e4	**(0) 2.14e4**
20	**1.25e-1**	6.18e-1	1.17	6.99e-1	**9.95e-1**	1.93	7.66	3.89
21	6.54	3.99	4.41	**3.99**	13.1	**10.6**	48.4	15.6
22	**3.19e-1**	1.12	6.38e-1	3.14	16.8	3731	**2.47e-25**	19.6
23	2.38	2.39	(0.96)	**2.38**	1.41	**1.41**	1.65	1.43
24	**2.36**	(0.84)	6.13	5.50	2.73	6.38	9.14	**2.48**
25	**0.00**	1.01	**0.00**	**0.00**	5.09	25.5	**0.00**	8.73
26	(0) 5.02	(0) 5.02	(0) 5.42	**(0) 4.79**	(0) 15.5	(0) 15.5	(0) 15.5	**(0) 15.5**
27	**35.6**	36.6	75.9	(0)	(0.96)	38.8	**36.6**	(0)
28	**(0) 6637**	(0) 6654	(0) 6640	(0) 6637	(0) 2.14e4	(0) 2.15e4	**(0) 2.14e4**	(0) 2.15e4

Table 7.8: Results on non-metameric benchmarks for problem dimensonalities of 50 and 100. If all trials produced a feasible solution then mean objective values are shown. Otherwise, the proportion of trials with a feasible solution is shown in parenthesis. If all algorithms produced no feasible solutions then the mean constraint violation is also shown. The best performing algorthm(s) are highlighted for each problem.

	DIM = 50				DIM = 100			
#	Score Selection	LSHADE44-IEpsilon	MA-ES	IUDE	Score Selection	LSHADE44-IEpsilon	MA-ES	IUDE
1	1.55e-6	1.62e-24	**2.87e-27**	3.85e-24	1.34e-1	1.90e-7	**3.98e-26**	9.17e-25
2	2.08e-6	6.54e-25	**2.89e-27**	5.09e-24	1.15e-1	1.03e-7	**3.98e-26**	5.28e-25
3	8.64e-8	2.43e4	**4.06e-27**	640	603	1.66e5	**4.43e-26**	(0.92)
4	15.1	**13.6**	119	72.1	40.2	**14.4**	251	283
5	2.37e-8	3.19e-1	**0.00**	9.19e-28	28.4	4.40	26.9	**4.27**
6	(0.88)	(0)	**287**	(0.08)	**541**	(0)	806	(0)
7	**-1927**	-322	-1375	(0.60)	**-2993**	-477	-2482	-276
8	1.82e-5	-1.10e-4	**-1.35e-4**	-5.57e-5	4.35e-4	1.68e-3	**-4.55e-5**	9.20e-5
9	-2.02e-3	-1.74e-3	6.66e-1	**-2.04e-3**	**1.00e-6**	4.18e-1	(0.92)	3.71e-1
10	2.32e-5	-4.79e-5	-4.83e-5	**-4.83e-5**	1.10e-4	1.53e-4	**-1.16e-6**	3.04e-6
11	(0)	(0)	**(0.76)**	(0)	(0)	(0)	(0.96)	**-4.64**
12	**3.98**	27.4	50.6	7.14	**6.11**	18.8	30.1	11.8
13	**4.10**	11.9	295	32.1	**41.0**	67.9	41.4	177
14	**1.10**	1.10	1.34	**1.10**	**7.84e-1**	8.36e-1	9.58e-1	7.87e-1
15	**7.89**	8.76	(0.68)	8.89	**5.75**	16.2	(0.88)	15.2
16	**0.00**	16.8	**0.00**	6.28	1.69e-16	188	**0.00**	96.4
17	**(0) 25.3**	(0) 25.5	(0) 25.5	(0) 25.3	**(0) 50.5**	(0) 50.5	(0) 50.5	(0) 50.5
18	**36.5**	36.8	36.6	(0)	(0.08)	37.9	**36.4**	(0)
19	(0) 3.61e4	**(0) 3.61e4**	(0) 3.61e4	(0) 3.61e4	(0) 7.30e4	**(0) 7.30e4**	(0) 7.31e4	(0) 7.30e4
20	**2.47**	3.50	15.2	8.62	**7.27**	8.83	35.4	17.3
21	26.9	**11.4**	55.3	12.0	33.6	9.34	31.6	**7.32**
22	**40.8**	(0.08)	976	929	(0.96)	(0)	**4231**	(0)
23	**1.10**	1.10	1.34	1.11	7.94e-1	**7.93e-1**	9.67e-1	7.98e-1
24	**2.98**	9.27	12.2	4.37	**8.76**	18.1	9.39	15.7
25	18.7	78.9	**0.00**	7.54	60.9	503	**4.24e-14**	289
26	(0) 25.5	(0) 25.5	(0) 25.5	**(0) 25.5**	(0) 50.5	(0) 50.5	(0) 50.5	**(0) 50.5**
27	(0.92)	40.3	**36.5**	(0)	(0.80)	(0.20)	**(0.96)**	(0)
28	(0) 3.63e4	(0) 3.63e4	**(0) 3.62e4**	(0) 3.63e4	(0) 7.33e4	(0) 7.34e4	**(0) 7.31e4**	(0) 7.34e4

Chapter 8

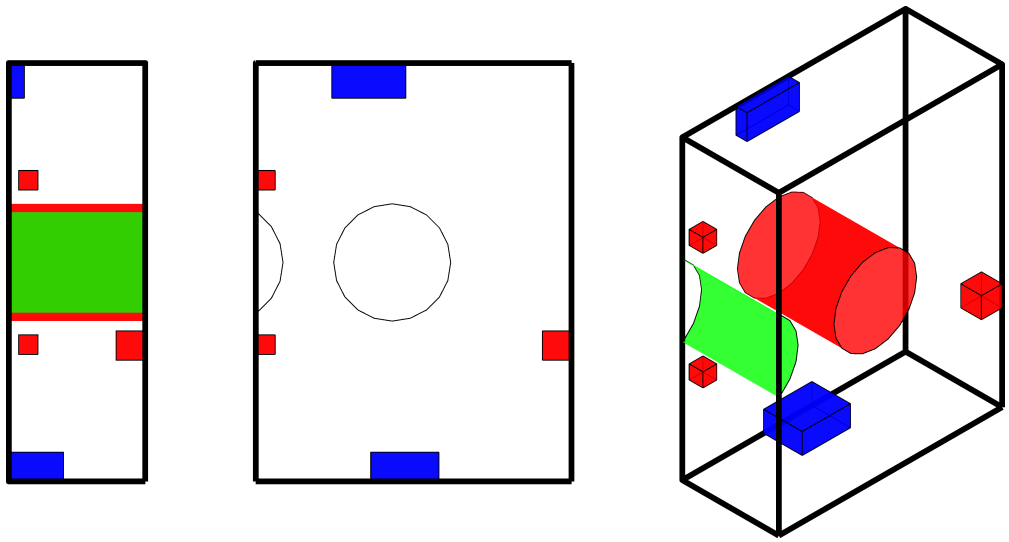
Design of Objects with a Lattice Microstructure

In recent work we used a static-metavariable representation, solved with the commercial software HEEDS, to aid in the design of objects with a lattice microstructure. This chapter describes an alternate metameric representation using a variable-length genome and applies the proposed metameric evolutionary algorithm to the problem. Section 8.1 gives background information, and Section 8.2 describes the optimization problem. Results are given in Section 8.3. It is found that the variable-length genome is able to produce lattices with lower masses than those found using the previous static-metavariable representation.

8.1 Background

Advancements in additive manufacturing is allowing for the fabrication of objects not possible, or impractical, through traditional processes. For example, an object could be made less dense if formed by a lattice microstructure rather than a continuous material. The lattice does not need to be uniform throughout the object. Particular regions could be made denser such that the object can satisfy the given loading conditions. Materials formed by such microstructures can also provide desirable thermal insulation, acoustic absorption, or vibration dampening properties [122].

