FACTORED EVOLUTIONARY ALGORITHMS: COOPERATIVE COEVOLUTIONARY OPTIMIZATION WITH OVERLAP

by

Shane Tyler Strasser

A dissertation submitted in partial fulfillment
of the requirements for the degree

of

Doctor of Philosophy

in

Computer Science

MONTANA STATE UNIVERSITY
Bozeman, Montana

December 13, 2016

DEDICATION

To my loving wife, Saundra, who's love and support made this possible.

ACKNOWLEDGEMENTS

I am very grateful to my advisor, Dr. John Sheppard, for welcoming me to Montana State University and introducing me to the wide world of artificial intelligence. His encouragement has pushed me to excel and has helped me mature as a researcher. This research would not have been possible without his support. I would also like to thank to my committee members, Mike Wittie, Brendan Mumey, and Tomáš Gedeon, for their helpful comments and advice.

I thank the members of the Numerical Intelligent Systems Laboratory at Montana State University for their comments and advice during the development of this work. Richard McAllister, Patrick Donnelly, and Liessman Sturlaugson, your friendship during my first years at MSU will always hold a special place in my heart. I would also like to thank Nathan Fortier, Stephyn Butcher, Rollie Goodman, Logan Perrault, Monica Thorton, and Karthik Ganesan Pillai for their thoughts and assistance during the development of this research.

I would also like to thank my parents, Dale and Ruth Strasser, who always encouraged me to ask questions and explore how the world works, and to my brother and sister, Nick and Portia, for always pushing me to strive for excellence. Also, my wonderful in-laws, Karen Searle and Lisa Armitage, who helped keep me fed.

Finally, I thank my beautiful wife, Saundra, for encouraging me through late nights and early mornings. Without her love and care, this work would not have been possible.

TABLE OF CONTENTS

TABLE OF CONTENTS - CONTINUED

TABLE OF CONTENTS - CONTINUED

TABLE OF CONTENTS - CONTINUED

LIST OF TABLES

LIST OF TABLES - CONTINUED

LIST OF FIGURES

LIST OF FIGURES - CONTINUED

LIST OF FIGURES - CONTINUED

LIST OF ALGORITHM

ABSTRACT

Factored Evolutionary Algorithms (FEA) define a relatively new class of evolutionary-based optimization algorithms that have been successfully applied to various problems, such as training neural networks and performing abductive inference in graphical models. FEA is unique in that it factors the function being optimized by creating subpopulations that optimize over a subset of dimensions of the function. However, unlike other optimization techniques that subdivide optimization problems, FEA encourages subpopulations to overlap with one another, allowing subpopulations to compete and share information. Although FEA has been shown to be very effective at function optimization, there is still little understanding with respect to its general characteristics. In this dissertation, we present seven results exploring the theoretical and empirical properties of FEA.

First, we present a formal definition of FEA and demonstrate its relationships to other multiple population algorithms. Second, we demonstrate that FEA's success is independent of the underlying optimization algorithm by evaluating the performance of FEA using a wide variety of evolutionary- and swarm-based algorithms over single-population and non-overlapping versions. Third, we demonstrate that for a given problem, there is an optimal way to generate groups of overlapping subpopulations derived using the Markov blanket in Bayesian networks. Fourth, we establish that a class of optimization functions like NK landscapes can be mapped directly to probabilistic graphical models. Additionally, we demonstrate that factor architectures derived from Markov blankets maintain better diversity of individuals in their population. Fifth, we present a new discrete Particle Swarm Optimization (PSO) algorithm and compare its performance to competing approaches. In addition, we analyze the performance of FEA versions of discrete PSO and discover that FEA masks the poor performance of search algorithms. We show what conditions are necessary for FEA to converge and scenarios where FEA may become stuck in suboptimal regions in the search space. Finally, we explore the performance of FEA on unitation functions and discover several instances where FEA struggles to outperform single-population algorithms. These results allow us to determine which situations are appropriate for FEA when using solving real-world problems.

CHAPTER ONE

INTRODUCTION

Bin packing, the traveling salesperson problem, job shop scheduling, training neural networks, and inference in Bayesian networks are all important problems that can be solved using optimization [103]. However, many problems, including the ones mentioned above, are NP-complete, and there is a need for algorithms that can find good approximate solutions to such problems. Additionally, there are problems that are not NP-complete but are still difficult to solve. This includes problems where the output of a function may be non-convex, noisy, stochastic, or dependent on human feedback.

One class of algorithms for optimizing these types of problems is local search algorithms. Local search algorithms work by maintaining a solution or set of solutions and attempting to locate better solutions by exploring variations of the current solution or set of solutions [51]. The most basic local search algorithm is a Hill Climber, which maintains a single point that is updated by moving to a neighbor of the current point that is the best solution to the optimization problem [50, 67]. These types of algorithms are popular because they are *weak methods*, meaning very few assumptions are made about the search space, which allows an algorithm to be applied to a wide range of problems with little or no modifications to algorithm. This is different than *strong methods*, which make several assumptions about the underlying optimization problem [18, 100].

While local search algorithms like Hill Climbers are able to locate good solutions to a wide range of problems, they often struggle on more complex problems with rugged search spaces. For example, while exploring the search space, an algorithm may become stuck in what are called *local optima*, which are points in the space that are better than all other points in the immediate neighborhood. This is different than a *global optimum*, which is a point in the search space that is better than all other points in the space [89, 103]. Local optima are a challenge for search algorithms to escape because they require a local search algorithm to first move toward worse solutions before the algorithm can move toward better solutions. In this dissertation, we discover search properties of a local search algorithm, called Factored Evolutionary Algorithms, by exploring empirical and theoretical experiments.

1.1 Evolutionary Algorithms

Stochastic search algorithms are a class of a Local Search algorithms often used to solve difficult problems because the randomness used in the algorithms allows the possibility for them to escape local optima. One of the best-known families of stochastic search algorithms are Evolutionary Algorithms (EA). Algorithms such as Genetic Algorithms (GA) [46], Genetic Programming (GP) [55], Evolution Strategies (ES) [86], Differential Evolution (DE) [105], and Estimation of Distribution Algorithms (EDA) [3] are types of EAs. These algorithms work by maintaining a population of individuals and use a fitness function to evaluate the quality of an individual. An EA then modifies its population according to a set of update rules. This process is repeated until a convergence criterion is met. EAs are inspired by *Darwinian evolution*: the best solutions in the population are more likely to survive and pass their traits onto future generations [22].

Another class of stochastic search algorithms is Swarm Intelligence (SI), which includes algorithms such as Ant Colony Optimization (ACO) [25] and Particle Swarm Optimization (PSO) [48]. ACO and PSO are based upon the movement in a population of animals, such as ants searching for food or fish moving in a school [29]. Similar to EA, SI maintains a population of individuals and moves the individuals based on either the population or swarm dynamics. For the sake of simplicity, in this dissertation we will refer to an EA as any population-based stochastic search algorithm, including SI.

Population-based algorithms like EAs are well suited for optimization problems where the search space is difficult to characterize. One reason EAs are so well suited is that the population of solutions allows an EA to explore a wide range of solution. Additionally, because EAs are weak methods, an EA can be applied to a wide range of tasks with minimal or no changes made to the EA algorithm [81].

While EAs have been successfully applied to a wide range of problems, there are drawbacks to EAs. The first is that they suffer from the *curse of dimensionality*, which says that as the number of variables in a problem increases, the number of samples required to cover the space grows exponentially [12, 45, 102]. Because more samples are required to cover the search space, an EA requires more iterations in order to locate a solution of a specified quality. For example, assume a random search algorithm is applied to a function with only eight variables, each of which can take on eight different values. Additionally, each state is ordered and distinct. This results in a total of 8^8 different states. If the search algorithm explores 10,000 samples, the probability of finding one of the top 500 solutions is 0.2980 percent. However, if the discrete problem has ten variables each of which can take on ten different values, there are now 10^{10} different states. With 10,000 randomly generated samples, the probability of finding one of the top 500 solutions is now only 0.0005 percent.

While this example assumed a random search, a GA or PSO will also struggle on locating good solutions as the dimensionality of the search space increases [114]. Because more samples are required for the EA to locate good solutions, the EA requires larger population sizes or additional iterations. However, simply adding more individuals or iterations does not always imply a larger exploration of the search space [12].

Another drawback to EAs is that certain algorithms, like a GA, are susceptible to *hitchhiking*, which is when poor values become associated with good schemas [99]. This is influenced by the he representation and the operators of the GA. This causes the GA to converge to suboptimal solutions or requires more generations to locate good solutions [66]. Similarly, PSO can be prone to what is called *two steps forward and one step back* (TSFOSB), which happens when near optimal parts of an individual's current position can be thrown away if the rest of an individual's position leads to low fitness [113]. The result is that PSO must spend additional iterations to recover the values that were thrown away. Additionally, the probability of hitchhiking or TSFOSB increases as the problem dimensionality increases [113].

Cooperative Coevolutionary Algorithm (CCEA) is an extension to EAs that addresses the curse of dimensionality and hitchhiking. The algorithm works by subdividing the dimensions into disjoint subsets which are then optimized over independently of other subpopulations. To evaluate a solution in a subpopulation, a process called *collaboration* is performed where a complete solution is created using the remaining subpopulations. ne of the earliest CCEA was developed by Potter and De Jong, the Cooperative Coevolutionary Genetic Algorithm (CCGA), in which the algorithm creates several subpopulations that represent a subcomponent of the full solution [82]. A full solution is then obtained by assembling representative members of each of the subpopulations. CCGA's assume that if a subpopulation locates a good

solution in its subspace, then the values will also lead to good solutions in the entire search space. When this assumption holds, the CCGA is able to explore the search space more efficiently than a full GA [82].

Van den Bergh and Engelbrecht extended the idea of CCGA by applying the concept to PSO in an algorithm called Cooperative Particle Swarm Optimization (CPSO) [113]. CPSO reduces the likelihood of the TSFOSB phenomenon by subdividing the dimensions of the search space similar to CCGA. Later work by Shi *et al.* developed an algorithm called Cooperative Coevolutionary Differential Evolutionary (CCDE) that extends CCGA and CPSO to Differential Evolution [99].

One drawback of CCEAs is that each of the subpopulations is assumed to be disjoint. Previous work on CCEA has shown that when relationships exist between subpopulations, the performance of the algorithm degrades [114]. However, for certain problems, CCEA will be unable to create disjoint subpopulations while minimizing interactions between subpopulations. For example, suppose in a function with three variables, X_1, X_2, and X_3, that there is a relationship between X_1 and X_2, and X_2 and X_3. Suppose in subdividing the dimensions for CCEA, each subpopulation optimizes over a single variable. But this means that a relationship exists between the subpopulation for X_1 and the subpopulation for X_2. A person can attempt to create a subpopulation that optimizes over both X_1 and X_2; however, there still exists a relationship with the population optimizing over variable X_3. This example demonstrates the need for a CCEA that is able to create more flexible subpopulations.

1.2 Factored Evolutionary Algorithms

One way to mitigate the interactions between factors like those in the previous example is to allow subpopulations to overlap with each another. In the previous example, one could create a subpopulation that optimizes over variables X_1 and X_2

and another subpopulation that optimizes over X_2 and X_3. This configuration allows all of the variable interactions in the optimization problem to be covered by at least one subpopulation. Factored Evolutionary Algorithm (FEA) is a generalization of CCEA that relaxes the assumption that subpopulations must be disjoint, allowing for a more flexible set of subpopulations. FEA generalizes Overlapping Swarm Intelligence (OSI) such that any EA can be used in an overlapping subpopulation.

The idea behind FEA is similar to how polynomials can be decomposed into a product of factors. FEA decomposes the optimization problem into a set of subpopulations, or *factors*, which, when put together, represent full solutions to the problem. Because there are multiple populations optimizing over the same variable, the algorithm requires a routine for resolving a conflict in values between populations when assembling a full solution. However, FEA encourages factors to overlap with one another, which allows the shared dimensions in overlapping factors to compete for inclusion in the full solution [33].

There are three main functions that FEA uses: Update, Compete, and Share. Update is the first step, which iterates over all of the factors and allows each factor to optimize over its individual variables. Next, FEA performs Compete, which finds sets of values from each factor that create a full solution with a high fitness. Finally, FEA performs Share by using the full solution to inject information back into the factors.

1.3 Research Questions

Prior work on CCEA, explored the most effective method to subdivide an optimization problem, which we call the *factor architecture*. For example, Shi *et al.*'s CCDE algorithm used two different factor architectures: one that divides the problem into only two populations, and another that assigns each variable to a

single subpopulation [99]. Van den Bergh and Engelbrecht used an architecture that subdivided the problem into six disjoint subpopulations [114]. However, both papers used the same factor architectures for all problems and failed to take into account variable interactions.

In the first application of OSI to train neural networks, Ganesan Pillai and Sheppard created a factor for each unique path of the neural network, starting and ending at an input and output node, respectively [80]. While this architecture was found to be effective for training deep neural networks, the number of factors grows exponentially as the number of layers in the deep neural network increases. Later work by Fortier *et al.* used a different factor architecture where each subpopulation learned the weights for each neuron in the neural network [35]. This architecture was also found to be an effective way to train neural networks and avoided the problem of the exponential increase in the number of subpopulations. This highlights that finding effective and efficient ways to create factor architectures in FEA is an important decision to consider when applying the algorithm to different problems.

Given a problem, one would like to derive the optimal factor architecture without empirically testing several different architectures. One open question is: Given an optimization problem that contains a set of variable interactions, is it possible to use the same procedure for deriving factor architecture from one set of problems and apply it to another group of problems? Additionally, if there exists a factor architecture that performs well on a set of problems, are there problems in which the same factor architecture does not perform well? This also raises the question of what causes the factor architecture to perform well on certain problems and poorly on other problems.

An even more general question is what types of problems is FEA effective on? Previous work has shown FEA to perform well on a variety of tasks, such as training neural networks, performing inference in Bayesian networks, learning

Bayesian networks, and routing data in a sensor network. However, the *No Free Lunch Theorem* states that there is no one algorithm that is universally the best search or optimization algorithm [122]. In other words, while one algorithm may be efficient on one class of problems, it is guaranteed to perform poorly on another class of problems. This raises the question of what sets of problems do FEAs perform well on and what sets do they perform poorly on?

Finally, we raise the questions, what are the general performance characteristics of FEA and where does it gain its performance advantage over traditional population algorithms? One specific question is what are the convergence properties of FEA? While some of the performance of FEA can be attributed to the subdivision of the problem over multiple subpopulations, the overlap of the subpopulations may also be a major contributor to FEA's performance. This also raises the question as to how much of a role the competition step in FEA contributes to FEA's performance.

In this dissertation, we attempt to answer the following three questions: what is the optimal configuration for FEA, what kind of problems does FEA work well on, and what are the general search characteristics of FEA? While there are many other questions concerning FEA, those questions can be classified under one or more of these three questions. Throughout the rest of the dissertation, we will refer back to how each contribution helps answer these questions.

1.4 Contributions

In this dissertation, we make several significant contributions to the fields of evolutionary computation, artificial intelligence, and computer science.

- We present the first formal definition of FEA. To do so, we provide a framework that allows us to define EAs containing multiple populations. This

framework allows us to formally define not only FEA, but also other multi-population algorithms, such as the Island Model and CCEA. In addition to this framework, we present the three primary subfunctions in FEA and a complexity analysis of each function. This formal definition of FEA advances the field of evolutionary computation by establishing the relationship between multi-population algorithms, regardless of the underlying optimization algorithm or the number of variables a subpopulation is optimizing. Additionally, the complexity analysis of FEA advances the understanding of FEA by showing which functions of FEA have the largest influence on its runtime. This helps answer what are the general search characteristics of FEA.

- We also demonstrate empirically that FEA's performance is not dependent on the underlying optimization algorithm. Previous work with FEA was limited to using PSO. Using our formal definition, we test FEA on a set of test functions using a variety of local search-based optimization algorithms and show that FEA almost always outperforms single-population and CCEA versions. These experiments advance the field of evolutionary computation by demonstrating that the FEA formalization of overlapping populations is what provides the main advantage.

- We study different factor architectures on a multiple groups of different problems. While there has been some work discussing different factor architectures, there has been little work to optimize these factor architectures for FEA. In this dissertation, we empirically demonstrate that the performance of FEA is tied to the factor architecture. These results advance the knowledge of how to efficiently create subpopulations in multi-population algorithms.

- Based on the above, we formally demonstrate how various optimization problems can be mapped to Factor Graphs. Furthermore, by mapping a problem to a factor graph, we can use the resulting structure as a means to derive a factor architecture for FEA. These results further our understanding of the relationship between optimization problems and probabilistic graphical models.

- While looking at the performance of FEA using PSO for different optimization problems, we discovered the opportunity to improve upon existing PSO algorithms for discrete and categorical optimization. As a result, we develop and analyze a new PSO algorithm, called Integer and Categorical PSO (ICPSO). ICPSO provides a novel and effective method to use PSO when solving discrete and categorical optimization problems.

- We prove the conditions required for FEA to converge and show that FEA is theoretically susceptible to becoming stuck in suboptimal locations in the search space. However, we provide experiments demonstrating that this rarely occurs. These results further our understanding of the impact an overlapping factor architecture has on the performance of FEA and on the overall dynamics of FEA.

- Finally, we also investigate the performance of FEA on deceptive problems like unitation and Royal Road functions. Much of the previous work on FEA was tested on commonly used test functions. However, no work explored FEA's performance on functions that are purposefully deceptive. By testing and exploring the performance of FEA on deceptive functions, we discover that FEA struggles on problems where there is no clear gradient in the search space. This is caused by the Compete step being unable to efficiently perform competition between overlapping factors.

1.5 Overview

In this section, we describe the organization of the remaining chapters of this dissertation and give a brief overview of each chapter. Chapter 2 presents the background material necessary to understand the work presented in this dissertation. This includes a definition of EAs followed by several examples of EAs, such as Genetic Algorithms, Particle Swarm Optimization, and Differential Evolution. Additionally, we present an in-depth review of related work.

In Chapter 3, we present a framework for defining multi-population algorithms. We then demonstrate how algorithms like the Island Model, CCEA's, and FEA fit into this framework. This is followed by presentation of the FEA algorithm and each of its subfunctions. Additionally, we present results comparing FEA with CCEA and single-population versions of different algorithms. In doing so, we show that FEA generalizes to other types of algorithms. Additionally, we demonstrate FEA's performance gains over CCEA and single-population versions.

In Chapter 4, we explore a variety of factor architectures on several different problems, such as abductive inference in Bayesian networks, maximizing NK landscapes, and optimizing commonly used benchmark test functions. Given a problem, we investigate different ways to create an overlapping factor structure. We then test each of those different architectures on a set of different problem instances demonstrate that there do exist factor architectures that are better than others. Additionally, we present theorems proving how various optimization functions can be mapped to factor graphs. These theorems are used to derive a general process for generating a factor architecture given an optimization problem and its variable interactions. We then present more results demonstrating the benefit of this factor

architecture and explore where this architecture gains its performance over other architectures.

In Chapter 5, we present our ICPSO algorithm. This includes a review of other discrete and categorical PSO algorithms, followed by several experiments demonstrating ICPSO performance over competing approaches. We show how FEA versions of the discrete PSO algorithms perform on NK landscapes and abductive inference in Bayesian networks and discover that FEA is able to mask the poor performance of several discrete PSO algorithms.

Chapter 6 presents an analysis of FEA convergence. First, we present results that prove the full global solution in FEA will converge to a single point. We also present a theorem showing that FEA may become stuck in suboptimal locations in the search space. However, we also empirically demonstrate that, in practice, the probability of becoming stuck in these suboptimal locations is very low.

In Chapter 7, we investigate the performance of FEA on unitation models, such as the Royal Road and Long Path functions. These results demonstrate instances where FEA struggles to perform as well as single-population algorithms. Additionally, we discover that in these cases where FEA struggles, it is because the Compete step in FEA is unable to efficiently perform competition between overlapping factors.

We conclude the dissertation in Chapter 8 with a summary of the dissertation and several areas for future work. Additionally, we summarize the key contributions.

CHAPTER TWO

BACKGROUND AND RELATED WORK

FEA is a generalization of EA that allow for the subdivision of the optimization problem. To understand the motivation for FEA's factoring of optimization problems, we first present a background of different EAs along with several examples. Next, we discuss drawbacks of different algorithms followed by related work on how to mitigate those issues in EAs.

2.1 Evolutionary Algorithms

EA are stochastic local search algorithms that use biologically inspired operations to optimize a given function [22, 89]. Given a set of solutions, or a population, EA modify each solution according to a specified update rule. These individuals help guide the search process to locate better solutions. One of the most common examples of an EA is a Genetic Algorithm (GA), which is based upon Darwinian Evolution [46, 65].

Often, the term EA is reserved for algorithms that contain the notion of evolving solutions. However, for the safe of brevity, we define an EA as any stochastic local search algorithm. For example, Particle Swarm Optimization (PSO) is a Swarm Algorithm based upon the swarming of insects or schooling of fish [22, 29, 48]. While PSO is biologically inspired, the update rules lack the notion of biological evolution. Additionally, our definition of EA covers stochastic search algorithms that are not biologically inspired, such as Hill Climbing.

EAs are popular search algorithms for several reasons. First, the algorithm is a *weak method*, which means it can be applied to a wide variety of search problems, often with no changes to the algorithm [89, 100]. This "black box" optimizer allows a user to implement a single algorithm that can then be applied to a wide range tasks [22]. Another reason EAs are popular is that they are often very successful in finding good solutions to difficult problems [89]. For example, if the problem has complex, nonlinear variable interactions, a user does not have to worry about making simplifying assumptions about the problem to allow for analytically solving the problem. Instead, the black box can be applied to solve the problem without any changes made to the task. Also, their performance on complex problems may often be better than problem-specific algorithms. For example, Fortier *et al.* presented a GA that outperformed domain-specific algorithms on abductive inference in Bayesian networks [33]. Finally, EAs are popular because their memory footprint is usually a constant or bounded amount [89]. One domain-specific algorithm for abductive inference in Bayesian networks is called mini-bucket elimination, which is time-and-space exponential in the induced width of the Bayesian networks ordered moral graph [23, 24]. However, the space complexity for PSO is a constant amount that is dependent only on the number of individuals in the population.

We will begin to define an EA as follows, but first we formally define what an optimization problem is. Assume that, for a given task to solve, there exists a function $f : \mathbf{D}^N \to \mathbb{R}$ with domain $\mathbf{D}^N$ with parameters $\mathbf{X} = \langle X_1, X_2, \ldots, X_N \rangle$. We refer to X_i as a dimension or variable in the function f. $\mathbf{D} = \langle D_1, D_2, \ldots, D_N \rangle$ and each D_i defines a domain for each variable X_i [103, 109]. The goal of *optimization* is to find a set of values for $\mathbf{x}_g = \langle x_1, x_2, \ldots, x_n \rangle$ such that $f(\mathbf{x}_g) \leq f(\mathbf{x})$ for all $\mathbf{x} \in \mathbf{D}$. If $\mathbf{x}_g$ is less than or equal to all other points in the search space, it is called a *global optimum*. *Minimization* attempts to find a set of values minimizing f, whereas *maximization*

tries to find a set of values maximizing f. Throughout this dissertation, we will assume we are dealing with minimization problems when describing algorithms.

However, finding a local optimum is often very difficult and often one can settle for finding a $\mathbf{x}_l$ such that $f(\mathbf{x}_l) \leq f(\mathbf{x})$ for all $\mathbf{x} \in \mathbf{D'}$, where **D'** is a subset of the search space. For example, **D'** may be the sets of points within some distance of $\mathbf{x}_l$. This is referred to as finding the *local optimum*.

EAs contain a population of D individuals at time t, which we can represent as $\boldsymbol{\mathcal{P}}^t = \{\mathbf{P}_1, \mathbf{P}_2, \ldots, \mathbf{P}_D\}$. Note that some EAs have a population size of one, which we will discuss later. Additionally, some EAs will also change the size of the population $\boldsymbol{\mathcal{P}}^t$ throughout time. Each individual $\mathbf{P}_i$ represents a vector $\langle X_{i,1}, X_{i,2}, \ldots, X_{i,n} \rangle$ that represents a candidate solution. These solutions have a *fitness* or *score* S_i, which in this dissertation will always be equal to $f(\mathbf{P}_i)$.

Algorithm 2.1 presents a general process for an EA. An EA first performs a one-time initialization of the individuals. Then, each candidate solution is modified according to a set of defined update rules. Ideally these modifications to the candidate solutions cause the average fitness of the population to increases. In addition, these modifications are usually stochastic in nature, enabling a broader exploration of the search space. After the evaluating all of the individuals on the fitness function, the EA updates the solution $\mathbf{X}_{Ans}$ that is returned by the algorithm . The fitness of $\mathbf{X}_{Ans}$ is denoted as S_{Ans}. This process is repeated until some stopping criterion is met [29]. We refer to an update to the entire population as an *iteration*.

The selection of final solution $\mathbf{X}_i$ is algorithm dependent. For example, GA's, often return the individual with the best current fitness after the algorithm has stopped at time t_E. This is represented as

$$\mathbf{X}_{Ans} = \mathbf{P}_i \text{ where } S_i \leq S_j \; \forall \; \mathbf{P}_j \in \boldsymbol{\mathcal{P}}^{t_E}$$

Algorithm 2.1: General Evolutionary Algorithm

Input: Function f to optimize
Output: Solution $\mathbf{X}_{Ans}$

1: $\boldsymbol{\mathcal{P}}^0 \leftarrow$ Initialize Individuals
2: $t \leftarrow 0$
3: **repeat**
4: Update Individuals $\boldsymbol{\mathcal{P}}^t$
5: Evaluate Individuals on f
6: $\mathbf{X}_{Ans} \leftarrow$ Get Candidate Solution from $\boldsymbol{\mathcal{P}}^t$
7: $t \leftarrow t + 1$
8: **until** Stopping Criteria Met
9: **return** $\mathbf{X}_{Ans}$

where t_E is the number of iterations the EAs performs. Other algorithms, such as PSO, return the best the solution found throughout the entire search process:

$$\mathbf{X}_{Ans} = \mathbf{P}_i \text{ where } S_i \leq S_j \; \forall \; X_j \in \boldsymbol{\mathcal{P}}^t \wedge \; \forall \; t \leq t_E.$$

Additionally, there are several ways that the stopping criterion can be defined. However, in this dissertation, we will only use the following two methods.

- **Set Iterations** — Stop when a defined total number of iterations have been performed. With this method, the user must specify the value for the maximum number of iterations t_E. This causes the EA to stop updating when $t_E < t$. Often this criteria is used because it allows for the specification of only a single parameter and allows the user to to have a fine control for how may iterations are performed. One drawback, however, is that if it is not set correctly, the algorithm may perform poorly because the EA was not allowed to run for enough iterations. Conversely, if it is set to high, the EA may spend an unnecessary

amount of time updating the individuals when a good solution has already been located.

- **Stagnation** — Stop when the fitness of $\mathbf{X}_{Answer}$ fails to improve by a certain value over a set number of iterations. This version requires the user to not only specify the number of iterations L for stagnation to occur, but also the maximum amount of change ϵ that is considered to be stagnation. Formally, this is represented as $|S_{Ans}^{t+L} - S_{Ans}^{t}| \leq \epsilon$ [29, 90].

As discussed, the choice of the different function implementations in the EA from Algorithm 2.1 is dependent on the specific EA algorithm. Though there are many more stopping criteria than the ones discussed here, this dissertation focuses on only four: Hill Climbing, Genetic Algorithms, Differential Evolution, and Particle Swarm Optimization.

2.1.1 Hill Climbing

One of the simplest EAs is the Hill Climbing (HC) algorithm, which is shown in Algorithm 2.2. In most cases, the algorithm works by maintaining only one individual, i.e., a population size of one. The HC shown in Algorithm 2.2 gives a generalized version of HC that allows for multiple individuals, which is used in experiments later in this dissertation.