As the resolution of additive manufacturing continues to improve more complex microstructures become possible. For example, every member in a lattice microstructure may itself be a lattice. If a lattice consists of 10^6 members, each of which is formed by a 10^6 member microlattice, the overall structure will contain 10^{12} members. The complexity of such a structure presents problems at each stage of the design process. Analysis



(a) (b) (c)

Figure 8.1: (a) Side, (b) front, (c) and isometric views of the upright suspension design envelope. The lattice is to be trimmed to the surfaces shown. Loads will be applied to the red surfaces, support surfaces are blue, and no material can be present within the green surface.

of the explicit structure (i.e., every member represented) is not practical. It may not even be possible to explicitly store the complete model.

This is an ongoing project and involves a collaboration of several groups. Each group focuses on a particular aspect of the design process (e.g., geometric representation, analysis, manufacturability). Our group at Michigan State University has been investigating the optimization of such structures. The results shown here are preliminary and do not incorporate all of the eventual tools for this design process. For example, the analysis tools required to evaluate extremely complex structures are not yet available. Instead, the structure is explicitly evaluated. Still, the early results are sufficient to demonstrate the metameric evolutionary algorithm.

One exploratory problem considered in this project is the design of an upright suspension of a race car. The load surfaces, support surfaces, and design envelope for the part is shown in Figure 8.1. The geometry and loading conditions are based on those provided in [88].

Software for procedurally generating the lattice microstructure is provided by a collaborating group. A lattice kernel is defined, formed by several nodes and connecting members. This kernel is repeated in the x-, y-, and z-directions to create a lattice. Certain affine transformations can be applied to each instance of the kernel, allowing for complex lattice geometries to emerge. The lattice is then trimmed to the design envelope shown in Figure 8.1. To avoid applying loads directly to the lattice, small regions of solid material, referred to as inserts, are placed at each load and support surface. The boundary conditions are applied to the inner surface of the inserts, and the lattice is attached to the outer surface.

The parameters controlling the lattice shape, topology, and inserts are not considered in this section. These values are fixed to those found by previous, non-metameric studies of this problem. The resulting lattice consists of approximately 25,000 members. Optimization is performed by altering the local densities of the lattice. It is assumed that direct control over the size of every lattice member is not possible in a procedurally generated lattice. Instead, field functions are employed that control the size of many members in a geometric region simultaneously.

8.2 Optimization Problem

Each node in the lattice is assigned a radius. The radius of each member is calculated as the average radius of the two nodes it connects. Field functions are used to alter the node radii. Let $\psi^{(i)}$ be a field function with a center at $(\psi_x^{(i)}, \psi_y^{(i)}, \psi_z^{(i)})$, a radius of $\psi_{rad}^{(i)}$, and peak value of $\psi_{peak}^{(i)}$. The value of a field function at its center is equal to its peak and linearly decays to 1 at a distance equal to its radius. A field function has a value of 1 at all points outside of its radius.

In this problem each lattice node has a base radius r_{base} of 0.2 mm. This radius is scaled by the maximum field function value at the location of the node. Let P be a lattice node located at (P_x, P_y, P_z). The radius of this node P_{rad} can be calculated by (8.1), where N is the number of field functions present. Each node has a maximum radius r_{max}, calculated such that the minimum aspect ratio (length/diameter) of any member attached to this node will be at least 5. This can vary among nodes, for the given lattice the maximum radius is 0.832 mm for most nodes. An example of how field functions can affect the resulting lattice is given in Figure 8.2.

$$P_{rad} = \min\left(r_{max}, r_{base} * \max_{i=1\ldots N}\left(1, \psi_{peak}^{(i)} - (\psi_{peak}^{(i)} - 1)\frac{\sqrt{(P_x - \psi_x^{(i)})^2 + (P_y - \psi_y^{(i)})^2 + (P_z - \psi_z^{(i)})^2}}{\psi_{rad}^{(i)}}\right)\right) \tag{8.1}$$

Finite element analysis is used to evaluate each design. Each member is represented as a Timoshenko beam element [72]. Tie constraints are used to attach the lattice to the solid material inserts. This results in large stress concentrations in both the inserts and attached members. In the current work, the factor of safety is calculated using only the stresses in members that are not subject to any tie constraints. Only material failure is considered here. Other modes of failure, such as buckling, are not included in the analysis. Both the lattice and inserts are assumed to be made of ULTEM 9085[1].

Two metameric optimization problems are formed for this problem. One uses a variable-length genome where each metavariable represents the position, peak, and radius of a single field function. The other problem

[1]Material properties available at https://www.stratasys.com/materials/search/ultem9085

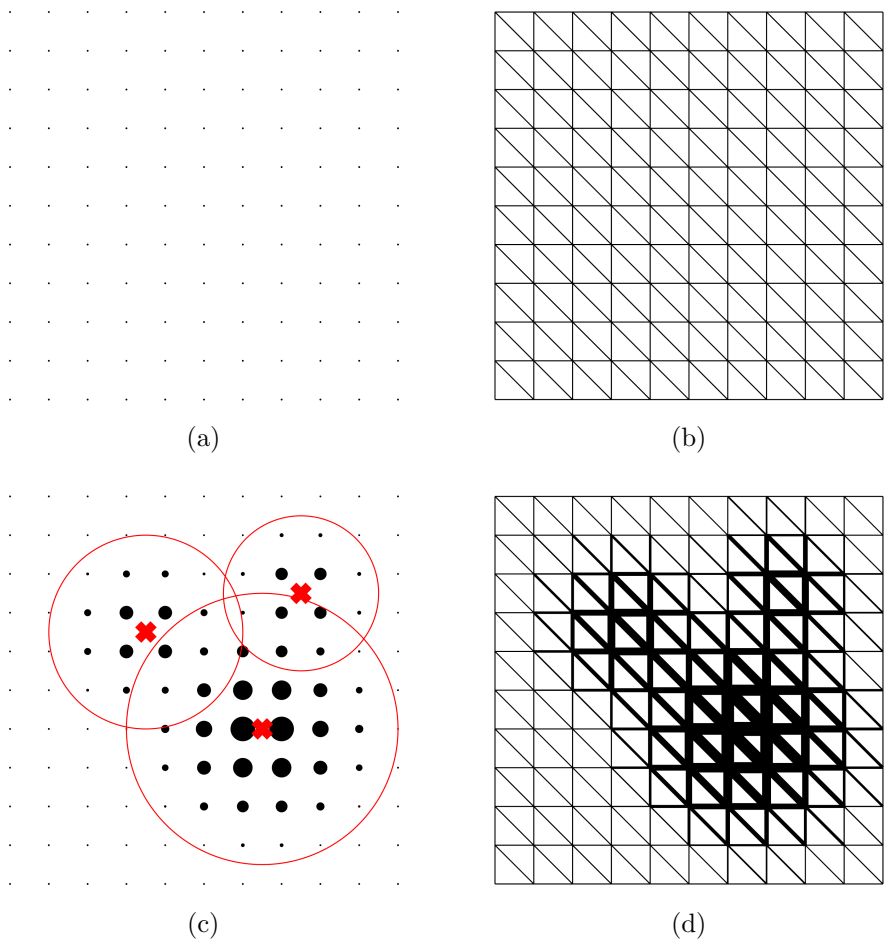


Figure 8.2: An example of field functions applied to a 2-D lattice. (a) Location and initial radius of each node. (b) Resulting lattice given the initial radii. (c) Field function centers are denoted by the thick red 'x', red circles show the radius of each field function. The size of the nodes is proportional to their new radii. (d) Resulting lattice after field function are applied.

uses a static-metavariable representation in which 90 field functions are uniformly spaced throughout the design envelope. Their positions are fixed and contained in the static-genotype. The peak and radius of each are considered design variables, along with the binary flags controlling metavariable expression. In addition to the metameric evolutionary algorithm, the commercial optimization software HEEDS[2] is also used to solve the problem using a static-metavariable representation.