During each iteration, an individual examines all positions neighboring its current location (line 5). Note there are various ways to generate neighbors of the current solution. For example, one way is to generate all points that are an ϵ distance away from the current solution [50]. However, this may generate an overwhelming number of solutions to evaluate and therefore may be unfeasible in practical applications. An alternate method is to generate a specific number of points based on random changes to the current solution. In this dissertation, we

use a neighborhood function that generates all points that differ by an ϵ distance in only one dimension.

If there is a neighboring solution with better fitness, then the individual moves to the new location (lines 8 - 10). This process is repeated until the individual can no longer move to a neighbor with a better fitness. Because hill climbing is a greedy search and allows only the individual to move to better locations, it can become trapped in local optima. There have been several extensions to hill climbing, such as simulated annealing, that allow the individual to move to locations with worse locations [51, 67]. For the purpose of this dissertation, we restrict ourselves to simple HC.

Algorithm 2.2: Hill Climbing Algorithm

Input: Function f to optimize
Output: Set of values $\mathbf{X}_{Ans}$

1: $\boldsymbol{\mathcal{P}}^0 \leftarrow$ Initialize Individuals
2: $t \leftarrow 0$
3: **repeat**
4: **for all** $\mathbf{P}_i \in \boldsymbol{\mathcal{P}}^t$ **do**
5: $\boldsymbol{\mathcal{P}}_N \leftarrow$ Neighbors($\mathbf{P}_i$)
6: Evaluate Population $\boldsymbol{\mathcal{P}}_N$ on f
7: $\mathbf{P}_{Best} \leftarrow$ Best Solution in $\boldsymbol{\mathcal{P}}_N$
8: **if** $S_{Best} < S_i$ **then**
9: $\mathbf{P}_i \leftarrow \mathbf{P}_{Best}$
10: **end if**
11: **end for**
12: $\mathbf{X}_{Ans} \leftarrow$ Get Candidate Solution from $\boldsymbol{\mathcal{P}}^t$
13: $t \leftarrow t + 1$
14: **until** Stopping Criteria Met
15: **return** $\mathbf{X}_{Ans}$

2.1.2 Genetic Algorithms

One of the most well-known Evolutionary Algorithms is the Genetic Algorithm (GA). GAs were inspired biologically by the idea of survival of the fittest. Each individual in a GA acts like a chromosome and is modified in a manner that mimics genetics [46, 65]. The pseudocode is shown in Algorithm 2.3.

Algorithm 2.3: Genetic Algorithm

Input: Function f to optimize
Output: Set of values $\mathbf{X}_{Ans}$

1: $\boldsymbol{\mathcal{P}}^0 \leftarrow$ Initialize Individuals
2: $t \leftarrow 0$
3: **repeat**
4: $\quad \boldsymbol{\mathcal{P}}' \leftarrow \emptyset$
5: $\quad$ **for** $i = 1$ to D **do**
6: $\quad\quad \boldsymbol{\mathcal{P}}_{Selected} \leftarrow$ Select Indivduals from $\boldsymbol{\mathcal{P}}$
7: $\quad\quad \mathbf{P}_i \leftarrow$ Perform Crossover $\boldsymbol{\mathcal{P}}_{Selected}$
8: $\quad\quad$ Mutate $\mathbf{P}_i$
9: $\quad\quad \boldsymbol{\mathcal{P}}' \leftarrow \boldsymbol{\mathcal{P}}' \cup \{\mathbf{P}_i\}$
10: $\quad$ **end for**
11: $\quad \boldsymbol{\mathcal{P}}^t \leftarrow \boldsymbol{\mathcal{P}}'$
12: $\quad t \leftarrow t + 1$
13: **until** Stopping Criteria Met
14: $\mathbf{X}_{Ans} \leftarrow$ Get Best Candidate Solution from $\boldsymbol{\mathcal{P}}^t$
15: **return** $\mathbf{X}_{Ans}$

In a GA, the update function from the EA in Algorithm 2.1 is replaced with lines 5 - 11 in Algorithm 2.3. During each iteration, or *generation*, there are two main steps that update the D individuals in the population: reproduction and mutation.

1. *Reproduction:* The candidate solutions first reproduce with one another. This is done in two steps: selection and crossover (lines 6 - 7).

(a) *Selection:* During selection, pairs of individual are selected for reproduction. There are several selection methods, such as fitness proportionate, tournament, and rank-based. All of these selection strategies are designed such that individuals with higher fitness are more likely to be selected for reproduction [22, 29].

(b) *Crossover:* Once the individuals, or *parents*, are selected, these parents are combined to create child individuals. Randomly selected dimensions from the first selected individual are taken and combined with the remaining dimensions of the second selected individual. The parents can be recombined with one another in several different ways, such as one-point, two-point, and uniform crossover [22].

2. *Mutation:* Each dimension in the individual in the new population may be mutated to introduce new information into the population. This is done by changing randomly selected dimensions in the individual to a randomly selected value. There are several different methods we can use, such as uniform mutation, inorder mutation, and Gaussian mutation. All of these methods are similar in that dimensions of an individual are changed with a probability p_m [29, 89].

Once reproduction and mutation have been performed, the child individuals are added to the population for the next generation [22, 29]. As with all Evolutionary Algorithms, these steps are repeated until the stopping criterion is met.

2.1.3 Differential Evolution

Differential Evolution (DE) is another Evolutionary Algorithm that has been found to perform successfully on a variety of optimization problems. It was originally proposed as an alternate optimization algorithm for nonlinear and non-differentiable

functions [105]. One reason for DE's popularity is that it has simple update equations and a low number of control parameters. Additionally, DE has been found to perform well in terms of accuracy and convergence rate on a wide range of problems [20, 29]. Algorithm 2.4 presents a the most commonly used version of DE. The DE's update has three main steps in each iteration: mutation, crossover, and selection.

Algorithm 2.4: Differential Evolution

Input: Function f to optimize
Output: Set of values $\mathbf{X}_{Answer}$

$\boldsymbol{\mathcal{P}}^0 \leftarrow$ Initialize Individuals
$t \leftarrow 0$
repeat
  for $i = 1$ **to** D **do**
    $\mathbf{P}_a, \mathbf{P}_b, \mathbf{P}_c \leftarrow$ Select Individuals from $\boldsymbol{\mathcal{P}}^t$
    for $j = 1$ **to** N **do**
      $M_{i,j}^{t+1} \leftarrow X_{a,j}^t + F\left(X_{b,j}^t - X_{c,j}^t\right).$
    end for
    $R_{Int} \leftarrow U_{\mathbb{Z}}(1, N)$
    for $j = 1$ **to** N **do**
      if $U_{\mathbb{R}}(0, 1) < p_c$ or $R_{Int} = j$ **then**
        $U_{i,j}^{t+1} \leftarrow M_{i,j}^{t+1}$
      else
        $U_{i,j}^{t+1} \leftarrow X_{i,j}^t$
      end if
    end for
    if $f(\mathbf{U}_i^{t+1}) < f(\mathbf{P}_i^t)$ **then**
      $\mathbf{P}_i \leftarrow \mathbf{U}_i^{t+1}$
    end if
  end for
  $t \leftarrow t + 1$
until Stopping Criteria Met
$\mathbf{X}_{Ans} \leftarrow$ Get Candidate Solution from $\boldsymbol{\mathcal{P}}^t$
return $\mathbf{X}_{Ans}$

1. *Mutation:* During mutation, a set of individuals are selected from the population $\boldsymbol{\mathcal{P}}^t$ at time t and combined to create a *mutation* vector (lines 5 - 8). There are several ways that this can be performed. In this dissertation, we restrict ourselves to what is referred to as the "DE/rand/1" method. In DE/rand/1, an individual i is mutated by first selecting three random individuals a, b, and c, from the population $\boldsymbol{\mathcal{P}}^t$ at time t where $a \neq b \neq c \neq i$. Dimension j of particle i's mutation vector $\mathbf{M}_i = \langle M_{i,1}, M_{i,2}, \ldots, M_{i,n} \rangle$ is calculated as

$$M_{i,j}^{t+1} = X_{a,j}^t + F\left(X_{b,j}^t - X_{c,j}^t\right).$$

   The values $X_{a,j}^t$, $X_{b,j}^t$, $X_{c,j}^t$, and $X_{d,j}^t$ are the current position of dimension j of individuals a, b, c, and d at time t. F is an integer in $[0, 2]$ and gives a weight to the difference between $X_{b,j}^t$ and $X_{c,j}^t$.

2. *Crossover:* This step creates a trial vector $\mathbf{U}_d$ for individual d by combining parts of the individual's mutation vector with its current position, shown in lines 9 - 16 in Algorithm 2.4. One of the most commonly used crossover operators is binomial crossover and proceeds as follows: Let $M_{d,j}^{t+1}$ be the value at position j in individual d's mutation vector at time $t+1$. Similarly, let $X_{d,j}^t$ be the value at position j in individual d's current position at time t. Individual d's trial vector $\mathbf{U}_d^{t+1}$ at time $t+1$ is computed as

$$U_{d,j}^{t+1} = \begin{cases} M_{d,j}^{t+1} & U_{\mathbb{R}}(0,1) < p_c \text{ or } U_{\mathbb{Z}}(1,N) = d \\ X_{d,j}^t & \text{otherwise} \end{cases}$$

   where p_c is a crossover rate specifying the probability of the items from the trial vector which will pass onto the offspring vector. The equation has two random values $U_{\mathbb{R}}(0,1)$ and $U_{\mathbb{Z}}(1,N)$. $U_{\mathbb{R}}(0,1)$ is a real-valued random number

between $[0, 1]$ that is generated for every dimension. $U_{\mathbb{Z}}(1, N)$ is a random integer between $[1, N]$ that is generated once during each individual's crossover phase. This ensures that, even with a very low crossover rate, an individual's trial vector is guaranteed to have at least one value that is different than its current position.

3. *Selection:* In lines 17 - 19 the trial vector is evaluated to see if the individual accepts the changes by comparing the fitness of the trial vector to that of the current position. If the fitness of the trial vector is either better than or equal to that of the current position for the individual, the current position is set to the trial vector. Otherwise, the individual keeps its current position. This guarantees that each new generation of individuals either gets better or remains the same. Similar to the GA, the DE algorithm performs these three steps for each individual until convergence occurs [20, 105].

In addition to the DE update equations presented above, several other update equations have been presented for the mutation and crossover steps. However, it has been noted that no single set of update equations has been found to perform best for all problems [20].

2.1.4 Particle Swarm Optimization

Particle Swarm Optimization (PSO) is a population-based approach to optimizing a function, usually over a continuous set of variables. PSO was developed by Kennedy and Eberhart and is based on the behavior of fish schools and bird flocks swarming towards food sources [48].

Whereas GAs use a population of individuals that reproduce with one another, PSO uses a swarm of particles that "fly" around the search space. In addition to the vector $\mathbf{X}_i$ that represent a candidate solution, particles use a velocity vector $\mathbf{V}_i$

Algorithm 2.5: Particle Swarm Optimization

Input: Function f to optimize
Output: Set of values $\mathbf{X}_{Ans}$

$\boldsymbol{\mathcal{P}}^0 \leftarrow$ Initialize Individuals
$t \leftarrow 0$
repeat
  for $i = 1$ **to** D **do**
    $\mathbf{V}_i = \omega\mathbf{V}_i + U_{\mathbb{R}}(0, \phi_1) \otimes (\mathbf{pBest}_i - \mathbf{P}_i) + U_{\mathbb{R}}(0, \phi_2) \otimes (\mathbf{gBest} - \mathbf{P}_i)$
    $\mathbf{P}_i = \mathbf{P}_i + \mathbf{V}_i$
    if $f(\mathbf{P}_i) < f(\mathbf{pBest}_i)$ **then**
      $\mathbf{pBest}_i \leftarrow \mathbf{P}_i$
    end if
    if $f(\mathbf{P}_i) < f(\mathbf{gBest})$ **then**
      $\mathbf{gBest} \leftarrow \mathbf{P}_i$
    end if
  end for
  $\mathbf{X}_{Ans} \leftarrow \mathbf{gBest}$
  $t \leftarrow t + 1$
until Stopping Criteria Met
return $\mathbf{X}_{Ans}$

to control how the particles move in the space. Each particle keeps track its own best position found in a vector $\mathbf{pBest}_i$ and the best position discovered by the entire swarm **gBest**. Algorithm 2.5 shows a general version of the PSO algorithm. During each iteration, a particle's position is updated as follows:

$$V_{i,j} = \omega V_{i,j} + U_{\mathbb{R}}(0, \phi_1) \otimes (pBest_{i,j} - X_{i,j}) + U_{\mathbb{R}}(0, \phi_2) \otimes (gBest_j - X_{i,j}) \tag{2.1}$$

$$X_{i,j} = X_{i,j} + V_{i,j} \tag{2.2}$$

$U_{\mathbb{R}}(0, \phi_1)$ and $U_{\mathbb{R}}(0, \phi_2)$ are random numbers between 0 and ϕ_1 and ϕ_2. This cause the particles have more random paths, aiding in the exploration of the search space.

$X_{i,j}$ and $V_{i,j}$ is the position and velocity for dimension j of particle $\mathbf{P}_i$ and ω is an inertia value that helps the velocity values from growing out of control.

The PSO algorithm first initializes a random swarm of particles over the search space. During each iteration, a particle's fitness is calculated using the fitness function. If the fitness of the particle is better than that of its local best or global best, the local and global best are updated accordingly. Finally, the velocities and positions of the particles are updated according to the update equations. This process is repeated until some convergence criterion is satisfied [48].

2.2 Related Work

While the update rules of the EAs are effective for finding good solutions to a wide range of problems, drawbacks to the update rules still exist [66, 99]. For example, an algorithm may locate good values for a particular variable but then lose that information in later iterations. Conversely, an algorithm may believe that a specific value for a variable correlates to high fitness when in reality the value is suboptimal.

In a GA, this is called hitchhiking. Simply stated, hitchhiking causes poor values become associated with good schemas [99]. Hitchhiking was first noticed by Mitchell *et al.* when applying a GA to a set of problems called the Royal Road, which were though to be easily solved by a GA. However, the authors noticed that on certain versions of the Royal Road, the GA actually performed worse than expected. This was due to hitchhiking: a bad value or set of values often got entangled or associated with a set of good values. As the GA copies the good values to future generations, the bad values also got carried along, hence, the term, hitchhiking.

Similarly, Van den Bergh and Engelbrecht discovered that PSO can be prone to what is called *two steps forward and one step back* (TSFOSB), which happens when

near-optimal or nearly optimal parts of a individual's current position are thrown away when the rest of an individual's position causes the individual to have low fitness [113].

The authors use the following example: suppose the PSO is optimizing a three-dimensional function $f(\mathbf{X}) = ||x - 20||^2$ with a global minimum at $\langle 20, 20, 20 \rangle$. Additionally, the global best **gBest** is at point $\langle 17, 2, 17 \rangle$ and an individual $\mathbf{P}_i$ is at $\langle 5, 20, 5 \rangle$. Given these positions, the fitness of **gBest** and $\mathbf{P}_i$ are 342 and 450, respectively. During the update, individual $\mathbf{P}_i$ will be pulled towards **gBest**. For the sake of discussion, suppose that the new location for $\mathbf{P}_i$ is $\langle 15, 5, 15 \rangle$, which has a fitness of 275. Because this new position has a better fitness than the original **gBest**, the algorithm will update the global best to $\langle 15, 5, 15 \rangle$. While the overall fitness of the global best improved, the quality of $\mathbf{P}_i$'s second variable decreased, and in fact, the previous value for the second variable was optimal. This information has thus been lost and the PSO must spend more iterations rediscovering this value [114].

While this is a simple example with only three variables where only one variable's information deteriorated, it can quickly be generalized to any number of variables. For example, a scenario could be conceived where in a problem with ten variables, the good values for four variables are lost during an update. Even worse, there could exist situations where the PSO knows the optimal values for $N - 1$ variables, but loses all that information because of a new value for the Nth variable [112, 114].

Differential Evolution also suffers from TSFOSB [72,99]. Using the same function for the PSO, an individual $\mathbf{P}_i$ is at $\langle 5, 20, 5 \rangle$ and has a fitness of 450. Suppose that after mutation and crossover, the new individuals position is $\langle 15, 5, 15 \rangle$ and has a fitness of 275. Because the fitness of the new position is better than before, during selection, the individual will accept the new position $\langle 15, 5, 15 \rangle$. Just like in the PSO,

the optimal information for the second variable was lost by the individual, and DE will need to spend more iterations rediscovering the lost information.

One possible way to help fix these shortcomings of EAs is to evaluate the function after updating each variable [112, 114]. However, this may result in an increase the number of fitness evaluations required by the algorithm. Additionally, this method may apply only to certain EAs, like PSO, and not generalize to ones like GA. A more commonly used method is to use a set of disjoint populations [112, 120]. By having disjoint subpopulations, the algorithm is able to maintain better diversity in the individuals [117], which in turn helps prevent hitchhiking and TSFOSB.

2.2.1 Island Model

One of the most common multi-population algorithms is the Island Model. The Island Model was first proposed by Grosso and was used to simulate the interactions of several evolving populations [39]. One reason for its popularity is that it offers an easy way in which to parallelize a GA. In this model, there exist M disjoint populations that evolve their individuals independent of one another. However, Grosso's algorithm allows for interactions between the populations, which is referred to as *migration*. After a specified number of iterations, each population sends a number of individuals to its neighbor. When a population receives a set of individuals from its neighbors, the population assimilates the new individuals into its population [5,11]. This process is repeated until convergence is reached.

There has been an extensive amount of research of various parameters of the Island Model, such as the rate of migration and migration topologies [5, 38, 118]. For example, Ishimizu and Tagawa compared different migration topologies that control which populations interact with one another [47]. The authors found that by using a

ring, torus, or hypercube migration topology, better performance was achieved than just using a fully connected set of populations [38, 47].

There has also been investigation of how to set the the migration rate in the Island Model [39, 81]. Grosso showed that too high of a rate of migration caused the Island Model to behave like a single-population GA, while too little migration causes the algorithm to converge toward worse solutions. Others have explored having each population contain a different representation of the problem. For example, Skolicki and De Jong proposed an Island model in which each population contained a different representation to the solution, such as binary or gray-encoding [101]. The authors used a transformation step during migration that allowed for one solution from one population to be inserted into a different population.

Because the populations in the Island Model still represent full solutions to the problem, each population is still susceptible to hitchhiking and TSFOSB. Consequently, the entire set of populations are still susceptible to hitchhiking and TSFOSB. One way to reduce the probability of each population encountering hitchhiking and TSFOSB is to have each population optimize over a subset of variables in the problem [81].

2.2.2 Cooperative Coevolutionary Algorithms

Cooperative Coevolutionary algorithms (CCEA), originally proposed by Potter and De Jong, are some of the earliest algorithms to subdivide an optimization problem in an evolutionary setting. [82]. In their work, the authors developed an algorithm called the Cooperative Coevolutionary Genetic Algorithm (CCGA) that uses *subspecies* to represent non-overlapping subcomponents of a potential solution. Complete solutions are then built by assembling the current best subcomponents of the subspecies. Their work showed that in most cases CCGA significantly

outperformed traditional GAs. Only in cases where the optimization function had high interdependencies between the function variables (i.e., epistasis) did CCGA struggle because relationships between variables were ignored.

More dynamic versions of CCGA, that allow for subpopulations to evolve over time, have been proposed [83]. When stagnation is detected in the population, a new subpopulation is randomly initialized and then added to the set of subpopulations. Similarly, a subpopulation is removed if it makes only small contributions to the overall fitness. Because of the dynamic subpopulations, the possibility exists that two subpopulations may overlap with another. However, there is no guarantee that subpopulations will overlap, and the algorithm does not have a function to resolve discrepancies between the subpopulations. The authors were able to demonstrate that their algorithm could evolve the correct number of subpopulations and was competitive with domain-specific algorithms on training cascade networks [83].

This idea of CCEA was extended by Van den Bergh and Engelbrecht to use PSO to train neural networks [113]. In their paper, the authors tested four fixed subpopulation architectures of their own design: Plain, Lsplit, Esplit, and Nsplit. Plain used a full single-population PSO. Lsplit divides the weights in the neural network by layer, whereas Esplit serializes the weights in the network and divides the vector into two even subswarms. Nsplit extends Lsplit by creating subswarms for all weights entering a single node in the neural network. Comparing these four different architectures, the success of the algorithms was highly dependent on the architecture used due to the interdependencies between the variables. By keeping variables with interdependencies together, the algorithm was more effective at exploring the search space [113].

Later, Van den Bergh and Engelbrecht extended their work by applying it to a wider range of optimization problems [114]. Van den Bergh and Engelbrecht

introduced the cooperative PSO (CPSO), which was able to get around the problem of losing good values since each dimension is optimized by a single subpopulation. The paper introduced CPSO as a general search algorithm not restricted to just training neural networks. However, one drawback to CPSO is that it can become trapped in what the authors call *pseudominima*, which are places that are minima when looking at a single dimension but are not local minima over the entire search space. To avoid this problem, the authors describe a hybrid algorithm that alternates between CPSO and PSO. The result is an algorithm that always outperforms PSO and is competitive with but more robust than CPSO.

CCEA has also been applied to Differential Evolution. Shi *et al.* proposed a simple extension of CCGA to DE, called CCDE [99]. Other extensions have been more complex, such as those presented by Yang *et al.*, where the authors developed a weighted cooperative algorithm that used DE to optimize problems with over 100 dimensions [123]. This algorithm utilized a weighting scheme to allow for the evolution of subpopulations where the function was optimized within the subpopulations. The authors' algorithm was found to outperform regular CCEA algorithms on most of the test functions explored [123].

A variation of CCEA that used evolving subpopulations was also proposed by Li and Yao [62]. Here, the subpopulations were allowed to grow or to shrink when stagnation was detected, creating a wider range of variable groups. The authors showed that their algorithm performed better than others on functions that had complex multi-modal fitness landscapes, but performed slightly worse than PSO on unimodal functions. They noted that, while subpopulations of random variables perform well, there should exist more intelligent ways of creating subpopulations.

One open question with CCEAs is how to evaluate an individual. Specifically, when evaluating an individual, which values should be pulled from other subpopula-

tions to allow for calculating fitness? This is referred to as *collaboration* in the CCEA and has some similarity to migration in the Island Model in that both of these steps allow the otherwise independent populations to interact with one another. Wiegand *et al.* presented some of the earliest work looking at the collaboration in CCEA and presented results comparing several different methods [119]. They proposed methods that used the best, random, or the worst individuals from other swarms to evaluate an individual from a different swarm. Additionally, they explored combinations of the the above methods by varying the number of collaborators. The authors found that using the best known individuals generally provided the best performance and that only one or two collaborators are required.

Other work with CCEA has focused on its convergence properties. Wiegand *et al.* developed a framework using evolutionary game theory to model the interaction of two subpopulations [120]. The authors calculated the next generation by using a payoff matrix A and proved that when trajectories converge to a fixed point, the populations become homogeneous. Additionally, the authors showed that subpopulations may converge to points with suboptimal fitness values. Finally, the authors used their framework to investigate the effects of uniform crossover and bit-flip mutation.

One of the drawbacks to this approach is that the authors assumed infinite populations in each of the subpopulations and that an individual's fitness is given as the average fitness using all individuals from the other subpopulation during collaboration [75]. Panait *et al.* relaxed this requirement by modeling the fitness evaluation of an individual as the average fitness of N randomly chosen individuals from the other subpopulation. Additionally, the authors used evolutionary game theory as a way to visualize different convergence properties of CCEA [75].

Later work by Panait argued that the primary reason for poor performance in CCGAs was the poor selection of individuals during collaboration [74]. The author

went on to use a refined evolutionary game theory model and showed that a CCGA will converge to the globally optimal solution when the collaboration process is set properly. Additionally, Panait verified previous results by Wiegand *et al.* showing that a collaboration process that uses the best individuals from subpopulations outperforms a collaboration process that uses the average or worst individuals.

Other theoretical work with CCGAs has investigated their *robustness*, which the authors define as the ability of an algorithm to consistently find good solutions. Wiegand and Potter first defined a framework for characterizing robustness in evolutionary algorithms [121]. Using this framework, the authors were able to show that CCGAs exploit this robustness during search and empirically demonstrate how this is done [121].

2.2.3 Overlapping Swarm Intelligence

Overlapping Swarm Intelligence (OSI) is a version of FEA that uses PSO as the underlying optimization algorithm. Introduced in 2012, OSI works by creating multiple swarms that are assigned to overlapping subproblems [41]. It was first used as a method to develop energy-aware routing protocols for sensor networks that ensure reliable path selection while minimizing energy consumption during message transmission [41]. OSI was shown to be able to extend the life of the sensor networks and to perform significantly better than preexisting energy-aware routing protocols.

The OSI algorithm was later extended by Ganesan Pillai and Sheppard to learn the weights of deep artificial neural networks [80]. In that work, each swarm represents a unique path starting at an input node and ending at an output node. A common vector of weights is also maintained across all swarms to describe a global view of the network, which is created by combining the weights of the best particles in each

of the swarms. The authors showed that OSI outperforms several other PSO-based algorithms as well as standard backpropagation on deep networks.

A distributed version of OSI was subsequently developed by Fortier *et al.* called Distributed Overlapping Swarm Intelligence (DOSI) [35]. In that paper, a communication and sharing algorithm was defined that allowed swarms to share values while also competing with one another. The key distinction from OSI was that a global solution was not used for fitness evaluation. The authors were able to show that DOSI's performance was close to that of OSI's on several different networks but there were several instances when OSI outperformed DOSI.

OSI and DOSI have also been used for inference tasks in Bayesian networks, such as abductive inference, where the task is finding the most probable set of states for some nodes in the network given a set of observations. Fortier *et al.* used OSI [33] and DOSI [31] to perform full and partial abductive inference in Bayesian networks. The authors were able to show that OSI and DOSI outperformed several other population-based and traditional algorithms, such as PSO, GA, simulated annealing, stochastic local search, and mini-bucket elimination.

Other applications of OSI and DOSI include learning Bayesian networks. Fortier *et al.* adapted OSI to learn the structure of Bayesian classifiers by allowing subswarms to learn the links for each variable in the network, where each variable represents an attribute in the data [32]. For each variable in the network, two subswarms were created: one comprised of the incoming links and one of the outgoing links. The authors were able to show that in most cases OSI was able to outperform the competing approaches significantly.

When learning Bayesian networks, latent or unobserved variables are often introduced into the network. Fortier *et al.* used OSI to learn the parameters of these latent variables [34]. A subswarm was created for each node with unlearned

parameters and all of the variables in that node's Markov blanket. The authors were able to show that OSI outperformed the competing approaches, including Expectation Maximization (EM), Monte-Carlo EM, Genetic Algorithm EM, and Age-Layered EM, and that the amount of overlap between the subswarms can impact the performance of OSI.

Fortier later adapted OSI to be a general structure learner for Bayesian networks [36]. The authors showed that OSI and PSO approaches to structure learning outperformed domain-specific algorithms, like K2. Additionally, Fortier showed that an OSI approach for learning the structure and parameters for latent variables simultaneously outperformed Expectation Maximization.

While DOSI allows for a full distributed implementation, it requires an increase in computational complexity due to the communication required between neighboring subswarms. Butcher *et al.* investigated the performance of DOSI when full consensus was not required between neighboring subswarms and showed that DOSI still performs well even with only a moderate amount of consensus between subswarms [10]. Additionally, the paper contained the first definition of Distributed Factored Evolutionary Algorithms (DFEA), which is a generalization of DOSI that allows for any optimization algorithm.

2.2.4 Related Approaches

In addition to CCEA and OSI, other work exists with concepts similar to CCEA and OSI. For example, a relatively new approach by Srivastava *et al.* is a variation of backpropagation called Dropout in which random parts of the neural network are deleted and the resulting "thinned" network is trained [104]. After training several thinned networks, these networks are combined to create the full neural network. The authors found that Dropout outperformed standard backpropagation on full neural

networks. In addition, Dropout performed well on other types of networks, such as Restricted Boltzmann Machines and Deep Belief Networks.

Tosun presented an approach to training Restricted Boltzman Machines (RBMs) that was heavily influenced by OSI [110, 111]. Specifically, Tosun developed an algorithm called Partitioned Learning that trains an RBM by subdividing the weights of the network. During each step, the algorithm trains only a specified set of weights that are independent of the weights in the rest of the network. The algorithm works by first splitting the RBM in a sets of weights and training them independently using Contrastive Divergence. After training the partitions, the algorithm generates a new set of partitions. However, the new set of partitions contain fewer splits, and consequently, partitions more sets of values to update. This process is repeated until the full RBM is trained.

Tosun provided experiments on both overlapping and nonoverlapping partitions and showed that overlapping partitions resulted in lower reconstruction error but required more operations. Furthermore, the author was able to show that his approach had lower reconstruction error and also took less time than regular Contrastive Divergence when applied to image reconstruction. Finally, Tosun adapted his approach to other tasks, such as RBM classifiers and Time Series Prediction [110].

Another example is the Multifactorial Evolutionary Algorithm (MFEA) introduced by Gupta *et al.* [40]. MFEA uses a single-population to simultaneously solve multiple optimization problems. In their algorithm, the authors assigned each individual in the population to optimize one problem from the set of problems MFEA is optimizing over. Each individual is only allowed to interact only with individuals that are assigned the same problem. Because an individual is assigned only a single function to optimize, MFEA essentially turns into a multi-population algorithm. By optimizing different tasks that have additional influence over the search process,

MFEA creates an exchange of information between the tasks. This creates an overlap between the tasks, which is similar in concept to how OSI uses sets of overlapping subpopulations. However, the focus of OSI is on subdividing a single function (task) into overlapping sets of variables and using subpopulations to optimize over sets of variables.

2.3 Conclusion

While EAs have been successfully applied to a wide range of problems, there are still drawbacks to the algorithms. One of the most commonly used variation of EA that addresses many of these issues is CCEA. CCEA is a general method that can be applied to any EA and often outperforms traditional single-population algorithms. However, CCEAs assume disjoint subpopulations. Furthermore, much of the previous work with CCEA fails to provide a general method for how to derive these subpopulations. In the next chapter, we present a new algorithm that directly addresses these these issues.

CHAPTER THREE

FACTORED EVOLUTIONARY ALGORITHMS

FEA is a generalization of EAs and OSI that allow for the subdivision of the optimization problem. In this chapter, we present a formal definition of FEA, including a framework for defining multi-population algorithms. First, we present our framework and demonstrate how different multi-population algorithms map to this framework. Then, in Section 3.2 we present the FEA algorithm and its subfunctions, followed by a complexity analysis of FEA. Finally, we compare four different versions of FEA that each use a different underlying optimization algorithm. In particular, we focus on how FEA performs using different algorithms like HC, PSO, GA, and DE. Each algorithm is then compared with single-population and Cooperative Coevolutionary (CC) versions of the underlying algorithm.

3.1 Framework for Multipopulation EA

A wide variety of multipopulation EAs exist. However, almost all of them share several commonalities. Here, we present a framework that allows for the definition of different multi-population EAs and use it to define several of the previously mentioned algorithms, allowing for a unification of the different approaches.

3.1.1 Defining the Framework

Similar to single-population EA's, we assume that we are given a function $f : \mathbf{D}^N \to \mathbb{R}$ with domain $\mathbf{D}^N$ to be optimized with parameters $\mathbf{X} = \langle X_1, X_2, \ldots, X_N \rangle$. $\boldsymbol{\mathcal{S}}$ represents the set of M populations $\{\boldsymbol{\mathcal{P}}_1, \boldsymbol{\mathcal{P}}_2, \ldots, \boldsymbol{\mathcal{P}}_M\}$. Each population $\boldsymbol{\mathcal{P}}_i$ is optimized over some set of variables $\mathbf{U}_i \subseteq \mathbf{X}$. Note that f can still be optimized over

Algorithm 3.6: Multiple Population Evolutionary Algorithm

Input: Function f to optimize
Output: Set of values $\mathbf{X}_{Ans}$

```
1: Initialize Populations 𝓢
2: t ← 0
3: repeat
4:    for i = 1 to D do
5:       Optimize f with Population 𝓟_i
6:    end for
7:    Interaction between Populations
8:    X_Ans ← Get Candidate Solution from 𝓢
9:    t ← t + 1
10: until Stopping Criteria Met
11: return X_Ans
```

the variables in $\mathbf{U}_i$ even if $\mathbf{U}_i \subset \mathbf{X}$. However, the fitness function requires values for variables that are not included in $\mathbf{U}_i$. We refer to these as the *remaining variables* and denote them as $\mathbf{R}_i = \mathbf{X} \setminus \mathbf{U}_i$.

Given this notation, we can present a general multi-population EA in Algorithm 3.6. The algorithm works as follows: the algorithm iterates over each population and allows them to update their individuals (lines 4 – 6). After each population updates its own individuals, the algorithm allows the populations to interact in some way followed and updates the candidate to be returned. This process is repeated until a stopping criterion is met. Note that the user must not only specify the stopping criteria for the entire multi-population, but also for each individual population.

3.1.2 Application of Subpopulation Framework

Given the above framework, we can now demonstrate how it is able to define a wide range of multi-population algorithms. We begin by showing that even a single-population EA fits the framework.

3.1.2.1 Single-Population Algorithms

In the case that the multi-population algorithm has only one population, then $\boldsymbol{\mathcal{S}} = \{\boldsymbol{\mathcal{P}}_1\}$. Additionally, $\mathbf{S}_1 = \mathbf{X}$. This is equivalent to the single-population algorithms discussed in Section 2.1. Additionally, because there is only one population, there is no need for Algorithm 3.6 to run line 7, which causes the populations to interact with one another.

3.1.2.2 Island Model

In the Island Model, $D > 1$. Additionally, $\mathbf{S}_i = \mathbf{X}$ for all $\boldsymbol{\mathcal{P}}_i \in \boldsymbol{\mathcal{S}}$. The Island Model iterates over all populations and allows each population to update its individuals independently of one another. While most work assumes that each population $\boldsymbol{\mathcal{P}}_i$ uses the same EA, there is nothing in the core Island Model that requires this. After updating each of the populations, the Island Model allows the populations to interact with one another by performing migration. The most commonly used method for migration is to take a set of individuals and send them to a set of different populations. A population that receives migrating individuals then adds them to its population.

There are two important considerations a user must make when implementing an Island Model. The first is the migration topology, which controls where individuals migrate to during the migration step, with the most popular topologies being rings, tori, and hypercubes. Another important consideration is how often migration should occur. As previously discussed, the Island Model is sensitive to the rate of migration; therefore, finding an appropriate value can be difficult.

3.1.2.3 CCEA CCEAs relax the Island Model such that each population or subpopulation does not represent full solutions and each subpopulation optimizes over disjoint sets of variables. Formally, this occurs when $D > 1$ and $\mathbf{U}_i \subset \mathbf{X}$ for all $\boldsymbol{\mathcal{P}}_i \in \boldsymbol{\mathcal{S}}$. Additionally, $\mathbf{U}_i \cap \mathbf{U}_j = \emptyset$ for all $\boldsymbol{\mathcal{P}}_i$ and $\boldsymbol{\mathcal{P}}_j$ in $\boldsymbol{\mathcal{S}}$. Because each population is only optimizing over a subset of values in $\mathbf{X}$, subpopulation $\boldsymbol{\mathcal{P}}_i$ optimizing over values $\mathbf{U}_i$ needs to know the values of $\mathbf{R}_i$ for local fitness evaluations. The fitness for an individual in $\mathbf{P}_i$ can then be calculated as $f(\mathbf{U}_i \cup \mathbf{R}_i)$.

CCEAs must define a process for finding values for $\mathbf{R}_i$, which the literature refers to as the *collaboration* step [119,120]. In almost all algorithms, these values are directly derived from other subpopulations. In the most commonly used process, the value for variable X_i in $\mathbf{R}_j$ is obtained from the best individual $\mathbf{P}_k$ in population $\boldsymbol{\mathcal{P}}_l$ where $X_i \in \mathbf{U}_l$. Note that because all subpopulations are disjoint, there exists only one population $\boldsymbol{\mathcal{P}}_j$ that optimizes over variable X_i for all variables $\mathbf{X}$. Collaboration is performed in line 7 of Algorithm 3.6.

While subpopulations in CCEAs do not interact in the same manner as the Island Model, the exchanging of values between subpopulations to update variables in the remaining set $\mathbf{R}_i$ creates an interaction between the subpopulations. The value subpopulation $\boldsymbol{\mathcal{P}}_i$ sends to $\boldsymbol{\mathcal{P}}_j$ for X_k will affect how $\boldsymbol{\mathcal{P}}_j$ evaluates individuals in its population. This is analogous to the migration step in the Island Model; subpopulations are interacting with one another.

3.1.2.4 MFEA While all of the previous methods assumed a single objective function, the above framework can be extended to allow for multiple fitness functions. Assume we are given a set of functions $\mathbf{f} = \{f_1, f_2, \ldots, f_K\}$ where each function is defined as $f_i : \mathbf{D}^N \rightarrow \mathbb{R}$ with domain $\mathbf{D}^N$. While the original work on MFEA allows for fitness functions with different size inputs, we assume here that each fitness

function has the same number of variables. However, our framework can be modified to allow for different sizes of input by setting each function f_i' to have an input of size N' where N' is equal to the fitness function with the most variables. Each function then uses a mapping of inputs f' to f'. A population $\boldsymbol{\mathcal{P}}_i$ is defined for each fitness function f_i.

During each update step, the population optimizes over its own individual fitness function. Note that the original MFEA assumes a GA being used to update individuals in the population. However, the interaction between the subpopulations is more complex than the Island Model. During the population interaction step, individuals are randomly selected from the populations. If the individuals are selected from the same population, crossover is performed to generate two new individuals that are added into the original subpopulation. However, if the individuals are from selected different populations, crossover is performed based on a cross-cultural crossover probability. If the probability allows the two individuals from different subpopulations to mate, two new individuals are produced that are randomly inserted into one of the parent's population. Otherwise, the individuals from different subpopulations undergo mutation to produce new individuals that are then inserted into their parents' population.

This process is directly analogous to the migration step in the Island Model. However, instead of directly inserting individuals into populations, the individuals selected for migration interact with individuals from other populations before finishing their migration.

3.2 Defining FEA

FEA is similar to cooperative EAs like CCGA and CPSO; however, FEA encourages subpopulations to overlap with one another. This overlap allows for

subpopulations to compete and share information. FEA is also a generalization of the OSI algorithm since it allows for any EA to be used as the underlying optimization algorithm. This allows for FEA to be part of a genera class of algorithms that includes CPSO, CCGA, OSI, and Island Model. Here, we present the three main steps of FEA: Update, Compete, and Share. We begin with defining the subpopulations in FEA.

3.2.1 Defining Factors

Using our framework for defining multipopulation algorithms, we define the subpopulations in FEA as follows. FEA uses a set of subpopulations, or *factors*, which are subsets of $\mathbf{X}$. Formally, we can represent this as $1 < D$, $\mathbf{U}_i \subset \mathbf{X}$, and $\bigcup_i \mathbf{U}_i = \mathbf{X}$ for all populations. Additionally, FEA encourages factors to overlap with one another. Without a loss of generality, assume every subpopulation overlaps some other subpopulation. To evaluate individuals, a factor must have a set of values for its remaining variables $\mathbf{R}_i$. FEA uses a *full global solution* $\mathbf{G} = \langle X_1, X_2, \ldots, X_N \rangle$ to fill in a factor's remaining variables.

Finally, we define a *factor architecture* which defines the variables the factors optimize. This also affects the number of factors created, the size of each factor, and the amount of overlap between two factors. Previous work has shown that generating efficient factor architectures is crucial for CCEA to be able to locate good solutions [82, 113]. Additionally, a factor architecture can cause an exponential increase in computational complexity [35, 80].

3.2.2 Update

The first step of FEA is to allow each factor to update its individuals. This is done in the same manor as the general multi-population EA shown in Algorithm 3.6, lines 4 – 6. FEA iterates over every factor and allows each factor to update its individuals until some stopping criterion is met. Throughout the dissertation, we will

Algorithm 3.7: FEA Compete Algorithm

Input: Function f to optimize, subpopulations $\boldsymbol{\mathcal{S}}$
Output: Full solution $\mathbf{G}$

```
1: randVarPerm ← RandomPermutation(N)
2: for ranVarIndex = 1 to n do
3:    i ← randVarPerm[ranVarIndex]
4:    B_Fit ← f(G)
5:    B_Val ← G[X_i]
6:    S_Opt ← {P_k | X_i ∈ U_k }
7:    randPopPerm ← RandomPermutation(|S_Opt|)
8:    for ranPopIndex = 1 to |S_Opt| do
9:       P_j ← S_Opt[randPopPerm[ranPopIndex]]
10:      G[X_i] ← Get Value From P_j for Variable X_i
11:      if f(G) < B_Fit then
12:         B_Val ← G[X_i]
13:         B_Fit ← f(G)
14:      end if
15:   end for
16:   G[X_i] ← B_Val
17: end for
18: return G
```

refer to a single factor updating its population as a *factor update* and the number of iterations a single factor performs during a factor update as *factor iterations*.

3.2.3 Compete

The goal of competition in FEA is to find the state assignments with the best fitness from the factors. FEA updates the full global solution $\mathbf{G}$ which allows factors to evaluate their individuals. For every $X_i \in \mathbf{X}$, the algorithm iterates over every factor containing X_i and finds the best value from those factors.

Algorith 3.7 gives the pseudocode for Compete. The algorithm first iterates over a random permutation of all the variables in $\mathbf{X}$ generated in line 1. Note that this permutation changes each time the algorithm is run. Lines 4 and 5 initialize

variables that are used for the competition. Next, the algorithm iterates over another random permutation of all the factors that are optimizing the variable X_i. Lines 10 – 15 then compare the individual values of variable X_i by substituting the factors' values into $\mathbf{G}$. In our implementation, the factor uses the best value found during the entire search process as its candidate value which is then evaluated in lines 10 – 15. The values yielding the best fitness from the overlapping factors are saved and then inserted into $\mathbf{G}$. Once the algorithm has iterated over all variables in $\mathbf{X}$, the algorithm exits and returns $\mathbf{G}$.

Note that competition is not guaranteed to find the best combination of values from each factor, nor does it guarantee the combination of values is better than the previous $\mathbf{G}$. However, by iterating over random permutations of $\mathbf{X}$ and $\boldsymbol{\mathcal{S}}$, the algorithm is able to explore different combinations and is still able to find good combinations of values.

3.2.4 Sharing

The sharing step serves two purposes. First, it allows overlapping factors to inject their current knowledge into other factors. Previous work by Fortier *et al.* discovered that this is one of the largest contributors to the FEA's performance [33]. Second, it sets each factor's $\mathbf{R}_i$ values to those in the full global solution $\mathbf{G}$ so that each factor $\boldsymbol{\mathcal{P}}_i$ can evaluate its partial solution on the function f. The sharing algorithm is provided in Algorithm 3.8.

The share algorithm iterates over all the factors and updates each factor's remaining values by setting each variable $X_j \in \mathbf{R}_i$ to the value in $\mathbf{G}$ (lines 2 – 4). Next, the algorithm injects information from $\mathbf{G}$ into factor $\boldsymbol{\mathcal{P}}_i$. To accomplish this, the algorithm finds the individual with the worst fitness in $\boldsymbol{\mathcal{P}}_i$ (line 5). Then,

Algorithm 3.8: FEA Share Algorithm

Input: Function f, Full global solution $\mathbf{G}$, Factors $\boldsymbol{\mathcal{S}}$
Output: Updated Factors $\boldsymbol{\mathcal{S}}$

```
1: for all P_i ∈ S do
2:    for all X_j ∈ R_i do
3:       R_i[X_j] ← G[X_j]
4:    end for
5:    P_w ← Get Worst Individual From P_i
6:    for all X_j ∈ U_i do
7:       P_w[X_j] ← G[X_j]
8:    end for
9:    P_w.fitness ← Fitness of G
10: end for
11: return S
```

the share algorithm sets the worst individual $\mathbf{P}_w$'s current position to the values in $\mathbf{G}$ (lines 6 – 8). Finally, the fitness for $\mathbf{P}_w$ is updated in line 9.

3.2.5 FEA

Now that the share and competition algorithms have been defined, we can give the full FEA (Algorithm 3.9). The algorithm works as follows. All of the subpopulations are first initialized according to the optimization algorithm being used and the subpopulation architecture (line 1). The full global solution $\mathbf{G}$ is initialized in line 2. Next, the algorithm begins updating the factors using three steps (lines 3 – 11). First, the algorithm iterates over each factor and optimizes the values using the corresponding optimization algorithm until some stopping criterion is met (line 6). Following the factor update steps, competition occurs between factors in the Compete function on line 9. Finally, the Share function on line 10 shares the updated best states between the factors. We refer to one iteration of Update, Compete, and

Algorithm 3.9: Factored Evolutionary Algorithm

Input: Function f to optimize, optimization algorithm A
Output: Full solution $\mathbf{G}$

1: $\boldsymbol{\mathcal{S}} \leftarrow$ Initialize factors from f, $\mathbf{X}$, and A
2: $\mathbf{G} \leftarrow$ Initialize full global from $\boldsymbol{\mathcal{S}}$
3: **repeat**
4: **for all** $\boldsymbol{\mathcal{P}}_i \in \boldsymbol{\mathcal{S}}$ **do**
5: **repeat**
6: Optimize f with population $\boldsymbol{\mathcal{P}}_i$
7: **until** Termination criterion is met
8: **end for**
9: $\mathbf{G} \leftarrow$ Compete$(f, \boldsymbol{\mathcal{S}})$
10: $\boldsymbol{\mathcal{S}} \leftarrow$ Share$(f, \mathbf{G}, \boldsymbol{\mathcal{S}})$
11: **until** Termination criterion is met
12: **return** $\mathbf{G}$

Share as an *FEA iteration*. These steps are repeated until the stopping criterion is met.

3.3 Complexity

Given the definition of FEA, we now analyze the the computational complexity of FEA and show which steps have the most computational burden by breaking down each step in the algorithm. Additionally, we examine the number of fitness evaluations used by each function. We refer to the pseudocode shown in Algorithm 3.9.

3.3.1 Fitness Evaluation

We first begin by determining the complexity to evaluate the fitness of an individual and approximate it as $\mathcal{O}(\Lambda(N))$, where N is $|\mathbf{X}|$, and $\mathbf{X}$ is the vector of variables. $\Lambda(N)$ is a function that returns the complexity in calculating fitness of an individual of N variables and is problem specific. In some cases, such as in

the sphere function, the complexity of a fitness evaluation requires one iteration over all N variables ($\Lambda(N) = N$). Other problems may be much more computationally complex, such as evaluating neural networks.

3.3.2 Optimization Algorithm

Next, we determine the complexity of the underlying optimization algorithm. We assume we are using a simple population-based algorithm such as a GA or PSO. We assume also that the population algorithm has $P = |\boldsymbol{\mathcal{P}}|$ individuals, where $\boldsymbol{\mathcal{P}}$ is the set of all the individuals in the factor. During each iteration, an individual with N variables has its position and fitness updated. We denote the complexity of updating a single individual of N variables as $\mathcal{O}(U(N))$. In most EAs, $U(N) = N+\Lambda(N)$. For example, in PSO, the algorithm must iterate over all N variables in the individual and velocity vectors. Then, the algorithm must evaluate the fitness of the updated individual, giving a complexity of $\mathcal{O}(U(N)) = \mathcal{O}(N + \Lambda(N))$. This is done P times, once for each individual; therefore, the complexity during each iteration is $\mathcal{O}(P \times U(N))$.

3.3.3 FEA

To determine the complexity of FEA, we first show the complexity of the three mains parts: Update, Compete, and Share.

- *Update:* The Update step in FEA involves iterating over the set of factors $\boldsymbol{\mathcal{S}}$ and maximizing each factor's local fitness for K iterations. Using the complexity for the optimization algorithms, each factor update has a complexity of $\mathcal{O}(P \times U(N))$, assuming each factor has P individuals. Since this is done for $D = |\mathcal{S}|$ factors and K times for each factor, the total complexity of this step is $\mathcal{O}(DKP \times U(N))$.

- *Compete:* In FEA, the Compete step is used to find the optimal set of values for the variables in $\mathbf{X}$. This is done by iterating over all the variables and then iterating over each factor. For every iteration there is a fitness evaluation. Assume for each variable X_i, there are M factors optimizing X_i. Note that there are N variables and M factors and that the time complexity for a fitness evaluation is $\mathcal{O}(\Lambda(N))$. Therefore, the total complexity of the Compete step in FEA is $\mathcal{O}(MN \times \Lambda(N))$.

- *Share:* Share involves distributing the values found during competition to other subpopulations. The algorithm iterates over all factors and seeds values by replacing the lowest ranked individual in the algorithm. The seeding step requires iterating over the N variables to set values for the worst-ranked individual in the factor to the full global solution. This is done for all D factors, which gives the Share step a complexity of $\mathcal{O}(DN)$.

FEA iterates over Update, Compete, and Share steps L times. Summing the complexity for Update, Compete, and Share and multiplying it by L gives a complexity of

$$\mathcal{O}(\text{FEA}) = \mathcal{O}(DKLP \times U(N) + LMN \times \Lambda(N) + DLN)$$

In many of the applications to which FEA has been applied, the architecture for the subpopulations creates a subpopulation for every dimension in $\mathbf{X}$, resulting in $D = N$ [33]. Substituting this into the above equation, we now have

$$\mathcal{O}(\text{FEA}) = \mathcal{O}(KLNP \times U(N) + LMN \times \Lambda(N) + LN^2)$$

If the complexity of $U(N) = N + \Lambda(N)$, then the total complexity of FEA is

$$\begin{aligned}\mathcal{O}(\text{FEA}) &= \mathcal{O}(KLNP \times (N + \Lambda(N)) + LMN \times \Lambda(N) + LN^2) \\ &= \mathcal{O}(KLN^2P + KLNP \times \Lambda(N) + LMN \times \Lambda(N) + LN^2)\end{aligned}$$

The first term in the above equation is the complexity required to update the individuals. The total number of individuals in FEA is calculated as $Total_P = NP$ while the total number of update steps used by FEA is $Total_{Up} = KL$. If there is an EA and FEA with the same number of individuals and update steps, then $Total_P \times Total_{Up} \times N$ is equal for both the EA and FEA. This demonstrates that FEA does not add any additional complexity when updating individuals.

While FEA does not add any additional complexity to the Update step, it does add additional fitness evaluations. One of the most commonly used measures of time complexity for EAs is the number of fitness evaluations required. This is because the number of fitness evaluations is a measure that can be applied to any local search algorithm [28]. Additionally, fitness evaluations are used as a stopping criteria for optimization algorithm competitions [63, 108]. We can formally analyze the number of fitness evaluations required by Update and Compete by analyzing the percent of fitness evaluations used by Compete, which is given as

$$\frac{LMN}{KLNP + LMN} = \frac{M}{KP + 1}$$

Figures 3.1 and 3.2 present plots of the percentage of fitness evaluations used by Compete with varying numbers of K and M with 20 individuals. In Figure 3.1, the number of factor iterations K is on the x-axis, whereas Figure 3.2 presents the number of factors optimizing a single variable. In the first graph, we observe that

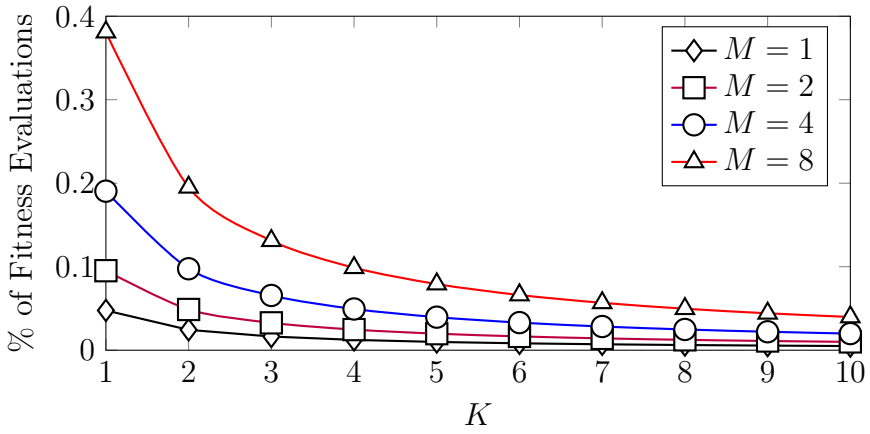


Figure 3.1: The percent of fitness evaluations used by Compete in FEA varying K.

as the number of update iterations K increases, the percentage of fitness evaluations used by Compete decreases. However, in the second graph, we see that the percent of evaluations used by Compete increases linearly as M increases.

From this, we conclude that while FEA does use more fitness evaluations than a regular EA, this is often less that 10% of the total number of fitness evaluations used by the algorithm. Furthermore, if there is a high number of factors optimizing a variable, the complexity can be managed by increasing the number of factor iterations K.

3.4 Comparing FEA with Single-Population and CCEA

To demonstrate general performance, we applied FEA to abductive inference in Bayesian networks, maximizing NK landscapes, and a set of benchmark test functions using Hill Climbing (HC), Genetic Algorithm (GA), Differential Evolution (DE), and Particle Swarm Optimization (PSO) as the underlying algorithms. On Bayesian networks and NK landscapes, we used Discrete DE (DDE) and Discrete Multi-Valued Particle Swarm Optimization (DMVPSO). We now describe the background

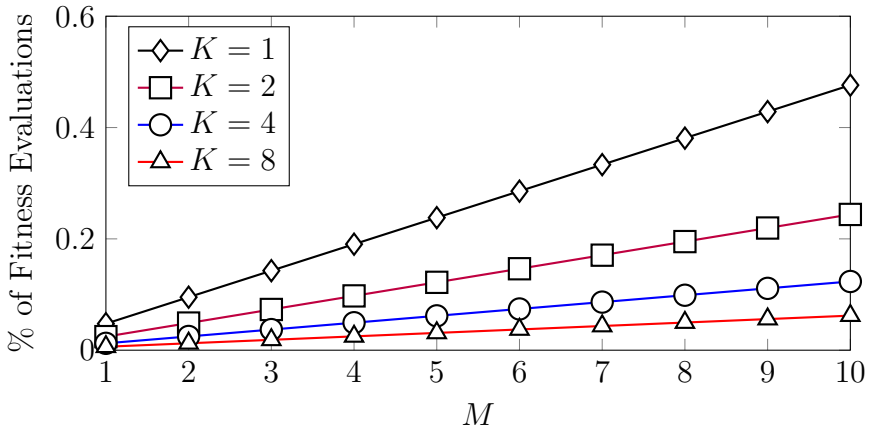


Figure 3.2: The percent of all fitness evaluations used by Compete in FEA varying M.

information on DMVPSO and the test problems necessary to understand the experiments in this chapter.

3.4.1 Discrete Multi-Valued Particle Swarm Optimization PSO

PSO has been successfully applied to a wide variety of optimization problems, but the algorithm is unable to handle discrete problems, which are functions in which the domain for the variables can only assume a finite number of states [53]. For example, a discrete variable may have a domain of different kinds furniture like chair, couch, table, and dresser. NK landscapes and Bayesian networks are both instances of discrete functions. For these problems, we used Discrete Multi-Valued Particle Swarm Optimization (DMVPSO), proposed by Veeramachaneni *et al.* [115], as the underlying algorithm as it has been shown by Fortier *et al.* to perform well in OSI [33].