Both optimization problems have the objective of minimizing the resulting mass while achieving a factor of safety of at least 1. It is possible that some field functions may not affect any part of the lattice. This can occur if no nodes are within their radius, or the nodes within their radius are more strongly affected by another field function. As a result bloat is a concern in this problem, to mitigate this a small parsimony pressure (Section 3.2.4.2) is applied to the objective function. The metameric optimization statement when

[2]Information on HEEDS available at https://www.redcedartech.com/index.php/solutions/heeds-software

using a variable-length genome is given in (8.2).

$$\begin{array}{lll} \text{Minimize} & \text{mass}(\mathbf{x}) * (1 + 0.001L(\mathbf{x})) & \mathbf{x} \in \mathbb{R}^{L(\mathbf{x})\times v}, L(\mathbf{x}) \in \mathbb{Z}^{+} \\ \text{Subject to} & \text{factor of safety}(\mathbf{x}) \geq 1.00 & \\ \text{(x-position)} & 0\,\text{m} \leq x_{n,1} \leq 0.162\,\text{m} & \\ \text{(y-position)} & 0\,\text{m} \leq x_{n,2} \leq 0.070\,\text{m} & \\ \text{(z-position)} & 0\,\text{m} \leq x_{n,3} \leq 0.214\,\text{m} & n = 1, 2, ..., L(\mathbf{x}) \\ \text{(peak)} & 10 \leq x_{n,4} \leq 100 & \\ \text{(radius)} & 0.010\,\text{m} \leq x_{n,5} \leq 0.050\,\text{m} & \end{array} \tag{8.2}$$

8.3 Results and Discussion

Each algorithm was only run once due to the cost of evaluating each design. Studies using the metameric evolutionary algorithm (MEA) were run for 100,000 function evaluations, while the HEEDS study was terminated after approximately 40,000. Results can be seen in Figure 8.3. The reported modified mass includes the parsimony term, the actual mass will be 1-2% below these values. HEEDS occasionally reports an increase in modified mass, this is due to small relaxations allowed in the factor of safety constraint.

The best solution was found by the MEA with a variable-length genome, achieving a mass of 0.426 kg. When a static-metavariable representation was used the MEA and HEEDS achieved masses of 0.455 kg and 0.502 kg, respectively. The inserts have a total mass of 0.132 kg for all designs, this is included in the reported masses.

Figure 8.4 shows the optimized field functions, and resulting relative node radii, found by the MEA with a variable-length genome. A total of 9 field functions are used, however only 8 are clearly visible in

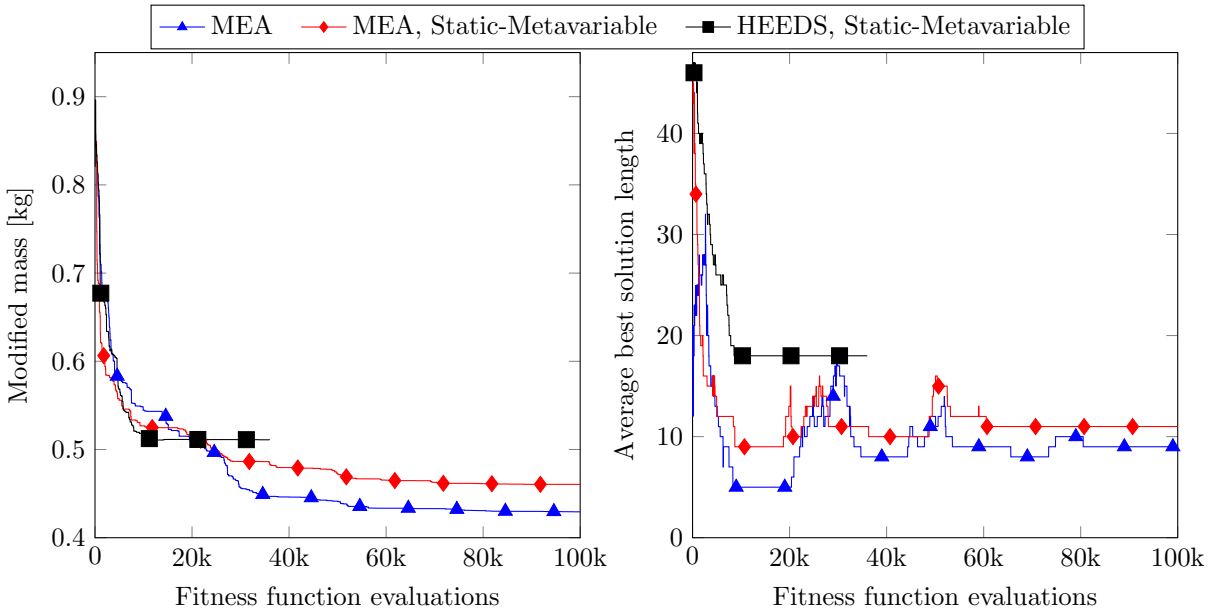
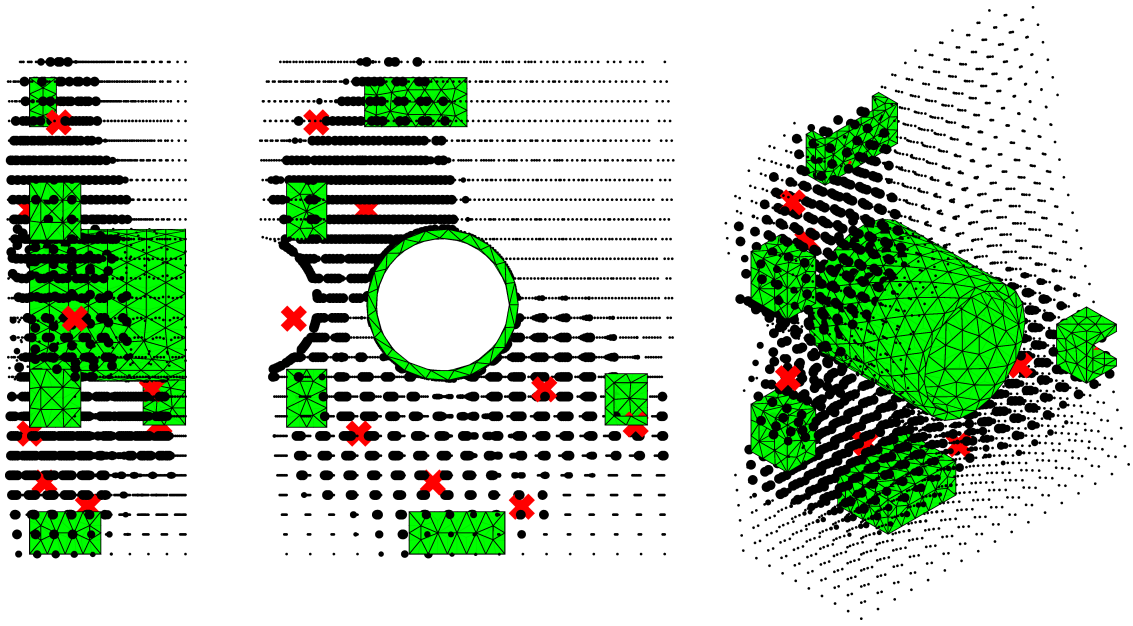


Figure 8.3: Modified mass and average best solution length for the lattice microstructure.

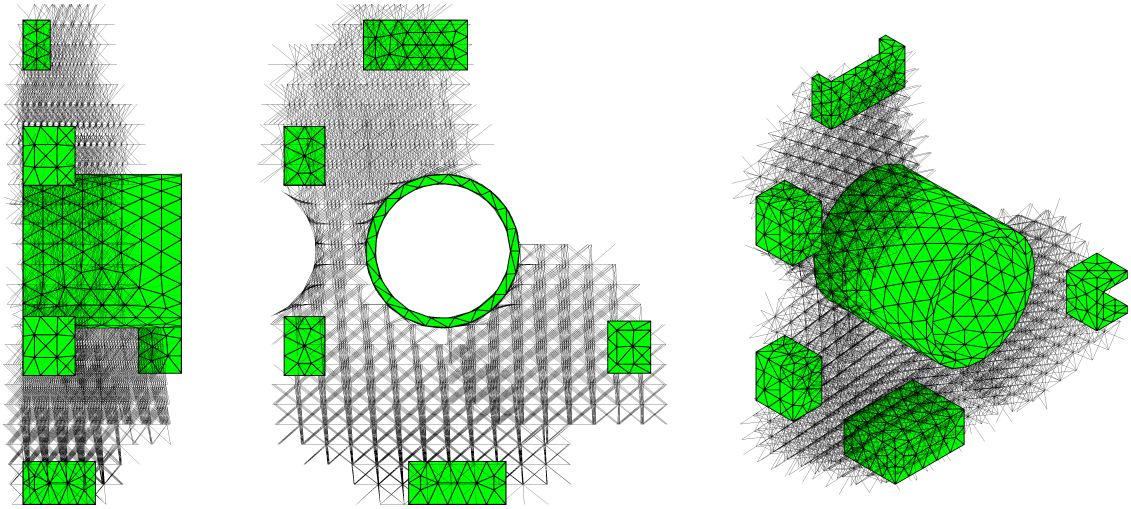
the figure. One field function is located inside of the top support insert. The field functions are located primarily at positions between the load and support surfaces. The resulting lattice is shown in Figure 8.5. For clarity, members with a radius under 0.4 mm are not shown in this figure. The members shown represent approximately 87% of the total lattice mass.



(a) (b) (c)

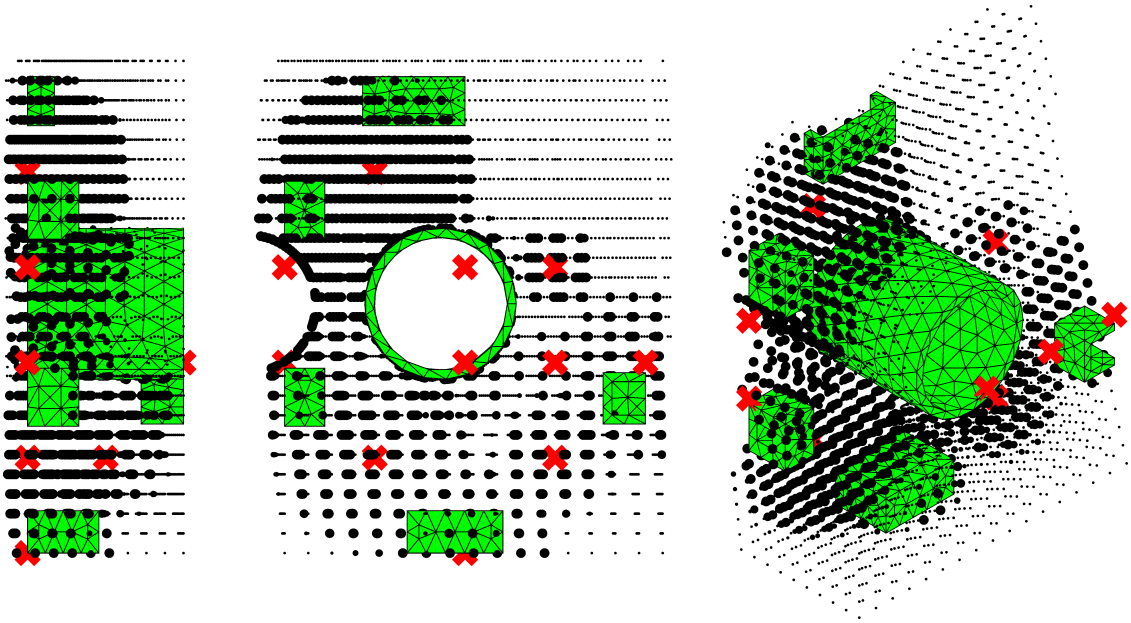
Figure 8.4: (a) Side, (b) front, (c) and isometric views of the relative lattice node radii optimized by the metameric evolutionary algorithm with a variable-length genome. Green material represents the inserts of solid material created at the load and support surfaces. Each field function center is represented by a thick, red 'x'.

The relative node radii and optimized field functions for the MEA using a static-metavariable representation are shown in Figure 8.6. Only the field functions expressed in the phenotype are shown here. Note the grid-like spacing of possible field function locations. The static-metavariable representation reduces the size of the search space, but in doing so it eliminates more optimal solutions. This results in a nearly 10% increase in mass of the lattice (i.e., not including the inserts) compared to the solutions found by the variable-length genome.



(a) (b) (c)

Figure 8.5: (a) Side, (b) front, (c) and isometric views of the lattice members optimized by the metameric evolutionary algorithm with a variable-length genome. The members are not scaled to size here, only members with a radius greater than 0.4 mm are shown.



(a) (b) (c)

Figure 8.6: (a) Side, (b) front, (c) and isometric views of the relative lattice node radii optimized by the metameric evolutionary algorithm with a static-metavariable representation. Green material represents the inserts of solid material created at the load and support surfaces. Each field function center is represented by a thick, red 'x'.

Chapter 9

Summary, Conclusions, and Future Work

9.1 Summary and Conclusions

Chapter 2 gave detailed definitions for metameric representations, problems, and algorithms. A metameric representation is one in which the genome is segmented into a number of metavariables. Metameric problems are optimization problems that encode their solutions using a metameric representation. It is noted that alternative, non-metameric representations also exist for many problems. Metameric algorithms are ones which have been designed or modified to take advantage of the metameric representation.

A survey of literature using metameric representations was performed in Chapter 3. Brief descriptions of the application of metameric representations to a number of problems are provided. Studies are grouped depending on their general methodology, including the encoding, variational, and selection operators. Despite strong similarities in their approaches, no commonly accepted metameric algorithm has emerged. By bringing greater recognition to the metameric class of problems a greater level of knowledge dissemination could be achieved.

Chapter 4 describes the general framework of our proposed metameric evolutionary algorithm. Several benchmark metameric problems are proposed.

Chapter 5 compared the variable-length and fixed-length encodings for metameric representations. Variable-length genomes are found to produce the best results due to the increased flexibility they provide to the variational operators. Static-metavariable representations, which define some underlying structural information of the solution in a static-genotype, may be useful or even necessary for certain problems. If possible,

they should be formulated in such a way that does not eliminate the optimal regions of the search space.

Chapter 6 presented several different crossover operators to be applied to metameric representations. The best crossovers are those that are able to minimize disruption when exchanging metavariables. Spatial crossover gave the best overall results, but it is less generalizable than the similar-metavariable crossover. Cut and splice crossover, which is frequently applied in literature, is found to be highly disruptive.

Chapter 7 proposed length niching selection. Each generation the parent population is partitioned into niches based on solution length. A window function determines the subset of lengths at which niches are formed. A biased window function, which shifts and stretches its bounds in response to observed trends in best solution length, generally gave the best results. Once the niches are formed, a local selection operator is applied. Score selection is proposed as a new local selection operator and is found to significantly outperform tournament selection. The diverse populations produced by length niching and score selection make the algorithm more likely to reach optimal solution lengths, and less likely to converge to suboptimal solutions. Score selection was also found to be effective for non-metameric problems.

Chapter 8 demonstrated the application of the metameric evolutionary algorithm to the design of objects with a lattice microstructure. The use of a variable-length genome was found to be more flexible, and produced better results, compared to a static-metavariable representation.

The proposed metameric evolutionary algorithm is expected to be a powerful tool for studies of metameric problems. It assumes no problem-specific heuristics and, compared to a fixed-length algorithm, it will be much easier to modify as needed. It is hoped that bringing recognition to the class of metameric algorithms will result in the emergence of further improved metameric algorithms going forward.

9.2 Future Work

This section briefly describes several potential areas of future work. These suggestions focus primarily on improving the utility and overall performance of the metameric evolutionary algorithm.

Global variables. Section 2.1 mentioned the possibility that metameric problems may include a fixed number of global variables in the genome. None of the benchmark problems considered in this work use global variables. Effectively handling global variables would require modifications to the variational operators as well as the measures of solution distance, if required by the crossover or selection operators.