In DMVPSO, the velocity update equations remain mostly unchanged from regular PSO [48]. However, the semantics of the velocity vector are changed such that it denotes the probability of a particle's position taking on a specific value. After the velocity is updated, it is transformed into the interval $[0, M-1]$, where M

is the number of values the variable may take on, using the sigmoid function

$$S_{i,j} = \frac{M-1}{1+\exp(-V_{i,j})}.$$

Next, each particle's position is updated by generating a random number according to the Gaussian distribution, $X_{i,j} \sim N(S_{i,j}, \sigma \times (M-1))$ and rounding the result. To ensure the particle's position remains in the range $[0, M-1]$, the following piecewise function is applied to the $X_{i,j}$

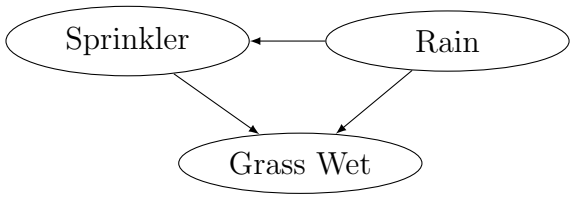
$$X_{i,j} = \begin{cases} M-1 & X_{i,j} > M-1 \\ 0 & X_{i,j} < 0 \\ X_{i,j} & \text{otherwise} \end{cases}$$

In all our experiments, the PSO and DMVPSO ω parameters were set to 0.729, and ϕ_1 and ϕ_2 were both set to 1.49618. These values guarantee convergence of PSO and have been found to perform well on a wide range of problems [16, 26, 98]. Each subpopulation for FEA had a population size of 10. We will now describe the three different sets of experiments and the results for each set.

3.4.2 Bayesian Networks

Bayesian networks are a probabilistic graphical model [54, 76]. A full joint probability distribution $P(X_1, \ldots, X_n)$ requires an exponential number of parameters to define every probability for every combination of events. However, a Bayesian network is able to reduce the number of parameters by taking advantage of independence properties between variables in the distribution. We define a Bayesian network as follows.

Parents	Sprinkler	
Rain	T	F
F	0.40	0.60
T	0.01	0.99



Rain	
T	F
0.2	0.8

Parents		Grass Wet	
Sprinkler	Rain	T	F
F	F	0.40	0.60
F	T	0.80	0.20
T	F	0.90	0.10
T	T	0.99	0.01

Figure 3.3: Example of a Bayesian network.

Definition 3.1. Let $G = (\mathbf{X}, \mathbf{E})$ be a directed acyclic graph. $\mathbf{X}$ is the set of random variables in a joint probability distribution $P(X_1, \ldots, X_N)$ and $\mathbf{E}$ represents relationships between the random variables. Specifically, an edge $e_{i,j} \in \mathbf{E}$ means that X_i is conditionally dependent on X_j. A joint distribution for a *Bayesian network* is then defined as

$$P(X_1, \ldots, X_N) = \prod_{i=1}^{N} P(X_i | \mathbf{Pa}(X_i))$$

where $\mathbf{Pa}(X_i)$ corresponds to the parents of X_i.

Given a Bayesian network, each random variable X_i in the probability distribution P must define a conditional probability $P(X_i|\mathbf{Pa}(X_i)$. This defines the probability of the random variable given all combinations of the variable's parent states. We will refer to this as the *conditional probability table* of a variable X_i.

Example 1. An example Bayesian network is shown in Figure 3.3. In this example, there are three random variables: Rain, Sprinkler, and Grass Wet. Each variable has two possible states: True (T) or False (F). Note that Rain does not have any parents; therefore, only the probability of its states are required. The conditional probability

table for Sprinkler, however, requires defining the probability of Sprinkler being True or False given the different states of Rain. Finally, Grass Wet is dependent on both the states of Rain and Sprinkler.

One common task in a Bayesian network is to query for the probability of an event occurring given some set of evidence. Note that this evidence set can be empty. Answering such a query is known as *inference* [17,23,54]. With Figure 3.3, one might know that it rained during the night. Given this information, we might ask, "what is the probability of the grass being wet?" This particular query can be answered quickly using marginalization, which in the above network, gives an answer of 0.802. However, in general, exact inference in Bayesian networks is NP-complete [17,54,76]. Additionally, approximate inference in Bayesian networks is NP-hard [19].

An important feature of Bayesian networks is the concept of a Markov blanket, which we define as follows.

Definition 3.2. In a Bayesian network, the *Markov blanket* of a node consists of the node's parents, children, and children's parents, denoted as

$$MB(X_i) = \{\mathbf{Ch}(X_i) \cup \mathbf{Pa}(X_i) \cup \mathbf{Pa}(\mathrm{Ch}(X_i)) \setminus X_i\}$$

where **Ch** corresponds to the child nodes of X_i and **Pa** corresponds to the parent nodes of node X_i. A node is conditionally independent of all other nodes in the network given its Markov Blanket.

Example 2. Suppose we have the Bayesian network in Figure 3.4. In this example, there are 12 random variables. Given the variable X_4, its Markov blanket consists of its parents X_1; its children X_8, X_9 and X_{10}; and its children's parents X_2, X_3 and X_5. Suppose we are given evidence for all the states in X_4's Markov blanket and

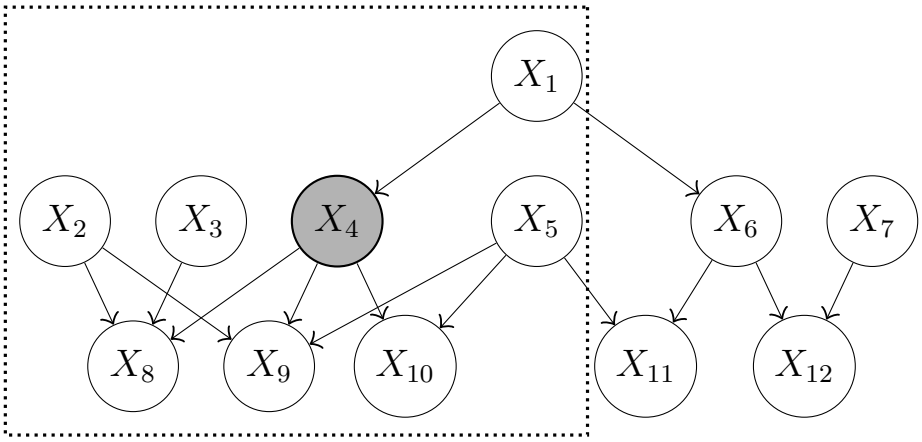


Figure 3.4: Example of a Markov Blanket in a Bayesian network.

after performing inference, we find that the probability of X_4 being in state 0 is 0.25. Now, if we are given additional information about another variable, such as variable X_6 being in state 1, the probability for variable X_4 does not change because it is separated from X_6 by its Markov blanket.

Abductive inference is a specific type of inference in Bayesian networks that tries to find the most likely explanation for some set of evidence. This is also known as the finding the maximum *a posteriori* (MAP) probability state of the remaining variables of a network.

Definition 3.3. Let $\mathbf{X}_U = \mathbf{X} \backslash \mathbf{X}_O$, where $\mathbf{X}$ denotes the variable nodes in the network. The problem of *abductive inference* is to find the most probable state assignment to the unobserved variables in $\mathbf{X}_U$ given the evidence $\mathbf{X}_O = \mathbf{x}_o$,

$$\begin{aligned} MAP(\mathbf{X}_U, \mathbf{x}_o) &= \underset{\mathbf{x} \in \mathbf{X}_U}{\operatorname{argmax}} \; P(\mathbf{x}|\mathbf{x}_O) \\ &= \underset{\mathbf{x} \in \mathbf{X}_U}{\operatorname{argmax}} \; \prod_{i=1}^{N} P(x_i|\mathbf{Pa}(X_i)) \end{aligned}$$

where $\mathbf{x}_o$ denotes the set of observed states for variables $\mathbf{X}_O$ and $\mathbf{x}$ is a set of state assignments for variables in $\mathbf{X}_U$. $\mathbf{Pa}(X_i)$ represents the complete state assignment to the parents of X_i.

Note that multiplying small probabilities several times can cause the values to go to zero. To deal with this issue, the log is often used instead of the raw probabilities. Since taking the log does not change the optimization problem, the MAP equation becomes

$$\begin{aligned} MAP(\mathbf{X}_U, \mathbf{x}_O) &= \underset{\mathbf{x} \in \mathbf{X}_U}{\operatorname{argmax}} \log \prod_{i=1}^{N} P(X_i | \mathbf{Pa}(X_i)) \\ &= \underset{\mathbf{x} \in \mathbf{X}_U}{\operatorname{argmax}} \sum_{i=1}^{N} \log P(X_i | \mathbf{Pa}(X_i)) \end{aligned}$$

3.4.3 NK Landscapes

The NK landscape is a mathematical framework that generates tunable fitness landscapes often used as test functions for evaluating evolutionary algorithms [52]. They were first proposed by Kaufman [52], and Weinberger showed that optimizing an NK landscape is NP-complete [116].

While NK landscapes usually assume binary variables, they have been extended to allow for mixed variables, such as continuous, integer, and nominal [61]. Li *et al.* first extended the binary NK landscape to continuous by mapping it to an N-dimensional hypercube $[0, 1]^N$ and mapping values on the interior of the hypercube using a multi-linear interpolation technique. For integers, values are first mapped to $[0, 1]$ and then to a continuous NK landscape. Nominal values required the landscapes to define fitness tables that allow L values for the discrete variables.

NK landscapes contains two parameters, N and K, that control the overall size of the landscape and the structure or amount of interaction between each dimension, respectively [116]. Increasing K makes the variables more dependent on one another, resulting in more rugged landscapes [2]. This interaction is often referred to as epistasis. Formally, we can define it as follows.

Definition 3.4. An *NK landscape* is a function $f : \mathcal{B}^N \rightarrow \mathbb{R}^+$ where $\mathcal{B}^N$ is a bit string of length N. K specifies the number of other bits in the string that a bit is dependent on. Given a landscape, the fitness value is calculated as

$$f(\boldsymbol{X}) = \frac{1}{N} \sum_{i=1}^{N} f_i(X_i, \mathbf{Nb}_K(X_i)) \tag{3.1}$$

where $\mathbf{Nb}_K(X_i)$ returns the K bits that are located within X_i's neighborhood. The individual functions are defined as $f_i : \mathcal{B}^K \rightarrow \mathbb{R}^+$.

Note that there are multiple ways to define the neighborhood function $\mathbf{Nb}_K(X_i)$. For example one commonly used method is that the function returns the next K contiguous bits of the string starting at X_i. If the end of the string is reached, then the neighborhood wraps back around to the beginning of the string. Another commonly used method when generating the landscape, is to pick K bits randomly to be in variable X_i's neighborhood. Additionally, the individual functions are defined as f_i are usually created randomly.

3.4.4 Experimental Setup

For the Bayesian networks, we used the Hailfinder, Hepar2, Insurance, and Win95pts Bayesian networks from the Bayesian Network Repository [93]. Previous work by Fortier *et al.* used these four networks to evaluate OSI as the represent an average size Bayesian network with varying number of parameters [33]. Table 3.1

Table 3.1: Properties of the Test Bayesian Networks.

Network	Nodes	Arcs	Parameters	Avg. MB Size
Hailfinder (Ha)	56	66	2656	3.54
Hepar2 (He)	70	123	1453	4.51
Insurance (I)	27	52	984	5.19
Win95pts (W)	76	112	574	5.92

lists the number of nodes, edges, parameters, and average Markov blanket size for the selected networks. NK landscapes were generated randomly using combinations of $N = 2, 5, 10$ and $K = 2, 5, 10$. For the benchmark optimization functions, we used the Ackley's, Dixon-Price, Exponential, Griewank, Rosenbrock, Schwefel 1.2, and Sphere functions as defined in Appendix A.

For each Bayesian network, 50 trials were performed for each algorithm. Because NK landscapes are randomly generated, we generated 50 landscapes, and each algorithm was then run 50 times on each landscape. Similarly, 50 trials were performed on each benchmark function.

We used the DDE algorithm proposed by Lampinen and Zelinka, which rounds the individual's position during fitness evaluations to handle integer values [59]. An indexing scheme is then used to map the integer value to the discrete value [59]. During tuning, we found that a value of 0.25 for both DE and DDE's mutation rate and a differential factor of 0.55 performed the best. For the GA, tournament section and one-point crossover were used along with uniform mutation at a rate of 0.02.

The FEA versions of the algorithms used the Markov architecture for the Bayesian networks by Fortier *et al.* [33]. On the NK landscapes, we created a factor for each variable and included every variable in its neighborhood and any variable that contained the current variable in its neighborhood. For the benchmark functions, we

created a factor for each neighboring pairs of variables in the ordered list of variables. Each factor performed ten iterations before Compete and Share were performed.

For the CC versions of DMVPSO, GA, DDE, and HC, $N/2$ factors optimizing over two variables each were created, since these parameters gave the best results during preliminary testing across all algorithms. These algorithms are similar to the CPSO-S_K model presented by Van den Bergh and Engelbrecht [113]. When evaluating the fitness of a factor for the CC algorithm, we use the best known value from the other factors to fill in the remaining values. While this is different than other implementations, such as those presented in [82], it uses the same source for values that FEA uses when it constructs the global solution $\mathbf{G}$.

FEA, CC, and the single-population algorithms were given a total of 350 individuals for GA, PSO, and DE. Often, HC uses only one individual, but, because CC and FEA requires at least one individual per factor, we used multiple individuals in the single-population. For the HC versions, 75 individuals were used, since that allows FEA to have at least one individual per factor on the largest optimization functions. Furthermore, these values were found to perform well during tuning for all algorithms. On the CC and FEA algorithms, individuals were distributed evenly across each of the factors. All algorithms were stopped once the best fitness did not improve after 15 iterations.

3.4.5 Results

Table 3.2 shows the results of comparing single-population and CC algorithms to the FEA versions on performing abductive inference on the four different Bayesian networks. Bold values indicate a statistically significant difference between the single-population, CC, and FEA versions of the corresponding underlying algorithms, using Paired Student t-Tests with a 0.05 significance level. If two algorithms tied, both

Table 3.2: Comparison of single-population, cooperative coevolutionary, and FEA algorithms on abductive inference on Bayesian Networks.

	Hailfinder	Hepar2	Insurance	Win95pts
DMVPSO	−5.05E+2(1.21E+2)	−5.29E+1(9.89E−1)	−2.54E+1(8.48E−1)	−1.80E+2(6.68E+1)
CC-DMVPSO	−1.16E+3(5.13E+2)	−2.00E+1(1.48E+0)	−1.76E+2(1.00E+2)	−4.99E+2(1.52E+2)
FEA-DMVPSO	**−3.45E+1(1.87E−1)**	**−1.75E+1(5.82E−1)**	**−1.07E+1(1.14E−1)**	**−1.43E+1(7.21E−1)**
DDE	−5.77E+1(1.76E+0)	**−1.98E+1(1.11E+0)**	−1.74E+1(1.04E+0)	−7.04E+1(3.11E+0)
CC-DDE	−4.13E+2(1.24E+2)	−2.35E+1(6.11E−1)	−1.74E+2(1.52E+2)	−1.86E+2(9.18E+1)
FEA-DDE	**−3.60E+1(5.68E−1)**	**−2.04E+1(9.79E−1)**	**−9.21E+0(9.11E−1)**	**−2.24E+1(1.70E+0)**
GA	−3.83E+1(1.58E+0)	−1.75E+1(4.35E−1)	−1.26E+1(8.73E−1)	**−3.11E+1(7.77E−1)**
CC-GA	−3.78E+1(5.67E−1)	−2.10E+1(1.07E+0)	−1.49E+1(2.28E+0)	−3.44E+2(2.10E+2)
FEA-GA	−3.67E+1(3.62E−1)	−1.98E+1(9.68E−1)	−1.19E+1(5.31E−1)	**−2.93E+1(1.57E+0)**
HC	−3.86E+1(6.91E−1)	**−1.67E+1(2.44E−1)**	**−1.14E+1(4.64E−1)**	−2.56E+1(1.09E+0)
CC-HC	−8.66E+2(2.37E+2)	−2.08E+1(1.30E+0)	−9.45E+1(7.76E+1)	−4.92E+2(2.33E+2)
FEA-HC	**−3.62E+1(5.66E−1)**	−1.81E+1(5.16E−1)	**−1.11E+1(8.00E−1)**	**−1.74E+1(9.59E−1)**

Table 3.3: Comparison of single-population, cooperative coevolutionary, and FEA algorithms on maximizing NK landscapes.

	25			40		
	2	5	10	2	5	10
DMVPSO	1.76E+1(4.35E−2)	1.79E+1(3.59E−2)	1.80E+1(3.23E−2)	2.68E+1(8.03E−2)	2.68E+1(5.16E−2)	2.68E+1(4.01E−2)
CC-DMVPSO	1.84E+1(6.53E−2)	1.78E+1(8.72E−2)	1.72E+1(7.98E−2)	2.95E+1(1.16E−1)	2.77E+1(1.86E−1)	2.63E+1(1.34E−1)
FEA-DMVPSO	**1.87E+1(3.93E−2)**	**1.93E+1(4.28E−2)**	**1.92E+1(4.77E−2)**	**3.02E+1(7.82E−2)**	**3.06E+1(6.15E−2)**	**3.07E+1(4.41E−2)**
DDE	**1.88E+1(3.94E−2)**	1.88E+1(3.60E−2)	1.84E+1(3.84E−2)	2.98E+1(8.04E−2)	2.82E+1(5.32E−2)	2.76E+1(4.59E−2)
CC-DDE	1.83E+1(5.49E−2)	1.77E+1(9.00E−2)	1.70E+1(7.98E−2)	2.95E+1(8.36E−2)	2.79E+1(1.66E−1)	2.65E+1(1.49E−1)
FEA-DDE	1.86E+1(4.16E−2)	**1.92E+1(4.18E−2)**	**1.91E+1(4.20E−2)**	**3.00E+1(7.88E−2)**	**3.02E+1(6.23E−2)**	**3.02E+1(4.75E−2)**
GA	**1.88E+1(3.98E−2)**	**1.91E+1(3.54E−2)**	1.84E+1(5.74E−2)	**3.03E+1(7.06E−2)**	**3.00E+1(7.66E−2)**	2.73E+1(5.30E−2)
CC-GA	1.82E+1(5.25E−2)	1.78E+1(9.79E−2)	1.70E+1(7.89E−2)	2.94E+1(8.13E−2)	2.78E+1(1.62E−1)	2.67E+1(1.37E−1)
FEA-GA	1.85E+1(4.97E−2)	1.89E+1(4.28E−2)	**1.87E+1(4.09E−2)**	2.98E+1(7.82E−2)	**2.99E+1(6.77E−2)**	**2.96E+1(5.50E−2)**
HC	**1.88E+1(3.98E−2)**	**1.94E+1(3.25E−2)**	**1.90E+1(3.35E−2)**	**3.03E+1(7.05E−2)**	**3.01E+1(4.56E−2)**	2.94E+1(4.41E−2)
CC-HC	1.83E+1(6.85E−2)	1.74E+1(1.12E−1)	1.70E+1(8.17E−2)	2.92E+1(1.10E−1)	2.73E+1(1.81E−1)	2.63E+1(1.65E−1)
FEA-HC	1.86E+1(4.27E−2)	1.91E+1(4.50E−2)	**1.89E+1(4.43E−2)**	**3.00E+1(7.68E−2)**	**3.02E+1(5.84E−2)**	**3.00E+1(5.63E−2)**

values are bolded. In all cases, the FEA versions of the algorithms performed better than the CC versions. However, FEA did not always perform better than the single-population algorithms. For example, the single-population GA tied with the FEA version on all four networks. The single-population DDE tied with FEA-DDE on the Hepar2. FEA-HC was significantly outperformed by HC on Hepar2, but tied on the Hailfinder and Insurance networks. The only instance in which FEA was significantly outperformed by the single-population algorithm was HC on the Hepar2 network. Finally, FEA-DMVPSO significantly outperformed DMVPSO on all four networks.

The results comparing single-population, CC, and FEA algorithms on the NK landscapes are shown in Tables 3.3. Similar to the Bayesian network results, the

Table 3.4: Comparison of single-population, cooperative coevolutionary, and FEA algorithms on optimizing benchmark functions.

	Ackley's	Dixon-Price	Exponential	Griewank
PSO	3.42E-2(3.42E-2)	6.76E-1(5.13E-3)	-9.97E-1(2.75E-4)	**1.01E-2(1.55E-3)**
CC-PSO	5.27E-14(2.19E-15)	**1.07E-8(4.92E-10)**	-1.00E+0(6.99E-17)	**1.30E-2(7.08E-3)**
FEA-PSO	1.37E-14(6.58E-16)	5.44E+0(3.75E+0)	**-1.00E+0(6.18E-17)**	9.10E-2(1.60E-2)
DE	1.09E+1(9.90E-1)	2.42E+5(5.94E+4)	-8.50E-1(3.95E-2)	1.17E+0(1.71E-1)
CC-DE	**1.53E-2(1.53E-2)**	1.84E+2(8.49E+1)	**-1.00E+0(6.18E-17)**	**9.51E-2(4.45E-2)**
FEA-DE	1.49E-1(5.13E-2)	**3.86E+1(9.20E+0)**	**-1.00E+0(1.96E-8)**	**5.52E-2(1.22E-2)**
GA	1.39E-1(1.11E-2)	1.47E+1(1.27E+0)	-9.66E-1(1.52E-3)	**2.06E-2(2.02E-3)**
CC-GA	**2.21E-2(1.56E-3)**	**3.61E+0(5.52E-1)**	**-9.94E-1(5.06E-4)**	5.44E-2(7.90E-3)
FEA-GA	**2.08E-2(1.21E-3)**	**4.43E+0(5.83E-1)**	**-9.94E-1(5.75E-4)**	9.88E-2(1.24E-2)
HC	1.92E+1(4.10E-2)	1.89E+2(2.21E+1)	-6.11E-6(1.36E-6)	1.08E+0(3.40E-2)
CC-HC	1.69E+0(7.93E-2)	1.22E+1(1.20E+0)	-9.99E-1(1.62E-4)	4.28E-1(8.34E-2)
FEA-HC	**5.95E-3(4.60E-4)**	**1.15E+0(4.33E-1)**	**-1.00E+0(7.87E-7)**	**3.34E-2(7.94E-3)**

	Rosenbrock	Schwefel	Sphere
PSO	6.09E+1(4.75E+0)	**1.99E+3(3.32E+2)**	3.60E-3(3.11E-4)
CC-PSO	2.43E+1(1.40E+1)	8.43E+5(2.03E+5)	**2.37E-17(6.24E-18)**
FEA-PSO	**2.99E+0(5.75E-1)**	**1.81E+3(6.54E+2)**	**5.79E-17(5.23E-17)**
DE	4.18E+5(9.13E+4)	1.15E+5(3.24E+3)	1.49E+3(4.49E+2)
CC-DE	3.20E+2(8.59E+1)	9.45E+5(1.89E+5)	**2.11E-30(4.94E-31)**
FEA-DE	**1.31E+2(1.78E+1)**	**9.18E+2(1.62E+2)**	**7.80E-3(7.80E-3)**
GA	2.06E+2(1.25E+1)	9.26E+4(5.68E+3)	7.15E-2(8.09E-3)
CC-GA	**4.74E+1(1.58E+1)**	1.90E+6(4.65E+5)	**9.81E-3(6.63E-4)**
FEA-GA	7.38E+1(1.10E+1)	**1.07E+2(1.23E+1)**	**9.39E-3(9.23E-4)**
HC	8.23E+2(6.67E+1)	3.66E+4(1.86E+3)	2.04E+2(1.33E+1)
CC-HC	6.53E+1(8.78E+0)	1.54E+6(5.63E+5)	2.11E+0(2.32E-1)
FEA-HC	**2.07E+0(3.67E-1)**	**9.71E+2(3.41E+2)**	**5.01E-3(3.88E-4)**

FEA versions of the algorithms significantly outperformed the CC versions. FEA-DMVPSO always outperformed the single-population versions. However, FEA-DDE was outperformed on the simplest landscapes ($N = 25, K = 2$). GA significantly outperformed FEA-GA on $N = 25, K = 2$, $N = 25, K = 5$, and $N = 40, K = 2$, but tied on $N = 40, K = 5$. FEA-GA outperformed the single-population algorithm when $K = 10$. Looking at the HC algorithms, FEA outperformed the single-population algorithm only on $N = 40, K = 10$. The single-population HC significantly outperformed the FEA version on $N = 25, K = 2$ and $N = 25, K = 5$. On the

Table 3.5: Results comparing the number of fitness evaluations on benchmark and Bayesian networks.

	Benchmark		Bayesian Networks	
	Rosenbrock	Schwefel	Win95pts	Hailfinder
PSO	3.68E+5(7.55E+3)	1.69E+5(2.18E+4)	1.10E+4(5.96E+2)	1.14E+4(6.25E+2)
CC-PSO	4.33E+5(9.54E+4)	6.09E+4(1.26E+2)	8.09E+4(2.68E+3)	8.06E+4(3.62E+3)
FEA-PSO	5.96E+5(9.23E+4)	4.85E+5(1.14E+4)	1.06E+5(3.84E+3)	1.22E+5(5.80E+3)
DE	4.69E+4(4.52E+3)	1.41E+4(9.44E+2)	2.04E+4(1.34E+3)	2.05E+4(1.51E+3)
CC-DE	2.41E+5(7.18E+3)	6.08E+4(0.00E+0)	1.03E+5(4.42E+3)	9.53E+4(4.73E+3)
FEA-DE	2.36E+5(4.83E+3)	1.81E+6(1.01E+5)	8.21E+4(2.12E+3)	8.50E+4(2.19E+3)
GA	1.45E+5(8.03E+3)	1.77E+4(1.78E+3)	3.66E+4(1.82E+3)	4.46E+4(1.44E+3)
CC-GA	1.63E+6(1.01E+5)	6.08E+4(0.00E+0)	9.49E+4(3.33E+3)	1.09E+5(4.12E+3)
FEA-GA	1.65E+6(1.10E+5)	4.05E+6(2.98E+2)	1.04E+5(7.24E+3)	1.01E+5(4.36E+3)
HC	1.62E+6(2.19E+4)	5.84E+5(2.78E+4)	1.30E+5(2.75E+3)	2.25E+5(4.40E+3)
CC-HC	8.48E+4(2.99E+3)	4.32E+4(4.08E+2)	3.54E+4(9.12E+2)	5.52E+4(1.94E+3)
FEA-HC	3.87E+5(2.02E+4)	4.07E+6(5.70E+5)	1.30E+5(1.71E+3)	8.87E+4(1.53E+3)

other landscapes ($N = 25, K = 10,\ N = 40, K = 2,\ N = 40, K = 5$), the FEA and single-population HC algorithms tied.

Table 3.4 shows the results of comparing single-population, CC algorithms, and FEA versions on optimizing the benchmark functions. Results are presented as the mean value over 30 trials along with the standard error shown in parentheses. Overall, the FEA versions of the algorithms performed the best. FEA-PSO was outperformed by CC-FEA on the Dixon-Price and Griewank functions. On the Ackley's function, CC-DE performed significantly better than FEA-DE. The GA outperformed FEA-GA on the Griewank function while CC-GA performed significantly better on the Rosenbrock function. FEA-HC performed significantly better than single-population and CC versions on all functions. Finally, there are several instances where the CC and FEA algorithms tied.

We also looked at the number of fitness evaluations each algorithm required. Table 3.5 presents the average number of fitness evaluations for two representative benchmark functions and Bayesian networks while Table 3.6 presents the results for

Table 3.6: Results comparing the number of fitness evaluations on NK landscapes.

	NK Landscapes	
	N = 25, K = 2	N = 40, K = 10
PSO	1.10E+4(2.63E+2)	1.11E+4(3.25E+2)
CC-PSO	7.91E+4(6.01E+2)	8.00E+4(1.26E+3)
FEA-PSO	7.20E+4(3.87E+2)	1.08E+5(2.38E+3)
DE	1.93E+4(4.30E+2)	1.32E+4(3.72E+2)
CC-DE	8.00E+4(7.40E+2)	8.65E+4(1.60E+3)
FEA-DE	7.53E+4(1.16E+3)	8.25E+4(7.40E+2)
GA	1.27E+4(1.84E+2)	1.17E+4(4.44E+2)
CC-GA	7.88E+4(7.91E+2)	8.38E+4(1.35E+3)
FEA-GA	7.41E+4(1.29E+3)	8.80E+4(1.75E+3)
HC	1.64E+4(2.96E+2)	2.47E+4(4.09E+2)
CC-HC	2.95E+4(2.21E+2)	3.41E+4(5.25E+2)
FEA-HC	4.26E+4(2.20E+2)	9.15E+4(4.89E+2)

two NK landscapes. For the sake of conciseness, we only present results for an easy and hard instance of each class of problem. Note that each algorithm stopped when the best solution failed to improve after 15 iterations. For the FEA, this correlates to the full global solution not improving after 15 iterations of Update, Compete, and Share. In almost all cases, the FEA algorithms required more fitness evaluations than the single-population and CC versions. There were a few exceptions where FEA versions required fewer iterations than the single population, HC on the Rosenbrock, and Hailfinder and PSO on the NK landscape $N = 40, K = 10$.