Multiple metavariable types. Several practical metameric problems define multiple types of metavariables. This is most common on problems whose solutions form graphs, including neural networks and trusses. In these examples some metavariables might represent nodes while others represent connections or members. As with global variables this would require significant changes to the variational and selection operators. It

is also unclear how solution length should be considered. Length may be the total number of metavariables, regardless of type, or the length of each type may be considered separately. These problems may also require additional considerations be made in regards to solution validity (Section 2.2.2).

Improved measures of distance. Section 4.2 proposed a measure of distance between metavariables and a measure between metameric genotypes. These measures give equal weight to all design variables in each metavariable, however this is not always ideal. Solution quality may be more sensitive to some design variables than others. It is likely desirable that the distance functions are weighted accordingly. For example, consider a problem where one design variable has little to no effect on solution quality. It's likely that a population would quickly emerge that is highly diverse, but only in terms of this particular variable. This would reduce the consideration given to the diversity of other variables, which would be detrimental to overall performance. Weighting the variables appropriately would prevent this, but it is unclear how such weights could be identified.

Multi-objective optimization. All benchmark problems in this study are single-objective, but each of them could also be formulated as multi-objective. Fully characterizing the Pareto-front for a multi-objective metameric problem would likely require a population of varying solution length. An example of this was given in Figure 3.5. Popular multi-objective algorithms, such a NSGA-II [31], use selection operators that encourage more diverse populations in terms of the objective values. Performance of such selection operators for metameric problems might be increased by also considering solution length.

Larger benchmark library. Six benchmark problems were used to evaluate the various metameric representations and operators. These were sufficient for this work, however many more metameric problems from literature were noted in Chapter 3. Some of these require global metavariables or multiple metavariable types. Incorporating these problems into the benchmark library would assist in the development of new metameric algorithms, and ensure the proposed algorithms perform well over a broader range of problems.

Reduced number of evaluations. Trials of the metameric evolutionary algorithm in this work generally used 100k function evaluations. This is not an unusually high amount when compared to other evolutionary algorithms in literature. However, this may make the metameric evolutionary algorithm's application to certain real-world problems impractical. The benchmark problems take hundredths of a second to evaluate. Evolutionary algorithms can be easily parallelized such that all solutions in the child population are evaluated simultaneously. The proposed algorithm would take roughly 3 days to complete if each evaluation took 1 minute to complete, including parallelization. High fidelity models of practical systems may take hours, or longer, to evaluate. Further developments should aim to reduce the total number of evaluation required. Application to expensive optimization problems may also benefit from the inclusion of problem-specific heuristics or surrogate modeling when available.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] S. Abdollahzadeh and N. J. Navimipour. Deployment strategies in the wireless sensor network: A comprehensive review. *Comput. Commun.*, 91-92:1–16, 2016.

[2] A. Ahrari and K. Deb. An improved fully stressed design evolution strategy for layout optimization of truss structures. *Comput. Struct.*, 164:127–144, 2016.

[3] S. Ando, M. Ishizuka, and H. Iba. Evolving analog circuits by variable length chromosomes. In A. Ghosh and S. Tsutsui, editors, *Advances in Evolutionary Computing*, pages 643–662. Springer, Berlin, Heidelberg, 2003.

[4] J. H. B. Ang, K. C. Tan, and A. Al Mamun. A memetic evolutionary search algorithm with variable length chromosome for rule extraction. In *Proc. of SMC 2008*, pages 535–540. IEEE, 2008.

[5] J. Bacardit and J. M. Garrell. Bloat control and generalization pressure using the minimum description length principle for a pittsburgh approach learning classifier system. In *Proc. Revis. Sel. Papers IWLCS 2003-2005*, pages 59–79. Springer, 2007.

[6] S. Bandyopadhyay and U. Maulik. Nonparametric genetic clustering: Comparison of validity indices. *IEEE Trans. Syst., Man, Cybern. C, Appl. Rev.*, 31(1):120–125, 2001.

[7] S. Bandyopadhyay and U. Maulik. Genetic clustering for automatic evolution of clusters and application to image classification. *Pattern Recognit.*, 35(6):1197–1208, 2002.

[8] S. Bandyopadhyay, C. A. Murthy, and S. K. Pal. VGA-classifier: Design and applications. *IEEE Trans. Syst., Man, Cybern. B Cybern.*, 30(6):890–895, 2000.

[9] W. Banzhaf. Artificial regulatory networks and genetic programming. In R. Riolo and B. Worzell, editors, *Genetic Programming Theory and Practice*, pages 43–61. Springer, Boston, 2003.

[10] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic Programming – An Introduction.* Morgan Kaufmann San Francisco, 1998.

[11] J. K. Bassett and K. A. De Jong. Evolving behaviors for cooperating agents. In *Proc. of ISMIS 2000*, pages 157–165. Springer, 2000.

[12] J. E. Beasley. OR-library: Distributing test problems by electronic mail. *J. Oper. Res. Soc.*, 41(11):1069–1072, 1990.

[13] P. J. Bentley. Evolving beyond perfection: An investigation of the effects of long-term evolution on fractal gene regulatory networks. *Biosyst.*, 76(1-3):291–301, 2004.

[14] B. Carse, T. C. Fogarty, and A. Munro. Evolving fuzzy rule based controllers using genetic algorithms. *Fuzzy Sets Syst.*, 80(3):273–293, 1996.

[15] T.-M. Chan, K.-F. Man, K.-S. Tang, and S. Kwong. A jumping-genes paradigm for optimizing factory wlan network. *IEEE Trans. Ind. Inform.*, 3(1):33–43, 2007.

[16] C. S. Chang and S. S. Sim. Optimising train movements through coast control using genetic algorithms. *IEE Proc.-Electr. Power Appl.*, 144(1):65–73, 1997.

[17] D. Chang, Y. Zhao, C. Zheng, and X. Zhang. A genetic clustering algorithm using a message-based similarity measure. *Expert Syst. Appl.*, 39(2):2194–2202, 2012.

[18] T.-J. Chang, N. Meade, J. E. Beasley, and Y. M. Sharaiha. Heuristics for cardinality constrained portfolio optimisation. *Comput. Oper. Res.*, 27(13):1271–1302, 2000.

[19] Y. Chen, H. Li, K. Jin, and Q. Song. Wind farm layout optimization using genetic algorithm with different hub height wind turbines. *Energy Convers. Manag.*, 70:56–65, 2013.

[20] Y. Chen, V. Mahalec, Y. Chen, X. Liu, R. He, and K. Sun. Reconfiguration of satellite orbit for cooperative observation using variable-size multi-objective differential evolution. *Eur. J. Oper. Res.*, 242(1):10–20, 2015.

[21] D. M. Cherba and W. Punch. Crossover gene selection by spatial location. In *Proc. of GECCO'06*, pages 1111–1116. ACM, 2006.

[22] C. A. C. Coello. Theoretical and numerical constraint-handling techniques used with evolutionary algorithms: A survey of the state of the art. *Comput. Methods Appl. Mech. Eng.*, 191(11-12):1245–1287, 2002.

[23] M. Črepinšek, S.-H. Liu, and M. Mernik. Exploration and exploitation in evolutionary algorithms: A survey. *ACM Comput. Surv.*, 45(3):35, 2013.

[24] S. Cussat-Blanc, K. Harrington, and J. Pollack. Gene regulatory network evolution through augmenting topologies. *IEEE Trans. Evol. Comput.*, 19(6):823–837, 2015.

[25] I. M. Daniel and O. Ishai. *Engineering Mechanics of Composite Materials*. Oxford University Press, New York, 2 edition, 2006.

[26] A. Das and R. Vemuri. An automated passive analog circuit synthesis framework using genetic algorithms. In *Proc. of ISLVSI'07*, pages 145–152. IEEE, 2007.

[27] S. Das and S. Sil. Kernel-induced fuzzy clustering of image pixels with an improved differential evolution algorithm. *Inf. Sci.*, 180(8):1237–1256, 2010.

[28] K. A. De Jong, W. M. Spears, and D. F. Gordon. Using genetic algorithms for concept learning. *Mach. Learn.*, 13(2-3):161–188, 1993.

[29] R. R. de Lucena, J. S. Baioco, B. S. L. P. de Lima, C. H. Albrecht, and B. P. Jacob. Optimal design of submarine pipeline routes by genetic algorithm with different constraint handling techniques. *Advanc. Eng. Softw.*, 76:110–124, 2014.

[30] K. Deb and S. Gulati. Design of truss-structures for minimum weight using genetic algorithms. *Finite Elem. Anal. Des.*, 37(5):447–465, 2001.

[31] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Trans. Evol. Comput.*, 6(2):182–197, 2002.

[32] H. Q. Dinh, N. Aubert, N. Noman, T. Fujii, Y. Rondelez, and H. Iba. An effective method for evolving reaction networks in synthetic biochemical systems. *IEEE Trans. Evol. Comput.*, 19(3):374–386, 2015.

[33] P. Dürr, C. Mattiussi, and D. Floreano. Neuroevolution with analog genetic encoding. In *Proc. of PPSN IX*, pages 671–680. Springer, 2006.

[34] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. Springer-Verlag Berlin Heidelberg, 2003.

[35] A. Emami and P. Noghreh. New approach on optimization in placement of wind turbines within wind farm by genetic algorithms. *Renew. Energy*, 35(7):1559–1564, 2010.

[36] Z. Fan, Y. Fang, W. Li, Y. Yuan, Z. Wang, and X. Bian. LSHADE44 with an improved constraint-handling method for solving constrained single-objective optimization problems. In *Proc. of CEC'18*. IEEE, 2018.

[37] Z. Fei, B. Li, S. Yang, C. Xing, H. Chen, and L. Hanzo. A survey of multi-objective optimization in wireless sensor networks: Metrics, algorithms and open problems. *IEEE Commun. Survey. Tutor.*, 19(1):550–586, 2017.

[38] K. P. Ferentinos and T. A. Tsiligiridis. Adaptive design optimization of wireless sensor networks using genetic algorithms. *Comput. Netw.*, 51(4):1031–1051, 2007.

[39] A. Fernández, S. García, J. Luengo, E. Bernadó-Mansilla, and F. Herrera. Genetics-based machine learning for rule induction: State of the art, taxonomy, and comparative study. *IEEE Trans. Evol. Comput.*, 14(6):913–941, 2010.

[40] M. V. Fidelis, H. S. Lopes, and A. A. Freitas. Discovering comprehensible classification rules with a genetic algorithm. In *Proc. of CEC 2000*, pages 805–810. IEEE, 2000.

[41] P. J. Fleming and R. C. Purshouse. Evolutionary algorithms in control systems engineering: A survey. *Control Eng. Pract.*, 10(11):1223–1241, 2002.

[42] D. Floreano, P. Dürr, and C. Mattiussi. Neuroevolution: From architectures to learning. *Evol. Intell.*, 1(1):47–62, 2008.

[43] S. Frandsen. On the wind speed reduction in the center of large clusters of wind turbines. *J. Wind Eng. Ind. Aerod.*, 39(1):251–265, 1992.

[44] A. Gad and O. Abdelkhalik. Hidden genes genetic algorithm for multi-gravity-assist trajectories optimization. *J. Spacecr. Rocket.*, 48(4):629–641, 2011.

[45] H. Ghiasi, K. Fayazbakhsh, D. Pasini, and L. Lessard. Optimum stacking sequence design of composite materials part II: Variable stiffness design. *Compos. Struct.*, 93(1):1–13, 2010.

[46] H. Ghiasi, D. Pasini, and L. Lessard. Optimum stacking sequence design of composite materials part I: Constant stiffness design. *Compos. Struct.*, 90(1):1–11, 2009.

[47] A. Ghozeil and D. B. Fogel. Discovering patterns in spatial data using evolutionary programming. In *Proc. of GP'96*, pages 521–527. MIT Press, 1996.

[48] M. Giger. *Representation Concepts in Evolutionary Algorithm-Based Structural Optimization.* PhD thesis, ETH Zurich, Zurich, 2006.

[49] M. Giger and P. Ermanni. Evolutionary truss topology optimization using a graph-based parameterization concept. *Struct. Multidiscip. Optim.*, 32(4):313–326, 2006.

[50] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley, 1989.

[51] D. E. Goldberg and J. Richardson. Genetic algorithms with sharing for multimodal function optimization. In *Proc. of ICGA'87*, pages 41–49, 1987.

[52] J. S. González, M. B. Payán, and J. M. R. Santos. Optimization of wind farm turbine layout including decision making under risk. *IEEE Syst. J.*, 6(1):94–102, 2012.

[53] J. S. González, M. B. Payán, J. M. R. Santos, and F. González-Longatt. A review and recent developments in the optimal wind-turbine micro-siting problem. *Renew. Sustain. Energy Rev.*, 30:133–144, 2014.

[54] S. A. Grady, M. Y. Hussaini, and M. M. Abdullah. Placement of wind turbines using genetic algorithms. *Renew. Energy*, 30(2):259–270, 2005.

[55] J. B. Grimbleby. Automatic analogue circuit synthesis using genetic algorithms. *IEE Proc.-Circuit, Devices, Syst.*, 147(6):319–323, 2000.

[56] J. A. Hageman, R. Wehrens, H. A. van Sprang, and L. M. C. Buydens. Hybrid genetic algorithm–tabu search approach for optimising multilayer optical coatings. *Anal. Chim. Acta*, 490:211–222, 2003.

[57] H. Hamda, F. Jouve, E. Lutton, M. Schoenauer, and M. Sebag. Compact unstructured representations for evolutionary design. *Appl. Intell.*, 16(2):139–155, 2002.

[58] J. K. Han, B. S. Park, Y. S. Choi, and H. K. Park. Genetic approach with a new representation for base station placement in mobile communications. In *Proc. of VTC 2001 Fall*, pages 2703–2707. IEEE, 2001.

[59] M. Hecker, S. Lambeck, S. Toepfer, E. van Someren, and R. Guthke. Gene regulatory network inference: Data integration in dynamic models—A review. *Biosyst.*, 96(1):86–103, 2009.

[60] M. Hellwig and H.-G. Beyer. A matrix adaptation evolution strategy for constrained real-parameter optimization. In *Proc. of CEC'18*. IEEE, 2018.

[61] J. F. Herbert-Acero, O. Probst, P.-E. Réthoré, G. C. Larsen, and K. K. Castillo-Villar. A review of methodological approaches for the design and optimization of wind farms. *Energies*, 7(11):6930–7016, 2014.

[62] M. Hifi and R. M'Hallah. Approximate algorithms for constrained circular cutting problems. *Comput. Oper. Res.*, 31(5):675–694, 2004.

[63] H. Holland John. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. MIT Press, 1992.

[64] G. A. Hollinger and D. A. Gwaltney. Evolutionary design of fault-tolerant analog control for a piezoelectric pipe-crawling robot. In *Proc. of GECCO'06*, pages 761–768. ACM, 2006.

[65] E. R. Hruschka, R. J. G. B. Campello, A. A. Freitas, and A. C. P. L. F. de Carvalho. A survey of evolutionary algorithms for clustering. *IEEE Trans. Syst., Man, Cybern. C, Appl. Rev.*, 39(2):133–155, 2009.

[66] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: A review. *ACM Comput. Surv.*, 31(3):264–323, 1999.

[67] N. O. Jensen. A note on wind generator interaction. Technical Report Riso-M-2411, Risø National Laboratory, Roskilde, 1983.

[68] J. Jia, J. Chen, G. Chang, and Z. Tan. Energy efficient coverage control in wireless sensor networks based on multi-objective genetic algorithm. *Comput. Math. Appl.*, 57(11-12):1756–1766, 2009.