To further investigate the number of fitness evaluations required for FEA to find good solutions, we present fitness curves from DMVPSO maximizing NK landscapes $N = 25, K = 2$ and PSO minimizing the Rosenbrock function. Figure 3.5 presents the average best fitness over time on the NK landscape problem while Figure 3.6 shows the same for the Rosenbrock function. The curve labeled "Single" refers to the single-population algorithm while "CC" and "FEA" refer to the CC and FEA versions, respectively. Note that the y-axis in Figure 3.6 is on a log scale to allow for a compact representation of the results. Results are average from 500 trials. Note

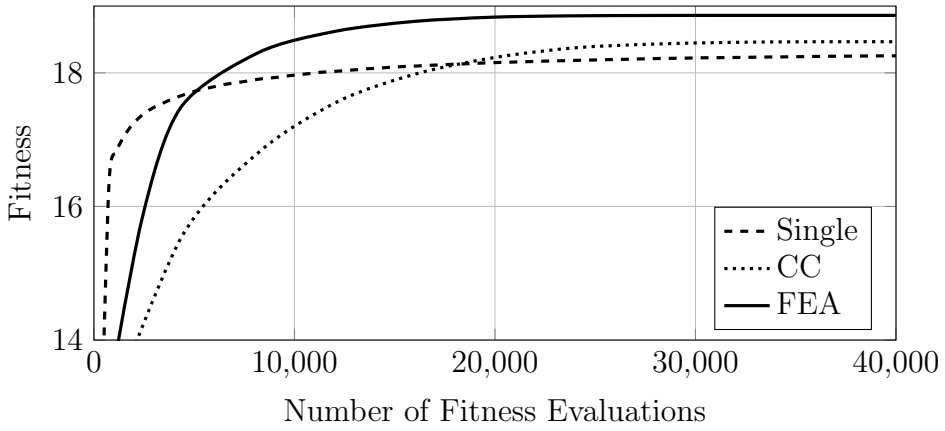


Figure 3.5: A fitness curve plotting fitness versus number of fitness evaluations on single-population, CC, and FEA DMVPSO algorithms maximizing NK landscapes with $N = 25$ and $K = 2$.

that the best values were taken at each iteration in the single-population and an FEA iteration for CC and FEA. A single iteration for CC and FEA involves iterating over all subpopulations and allowing each subpopulation to run for ten iterations while an iteration for a single-population is given after updating each of the individuals one time. The consequence is that the fitness curves for CC and FEA have fewer data points. For both figures, the single-population was the best algorithm at improving its initial solutions with the fewest number of fitness evaluations. However, the single-population algorithm required several more fitness evaluations to further converge. Furthermore, the value the single-population converged to was a worse solution than both CC and FEA. FEA, on the other hand, was able to quickly converge to a fit solution in the fewest number of fitness evaluations. Finally, CC took more fitness evaluations and also converged to a worse solution than FEA.

3.4.6 Analysis

The Paired Student t-Tests on the mean fitness show that FEA versions of DMVPSO all performed significantly better than the corresponding single-population

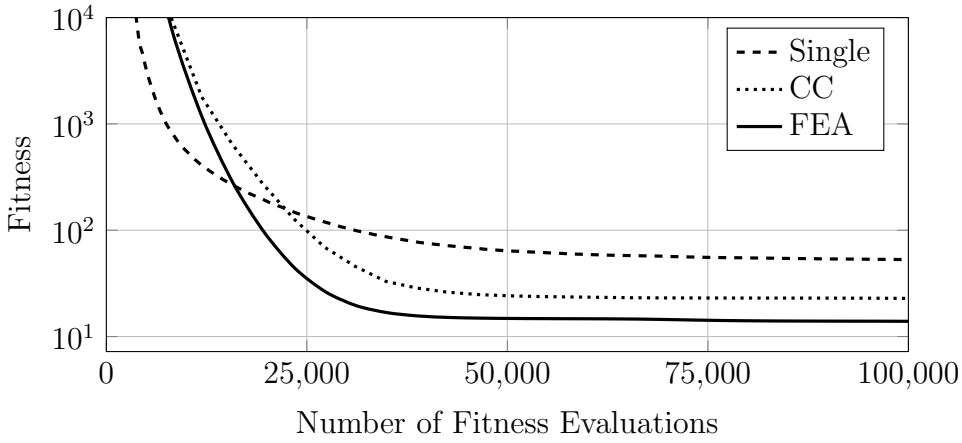


Figure 3.6: A fitness curve plotting fitness versus fitness evaluations on single-population, CC, and FEA PSO algorithms minimizing the Rosenbrock function. Note that the y-axis is shown on a logarithmic scale to allow for a compact display of fitness values over time.

and CC versions. This demonstrates that FEA's use of overlapping subpopulations is more effective than the non-overlapping subpopulations used by the CC algorithm.

Additionally, the results suggest that on difficult problems, such as the Hailfinder Bayesian network, FEA offers an increase in performance for almost all algorithms. We believe that this is because FEA's overlapping subpopulations help the algorithm avoid hitchhiking and TSFOSB. If a subpopulation contains a poor value due to hitchhiking, a better value from a different subpopulation may be selected during the competition. This new value will then be injected into the other subpopulation during the Share step of FEA, thus eliminating the value that is hitchhiking.

The results from the NK landscapes further support this hypothesis. FEA's performance over single-population and CC algorithms is significant on the landscapes with high variable interaction ($K = 10$). On those problems, the landscape is more complex, which increases the probability of the algorithms becoming trapped in a local optimum. As shown in previous chapters, the Markov architecture is able to maintain better diversity than other factor architectures. Furthermore, we believe that FEA

is able to maintain more diversity during the search process than single-population algorithms because the factors are updated independently during the Update step. This is a similar concept to the Island model for EAs, where (full) subpopulations are updated independently of one another.

Finally, the the benchmark function results further support our hypothesis that FEA versions of population-based algorithms perform better than both single-population and CC algorithms on complex or difficult problems. Many of the test functions used are designed to be difficult problems with many traps and valleys. FEA allows the algorithm to escape these by breaking up variables that are hitchhiking and allowing the algorithm to maintain diversity between the subpopulations. This is especially noticeable when comparing HC and FEA-HC, where FEA-HC always outperformed HC. Because HC is a greedy algorithm that only moves in directions with better fitness, it is susceptible to becoming stuck in local optima. FEA-HC allows the individuals to escape those suboptimal locations by sharing information between those individuals.

The results also support the claims by Fortier *et al.* that the increased performance obtained by FEA is due to a representation of each subpopulation that allows communication and competition to occur between overlapping populations [33] . When there was no overlap between subpopulations, such as in the CC algorithms, the performance was significantly worse than when there was overlap, like in FEA. By defining each node's subpopulation to cover its Markov blanket when performing abductive inference, we ensure that each population learns the state assignments for all variables upon which that node may depend. Also, because multiple subpopulations optimize over each variable, the FEA algorithm allows greater exploration of the search space. Through competition and sharing, FEA is

able to find good combinations of variable state assignments from the subpopulations are used in the final solution.

While FEA does find better solutions, we observe that FEA almost always requires roughly twice as many fitness evaluations. One source of the additional fitness evaluations come from the Compete step. However, based on the results in Figure 3.2, this corresponds to around 10% of the total number of fitness evaluations. As shown in the complexity analysis of FEA, this percentage is related to the number of iterations FEA uses in the Update step and the number of overlapping factors optimizing a variable. If FEA only ran each factor for one iteration, then the percentage of fitness evaluations used during the Compete step would increase. However, if the FEA had each factor run for more iterations, then the percentage of fitness evaluations incurred during Compete would decrease. Furthermore, the graphs in Figures 3.5 and 3.6 show that FEA quickly converges and fewer iterations could have been performed with little to no dropout in performance. The other source of the extra fitness evaluations is how FEA determines that convergence has been reached. In these experiments, FEA updated each factor ten times before performing competition and then checking for convergence. If FEA has begun to converge, FEA will be performing ten times more fitness evaluations than a single population; therefore, FEA performs unnecessary fitness evaluations towards the end of its convergence.

However, based on the fitness curves in Figures 3.5 and 3.6, FEA performs better than CC and single-population algorithms when limiting all the algorithms to the same number of fitness evaluations. While the single-population algorithm is faster early in the search process, it takes more fitness evaluations to converge while converging to worse solutions that CCEA and FEA. These results also demonstrate that the convergence criteria used in these experiments for FEA could be lowered without negatively affecting the performance. Here, convergence was reached when

the full global solution did not improve after 15 iterations, however, we could have used a much lower threshold.

3.5 Conclusion

In this chapter, we presented a framework for defining multi-population EAs. We then demonstrated how this framework can define the Island Model, CCEA, and MFEA. Next, we defined FEA and defined its three major subfunctions: Update, Compete, and Share. We then provided a complexity analysis of FEA and showed the conditions under which Update and Compete functions dominate the runtime of FEA. From this, we observe that while FEA will increase the number of fitness evaluations, the amount can be managed by adjusting parameters in FEA. Furthermore, FEA does not increase the complexity to update individuals.

Additionally, we examined the generality of FEA by performing experiments comparing single-population, CC, and FEA versions of GA, PSO, DE, and HC. In almost all cases, FEA outperformed single-population and CC algorithms. We also demonstrated that the performance of FEA is not dependent on the optimization algorithms. While FEA requires more fitness evaluations, the majority of these extra evaluations are caused by FEA performing extra fitness evaluations when the algorithm begins to converge. Additionally, the criteria for convergence could be lowed so that FEA performs fewer fitness evaluations without affecting the performance of FEA.

CHAPTER FOUR

FACTOR ARCHITECTURES

While there has been some work discussing different factor architectures, there has been little work optimizing factor architectures for FEA. In this chapter, we verify empirically that the performance of FEA is tied to the factor architecture. To do so, we test a variety of architectures on three different problems: abductive inference in Bayesian networks, maximizing NK landscapes, and optimizing a set of commonly-used benchmark functions for EAs.

From those results, we discover that factor architectures affect the performance of FEA. Additionally, there appears to be a relationship between different architectures. We then prove that a specific class of optimization problems can be mapped to factor graphs. Factor graphs are similar to Bayesian networks in that they can be used to encode a joint probability distribution over a set of random variables. Additionally, we use this mapping as a general method for deriving factor architectures when applying FEA to a problem by mapping the problem to a factor graph and use the resulting dependencies as a way to derive the factors. Finally, we demonstrate this general method by using an FEA version of GA on a set of benchmark functions, including an investigation into how the architecture outperforms other factor architectures.

4.1 Comparing Factor Architectures

We begin by comparing a set of factor architectures for performing abductive inference in Bayesian networks, maximizing NK landscapes, and optimizing a set of commonly-used benchmark functions for EAs.

4.1.1 Bayesian Networks

For all our experiments, we used an empty evidence set. Previous work by Fortier *et al.* randomly selected a 50% of the leaf variables as nodes to apply evidence when using OSI to perform full and partial abductive inference [33]. However, this raises the question as to how OSI performs when using other methods for selecting evidence. To eliminate these experimental decisions, we we do not apply any evidence to the Bayesian network; therefore, we are searching for the most probable state assignment for all variables, i.e. $\mathbf{X}_U = \mathbf{X}$. Additionally, when evidence is applied to the network, the number of variables to be optimized is reduced. Thus, by choosing not to apply evidence, we are testing on a more difficult optimization problem.

To test our architectures, we used the Hailfinder, Hepar2, Insurance, and Win95pts Bayesian networks from the Bayesian Network Repository [93]. Table 3.1 lists the number of nodes, edges, parameters, and average Markov blanket size for the selected networks. For each Bayesian network, we compared four different factor architectures, which are described below.

- **Random Architecture** — Random factors are considered as the baseline architecture. For this approach, a random subpopulation is constructed for each of the N variables. K variables are then added to each of the N subpopulations. For each individual Bayesian network, K was set to be equal to the rounded average Markov blanket size for the network.

- **Parents Architecture** — For each variable X_i, we construct a subpopulation of individuals consisting of $X_i \cup \mathbf{Pa}(X_i)$. This is one of the simplest ways to subdivide a Bayesian network and provide overlap. Additionally, this architecture takes advantage of the structure of the log likelihood function that is used to evaluate the fitness of individuals in the population.

- **Markov Architecture** — This architecture uses the Markov blanket of the nodes to create subpopulations, which offers arguably one of the most natural ways to subdivide a Bayesian network and provide overlap. For every variable, a factor was created consisting of the variable itself, and every node in its Markov blanket. This architecture may provide an advantage when performing inference because every node in the network is conditionally independent of all other nodes given its Markov blanket.

- **Clique Tree Architecture** — This architecture is one of the more complicated methods to create subpopulations. The Bayesian network is first moralized, which consists of connecting parents of variables with an undirected edge. Next, the directed edges of the Bayesian network are made to be undirected, followed by triangulating the network. Finally, the graph is decomposed into a clique tree by first computing the maximal cliques and then finding the maximum spanning tree weighted by the overlap size of neighboring cliques. Note that unlike the previous architectures, this method does not build a subpopulation by centering on a single variable. Instead, each subpopulation corresponds to the variables in a clique in the resulting clique tree.

Table 4.1 shows the results for the FEA-DMVPSO when performing abductive inference on the four different Bayesian networks using the four different factor architectures. The average fitness values, along with the standard deviation for error bounds, are displayed. Additionally, the results of statistical significance testing using a Paired Student t-Test with $\alpha = 0.05$ between each pairs of factor architectures are shown in Table 4.2. If an architecture performed significantly better than another architecture on a particular Bayesian network, the network's abbreviation is shown

Table 4.1: Average fitness of factor architectures for FEA-DMVPSO performing abductive inference on Bayesian networks.

Network	Random	Parents	Markov	Clique
Hailfinder	-37.98±2.17	-36.81±2.08	-35.13±1.59	-95.88±204.36
Hepar2	-19.46±2.73	-17.16±1.38	-16.37±0.00	-16.57±0.83
Insurance	-12.96±2.24	-12.32±2.26	-11.61±1.81	-12.73±2.10
Win95pts	-39.02±5.60	-48.24±105.62	-33.25±5.95	-68.22±148.80

Table 4.2: Hypothesis tests of factor architectures for FEA-DMVPSO performing abductive inference on Bayesian networks.

Architecture	Random	Parents	Markov	Clique
Random	–	–	–	–
Parents	Ha,He,–,–	–	–	Ha,–,–,–
Markov	Ha,He,I,W	Ha,He,–,–	–	Ha,–,I,–
Clique	–,He,–,–	–,He,–,–	–	–

where the row architecture outperforms the column architecture. If there is no significant difference, a "—" is shown.

The Markov architecture outperformed all other architectures on all networks. However, it only significantly outperformed the Parents architecture on the Hailfinder and Hepar2 networks, and the Clique architecture on the Hailfinder and Insurance networks. The Parents architecture also performed well on all networks except the Win95pts network, whereas the Clique architecture performed well only on the Hepar2 and Insurance networks. Finally, the Random architecture performed the worst on the Hepar2 and Insurance networks and did not significantly outperform any of the other architectures.

4.1.2 NK Landscapes

We tested each architecture on NK landscapes with parameters $N = 25, 40$ and $K = 2, 5, 10$. In our experiments, we compared the following factored architecture strategies on 50 randomly generated landscapes.

- **Random Architecture** — Random subpopulations for NK landscapes are constructed similarly to the Random architecture used in abductive inference in Bayesian networks. A subpopulation is created for every variable, and M variables are then added to each of the N subpopulations. For these experiments we set M to be equal to K, giving each factor a size of $K + 1$.

- **Neighborhood Architecture** — For each variable X_i, we create a subpopulation and add all the variables in the set $\mathbf{Nb}_K(X_i)$. In all of our experiments, we used the neighborhood function that returns the next K contiguous bits of the string starting at X_i. This results in subpopulation sizes of $K + 1$.

- **Loci Architecture** — The loci subpopulation extends the neighborhood architecture. Each variable X_i is still used to create a subpopulation along with the variables in $\mathbf{Nb}_K(X_i)$. Therefore, we add variable X_j to the subpopulation if $X_i \in \mathbf{Nb}_K(X_j)$. In other words, the subpopulation consists of all the variables in its neighborhood and all variables that contain X_i in their neighborhood. The neighborhood architecture will result in factors similar to those created by the Markov architecture for Bayesian networks. This creates subpopulations of size $2K + 1$.

Table 4.3 shows the results the FEA-DMVPSO on maximizing NK landscapes using different factor architectures. Results from the hypothesis testing are in Table 4.4, where a + is shown if the row architecture is better than the column architecture.

Table 4.3: Average fitness of factor architectures for FEA-DMVPSO maximizing NK landscapes.

N	K	Random	Neighborhood	Loci
25	2	18.01±0.78	18.41±0.76	18.42±0.75
	5	18.20±0.67	18.59±0.59	18.43±0.59
	10	17.81±0.58	18.03±0.58	17.67±0.57
40	2	28.51±1.25	29.34±1.18	29.37±1.13
	5	28.73±0.92	29.66±0.75	29.40±0.74
	10	28.04±0.82	28.81±0.75	28.25±0.72

If there is no significant difference or the row architecture is worse than the column architecture, a "−" is shown.

The Neighborhood architecture always outperforms the Loci and Random architectures. However, it is only significantly better than the Loci architecture for $K = 5$ and $K = 10$. Random was almost always the worst and was significantly outperformed by Neighborhood and Loci, except when $N = 25, K = 10$, where random performed statistically better than the Loci architecture. A key observation from these results is the Neighborhood architecture is never significantly outperformed on any of the landscapes.

4.1.3 Test Functions

In this section, we evaluate the performance of FEA on real-valued optimization problems by applying the algorithm to a set of test functions commonly used in the literature for testing EAs and swarm algorithms.

The seven functions selected for these experiments were the Ackley's (AK), Dixon-Price (DP), Exponential (EX), Griewank (GR), Rosenbrock (RO), Schwefel 1.2 (SH), and Sphere (SP) functions. All of the functions are defined in Appendix A We used the same set of standard function ranges as presented in [28]. All of the problems are minimization problems with global minima of 0.0, except for Exponential, which

Table 4.4: Hypothesis tests of factor architectures for FEA-DMVPSO maximizing NK landscapes.

N	K	Architecture	Random	Neighborhood	Loci
25	2	Random	−	−	−
		Neighborhood	+	−	−
		Loci	+	−	−
	5	Random	−	−	−
		Neighborhood	+	−	+
		Loci	+	−	−
	10	Random	−	−	−
		Neighborhood	+	−	+
		Loci	+	−	−
40	2	Random	−	−	−
		Neighborhood	+	−	−
		Loci	+	−	−
	5	Random	−	−	−
		Neighborhood	+	−	+
		Loci	+	−	−
	10	Random	−	−	−
		Neighborhood	+	−	+
		Loci	+	−	−

has a minimum of -1.0. Additionally, all of the problems are scalable, meaning they can be optimized for versions of any dimension. In our experiments, we used functions of 50 dimensions. For each function, we compared the following factored architecture strategies.

- **Simple Architecture** — Simple architecture are specified by two parameters, l and m, where l controls how many variables each factors optimizes over while m dictates how many variables each neighboring factor overlap. For example, values of 4 and 2 for l and m would create a factors $S_1 = \{X_1, X_2, X_3, X_4\}$, $S_2 = \{X_3, X_4, X_5, X_6\}$, $S_3 = \{X_5, X_6, X_7, X_8\}$, and so on. We will denote simple factor factor architecture with parameters l and m as SA-l, m.

Table 4.5: Average fitness of different factor architectures for FEA-PSO minimizing different benchmark test functions.

	CA-2	CA-5	CA-10	SA-4,2	SA-10,5
Ackley's	1.34E-14(1.03E-15)	1.30E-14(8.30E-16)	2.93E-2(2.93E-2)	5.86E-2(4.07E-2)	1.22E+1(1.00E+0)
Dixon-Price	7.99E-2(7.99E-2)	2.56E-1(7.99E-2)	3.63E+0(3.18E+0)	2.48E+0(4.24E-1)	1.33E+0(2.99E-1)
Exponential	-1.00E+0(6.18E-17)	-1.00E+0(6.18E-17)	-1.00E+0(6.18E-17)	-1.00E+0(6.18E-17)	-1.00E+0(6.18E-17)
Griewank	4.26E-3(2.04E-3)	4.00E-3(1.61E-3)	1.15E-2(3.85E-3)	2.15E-2(1.08E-2)	1.05E-1(2.73E-2)
Rosenbrock	4.57E+0(5.72E-1)	6.58E+0(2.75E+0)	3.98E+0(6.03E-1)	1.71E+1(5.21E+0)	6.29E+1(7.68E+0)
Schwefel	1.67E+4(1.88E+3)	8.45E+2(4.84E+2)	3.94E+2(3.32E+2)	1.97E+4(4.79E+3)	1.22E+4(3.05E+3)
Sphere	9.50E-42(8.87E-42)	9.74E-36(5.36E-36)	6.75E-32(3.37E-32)	9.69E-36(4.81E-36)	2.82E-19(2.62E-19)

Table 4.6: Hypothesis tests of factor architectures for FEA-PSO optimizing benchmark functions.

	CA-2	CA-5	CA-10	SA-4,2	SA-10,5
CA-2	—	—	—	Dp, Ro	Ak, Dp, Gr, Ro
CA-5	Sh	—	—	Dp	Ak, Dp, Gr, Ro
CA-10	Sh	—	—	Sh	Ak, Gr, Ro, Sh
SA-4,2	—	—	—	—	Ak, Dp, Sh
SA-10,5	—	—	—	—	—

- **Centered Architecture** — The centered factor generates a factor for each variable in the optimization problem. For each factor, the next l variables are included in the factor. The algorithm starts with variable X_1 and adds the next l variables $\{X_2, X_3 \dots X_{l+1}\}$. This is repeated for all variables. The architecture is called Centered Architecture because in the case that l is odd, the architecture can be viewed as adding the previous $(l-1)/2$ and next $(l-1)/2$ variables to a factor centered around variable X_i. We note that in our implementation, we did not use any wrap-around upon reaching the end of the list of variables. Because of this, factors centered around variables towards the end of the list of variables will be smaller. We denote this architecture as CA-l.

Table 4.5 displays the results of the different factor architectures on minimizing the benchmark test functions. Results are displayed as average fitness over 30 trials with the standard error shown in parentheses. CA-2 outperformed all other architectures on the Dixon-Price and Sphere functions. It was outperformed by CA-5

on Ackley's and Griewank. CA-10 performed the best on Rosenbrock and Schwefel. All of the architectures performed equally well on the Exponential function. Neither of the SS architectures performed better than the CS architectures.

Table 4.6 shows the pairwise significance tests. If an architecture performed significantly better than another architecture on a particular benchmark function network, the function's abbreviation is shown where the row architecture outperforms the column architecture. We note that in almost all cases, all of the CA architectures tied in terms of performance. Only on the Schwefel problem did CA-5 and CA-10 perform significantly better than CA-2. The difference between SA-4,2 and SA-10,5 was significant for the functions Ackley's, Dixon-Price, and Schwefel. Similarly, the CA-2 and CA-5 were significantly different from SA-10,5 on all functions except the Exponential, Schwefel, and Sphere. Finally, we note that that CA-2 was only significantly outperformed on the Schwefel function.

4.1.4 Analysis

From the Bayesian network results, the Markov architecture performed the best across all networks. However, it was only statistically better than the other architectures on certain networks. We believe this is because the size of FEA's factors also affect FEA's performance. This can be demonstrated by comparing Markov's performance on the Win95pts and Hailfinder networks. In the Win95pts network, the average Markov blanket size of a variable in the network is 5.92. On this network, the Markov architecture failed to be statistically better than Clique and Parents. But on the Hailfinder network, where the average Markov blanket size is 3.54, the Markov architecture was statistically better than all other architectures. These results also demonstrate that the Markov architecture will usually perform the best on networks that have relatively small average Markov blankets (< 5).

Within the NK landscape experiments, the best-performing architecture was Neighborhood. In almost all cases, Neighborhood outperformed Loci and Random. The Neighborhood architecture is similar to the Parents architecture for abductive inference in Bayesian networks. However, when $K = 2$, the Loci architecture, which is similar to the Markov architecture for Bayesian networks, performed better than the Neighborhood. This supports our previous claim that the performance of FEA's factors also depends on the size of the factors. When K became large, Loci's performance became worse as compared to Neighborhood's.

Based on the benchmark results, CA-2 was the best-performing architecture except on the Schwefel function. In that case, CA-5 and CA-10 performed the best. The Schwefel function output is dominated by the summation of factors, where the factor is composed of a product of input variables and the number of variables in the product ranges from 1 to n. This results in a function with variables that interact with one another to a large degree. Based on the results, we see that the factor architecture with larger factors, CA-10, is better able to capture the high level of variable interaction in the Schwefel function.

For the other benchmark functions, the problems are either completely separable, like the Sphere function, or are dominated by the summation of the product of only one or two variables, like Rosenbrock. In those cases, the factor architecture with small factors, CA-2, is capable of capturing the variable interaction. These results indicate that the best factor architectures are those that are appropriately sized for the product of variables in the benchmark function. Factors that are too large can lead to problems such as TSFOSB, like regular PSO. However, if the factors are too small, then FEA loses its effectiveness because variable interactions may not be captured by the factors.

4.2 Mapping Functions to Factor Graphs

Based on the results in the previous section, we discovered that factor architectures affect the performance of FEA. Furthermore, we demonstrated that taking advantage of groupings of highly related variables is the best way to create factor architectures. From these results, there appears to be a relationship between the Markov architecture for Bayesian networks and the Lock architecture for NK landscapes. If the problem can be represented as a Bayesian network, then the Markov blanket should be used to derive the factor architectures. This is because the Markov blanket for a variable makes it independent of all other variables outside of its Markov blanket, allowing the factor to discover good values in the search space.

In this section, we prove that a specific class of optimization problems can be mapped to factor graphs. Factor graphs are similar to Bayesian networks in that they can be used to encode a joint probability distribution over a set of random variables. Additionally, we use this mapping as a general method for deriving factor architectures when applying FEA to a problem by mapping the problem to a factor graph and use the resulting dependencies as a way to derive the factors. Finally, we demonstrate this general method by using an FEA version of GA on a set of benchmark functions, including an investigation into how the architecture outperforms other factor architectures.

4.2.1 Related Work

The idea of mapping an optimization to a probabilistic graphical model is similar to Estimation of Distribution Algorithms (EDA) [8]. EDA are a class of GAs that use a probability distribution to estimates the fitness function being optimized. While GAs are effective for a wide range of problems, they often struggle when building

blocks are not contained in contiguous sets of variables. This is because GAs struggle by breaking up building blocks too frequently or not mixing different combinations of the blocks together [78]. EDA attempts to alleviate these problems by using a learning algorithm to build a probability distribution while optimizing the function. During each iteration, an EDA will select individuals similar to a regular GA. The algorithm then uses the selected individuals to update the structure and parameters of a probabilistic graphical model. Finally, EDA generates a new set of individuals based on the distribution [77].

Most EDAs perform selection and generate new individuals in a similar fashion in that a probability vector is used to generate new individuals. However, there is a great deal of variation in how algorithms update their estimated distribution. One of the simplest ways is to assume independence between all variables in the problem [78]. Baluja first samples individuals from a probability vector [3]. From the sample, the best individuals are selected and used to update the probability by scaling the probability vector by the number of state occurrences in the selected individuals. Mühlenbein and Paass use a similar method in that individuals are sampled from a probability vector [70]. However, the probability vector is completely replaced by setting the distribution of a variable equal to the number of state occurrences in the selected individuals.

One issue with assuming independence between all variables is that the algorithms will be misled when there is a strong interaction between variables. De Bonet *et al.* relax the independence assumption and allow variables to be connected to two other variables [21]. Once samples have been generated and selected, the algorithms builds a Bayesian network by using a greedy chain search that minimizes mutual information between neighboring variables. The result is a chain of random variables that the authors also show minimizes the Kullback-Leibler

divergence between the estimated distribution and the true distribution [21]. Similar to the methods that assume independence between variables, the Bayesian network is sampled to generate new individuals, which are used to update the Bayesian network. This process is repeated until a stopping criterion is met.

Baluja presented an algorithm that relaxed the chain requirement to allow for building a tree over the variables [4]. After samples have been generated, a model is constructed using the Chow-Liu algorithm, which constructs a fully connected graph weighted by the mutual information between variables followed by finding a maximum spanning tree [14]. Next, the parameters for variables in the tree are estimated using the individuals in the population. This Bayesian network is then used to generate the individuals and the process is repeated until the stopping criteria is met. A forest version was presented by Pelikan and Mühlenbein [79], which used the Pearson's chi-square test instead of the using mutual information [79].

Many of the previously mentioned EDA algorithms for optimizing a function have been extended to allow for models that allow variables to interact with more than two other variables. While learning general Bayesian networks often requires more computational time than that of simplified learners, such as the minimum spanning tree algorithm by Chow-Liu algorithm, the result is often an overall reduction in the computational time of the EDA algorithm [14, 78]. There are several variations of EDA that use a general probability distribution to generate individuals. One of the earliest versions was called the factorized distribution algorithm (FDA) [69]. In this algorithm, it is assumed that the Bayesian network structure is given along with the optimization problem. During each step, the FDA uses the individuals in the population to update the local probability distributions in the network. This is repeated until some stopping criterion is met.

Harik used a more general algorithm called the Extended Compact Genetic Algorithm (ECGA) that would update both the structure and parameters of the Bayesian network [42]. ECGA first assumes all variables are independent and adds connections between variables that minimizes the minimum description length (MDL). If there are no connections that reduce the MDL, the algorithm stops adding links and then uses the Bayesian network to generate new individuals, which are then used to update the network The resulting Bayesian network often has groups of variables that are independent of all other groups. During each generation, a completely new Bayesian network is generated.

The Bayesian Optimization Algorithm (BOA) presented by Pelikan *et al.* extends ECGA to learn more general Bayesian networks [77]. Individuals are first randomly generated and selected based on fitness. Next, the K2 algorithm developed by Heckerman *et al.* is used to build a Bayesian network [43]. In BOA, the authors restrict variables to have no more than two incoming edges and used the Bayesian-Dirichlet score as the measure of network quality. Similar to other EDAs, BOA uses the Bayesian network to generate and selected new individuals which are then used update the network. Other authors have presented similar algorithms to BOA that use the Bayesian Information Criterion (BIC) to drive the Bayesian learning algorithm [30, 68].

While most of the work with EDAs involves learning Bayesian networks, there has also been some work in using Markov networks in EDAs instead of Bayesian networks. Distribution Estimation Using Markov Networks (DEUM) is one of the first algorithms focusing on Markov networks [96, 97]. DEUM is similar to FDA in that the structure of the network is given upfront. During each iteration, the algorithm updates the parameters to the Markov network using the current generation of individuals.

An extension was made to DEUM by Shakya *et al.* that allows for learning the structure of the Markov network [95]. The algorithm does this by calculating the mutual information between variables to determine pairs of neighboring variables. Once a structure is learned, the algorithm is able to compute the parameters of the network using the individuals in the population. There has also been some work using Walsh functions to investigate how the learned Markov network can affect the search process [8].

EDAs are related to our work here, where we show how a Factor Graph can represent NK landscapes, because they use a probability distribution to represent a fitness function. However, our work differs in that we create an exact mapping from NK landscapes to factor graphs and therefore do not have to learn a probability distribution. Furthermore, the factor graph is not used to generate individuals during the search process.

4.2.2 Factor Graphs

We begin by presenting a formal definition of a factor followed by factor graphs and abductive inference in factor graphs. In all the following definitions, we define $\mathbf{D}$ to be a set of random variables.

Definition 4.1. A *factor potential* is a function ϕ: $Val(\mathbf{D}) \rightarrow \mathbb{R}$, where $Val(\mathbf{D})$ is the set of values for a set of variables $\mathbf{D}$. The function ϕ defines the affinity or strength of relationship between states for a variable.

Example 3. Suppose we have two random variables A and B, each of which can take two states: Yes and No. An example factor potential for the two variables is:

A	B	ϕ
Yes	Yes	30
Yes	No	5
No	Yes	1
No	No	10

Here, this potential defines a relationship where there is a high likelihood that A and B agree. The highest potentials are for (Yes, Yes) and (No, No) while the lowest values are for (Yes, No) and (No, Yes).

Definition 4.2. Let $G = (\mathbf{V}, \mathbf{E})$ be the set of vertices and edges in a graph where $\mathbf{V} = \{\mathbf{X} \cup \boldsymbol{\phi}\}$. $\mathbf{X} = \{X_1, X_2, \ldots, X_n\}$ is the set of variables and each X_i is a random variable. $\boldsymbol{\phi}$ is the set of factors that define the potentials between variables. An edge $e_{i,j} \in \mathbf{E}$ connects a factor ϕ_i to a variable X_j if X_j is an argument of factor ϕ_i. Note that edges only exist between variable nodes and factor nodes. The set of variables that are connected to factor ϕ_i is denoted as $\mathbf{D}_i$. A joint distribution P_Φ for a *factor graph* is then defined as

$$P_\Phi(X_1, \ldots, X_n) = \frac{1}{Z} \prod_{i=1}^{m} \phi_i(\mathbf{D}_i)$$

where Z is a normalization factor that is calculated as

$$Z = \sum_{X_i \in \mathbf{X}} \left(\prod_{i=1}^{m} \phi_i(\mathbf{D}_i) \right).$$

Example 4. Figure 4.1 shows an example factor graph. The square nodes represent the factor potentials while the circles are random variables. In this example, there are three factor potentials and four random variables. The potential ϕ_1 defines a function

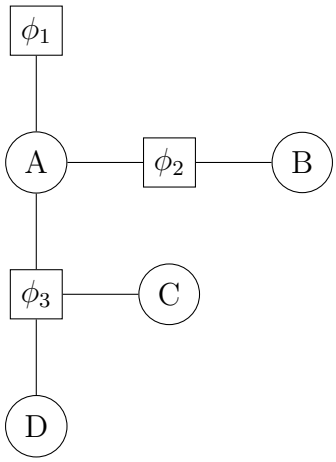


Figure 4.1: Example of a factor graph.

for $\mathbf{D}_1 = \{A\}$ while ϕ_2 defines the potentials for $\mathbf{D}_2 = \{A, B\}$. Finally, the function ϕ_3 defines the potentials for $\mathbf{D}_3 = \{A, C, D\}$.

We can also define a Markov blanket in a factor graph.

Definition 4.3. Given a variable X_i in a factor graph, the *Markov blanket* of X_i is defined as all of the variables X_k that are connected to any of the factor nodes ϕ_j connected to the node X_i [1].

Example 5. In Figure 4.1, the Markov blanket for A are all other nodes in the network because there is a factor connecting A to B, C, and D. However, variable B is only connected to A through ϕ_2, and therefore, the Markov blanket for B is A. Finally, the Markov blanket for C is A and B while the Markov blanket for D is A and B.

Next, we give a definition for abductive inference in factor graphs.

Definition 4.4. Given a set of evidence variables and their corresponding states in a factor graph, *abductive inference* is the problem of finding the maximum *a posteriori* (MAP) probability state of variables without any evidence. If we let $\mathbf{X}_U = \mathbf{X} \backslash \mathbf{X}_O$,

where $\mathbf{X}$ denotes the variable nodes in the network, the problem of abductive inference is to find the most probable state assignment to the variables in $\mathbf{X}_U$ given the observation of evidence $\mathbf{X}_O = \mathbf{x}_O$. MAP is performed by maximizing the equation

$$MAP(\mathbf{X}_U, \mathbf{X}_O) = \underset{\mathbf{x} \in \mathbf{X}_U}{\operatorname{argmax}} \frac{1}{Z} \prod_{i=1}^{m} \phi_i(\mathbf{d}_i)$$

where $\mathbf{d}_i$ is equal to the (full or partial) state assignments for the variables connected to the factor ϕ_i defined by $\mathbf{x}_O$. Z is a normalization factor and is calculated over the restricted potentials defined by $\mathbf{x}_o$ as

$$Z = \sum_{X_i \notin \mathbf{X}_o} \left(\prod_{i=1}^{m} \phi_i(\mathbf{d}_i) \right).$$

Because MAP is an argmax and only concerned with the most likely state assignments and not the probability of the assignment, the normalization factor Z can be dropped [1]. Note that multiplying several small probabilities together can cause the values to go to zero. To address this issue, the log of the probabilities is often used instead of the raw probabilities [1]. Since taking the log does not change the optimization problem, the MAP equation becomes

$$MAP(\mathbf{X}_U, \mathbf{X}_O) = \underset{\mathbf{x} \in \mathbf{X}_U}{\operatorname{argmax}} \sum_{i=1}^{m} \log \phi_i(\mathbf{d}_i). \tag{4.1}$$

We now give a theorem about how optimizing NK landscapes can be reduced to abductive inference in factor graphs followed by the proof. Recall that an NK landscape is a function $f : \mathcal{B}^N \to \mathbb{R}^+$ where $\mathcal{B}^N$ is a bit string of length N and K specifies the number of other bits in the string that a bit is dependent on. Fitness is

calculated as

$$f(\mathbf{X}) = \frac{1}{N}\sum_{i=1}^{N} f_i(X_i, \mathbf{Nb}_K(X_i))$$

where $\mathbf{Nb}_K(X_i)$ returns the K bits that are located within X_i's neighborhood.

Theorem 4.2.1. Given an NK landscape, a corresponding factor graph can be defined such that all factors ϕ_i correspond to a functions f_i from the landscape and the optimization problem in the NK landscape is identical to abductive inference over the factor graph.

Proof. Assume we are given an NK landscape L with N bits and K interactions per bit. Remember the task for optimizing an NK landscape is to find state assignments $\mathbf{x} = \langle x_1, x_2, \ldots, x_n \rangle$ for $\mathbf{X}$ that maximizes the function

$$f(\mathbf{X}) = \frac{1}{N}\sum_{i=1}^{N} f_i(X_i, \mathbf{Nb}_K(X_i))$$

This maximization problem can be formulated as

$$M(L) = \underset{x \in Val(\mathbf{X})}{\operatorname{argmax}} \sum_{i=1}^{n} f_i(x_i, nb_K(x_i)).$$

We will construct a factor graph $G = (\mathbf{V}, \mathbf{E})$ such that the vector defining the most probable explanation for G is equal to the vector denoting the global maximum of L.

For each bit X_i in L, a variable node X_i and a factor node ϕ_i are inserted into the factor graph G. Edges are then added between ϕ_i and each node X_j in $nb_K(X_i)$. Due to our factor graph construction, $\mathbf{D}_i = \{X_i, nb_K(X_i)\}$. Parameterize each factor ϕ_i as $\phi_i(\mathbf{D}_i) \leftarrow e^{f_i(X_i, nb_K(X_i))}$ for each entry in the factor table. Note that if given an empty evidence set $\mathbf{X}_O$, performing abductive inference on the newly constructed

factor graph is

$$MAP(\mathbf{X}_U) = \underset{\mathbf{x} \in Val(\mathbf{X}_U)}{\operatorname{argmax}} \sum_{i=1}^{m} \log \phi_i(\mathbf{D}_i).$$

Since we defined $\phi_i(\mathbf{D}_i) = e^{f_i(X_i, nb_K(X_i))}$, we have $\log \phi_i(\mathbf{D}_i) = f_i(x_i, nb_K(x_i))$. Therefore the optimization problem can be rewritten as

$$MAP(\boldsymbol{X}_U) = \underset{x \in Val(\boldsymbol{X})}{\operatorname{argmax}} \sum_{i}^{n} f_i(x_i, \boldsymbol{Nb}_K(x_i)).$$

This equation is identical to the equation maximizing NK landscapes, therefore, $M(L) = MAP(X)$. □

Theorem 4.2.1 has several implications. The first is that it shows a direct relationship between NK landscapes and probabilistic graphical models, allowing us to use any results for analyzing probabilistic graphical models on NK landscapes and vice versa. For example, this theorem can be used as an alternative reduction to prove the NP-hardness of abductive inference in factor graphs, and by extension, on Bayesian networks. Second, it can allow for an alternative way define NK landscapes. For example, K could be viewed as an upperbound to the number of variables connected to a factor. Additionally, the discrete factors in the factor graph could be replaced with continuous factors, which would define a new class of continuous NK landscapes. Note that would be an alternative model than the continuous NK landscape presented by Li *et al.* [61]. For the purposes of this paper, we will restrict our use of Theorem 4.2.1 to informing us on how to apply FEA to NK landscapes.

We also extend this to any maximization function with a similar form. This is expressed in the following corollary.

Corollary 4.2.2. Given a function to be maximized with the the form

$$f(\mathbf{X}) = \sum_{i=1}^{N} f_i(\mathbf{D}_i)$$

where $\mathbf{D}_i = \{X_j, X_k, \ldots X_l\}$, a corresponding factor graph can be defined such that optimizing the function is identical to abductive inference of the factor graph.

Proof. Assume we are given a maximization function f with N variables and that this maximization problem can be formulated as

$$M(f) = \underset{x \in Val(\mathbf{X})}{\operatorname{argmax}} \sum_{i=1}^{n} f_i(\mathbf{D}_i).$$

We will construct a factor graph $G = (\mathbf{V}, \mathbf{E})$ such that the vector defining the most probable explanation for G is equal to the vector denoting the global maximum of f.

For each dimension X_i in f, a variable node X_i and a factor node ϕ_i are inserted into the factor graph G. Edges are then added between ϕ_i and each node X_j in $\mathbf{D}_i = \{X_j, X_k, \ldots X_l\}$. Each factor ϕ_i is parameterized as

$$\phi_i(\mathbf{D}_i) \leftarrow \exp(f_i(X_j, X_k, \ldots X_l))$$

for each entry in the factor table. Note that if given an empty evidence set $\mathbf{X}_O$, performing abductive inference on the newly constructed factor graph is calculated as

$$MAP(\mathbf{X}_U) = \underset{\mathbf{x} \in Val(\mathbf{X}_U)}{\operatorname{argmax}} \sum_{i=1}^{m} \log \phi_i(\mathbf{D}_i).$$

Since we defined $\phi_i(\mathbf{D}_i) = e^{f_i(\{X_j, X_k, \ldots X_l\})}$, we have $\log \phi_i(\mathbf{D}_i) = f_i(\{X_j, X_k, \ldots X_l\})$. Therefore the optimization problem can be rewritten as

$$MAP(\boldsymbol{X}_U) = \underset{x \in Val(\boldsymbol{X})}{\operatorname{argmax}} \sum_{i}^{n} f_i(x_j, x_k, \ldots x_l)).$$

This equation is identical to maximizing the function f, therefore, $M(f) = MAP(X)$.

□

We note two major differences between Theorem 4.2.1 and Corollary 4.2.2. The first is that Corollary 4.2.2 does not place any restriction on the variables being continuous or discrete. Second, Corollary 4.2.2 does not require that each f_i use the same size $\mathbf{D}_i$ input that NK landscapes require. Furthermore, the sets $\mathbf{D}_i$ can be any subset of variables.

Note that this corollary only applies to maximization. However, any maximization problem can be converted to a minimization by creating a new function $f'(\mathbf{X}) = -f(\mathbf{X})$. This means that any function of the form

$$f(\mathbf{X}) = \sum_{i=1}^{N} f_i(X_i, X_{i+1}, \ldots, X_{i+k_i}) \tag{4.2}$$

can be mapped to a factor graph.

We give the following example to demonstrate the mapping.

Example 6. Suppose we are given the Rosenbrock function

$$f(\mathbf{X}) = \sum_{i=1}^{N-1} 100(X_{i+1} - X_i)^2 + (1 - X_i)^2.$$

Note that this is a minimization function that can be transformed into a maximization function, giving us the folloing

$$f'(\mathbf{X}) = -f(\mathbf{X}) = -\sum_{i=1}^{N-1} 100(X_{i+1} - X_i)^2 + (1 - X_i)^2.$$

Note that f_i is equal to

$$f'_i(X_i, X_{i+1}) = -100(X_{i+1} - X_i)^2 - (1 - X_i)^2.$$

This gives us the following equation

$$f'(\mathbf{X}) = \sum_{i=1}^{N-1} f'_i(X_i, X_{i+1}).$$

Each function ϕ_i is then set as

$$\phi_i(X_i, X_{i+1}) \leftarrow \exp(-100(X_{i+1} - X_i)^2 - (1 - X_i)^2).$$

4.3 Empirical Analysis

We extended the experiments in the previous experiments by first analyzing FEA using a GA instead of a PSO. In that section, we analyzed factor architectures for FEA using only PSO as the underlying optimization algorithm. Additionally, we restricted their analysis to the average fitness of each architecture on only four networks. Here, we incorporate more networks and look at other performance characteristics, such as population diversity, to gain a better understanding where each architecture gains its advantage.

In these experiments, we tested each of the architectures on three different problems: abductive inference in Bayesian networks, maximizing NK landscapes, and

Table 4.7: Properties of Bayesian networks.

	Nodes	Edges	Parms.	Avg. MB Size
Alarm	37	46	509	3.51
Andes	223	338	1157	5.61
Barley	48	84	114,005	5.25
Child	20	25	230	3.00
Diabetes	413	602	429,409	3.97
Hailfinder	56	66	2656	3.54
Hepar2	70	123	1453	4.51
Insurance	27	52	984	5.19
Link	724	1125	14,211	4.80
Mildew	35	46	540,150	4.57
PathFinder	135	200	77,155	3.04
Pigs	441	592	5,618	3.66
Water	32	66	10,083	7.69
Win95pts	76	112	574	5.92

minimizing a set of commonly-used benchmark functions. For the Bayesian networks, we used a set of networks from the Bayesian Network Repository [93]. The networks used and parameters are shown in Table 4.7. NK landscapes were generated randomly using combinations of $N = 25, 40$ and $K = 2, 5, 10$. For each Bayesian network, 30 trials were performed for each factor architecture. Because NK landscapes are randomly generated, we generated 30 landscapes. Each version of FEA was then run 30 times on each landscape. On the benchmark test functions, we used the Brown, Dixon & Price, Powell Singular, Rana, and Rosenbrock. These functions were chosen because they match the form of the function shown in Equation 4.2, and each f_i is comprised of more than one variable.

The GA used in our experiments used tournament selection and uniform crossover. The mutation rate was set to 0.15. Additionally, each factor contained ten individuals. For each of the problems, we used three different methods for deriving factor architectures.

Table 4.8: Results of FEA-GA performing abductive inference on Bayesian networks.

	Markov	Parents	Random (M)	Random (P)
Alarm	**−1.26E+01 (7.17E−01)**	−1.64E+01 (7.80E−01)	−1.76E+01 (9.70E−01)	−1.94E+01 (1.06E+00)
Andes	**−6.92E+01 (6.33E−01)**	−7.47E+01 (8.41E−01)	−8.27E+01 (8.90E−01)	−8.35E+01 (1.45E+00)
Barley	**−5.10E+01 (9.69E−01)**	−5.62E+01 (1.35E+00)	−5.48E+01 (1.43E+00)	−6.17E+01 (1.78E+00)
Child	**−7.61E+00 (3.35E−01)**	−9.13E+00 (4.07E−01)	−9.64E+00 (3.67E−01)	−1.04E+01 (4.16E−01)
Diabetes	**−1.42E+04 (5.42E+02)**	−1.63E+04 (5.81E+02)	−1.79E+04 (3.78E+02)	−1.89E+04 (9.45E+02)
Hailfinder	**−3.63E+01 (4.26E−01)**	−8.79E+01 (3.45E+01)	−1.14E+02 (4.15E+01)	−6.39E+01 (2.48E+01)
Hepar2	**−2.02E+01 (5.74E−01)**	−2.13E+01 (6.03E−01)	−2.19E+01 (6.61E−01)	−2.14E+01 (6.35E−01)
Insurance	**−1.26E+01 (4.39E−01)**	−1.43E+01 (6.52E−01)	−1.45E+01 (5.03E−01)	−1.54E+01 (6.18E−01)
Link	**−3.61E+03 (3.61E+02)**	−5.70E+03 (5.84E+02)	−6.04E+03 (4.94E+02)	−7.28E+03 (5.14E+02)
Mildew	**−1.02E+03 (9.64E+01)**	−1.30E+03 (1.02E+02)	−1.32E+03 (1.28E+02)	−1.27E+03 (1.64E+02)
PathFinder	**−5.27E+02 (7.42E+01)**	−8.79E+02 (9.89E+01)	−1.42E+03 (1.28E+02)	−1.67E+03 (1.84E+02)
Pigs	**−2.67E+02 (4.70E+00)**	−2.85E+02 (2.66E+00)	−2.76E+02 (1.85E+00)	−2.83E+02 (5.86E+00)
Water	**−4.08E+02 (6.90E+01)**	−5.84E+02 (8.50E+01)	−4.84E+02 (1.04E+02)	−6.81E+02 (1.03E+02)
Win95pts	**−2.21E+01 (8.35E−01)**	−2.90E+01 (1.01E+00)	−6.31E+01 (2.45E+01)	−3.95E+01 (1.46E+00)

- **Parents Architecture** — For each variable X_i, we construct a subpopulation of individuals consisting of elements in the neighborhood of X_i.

- **Markov Architecture** — This architecture uses the Markov blanket of the nodes to create subpopulations. For each problem, a factor is created for each variable X_i consisting of X_i and all the nodes in X_i's Markov blanket.

- **Random Architecture** — Random subpopulations are considered as the baseline architecture. For this approach, a random subpopulation is constructed for each of the N variables. K variables are then added to each of the N subpopulations. We used two different values for K. The first is setting it equal to the number of variables in a node's neighborhood and denote it as Random (P). Random (M) denotes a random architecture where M is set equal to the average size of a Markov blanket for each test problem.

Similar to the previous chapter, we hypothesize that the Markov architecture will outperform the other architectures [106]. Additionally, we hypothesize that in the Random architectures, the subpopulations with fewer variables, Random (P), will outperform Random (M) because it will be less susceptible to hitchhiking.

Table 4.9: Results of FEA-GA maximizing NK landscapes.

		Markov	Parents	Random (M)	Random (P)
N = 25	K = 2	**1.87E+01 (2.33E-02)**	1.85E+01 (2.54E-02)	1.82E+01 (2.82E-02)	1.80E+01 (2.99E-02)
	K = 5	**1.91E+01 (1.73E-02)**	1.88E+01 (2.02E-02)	1.87E+01 (2.02E-02)	1.81E+01 (2.32E-02)
	K = 10	**1.87E+01 (1.70E-02)**	1.84E+01 (1.73E-02)	1.86E+01 (1.55E-02)	1.82E+01 (1.96E-02)
N = 25	K = 2	**2.96E+01 (3.23E-02)**	2.93E+01 (3.30E-02)	2.86E+01 (3.72E-02)	2.85E+01 (3.73E-02)
	K = 5	**3.05E+01 (2.34E-02)**	2.99E+01 (2.56E-02)	2.93E+01 (2.92E-02)	2.87E+01 (3.13E-02)
	K = 10	**2.98E+01 (2.24E-02)**	2.95E+01 (2.26E-02)	2.93E+01 (2.45E-02)	2.86E+01 (2.60E-02)

Table 4.10: Results of FEA-GA minimizing benchmark functions.

	Markov	Parents	Random (M)	Random (P)
Brown	**1.67E−02 (1.52E−03)**	3.43E−02 (3.24E−03)	4.17E−02 (7.06E−03)	6.15E−02 (6.29E−03)
Dixon & Price	**3.64E+00 (4.26E−01)**	4.14E+00 (3.59E−01)	**3.67E+00 (4.38E−01)**	5.69E+00 (5.81E−01)
Powell Singular	**4.80E−01 (2.77E−02)**	6.75E−01 (5.10E−02)	6.51E−01 (5.09E−02)	9.66E−01 (1.22E−01)
Rana	**−1.41E+04 (9.56E+01)**	−1.33E+04 (1.30E+02)	−1.32E+04 (1.29E+02)	−1.25E+04 (1.32E+02)
Rosenbrock	6.99E+01 (9.10E+00)	7.01E+01 (1.29E+01)	5.72E+01 (1.20E+01)	4.60E+01 (6.34E+00)

4.3.1 Results

Table 4.8 shows the results for performing abductive inference on the Bayesian networks using FEA-GA. Results are reported for each of the four different factor architectures in terms of average fitness values. The standard error for confidence bounds is given in parentheses. Similarly, Table 4.9 displays the results of FEA-GA to maximize NK landscapes. Finally, we present the benchmark functions results in Table 4.10. All results are averaged over 30 trials. In all tables, a bold value indicates that a Paired Student t-Test with $\alpha = 0.05$ determined that factor architecture significantly outperformed the others on that specific problem. If multiple values in a single row are bold, that means that the bolded architectures were not statistically significantly different from each other but significantly outperformed all non-bolded architectures.

Looking at Table 4.8, we can see that Markov always significantly outperforms all competing architectures. The next best performing architecture was Parents, as it outperformed both Random architectures on the Alarm, Andes, Child, Diabetes,

Hepar2, Insurance, Link, Pathfinder, and Win95pts networks. On the Barley, Water, and Pigs networks, the Random (M) architecture performed second best, while on Mildew and Hailfinder, Random (P) was second.

For the NK landscapes, Markov again demonstrated the best performance to a statistically significant level. Out of the remaining architectures, Parents performed best and was only outperformed by Random(M) on $N = 25$ and $K = 10$.

Finally, Markov had the best overall performance on the benchmark functions, statistically outperforming the other methods in most cases. However, it failed to differ statistically from Random (M) on Dixon & Price and was outperformed by Random (P) on the Rosenbrock function. Parents was second only on Brown while Random (M) performed best on the Dixon & Price and second best on the Powell Singular functions. Random (P) was best on Rosenbrock and second best on the Rana.

4.3.2 Analysis

Based on our results, the Markov architecture performs best on the majority of problems. This is because this architecture groups variables together that are highly related, allowing interactions to be captured by inter-swarm optimization. Similarly the Parents architecture groups variables that are highly related; however, the resulting swarm for a variable X_i contains the variables that X_i is dependent on, but does not contain variables that depend on it. Thus, the superior performance of the Markov architecture can be attributed to the fact that it includes both sets of variables in each swarm, and thus does a better job of grouping highly related variables.

Even though the Markov architecture has larger factors, it does not appear to suffer from hitchhiking, which often affects populations that optimize over large sets

of variables. This is illustrated by the fact Random (M), which has larger factors, outperformed Random (P) on all networks except Hailfinder, Hepar2, Mildew, and Win95pts. If hitchhiking had been present, we would have expected the larger factors to impede the algorithm's performance. The NK landscape experiments further support these results: the Random (M) architecture outperformed the smaller random architecture, Random (P), on all landscapes tested.

Additionally, Parents outperformed Random (M) on almost all NK Landscapes. This suggests that the level of interaction between factor variables, rather than the sizes of the factors, is more important when creating a factor architecture.

To investigate further where the Markov architecture's performance gains originate, we analyzed the diversity over time for each of the different architectures. Because most individuals optimize over different subsets of variables, we used the genotype variance, a measure of the distance between each individual and the "average" individual, to measure diversity [9]. This variance is calculated as

$$\frac{1}{N \times P} \sum_{i=1}^{N} \sum_{j=0}^{P} (\bar{x}_i - x_{i,j})^2$$

where $\bar{x}_i$ is the average value across the population for variable X_i, $x_{j,i}$ is individual j's value for variable X_i, P is the total number of individuals, and N is the number of variables in the problem being optimized. The graphs for both fitness and diversity over time for FEA-GA are shown in Figure 4.2.

Results are presented for 100 trials of FEA-GA optimizing an NK landscape with parameters $N = 25$ and $K = 2$. The y-axis on the chart on the top denotes the average diversity, while the y-axis on the bottom chart is the best fitness. The x-axis shows the number of FEA iterations, where each iteration consists of the update, compete, and share steps, and each factor performs five updates during a single iteration.