[69] J. Jia, J. Chen, G. Chang, Y. Wen, and J. Song. Multi-objective optimization for coverage control in wireless sensor network with adjustable sensing radius. *Comput. Math. Appl.*, 57(11):1767–1775, 2009.

[70] D. B. Jourdan and O. L. de Weck. Multi-objective genetic algorithm for the automated planning of a wireless sensor network to monitor a critical facility. In *Proc. of SPIE, Volume 5403*, pages 565–575. SPIE, 2004.

[71] I. Kajitani, T. Hoshino, M. Iwata, and T. Higuchi. Variable length chromosome GA for evolvable hardware. In *Proc. of ICEC'96*, pages 443–447. IEEE, 1996.

[72] H. Karadeniz. *Stochastic Analysis of Offshore Steel Structures: An Analytical Appraisal*. Springer-Verlag London, 2012.

[73] D. Keller. Global laminate optimization on geometrically partitioned shell structures. *Struct. Multidiscip. Optim.*, 43(3):353–368, 2011.

[74] M. M. Khan, A. M. Ahmad, G. M. Khan, and J. F. Miller. Fast learning neural networks using cartesian genetic programming. *Neurocomputing*, 121:274–289, 2013.

[75] S. A. Khan and S. Rehman. Iterative non-deterministic algorithms in on-shore wind farm design: A brief survey. *Renew. Sustain. Energy. Rev.*, 19:370–384, 2013.

[76] R. Kicinger, T. Arciszewski, and K. A. De Jong. Evolutionary computation and structural design: A survey of the state-of-the-art. *Comput. Struct.*, 83(23):1943–1978, 2005.

[77] K.-J. Kim and S.-B. Cho. Automated synthesis of multiple analog circuits using evolutionary computation for redundancy-based fault-tolerance. *Appl. Soft Comput.*, 12(4):1309–1321, 2012.

[78] K. Lakshmi and A. R. M. Rao. Multi-objective optimal design of laminated composite skirt using hybrid NSGA. *Meccanica*, 48(6):1431–1450, 2013.

[79] W. B. Langdon. The evolution of size in variable length representations. In *Proc. of ICEC'98*, pages 633–638. IEEE, 1998.

[80] R. Le Riche and R. T. Haftka. Improved genetic algorithm for minimum thickness composite laminate design. *Compos. Eng.*, 5(2):143–161, 1995.

[81] C.-Y. Lee. *Efficient Automatic Engineering Design Synthesis via Evolutionary Exploration*. PhD thesis, California Institute of Technology, Pasadena, 2002.

[82] C. Y. Lee and E. K. Antonsson. Variable length genomes for evolutionary algorithms. In *Proc. of GECCO'00*, page 806. Morgan Kaufmann, 2000.

[83] F. H.-F. Leung, H.-K. Lam, S.-H. Ling, and P. K.-S. Tam. Tuning of the structure and parameters of a neural network using an improved genetic algorithm. *IEEE Trans. Neural Netw.*, 14(1):79–88, 2003.

[84] J. D. Lohn and S. P. Colombano. A circuit representation technique for automated circuit design. *IEEE Trans. Evol. Comput.*, 3(3):205–219, 1999.

[85] F. Luna, J. J. Durillo, A. J. Nebro, and E. Alba. Evolutionary algorithms for solving the automatic cell planning problem: A survey. *Eng. Optim.*, 42(7):671–690, 2010.

[86] V. Maniezzo. Genetic evolution of the topology and weight distribution of neural networks. *IEEE Trans. Neural Netw.*, 5(1):39–53, 1994.

[87] S. Manos, M. Large, and L. Poladian. Evolutionary design of single-mode microstructured polymer optical fibres using an artificial embryogeny representation. In *Proc. of GECCO'07*, pages 2549–2556. ACM, 2007.

[88] V. J. C. Maranan. *Design and Processing of an Additive Manufacturing Component*. Baccalaureate thesis, The Pennsylvania State University, 2013.

[89] E. Mezura-Montes and C. A. C. Coello. A simple multimembered evolution strategy to solve constrained optimization problems. *IEEE Trans. Evol. Comput.*, 9(1):1–17, 2005.

[90] E. Mezura-Montes and C. A. C. Coello. Constraint-handling in nature-inspired numerical optimization: Past, present and future. *Swarm Evol. Comput.*, 1(4):173–194, 2011.

[91] Z. Michalewicz and M. Schoenauer. Evolutionary algorithms for constrained parameter optimization problems. *Evol. Comput.*, 4(1):1–32, 1996.

[92] J. F. Miller. Cartesian genetic programming. In J. F. Miller, editor, *Cartesian Genetic Programming*, pages 17–34. Springer, Berlin, Heidelberg, 2011.

[93] J. F. Miller, D. Job, and V. K. Vassilev. Principles in the evolutionary design of digital circuits—Part I. *Genet. Program Evolvable Mach.*, 1(1-2):7–35, 2000.

[94] J. F. Miller and S. L. Smith. Redundancy and computational efficiency in cartesian genetic programming. *IEEE Trans. Evol. Comput.*, 10(2):167–174, 2006.

[95] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT press, 1998.

[96] G. Molina, E. Alba, and E.-G. Talbi. Optimal sensor network layout using multi-objective metaheuristics. *J. Univers. Comput. Sci.*, 14(15):2549–2565, 2008.

[97] M. Montemurro, A. Vincenti, and P. Vannucci. A two-level procedure for the global optimum design of composite modular structures—Application to the design of an aircraft wing: Part 1: Theoretical formulation. *J. Optim. Theory Appl.*, 155(1):1–23, 2012.

[98] J. C. Mora, J. M. C. Barón, J. M. R. Santos, and M. B. Payán. An evolutive algorithm for wind farm optimal design. *Neurocomputing*, 70(16):2651–2658, 2007.

[99] G. Mosetti, C. Poloni, and B. Diviacco. Optimization of wind turbine positioning in large windfarms by means of a genetic algorithm. *J. Wind Eng. Ind. Aerod.*, 51(1):105–116, 1994.

[100] S. J. Nanda and G. Panda. A survey on nature inspired metaheuristic algorithms for partitional clustering. *Swarm Evol. Comput.*, 16:1–18, 2014.

[101] M. Nicolau, M. Schoenauer, and W. Banzhaf. Evolving genes to balance a pole. In *Proc. of EuroGP 2010*, pages 196–207. Springer, 2010.

[102] M. G. H. Omran, A. Salman, and A. P. Engelbrecht. Dynamic clustering using particle swarm optimization with application in image segmentation. *Pattern. Anal. Appl.*, 8(4):332–344, 2006.

[103] P. P. Palmes, T. Hayasaka, and S. Usui. Mutation-based genetic neural network. *IEEE Trans. Neural Netw.*, 16(3):587–600, 2005.

[104] S.-M. Pan and K.-S. Cheng. Evolution-based tabu search approach to automatic clustering. *IEEE Trans. Syst., Man, Cybern. C, Appl. Rev.*, 37(5):827–838, 2007.

[105] C. H. Park, W. I. Lee, W. S. Han, and A. Vautrin. Improved genetic algorithm for multidisciplinary optimization of composite laminates. *Comput. Struct.*, 86(19-20):1894–1903, 2008.

[106] J. L. Pelletier and S. S. Vel. Multi-objective optimization of fiber reinforced composite laminates for strength, stiffness and minimal mass. *Comput. Struct.*, 84(29):2065–2080, 2006.

[107] L. Poladian. A genotype-to-phenotype mapping for microstructured polymer optical fibres. In *Proc. of CEC'11*, pages 378–385. IEEE, 2011.

[108] R. Poli, W. B. Langdon, N. F. McPhee, and J. R. Koza. *A Field Guide to Genetic Programming.* Lulu Enterprises, UK, 2008.

[109] P. Pulkkinen and H. Koivisto. Fuzzy classifier identification using decision tree and multiobjective evolutionary algorithms. *Int. J. Approx. Reason.*, 48(2):526–543, 2008.