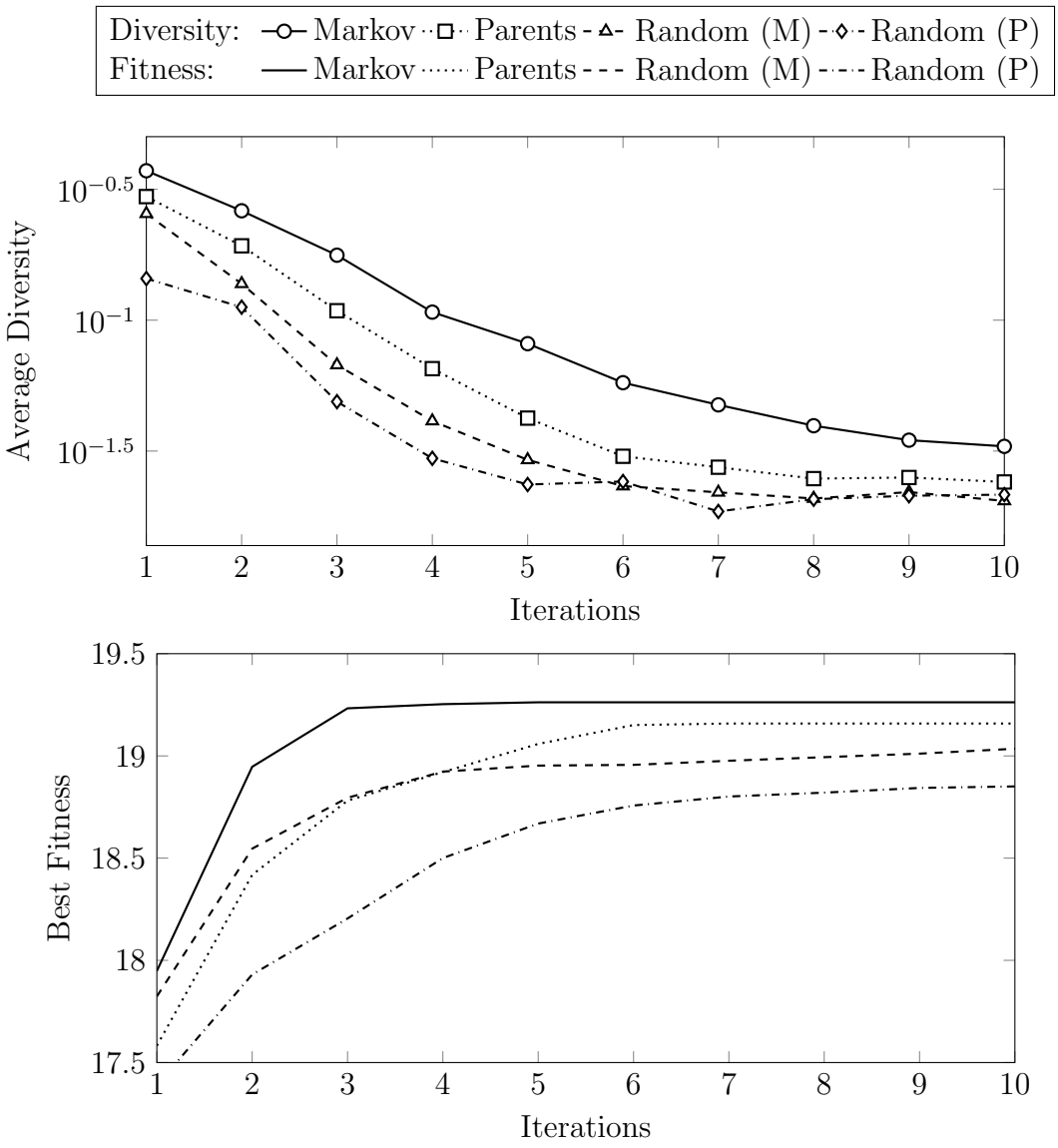


Figure 4.2: Average diversity and best fitness of all individuals for NK landscape N = 25, K = 2.

When considering fitness, the Markov architecture converges the fastest, and it maintains the best population diversity over time. We believe this is because the Markov architecture provides the best balance between the number of variables in the factors and the level of interaction being handled. Larger factors provide more variation between individuals' state assignments, thus leading to higher diversity in the population. Given this, we might expect the Markov and Random (M) architectures to have identical diversity curves, as they yield the same factor sizes for NK Landscapes; however, we observed that both Random architectures had relatively low diversity. Also, while Parents has smaller factors than Random (M), its diversity was higher on average. This implies that factor size is not the only influence on average diversity, and that grouping together highly interactive variables also increases the average diversity.

A similar analysis was performed on the Rosenbrock function, the results of which are in Figure 4.3. We note that the y-axis on the bottom, which shows the best fitness, is in log scale and inverted to allow for a better view of the graph. The first thing we observed is that the Markov architecture reaches a plateau in around 40 iterations, while the other architectures take around 60-80 iterations before reaching decreasing returns. This is consistent with the NK landscape results, where the Markov architecture also reaches convergence quickest. However, unlike in the NK landscape experiments, the other architectures continue to increase, albeit slowly after reaching this plateau. In fact, after roughly 120 iterations, the Random (M) architecture's fitness curve reaches slightly above the Markov fitness curve.

Looking at the diversity results, it appears that the Markov architecture tends to lose diversity quicker than the other architectures. Markov's diversity curve initially declines more steeply than those of other architectures, then appears to stabilize at that level of diversity for the remainder of optimization. On the other hand, the

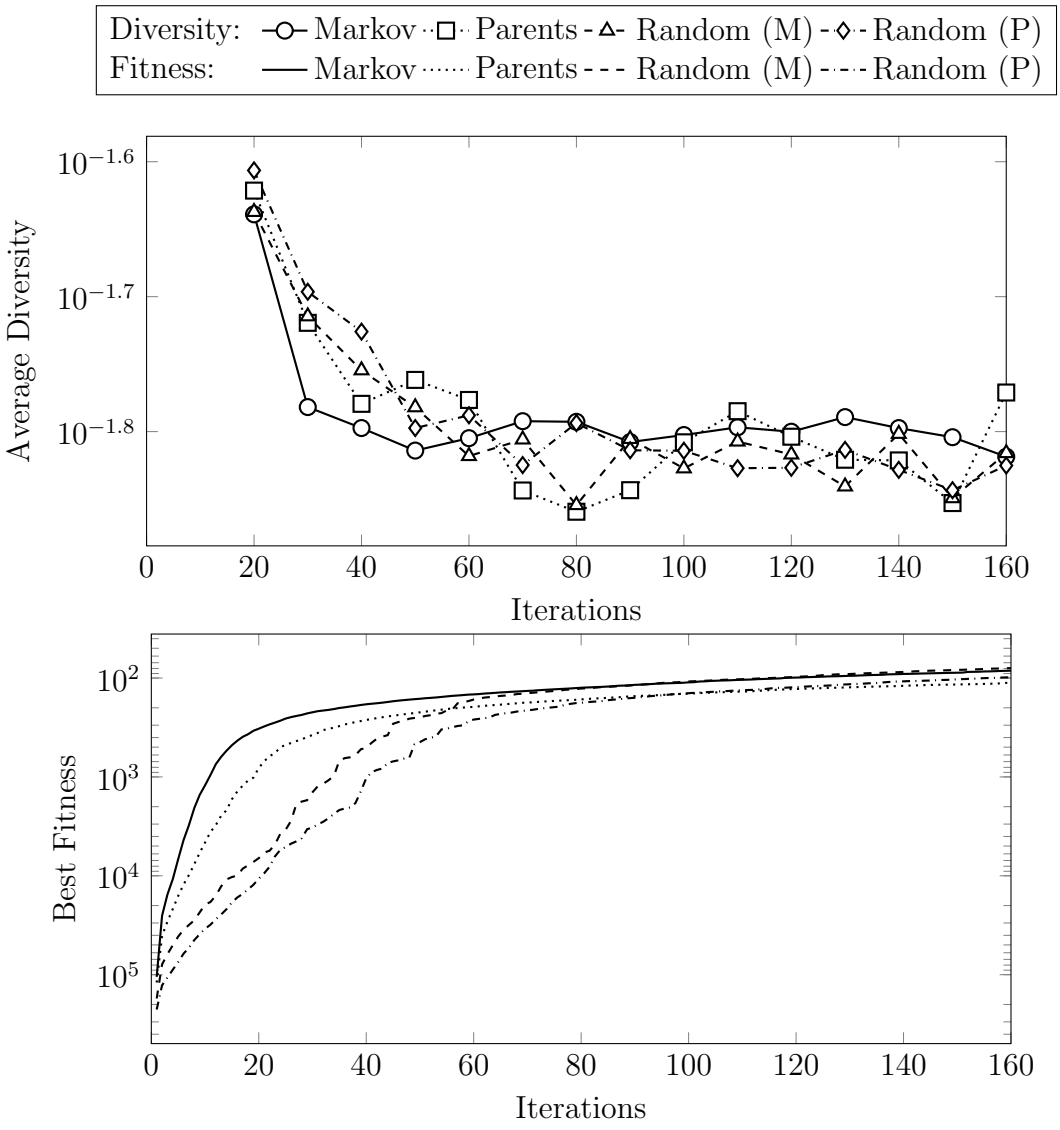


Figure 4.3: Average diversity of all individuals in FEA over time for Rosenbrock.

Parents, Random (M), and Random (P) architectures take slightly longer to reach a low level of diversity. These three architectures' diversity also fluctuate much more during the remainder of the iterations. This suggests that the Markov architecture reaches stagnation after its initial diversity loss, where solution values are changing relatively little, while the other architectures produce solutions that continue to vary significantly in later iterations.

We believe the reason for this is that the Markov architecture is able to find a good solution relatively quickly, but then becomes stuck in that local optimum. Ideally, an optimization method should be able to escape local optima in order to explore the search space further, balancing exploitation of previous-discovered good solutions with additional search. More research is needed in order to explore why this trade-off appears to be a problem for the Markov architecture on the Rosenbrock function, but not for the Bayesian networks and NK landscapes. One possible explanation is that the Rosenbrock function is by its nature able to deceive factors that are optimizing over highly interactive variables, that is, the function leads the factors to local optimum. We will further explore this hypothesis in Chapter 9.

4.4 Conclusion

In this Chapter, we empirically demonstrated that taking advantage of groupings of highly related variables is the best way to create factor architectures. From these results, there appears to be a relationship between the Markov architecture for Bayesian networks and the Lock architecture for NK landscapes. If the problem can be represented as a Bayesian network, then the Markov blanket should be used to derive the factor architectures. This is because the Markov blanket for a variable makes it independent of all other variables outside of its Markov blanket, allowing the factor to discover good values in the search space. We then proved that maximizing

an NK landscape is equivalent to abductive inference in factor graph. Furthermore, any function that can be expressed as a summation of functions can be mapped to a factor graph. These mappings can then be used as a general method for creating a factor architecture for FEA. The function is mapped to a factor graph and then the Markov blankets for a variable are used to create a factor architecture for the function. Next, we demonstrated the performance advantage of the Markov architecture over other approaches. Finally, we showed that the Markov architecture maintains better diversity in the population.

CHAPTER FIVE

DISCRETE PARTICLE SWARM OPTIMIZATION

Discrete optimization problems, such as feature selection or inference in Bayesian networks, represent an important and challenging set of problems. These differ from continuous problems in that each variable can take on only a finite number of states [29]. An example is integer problems, where variables are restricted to a set of integer values. For such problems, there exists a relationship between neighboring values. More generally, there is an implicit ordering in the integers: integers with a larger difference between them are considered to be further apart.

While integer problems are a subset of discrete problems, there are other types. For example, in abductive inference for Bayesian networks, the goal is to find the set of states that best explains a set of observations. Here, there may not exist a direct relationship or gradient between neighboring states. For example, say the set of states is a set of furniture Chair, Table, Dresser, Couch, and Desk. While these states may be represented with integers during optimization, there is no real ordered relationship between the values of this *encoding*. We refer to such problems as categorical optimization problems.

Particle Swarm Optimization (PSO) is a highly customizable, yet relatively simple search algorithm applicable to a wide variety of optimization problems. However, the original PSO algorithm is unable to handle discrete problems, such as the ones discussed above, as its velocity update requires continuous solution values [53, 73, 84]. Currently, there are several extensions to the PSO algorithm that allow discrete solution values, though the definition of "discrete" varies widely between applications and algorithms.

Previous chapters demonstrated that FEA can use a wide range of underlying optimization algorithms on various discrete problems. One design decision encountered during those experiments was which discrete PSO algorithm to use. Our initial investigation found that the discrete PSO by Veeramachaneni *et al.* performed well and used it in several experiments. However, a more careful analysis revealed that the discrete PSO by Veeramachaneni *et al.* did not actually behave as originally expected [115]. This discovery presented an opportunity to develop a new discrete PSO algorithm called Integer and Categorical PSO (ICPSO).

The goal of our algorithm is to keep the extensions to continuous PSO as simple as possible and preserve much of the original semantics, while also addressing some of the potential pitfalls of other discrete PSO algorithms. To achieve this, we alter the representation of the particle's position so that each attribute in a particle is a distribution over its possible values rather than a value itself. This is similar to Estimation of Distribution Algorithms (EDAs) where a set of fit individuals is used to generate a distribution vector that can then generate fitter solutions [60]. ICPSO differs from EDAs in that the algorithm uses a population of distribution vectors that are updated using the PSO update equations.

For ICPSO, evaluating a particle becomes the task of sampling a candidate solution from these distributions and then calculating its fitness. ICPSO also allows us to use the original PSO update equations and avoids problems associated with an implicit ordering of the possible solution values. Additionally, ICPSO modifies the global and local best solutions' distributions whenever a global best sample is produced. This serves to bias distributions toward the best sample they have produced while still allowing exploration of the search space.

In this chapter, we present our new PSO algorithm, including a review of related work followed by several experiments demonstrating the performance of ICPSO over

Table 5.1: Average iterations for an individual to find the optimal solution in a parabola.

Algorithm	Iterations
PSO	2.53
VPSO	12.09
RAN-R	9.99
RAN-NR	5.52

competing approaches. Additionally, we test the performance of FEA using different discrete PSO algorithms.

5.1 Revisiting the Veeramachaneni PSO Algorithm

We begin by first revisiting the discrete PSO algorithm proposed by Veeramachaneni *et al.* [115]. To do so, we performed a simple experiment to evaluate the number of iterations until an individual reaches a local optimum. Using only one individual, we set the global best for the individual to the optimal solution in the parabola $f(X) = -X^2 + 10X + 75$ restricted to integer values in the range [0, 10]. Furthermore, we removed the local best and any update rules related to the local best. We then examined the number of iterations required for the individual to converge to the optimal solution. Table 5.1 gives the average number of iterations for PSO and Veeramachaneni PSO (VPSO) to reach the global best. Additionally, we present results of a random search algorithm (RAN) with and without replacement. Results are over 30 trials.

RAN-R is random with replacement while RAN-NR is a random search without replacement. Here, we discovered that even on a simple problem, an individual in VPSO takes longer than expected to move towards the global best solution. In fact, the results are almost to that of a random search without replacement. From

these results, we conclude that the update rules for VPSO are ineffective at moving individuals towards the global best. If VPSO is unable to move the individuals towards the global best, then there is not a high probability of VPSO being able to move individuals towards good solutions when used in real world problems. These results motivate the need to develop a more effective discrete PSO algorithms.

5.2 Discrete Optimization

Across the literature, "discrete" PSO has taken on a number of meanings. Many methods address integer problems; however, there are many applications where the discrete solution values are not integers. Similarly, while categorical optimization is a form of discrete optimization, not all discrete optimization is necessarily categorical. Due to this ambiguity in the literature, we propose a more specific definition of discrete optimization problems.

Definition 5.1. Discrete Optimization: A class of problems where an objective function is to be optimized that has variables whose values are limited to finite sets, numerical or categorical, ordered or unordered.

The definition of discrete optimization is often a bit fuzzy because integers are frequently used as an example. This definition recognizes that discrete optimization problems often require integer solutions where the possible states for each variable are numerical, discrete, ordered and there is a fitness relationship between adjacent or nearby values. But it also emphasizes that some discrete problems are over variables with categorical values such as the previous emotion example. In this case, there is not necessarily a fitness relationship between the values Chair and Couch because the set is unordered.

More importantly, we could resort to an integer encoding for this set for representational convenience, just because the integers in the encoding are ordered, this does not mean there is now a fitness relationship between adjacent values in the encoding. For example, we could just as easily have encoded our set of emotions from above: {Chair, Couch, Desk, Table, Dresser} as $\{1, 2, 3, 4, 5\}$ or $\{3, 5, 4, 2, 1\}$.

Traditionally, discrete sets include both finite and countably infinite sets. For practical reasons, we limit ourselves to finite sets due to finite computer memory. However, our algorithm is applicable to all discrete problems falling within this definition. We refer to our algorithm as Integer and Categorical PSO (ICPSO) to emphasize that it can be applied to both integer and categorical discrete problems.

5.3 Related Work

In this section, we review other approaches to discrete particle swarm optimization, both binary and multi-valued. This analysis provides the necessary context for our experiments comparing the performance of ICPSO to competing approaches. Additionally, this section serves to highlight the problems, endemic in discrete PSO methods, that our contributions are intended to address.

First, we discuss PSO variants that mostly maintain the core update equations and have only been augmented with a few additional equations to handle discrete problems. These algorithms are the most similar to ours, and thus the bulk of the discussion will be focused here.

Next, we present versions of PSO that have been combined with other optimization algorithms, such as Estimation of Distribution Algorithms (EDA). We include these hybrid algorithms because ICPSO shares some similar ideas, though it differs in implementation. Finally, we discuss other PSO variants applied to specific discrete problems, such as the Traveling Salesperson Problem. While these algorithms

are only designed for specific applications, we include them in our discussion for the sake of completeness.

5.3.1 Discrete Variations of PSO

It is possible to use the original continuous PSO to solve problems with integer-valued solutions by rounding the particle's position at each iteration [48]. We will refer to this algorithm as Integer PSO (IPSO).

IPSO requires a relationship between neighboring states, as the velocity update equation uses subtraction to measure the distance between the particle's current position and the global/local best positions. As such, it is only applicable to certain problems where the states can be ordered or arranged in a way that provides the necessary relationship between neighboring states.

Binary PSO (BPSO) was proposed originally by Kennedy and Eberhart [53]. BPSO requires that the position vector be a binary representation of candidate solutions. This representation also changes the velocity interpretation: the velocity represents the probability of each variable assuming the value 0 or 1. While the velocity update for BPSO remains unchanged, the position update is modified to take advantage of the new semantics of the velocity vector. After updating the velocity vector, each term in the velocity is mapped into a [0,1] interval using the sigmoid function

$$S_{i,j} = \frac{1}{1 + \exp(-V_{i,j})} \tag{5.1}$$

where $V_{i,j}$ is the value of the jth variable for particle i [53]. Next, a random number is sampled from the normal distribution $X_{i,j} = \mathcal{N}(0, 1)$ and converted to be in $[0, 1]$ by first calculating $S_{i,j} - X_{i,j}$, and then using a unit step function to snap the difference to 0 or 1. This value is then assigned as particle i's current position for variable X_j. When velocity is high, the position update is more likely to select a value closer to

1 than 0. Conversely, when velocity is low, there is a higher likelihood of selecting 0 [53].

The main limitation of BPSO is that it requires a binary representation. Standard binary coding is commonly used for binary representation, but has the disadvantage of introducing Hamming cliffs. Hamming cliffs represent situations where adjacent binary-encoded numbers have a large Hamming distance between them, or where two binary-encoded numbers with a very small Hamming distance actually have a large difference in value.

An alternative strategy is to use gray coding, where the Hamming distance between neighboring values is set to 1, which reduces the Hamming Cliff problem. However, both methods' encodings may overrepresent the problem if the number of states is not a power of 2. This can also increase problem dimensionality, slowing optimization [64].

Veeramachaneni *et al.* developed an extension to binary PSO that relaxes the need for a binary representation of the problem [115]. This was discussed in earlier chapters under the name DMVPSO. Throughout the rest of the paper, we will refer to this algorithm as the Veeramachaneni PSO (VPSO). In VPSO, each variable is allowed to assume any of M discrete values. While the velocity update remains unchanged from the binary case, the position update is modified to allow for more than two states. After the velocity has been updated, it is mapped into the $[0, M-1]$ interval by first using a generalized version of the sigmoid function in Equation (5.1), which is given as

$$S_{i,j} = \frac{M-1}{1+\exp(-V_{i,j})}.$$

Next, each particle's position is updated by generating a random number according to the normal distribution $X_{i,j} = \mathcal{N}(S_{i,j}, \sigma \times (M-1))$ and rounding the result. Then

the piecewise function

$$X_{i,j} = \begin{cases} M-1 & X_{i,j} > M-1 \\ 0 & X_{i,j} < 0 \\ X_{i,j} & \text{otherwise} \end{cases}$$

is applied to ensure all values fall within $[0, M-1]$ [115].

While VPSO does extend BPSO so that any number of discrete values can be used, the algorithm requires a relationship between neighboring states in the range of variable values, just like in IPSO. In [115], the authors demonstrated that VPSO is able to outperform BPSO when mapping continuous variables to quaternary or ternary. However, all experiments contained relationships between neighboring states.

The multi-valued PSO extension most similar to ours was introduced by Pugh and Martinoli [84]. We will refer to this variant as PPSO. PPSO, like ICPSO, uses a probabilistic interpretation of a particle and evaluates fitness stochastically by generating a sample solution.

The position vector, however, does not represent a valid probability distribution explicitly in PPSO. To evaluate the position, the authors presented a process to generate a sample that can be directly evaluated in the fitness function. When generating a sample, each element in the position vector has a sigmoid transformation applied to each of its terms. The probability of state k for variable j is equal to the position vector's jth element divided by the weighted sum of all other elements. This allows the solution element to take on any value from 0 to a user-specified n. This sampling procedure involves several more steps than our implementation of discrete PSO.

An adjustment must also be applied after each modification of the particle's values in order for all of the particles to share a common reference frame [84]. To

achieve this adjustment, a value $c_{i,j}$ is subtracted from each value of particle P_i, element j's vector of values. This $c_{i,j}$ is calculated such that all values of the vector, when mapped to the sigmoid function, sum to 1. The resulting equation is solved for c via an approximate root-finding method to produce an adjustment for each value in each element of the position vector. To address the noisy fitness evaluation, the algorithm also reevaluates best particles at each iteration, averaging fitness over particles' lifetimes. Unlike in ICPSO, the sample's value is not used when setting the global or local best positions.

Angle Modulated PSO (AMPSO), reduces a high-dimensional binary search space into a smaller continuous search space using an angle modulation-based method, thus reducing the number of parameters to be optimized [73]. This speeds up optimization while potentially improving performance. The approach uses the angle modulation equation, which is given as

$$g(x) = \sin(2\pi(x - a) \times b \times \cos(A)) + d$$

where $A = 2\pi \times c(x - a)$ and x is a single input value.

AMPSO first optimizes over the parameters a, b, c, and d in the angle modulation equation. Next, for each variable, the algorithm generates k evenly-spaced values, where k is the number of bits needed to represent every state in the discrete problem. These k values are then transformed into a bit string by converting positive values to 1 and negative values to 0.

The novelty of AMPSO is more related to the transformation of the search space than the PSO implementation itself. Additionally, this approach requires a binary representation of the problem, which has similar limitations to BPSO.

5.3.2 Hybrid Algorithms

Another related class of algorithms combines Estimation of Distribution Algorithms (EDA) and PSO to create an EDA-PSO hybrid [6, 27, 124]. These approaches either use EDA to help guide the movement of particles in PSO or use PSO to generate or update individuals generated by an EDA. Our work differs from these hybrid approaches in that it retains the core PSO update equations by modifying the particle representation to fit discrete problems. Additionally, many of these hybrid algorithms are designed to operate on continuous problems, while ICPSO is specifically designed for discrete optimization.

In the work by El-Abd and Kamel, the authors propose a hybrid algorithm in which a particle is either updated according to the PSO update equations or replaced with a new individual sampled from the estimated distribution [27]. In another example of a hybrid algorithm, Zhou *et al.* developed an algorithm called Discrete Estimation of Distribution Particle Swarm Optimization (DEDPSO), in which the local best positions from all individuals are used to update the distribution vector. This update distribution vector is then sampled to update the existing individuals [124]. The algorithm is designed to operate on binary vectors.

In [87], Reynolds *et al.* first generate a set of individuals using EDA and insert them into a PSO swarm. PSO runs for a set number of iterations and then uses the updated positions to generate new individuals. Those individuals are then added back to the original set of individuals from EDA. The set of individuals from both algorithms are then used to update the probability distribution, and the process is repeated. This is similar to the work by Bengoetxea and Larrañaga [6]. Their algorithm generates individuals independently using EDA and PSO and uses the generated individuals from both algorithms to update the distribution.

Kulkarni and Venayagamoorthy use both EDA and PSO when updating an individual [58]. An individual is first updated according to the PSO update equations, and a new individual is generated using EDA. Of these, the individual with the best fitness is kept in the swarm [58]. Santucci and Milani take a different approach to hybridization by using PSO within an EDA framework [92]. This is done by replacing the PSO update equations with those found in EDA. Then each variable in an individual is updated using EDA. After all individuals have been updated, the distribution is updated using the new positions of the individuals [92].

5.3.3 Other Approaches

In addition to the hybrid and discrete PSO algorithms, there are several PSO variants tailored to specific discrete problems, such as the Traveling Salesperson Problem (TSP). In these cases, each application requires a specific mapping or transformation of the problem [13]. For example, Clerk uses PSO to solve TSP by representing the particle as a path through all nodes [15]. With this representation, the velocity is a set of changes to be made to the path. The addition and subtraction operators are then re-defined to fit the modified semantics of the optimization [15].

Sha *et al.* propose a PSO algorithm to solve the job shop scheduling problem [94]. Each particle represents a matrix, where each element is the priority of a job on a machine. The velocity represents a swap operator of a job to a different machine. To evaluate a solution, the matrix is decoded into a schedule using Giffler and Thompson's heuristic [37]. This schedule is then evaluated for fitness.

Other approaches use a variation of integer PSO by rounding continuous values to integer values during fitness evaluation. This is similar to the work done by Salman *et al.*, where the authors applied PSO to the job shop scheduling problem [91]. Each particle's position represents a matrix that contains assignments of a task to a machine

or processor. The velocity update remains unchanged, but is also represented by a matrix. Particle evaluation is done after rounding values in the matrix [91].

Hela and Abdelbar also used a matrix representation in using PSO to solve the quadratic assignment problem [44]. In that work, the velocity is represented by a matrix where an element (i, j) represents the likelihood of variable i taking on value j. The position is then updated by probabilistically selecting an element for variable i, using roulette wheel selection, based on the values in row i of the velocity matrix. To update the velocity, the position array is expanded to a binary position matrix where $(i, j) = 1$ if variable i is set to value j. In some ways, this type of matrix-based representation is similar to the one used in ICPSO. However, in our algorithm, an individuals position is also a matrix representation, which allows for the positions to contain a more detail representation of the problem. Consequently, when updating the velocity, the difference will contain a finer level of granularity, which also allows for better updates to the position of an individual.

A more recent algorithm for combinatorial optimization problems uses what is called set-based PSO [13]. Here, each individual represents a subset of values out of a universal set, and the velocity represents the probability of an element being selected for inclusion in the set. To fit the updated position and velocity semantics, the authors define new set-theoretic velocity and position update equations [13].

5.4 Integer and Categorical PSO

We propose an alternative position representation that supports discrete-valued solutions. We first describe the particle representation used in this PSO variant, followed by the necessary changes to the update equations. Finally, we give a modified fitness evaluation procedure and explain how to set the personal and global best vectors.

5.4.1 Representation

The current position for a particle in ICPSO is a set of probability distributions, one for each dimension of the solution. This differs from other PSO variants, where a particle's position is often a direct representation of the solution values. When optimizing over discrete values, this direct representation creates a problem: namely, there is an assumption that there must be a relationship between neighboring states, and that the arithmetic difference between states must be indicative of distance between them. While certain discrete applications may have a fairly natural way to order the possible states, others do not.