[110] S. D. Rajan. Sizing, shape, and topology design optimization of trusses using genetic algorithm. *J. Struct. Eng.*, 121(10):1480–1487, 1995.

[111] A. R. M. Rao and N. Arvind. A scatter search algorithm for stacking sequence optimisation of laminate composites. *Compos. Struct.*, 70(4):383–402, 2005.

[112] A. R. M. Rao and K. Lakshmi. Discrete hybrid PSO algorithm for design of laminate composites with multiple objectives. *J. Reinf. Plast. Compos.*, 30(20):1703–1727, 2011.

[113] J. N. Richardson, S. Adriaenssens, P. Bouillard, and R. F. Coelho. Multiobjective topology optimization of truss structures with kinematic stability repair. *Struct. Multidiscip. Optim.*, 46(4):513–532, 2012.

[114] F. Rothlauf. *Representations for Genetic and Evolutionary Algorithms.* Springer-Verlag, 2006.

[115] M. L. Ryerkerk, R. C. Averill, K. Deb, and E. D. Goodman. Improved local selection for metameric problems. In preparation.

[116] M. L. Ryerkerk, R. C. Averill, K. Deb, and E. D. Goodman. A length niching operator for metameric problems. In preparation.

[117] M. L. Ryerkerk, R. C. Averill, K. Deb, and E. D. Goodman. A survey of evolutionary algorithms using metameric representations. *Genet. Program Evolvable Mach.* submitted.

[118] M. L. Ryerkerk, R. C. Averill, K. Deb, and E. D. Goodman. Solving metameric variable-length optimization problems using genetic algorithms. *Genet. Program Evolvable Mach.*, 18(2):247–277, 2017.

[119] S. Sanchez and S. Cussat-Blanc. Gene regulated car driving: Using a gene regulatory network to drive a virtual car. *Genet. Program Evolvable Mach.*, 15(4):477–511, 2014.

[120] Y. A. Sapargaliyev and T. G. Kalganova. Open-ended evolution to discover analogue circuits for beyond conventional applications. *Genet. Program Evolvable Mach.*, 13(4):411–443, 2012.

[121] B. Sareni and L. Krähenbühl. Fitness sharing and niching methods revisited. *IEEE Trans. Evol. Comput.*, 2(3):97–106, 1998.

[122] T. A. Schaedler, A. J. Jacobsen, A. Torrents, A. E. Sorensen, J. Lian, J. R. Greer, L. Valdevit, and W. B. Carter. Ultralight metallic microlattices. *Science*, 334(6058):962–965, 2011.

[123] M. Schoenauer. Shape representations and evolutionary schemes. In *Proc. of Evolutionary Programming V*, pages 121–129. MIT Press, 1996.

[124] M. Schoenauer, L. Kallel, and F. Jouve. Mechanical inclusions identification by evolutionary computation. *Eur. J. Comput. Mech.*, 5-6:619–648, 1996.

[125] A. Sîrbu, H. J. Ruskin, and M. Crane. Comparison of evolutionary algorithms in gene regulatory network model inference. *BMC Bioinform.*, 11:59, 2010.

[126] S. Şişbot, Ö. Turgut, M. Tunç, and Ü. Çamdalı. Optimal positioning of wind turbines on gökçeada using multi-objective genetic algorithm. *Wind Energy*, 13(4):297–306, 2010.

[127] G. Soremekun, Z. Gürdal, R. T. Haftka, and L. T. Watson. Composite laminate design optimization by genetic algorithm with generalized elitist selection. *Comput. Struct.*, 79(2):131–143, 2001.

[128] A. Spirov and D. Holloway. Using evolutionary computations to understand the design and evolution of gene and cell regulatory networks. *Methods*, 62(1):39–55, 2013.

[129] R. Srikanth, R. George, N. Warsi, D. Prabhu, F. E. Petry, and B. P. Buckles. A variable-length genetic algorithm for clustering and classification. *Pattern Recognit. Lett.*, 16(8):789–800, 1995.

[130] K. O. Stanley. Compositional pattern producing networks: A novel abstraction of development. *Genet. Program Evolvable Mach.*, 8(2):131–162, 2007.

[131] K. O. Stanley, D. B. D'Ambrosio, and J. Gauci. A hypercube-based encoding for evolving large-scale neural networks. *Artificial life*, 15(2):185–212, 2009.

[132] K. O. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evol. Comput.*, 10(2):99–127, 2002.

[133] T. Y. Teck and M. Chitre. Direct policy search with variable-length genetic algorithm for single beacon cooperative path planning. In *Proc. of DARS 2014*, pages 321–336. Springer, 2014.

[134] C.-K. Ting, C.-N. Lee, H.-C. Chang, and J.-S. Wu. Wireless heterogeneous transmitter placement using multiobjective variable-length genetic algorithm. *IEEE Trans. Syst., Man, Cybern. B Cybern.*, 39(4):945–958, 2009.

[135] M. A. Trefzer, T. Kuyucu, J. F. Miller, and A. M. Tyrrell. On the advantages of variable length GRNs for the evolution of multicellular developmental systems. *IEEE Trans. Evol. Comput.*, 17(1):100–121, 2013.

[136] A. Trivedi, D. Srinivasan, and N. Biswas. An improved unified differential evolution algorithm for constrained optimization problems. Retrieved from http://web.mysites.ntu.edu.sg/epnsugan/PublicSite/Shared%20Documents/CEC-2018/Constrained/Improved_Unified_Differential_Evolution_CEC_2018_Report.pdf, 2018. Accessed: 2018-11-21.

[137] L. Y. Tseng and S. B. Yang. A genetic approach to the automatic clustering problem. *Pattern Recognit.*, 34(2):415–424, 2001.

[138] S. Tsutsui, M. Yamamura, and T. Higuchi. Multi-parent recombination with simplex crossover in real coded genetic algorithms. In *Proc. of GECCO'99*, pages 657–664. Morgan Kaufmann, 1999.

[139] J. R. Vinson and R. L. Sierakowski. *The Behavior of Structures Composed of Composite Materials*. Springer, 2002.

[140] Y. Wang, Z. Cai, Y. Zhou, and Z. Fan. Constrained optimization based on hybrid evolutionary algorithm and adaptive constraint-handling technique. *Struct. Multidiscip. Optim.*, 37(4):395–413, 2009.

[141] N. Weicker, G. Szabo, K. Weicker, and P. Widmayer. Evolutionary multiobjective optimization for base station transmitter placement with frequency assignment. *IEEE Trans. Evol. Comput.*, 7(2):189–203, 2003.

[142] D. Whitley, T. Starkweather, and C. Bogart. Genetic algorithms and neural networks: Optimizing connections and connectivity. *Parallel Comput.*, 14(3):347–361, 1990.

[143] D. Wilson, E. Awa, S. Cussat-Blanc, K. Veeramachaneni, and U.-M. O'Reilly. On learning to generate wind farm layouts. In *Proc. of GECCO'13*, pages 767–774. ACM, 2013.

[144] A. S. Wu, A. C. Schultz, and A. Agah. Evolving control for distributed micro air vehicles. In *Proc. of CIRA'99*, pages 174–179. IEEE, 1999.

[145] G. Wu, R. Mallipeddi, and P. Suganthan. Problem definitions and evaluation criteria for the CEC 2017 competition on constrained real-parameter optimization. Technical report, Nanyang Technological University, Singapore, 2017.

[146] X. Yao. Evolving artificial neural networks. *Proc. IEEE*, 87(9):1423–1447, 1999.

[147] X. Yao and Y. Liu. A new evolutionary system for evolving artificial neural networks. *IEEE Trans. Neural Netw.*, 8(3):694–713, 1997.

[148] J. Yu, S. Wang, and L. Xi. Evolving artificial neural networks using an improved PSO and DPSO. *Neurocomputing*, 71(4-6):1054–1060, 2008.

[149] R. S. Zebulum, M. Vellasco, and M. A. Pacheco. Variable length representation in evolutionary electronics. *Evol. Comput.*, 8(1):93–120, 2000.