Using our emotions example from before, suppose we have a solution with three attributes and {Dresser, Table, Chair, Couch, Desk} is encoded as $\{1, 3, 2, 4, 5\}$. Say we have two particles with positions $\mathbf{P_{p_1}} = (1, 4, 5)$ and $\mathbf{P_{p_2}} = (1, 3, 5)$, and the global best is at $\mathbf{gBest} = (1, 5, 5)$. The vector subtraction during the velocity update would imply that $\mathbf{p_1}$ is closer to the global best than $\mathbf{p_2}$ is. However, semantically, one could argue that Chair is closer to Desk than Couch is; therefore, the variable ordering is not indicative of a meaningful ordering of the states.

ICPSO's particle representation avoids this problem by using probability distributions rather than single values in the position vector. A particle P's position is represented as

$$\mathbf{X}_P = [\mathcal{D}_{P_1}, \mathcal{D}_{P_2}, \ldots, \mathcal{D}_{P_N}]$$

where each $\mathcal{D}_{p,i}$ denotes the probability distribution for variable X_i. In other words, each entry in the particle's position vector is itself comprised of a set of distributions:

$$\mathcal{D}_{P,i} = [d^a_{P,i}, d^b_{P,i}, \ldots, d^k_{P,i}],$$

where $d^j_{P,i}$ corresponds to the probability that variable X_i takes on value j for particle P.

A particle's velocity is a vector of n vectors ϕ, one for each variable in the solution, that adjust the particle's probability distributions. Formally, this is represented as:

$$\mathbf{V}_p = [\phi_{p,1}, \phi_{p,2}, \ldots, \phi_{p,n}]$$
$$\phi_{p,i} = [\psi^a_{p,i}, \psi^b_{p,i}, \ldots, \psi^k_{p,i}].$$

where $\psi^j_{p,i}$ is particle p's velocity for variable i in state j. Since these values are continuous, the velocity update equation remain unchanged from the update equation in traditional PSO.

5.4.2 Update Equations

The velocity and position update equations are identical to those of traditional PSO, seen in Equations (2.1) and (2.1). However, because we are working with distributions instead of real values, the difference and addition operators for the distribution and velocity vectors take on a slightly different meaning. For this reason, we will define them explicitly as follows. The difference operator is defined as a component-wise difference between the two position vectors, i.e. for each variable X_i and value $j \in Vals(X_i)$,

$$d^j_{(\mathbf{pBest_p}-\mathbf{P_p}),i} = d^j_{pB,i} - d^j_{p,i}.$$

Here, $d^j_{p_B}$ is the personal best position's probability that variable X_i takes value j. The global best equation is identical except $\mathbf{pBest_p}$ is replaced with **gBest** and $d^j_{pB,i}$ with $d^j_{gB,i}$.

The addition of the velocity vector to the position vector is similarly component-wise over each value in the distribution. For each probability for variable X_i and possible value j, the addition is $d^j_{p,i} + \psi^j_{p,i}$.

These operations have the potential to create probabilities that fall outside $[0, 1]$. In order to maintain a valid probability distribution, any value outside this range is mapped to the nearest boundary. The distribution is then normalized to ensure that its values sum to 1.

To evaluate a particle p, its distributions are sampled to create a candidate solution. This sample is denoted

$$\mathbf{S}_p = [s_{p,1}, s_{p,2}, \ldots, s_{p,n}]$$

where $s_{p,j}$ denotes the state of variable X_j.

The fitness function is used to evaluate the sample's fitness, which then is used to evaluate the distribution.

5.4.3 Setting the Best Vectors

When a particle produces a sample that beats the global or local best, we use both the distributions from that particle's position, $\mathbf{P}_p$, and the sample itself, $\mathbf{S}_p$, to update the best values. The goal of this update is to bias the distribution that produced the best sample toward producing similar samples in the future. This is accomplished by reducing the probability, for each variable, of taking on any state except its state in the best sample. Mathematically, for all states $j \in Vals(X_i)$ the

global best's probability is updated as

$$d^j_{gB,i} = \begin{cases} \epsilon \times d^j_{p,i} & \text{if } j \neq s_{p,i} \\ d^j_{p,i} + \sum\limits_{\substack{k \in Vals(X_i) \\ \wedge k \neq j}} (1-\epsilon) \times d^k_{p,i} & \text{if } j = s_{p,i} \end{cases}$$

where ϵ, the *scaling factor*, is a user-set parameter that determines the magnitude of the shift in the distribution. We restrict the scaling factor to values in $[0, 1)$. This increases the likelihood of the distribution producing samples similar to the best sample, while inherently maintaining a valid probability distribution. This can be shown as follows:

$$\begin{aligned} \sum_{j \in Vals(X_i)} d^j_{gB,i} &= \sum_{j \in Sv(X_i,k)} \left(d^j_{p,i} \times \epsilon\right) + d^k_{p,i} + \sum_{j \in Sv(X_i,k)} \left(d^j_{p,i} - d^j_{p,i} \times \epsilon\right) \\ &= \sum_{j \in Sv(X_i,k)} \left(d^j_{p,i} \times \epsilon\right) + d^k_{p,i} + \sum_{j \in Sv(X_i,k)} \left(d^j_{p,i}\right) - \sum_{j \in Sv(X_i,k)} \left(d^j_{p,i} \times \epsilon\right) \\ &= d^k_{p,i} + \sum_{j \in Sv(X_i,k)} \left(d^j_{p,i}\right) = 1 \end{aligned}$$

where $Sv(X_i, k) = \{j | j \in Vals(X_i) \wedge j \neq k\}$ and $k = s_{p,i}$. The procedure for setting the local best is directly analogous. The global best sample is returned as the solution at the end of optimization.

The pseudocode for setting the global best on a single variable's distribution is given in Algorithm 5.10. First, the distribution new $\hat{\mathcal{D}}_{p,i}$ is initialized. Given a state from the sample and a distribution for the corresponding variable, the algorithm iterates over all probabilities in the distribution. For probability indices k not equal to the index of the state $s^j_{p,i}$, the probability is multiplied by the scaling factor ϵ (line 6). This new probability is inserted into the variable $\hat{d}^k_{gB,i}$. Otherwise, the probability for state k' must be increased. To do so, the algorithm calculates the total change in

Algorithm 5.10: ICPSO Update Best

Input: Probability Distribution $\mathcal{D}_{p,i}$, State $s^j_{p,i}$
Output: Updated Distribution $\hat{\mathcal{D}}_{gB,i}$

1: Initialize $\hat{\mathcal{D}}_{gB,i}$
2: $k' \leftarrow s^j_{p,i}$
3: $\Delta \leftarrow 0$
4: **for all** $d^k_{p,i} \in \mathcal{D}_{p,i}$ **do**
5: **if** $k \neq k'$ **then**
6: $\hat{d}^k_{gB,i} \leftarrow \epsilon \times d^k_{p,i}$
7: **else**
8: **for all** $d^l_{p,i} \in \mathcal{D}_{p,i}$ and $l \neq k$ **do**
9: $\Delta \leftarrow \Delta + \left((1-\epsilon) \times d^l_{p,i}\right)$
10: **end for**
11: $\hat{d}^k_{gB,j} \leftarrow d^k_{p,i} + \Delta$
12: **end if**
13: **end for**
14: **return** $\hat{\mathcal{D}}_{gB,i}$

probability by iterating over all other variables and recording the difference (lines 8 - 10). This value Δ is then added to $d^k_{p,i}$ in line 11, which increases the likelihood of the distribution generating samples with state $s^j_{p,i}$ for variable X_i.

5.5 Single Population Experiments

We compare ICPSO against the algorithms from Section 5.3.1, as those were the closest in approach and interpretation. Specifically, we compared to the Angle Modulated PSO (AMPSO), Binary PSO (BPSO), Binary PSO using Gray coding (BGPSO), Integer PSO (IPSO), the PSO proposed by Pugh and Martinoli (PPSO), and the PSO proposed by Veeramachaneni *et al.* (VPSO). Additionally, we present results of a Random PSO (RAN), which gives a baseline to compare with the discrete

PSO algorithms. RAN updates its individuals by randomly generating new positions and only keeps track of the global best position as a final solution to be returned.

5.5.1 Design

We compare the algorithms on a set of benchmark functions (namely, Ackley, Griewank, Rastrigin, Rosenbrock, and Sphere) using the same function ranges as presented in the appendix of Eberhart and Shi [26]. Normally, these are studied as continuous functions to be optimized; however, we modify them to allow for both integer and categorical (discrete) optimization. For the integer discrete optimization problem, we restrict the states of the variables to integer values. This permits adjacent values to correlate with their fitness as defined by the original function. For categorical problems, and unique to our experimental design within the discrete PSO literature, we break the relationship between the adjacent numerical values and their fitness by mapping the integer values to a randomly chosen ("shuffled") integer encoding. For example, the state values of some variable x, $\{1, 2, 3, 4, 5\}$, might be shuffled to the encoding $\{4, 2, 1, 3, 5\}$.

We tested all discrete algorithms on both shuffled (categorical) and unshuffled (integer) versions. Functions were restricted to ten dimensions and ten states for each dimension. For the shuffled versions, we generated 30 different shuffles and ran each algorithm 30 times on the each function. Algorithms were also run 30 times on each of the unshuffled problems.

We also tested the algorithms on NK landscapes, which are usually binary strings. We, however, used a generalized version that allows for categorical strings, where each variable can take on D different values. For our experiments, we used an NK-landscape with $N = 10$ and $K = 2$. We varied the number of states, D, for each dimension to values 2, 4, 6, 8, 10, 15, and 20. For each set of NK landscape

parameters, we generated 30 different landscapes and ran each algorithm 30 times per landscape.

Finally, we tested the discrete PSO algorithms on abductive inference in Bayesian networks. We used the same set of Bayesian networks from the experiments in Section 3.4.2, Chapter 4: Hailfinder, Hepar2, Insurance, and Win95pts. Network details are shown Table 3.1. For fitness evaluations, we used the log likelihood and for each network, we ran each algorithm 30 times per landscape.

All PSO variants used the same set of parameters in order to make comparisons as consistent as possible. Parameters ϕ_1 and ϕ_2 were set to 1.49618, and $\omega = 0.729$, which has been found to encourage convergent trajectories [26]. Each algorithm used a swarm of size five and terminated once the global best did not change after 50 iterations. This is due to the recommendations of [28], which demonstrated that a large swarm may, counterintuitively, have difficulty exploring the search space. For our approach, we set the scaling factor ϵ to 0.75, and in the VPSO we set σ to the authors' recommended value of 0.2. All algorithms randomly initialized velocity and position vectors. Significance testing was done using a Paired Student t-Test with $\alpha = 0.05$.

5.5.2 Results

Results from the test functions, reported as average solution fitness, are shown in Table 5.2. The center column displays unshuffled function results, and results on the shuffled functions are presented in the far right column. All results are reported in terms of average solution fitness. Standard error is shown in parentheses. Bold values indicate algorithms that statistically significantly outperformed *all* other algorithms.

On the unshuffled functions, IPSO significantly performed the best. This was expected, given that IPSO is able to follow the gradient of the search space to good

Table 5.2: Results of discrete PSO algorithms minimizing benchmark functions.

	Ackleys	Shuffled Ackleys	Griewank	Shuffled Griewank
AMPSO	4.45E+00(5.79E-01)	5.59E+00(1.77E-01)	7.64E-01(6.54E-02)	9.58E-01(1.86E-02)
BGPSO	6.74E+00(8.64E-02)	5.23E+00(8.04E-02)	9.77E-01(8.29E-03)	9.44E-01(7.73E-03)
BPSO	6.94E+00(9.12E-02)	5.05E+00(1.05E-01)	9.62E-01(9.20E-03)	9.19E-01(1.06E-02)
ICPSO	2.49E+00(1.10E-01)	**2.45E+00(1.16E-01)**	3.92E-01(3.25E-02)	**3.85E-01(2.79E-02)**
IPSO	**7.47E-01(1.35E-01)**	4.74E+00(9.12E-02)	**1.63E-01(3.23E-02)**	6.54E-01(3.65E-02)
PPSO	4.27E+00(1.58E-01)	4.33E+00(1.34E-01)	8.28E-01(2.11E-02)	8.27E-01(1.58E-02)
VPSO	6.49E+00(1.01E-01)	4.32E+00(9.47E-02)	9.55E-01(7.28E-03)	9.03E-01(8.49E-03)
RAN	5.26E+00(9.93E-02)	5.07E+00(1.15E-01)	8.75E-01(1.03E-02)	8.62E-01(1.50E-02)

	Rastrigin	Shuffled Rastrigin	Rosenbrock	Shuffled Rosenbrock
AMPSO	2.65E+01(3.81E+00)	3.13E+01(3.22E+00)	1.72E+04(3.65E+03)	3.11E+04(5.25E+03)
BGPSO	4.45E+01(1.12E+00)	3.07E+01(1.17E+00)	3.61E+04(2.01E+03)	1.96E+04(1.20E+03)
BPSO	4.65E+01(1.51E+00)	3.01E+01(9.76E-01)	3.57E+04(2.37E+03)	2.12E+04(1.31E+03)
ICPSO	5.07E+00(6.96E-01)	**4.67E+00(3.12E-01)**	1.20E+03(1.93E+02)	**1.11E+03(1.55E+02)**
IPSO	**9.33E-01(2.09E-01)**	1.63E+01(1.09E+00)	**2.74E+02(5.26E+01)**	7.72E+03(1.37E+03)
PPSO	1.73E+01(1.09E+00)	1.93E+01(1.68E+00)	7.93E+03(1.10E+03)	8.40E+03(1.03E+03)
VPSO	3.83E+01(1.55E+00)	3.08E+01(1.10E+00)	1.80E+04(1.61E+03)	2.42E+04(1.43E+03)
RAN	2.36E+01(9.29E-01)	2.32E+01(7.92E-01)	8.98E+03(6.56E+02)	9.62E+03(7.20E+02)

	Sphere	Shuffled Sphere
AMPSO	2.74E+01(3.95E+00)	2.48E+01(2.45E+00)
BGPSO	4.17E+01(1.39E+00)	2.36E+01(1.15E+00)
BPSO	4.79E+01(1.38E+00)	2.60E+01(8.02E-01)
ICPSO	4.07E+00(4.52E-01)	**5.17E+00(6.49E-01)**
IPSO	**9.33E-01(2.03E-01)**	1.36E+01(1.06E+00)
PPSO	1.89E+01(1.20E+00)	1.64E+01(1.13E+00)
VPSO	3.58E+01(1.01E+00)	2.16E+01(7.86E-01)
RAN	2.26E+01(8.17E-01)	2.32E+01(1.17E+00)

Table 5.3: Results of discrete PSO algorithms maximizing NK landscapes.

D	AMPSO	BPSO	BGPSO	ICPSO	IPSO	PPSO	VPSO	RAN
2	7.01(0.02)	**7.35(0.01)**	**7.36(0.01)**	7.23(0.02)	6.73(0.02)	7.13(0.02)	**7.33(0.01)**	7.22(0.01)
4	7.43(0.02)	7.82(0.01)	7.81(0.01)	**8.07(0.01)**	7.42(0.02)	7.71(0.01)	7.57(0.01)	7.58(0.01)
6	7.44(0.02)	7.78(0.01)	7.77(0.01)	**8.14(0.01)**	7.61(0.02)	7.77(0.01)	7.66(0.01)	7.62(0.01)
8	7.55(0.02)	7.71(0.01)	7.69(0.01)	**8.09(0.01)**	7.66(0.02)	7.75(0.01)	7.64(0.01)	7.65(0.01)
10	7.47(0.02)	7.79(0.01)	7.78(0.01)	**8.11(0.01)**	7.66(0.02)	7.79(0.01)	7.65(0.01)	7.60(0.01)
15	7.47(0.02)	7.64(0.01)	7.66(0.01)	**8.02(0.01)**	7.73(0.02)	7.73(0.01)	7.63(0.01)	7.63(0.01)
20	7.47(0.02)	7.71(0.01)	7.69(0.01)	**7.98(0.02)**	7.77(0.01)	7.71(0.01)	7.65(0.01)	7.64(0.01)

Table 5.4: Results of discrete PSO algorithms performing abductive inference on Bayesian networks.

	Hailfinder	Hepar2	Insurance	Win95pts
AMPSO	-2.43E+03(2.14E+02)	-6.67E+01(1.05E+00)	-7.21E+01(2.44E+01)	-1.47E+03(1.72E+02)
BPSO	-2.01E+03(8.44E+01)	-6.64E+01(6.49E−01)	-4.67E+01(1.96E+00)	-1.37E+03(9.42E+01)
BGPSO	-2.06E+03(9.59E+01)	-6.72E+01(7.52E−01)	-4.49E+01(2.09E+00)	-1.51E+03(8.87E+01)
ICPSO	**-8.12E+02(1.39E+02)**	**-3.11E+01(1.12E+00)**	**-2.47E+01(1.91E+00)**	**-2.53E+02(7.89E+01)**
IPSO	-4.11E+03(1.62E+02)	-8.24E+01(1.31E+00)	-1.46E+03(1.44E+02)	-3.16E+03(1.31E+02)
PPSO	-1.66E+03(1.62E+02)	-5.76E+01(1.47E+00)	-4.07E+01(1.73E+00)	-8.73E+02(1.09E+02)
VPSO	-2.98E+03(1.09E+02)	-6.89E+01(6.48E−01)	-5.24E+02(6.60E+01)	-1.24E+03(7.71E+01)
RAN	-2.28E+03(8.35E+01)	-6.47E+01(5.95E−01)	-1.01E+02(3.39E+01)	-1.34E+03(8.92E+01)

solutions. The next best-performing algorithm was ICPSO. Even though it is not shown in the table, ICPSO significantly outperformed all other algorithms except IPSO. AMPSO, BPSO, and BGPSO all had comparable performance. VPSO was comparable with AMPSO, BPSO, and BGPSO and PPSO generally outperformed VPSO.

For the shuffled problems, ICPSO statistically performed the best, with IPSO as the runner-up. The rest of the algorithms have roughly the same performance, and are function-dependent as to which method performs the best.

Comparing each algorithm across the regular and shuffled problems, we found that ICPSO had the smallest change in performance. In some cases the performance on the unshuffled problems was better than the shuffled; however, this was not always the case. Meanwhile, IPSO consistently had poorer performance on the shuffled

problems. AMPSO usually performed better on the unshuffled problems, except on the Rastrigin function. BPSO and BGPSO both performed better on the shuffled functions than unshuffled.

Table 5.3 contains the results of the PSO variants on maximizing NK landscapes. The far left column indicates the number of states per dimension. On the NK landscapes, ICPSO almost always demonstrated the best performance significantly, and was only outperformed on binary strings ($D = 2$). In the binary case, BPSO, BGPSO, and VPSO significantly performed the best. Many of the other algorithms, such as AMPSO, BPSO, and BGPSO only varied slightly for $D = 4$ to 20. However, ICPSO had among the highest variance in performance between different D values.

Finally, the results from abductive inference are shown in Table 5.4. The far left column gives the different algorithm while the column shows the different Bayesian networks. In all networks, ICPSO significantly outperformed all other approaches. The 2nd best performing algorithm was the PPSO, which significantly outperformed all other approaches on all networks. The worst performing algorithm was IPSO, which was outperformed by all other approaches by a significant amount.

The fitness curves for all of the PSO algorithms on the sphere and shuffled sphere problems are shown in Figures 5.1 and 5.2. Figure 5.3 shows the fitness curves of the PSO algorithms on the NK landscape problem with $D = 4$. For ease of reading, the results in each case have been split between two graphs with the same scale. The X-axis is the number of iterations, while the Y axis is the **gBest** fitness averaged over 30 runs. For these experiments, we ran all algorithms for 200 iterations.

5.5.3 Analysis

Based on our results, ICPSO is generally more robust than the other approaches, as demonstrated by its consistent performance on the shuffled and unshuffled

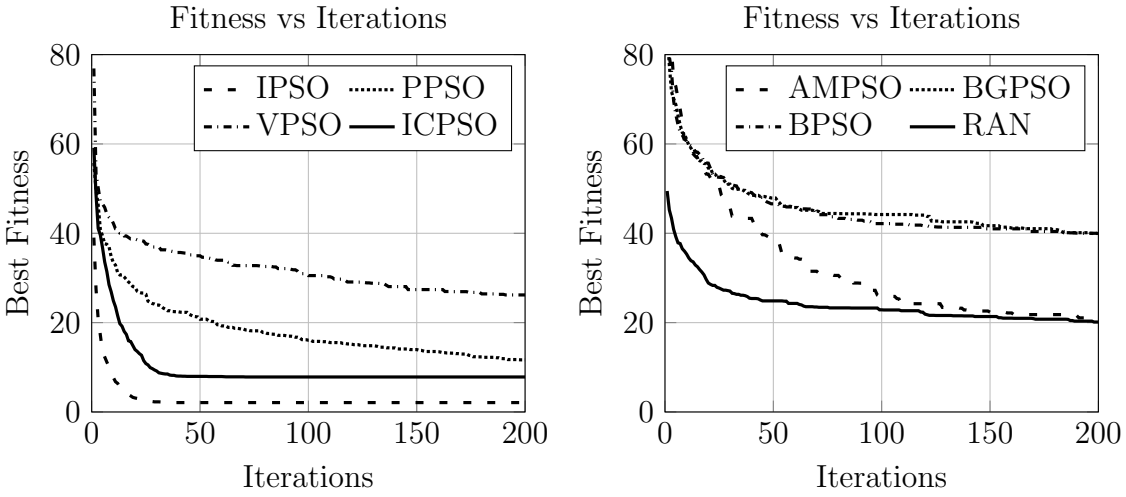


Figure 5.1: Fitness curves for minimizing the discrete sphere function.

functions. We believe this is because the particles represent distributions instead of candidate solutions, and thus do not rely on having a gradient or relationship between neighboring states.

While PPSO uses a very similar particle representation to ours, ICPSO always outperformed PPSO. We believe that this is due in part to how we set and bias the local and global bests. ICPSO uses the knowledge that if a particular sample has high fitness, more exploration should likely be performed around the sample. Another benefit to our approach is that we avoid the added complexity that PPSO incurs due to the approximate methods it requires to shift position values. ICPSO instead treats each variable as a probability distribution and normalizes after the position update.

As the number of states varied in NK landscapes, ICPSO had the highest variance in terms of its performance. This could be because it uses samples to set the local and global best vectors, which may make ICPSO more sensitive to bias associated with sampling. Additionally, this could be caused by under-sampling the distribution.

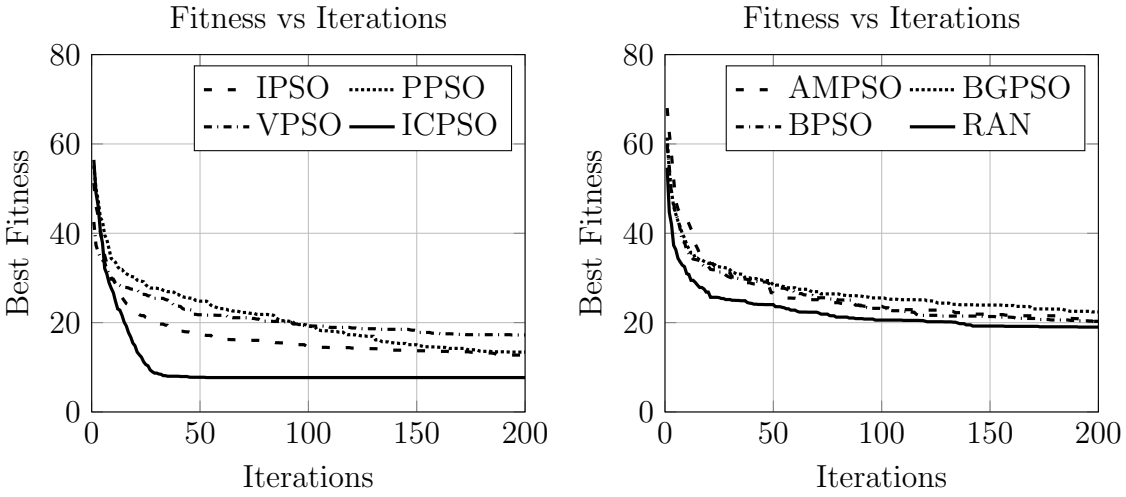


Figure 5.2: Fitness curves for minimizing the discrete sphere function.

Results also suggest that IPSO performs the best if a gradient exists between the states for a variable. If no such gradient exists, then IPSO suffers the largest drop in performance. This is demonstrated in the abductive inference results, where ICPSO significantly outperformed all other approaches.

The fitness curves for the sphere problems show that ICPSO has a steep initial curve. However, there appear to be decreasing returns to fitness after about 100 iterations. IPSO has a similar trend on the unshuffled sphere problem shown in the left graph in Figure 5.1. Some other approaches, such as PPSO and VPSO, converge at a comparatively slow pace. On more difficult problems, such as the NK landscapes in Figure 5.2, ICPSO has a more gradual fitness curve. However, ICPSO still converges faster than the competing approaches. Additionally, Figure 5.2 demonstrates that IPSO quickly converges to a poor solution. Again, these results demonstrate that in general, regular PSO requires a gradient between states in a variable in order to perform well.

Analysis of the fitness curves in Figures 5.1, 5.2, and 5.3 suggests that ICPSO might also be useful in applications where only a limited number of iterations may be

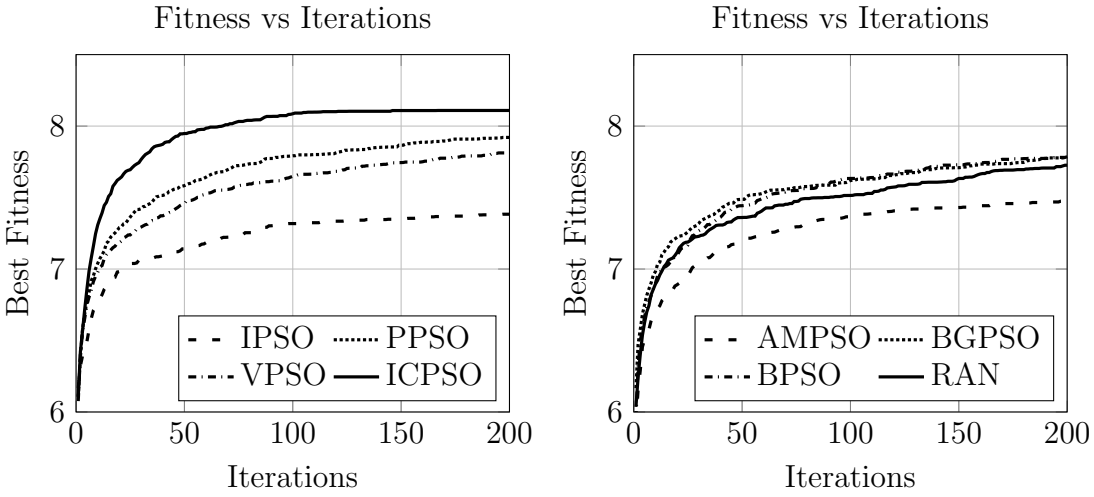


Figure 5.3: Fitness curves discrete PSO algorithms maximizing NK landscapes functions where $N = 10$, $K = 2$, and $D = 4$.

run. Since ICPSO generally has a sharp initial fitness gain and levels out within relatively few iterations, it can return a good solution sooner than many of the competing algorithms.

To further support our hypothesis that ICPSO is able to efficiently explore in the modified search space better than other discrete PSO algorithms, we used Multi-Dimensional Scaling (MDS) to visualize all the individuals in the PSO algorithms in 2 dimensions. MDS is method for reducing the dimensionality of data that attempts to preserve the distance between data points in a reduced [56, 57]. Let $\Delta \in \mathbb{R}^{M \times M}$ be a symmetric similarity matrix between M data points where each point $X_i \in \mathbb{R}^N$ Each entry $\delta_{i,j}$ in Δ represents the distance between two individuals. The goal of MDS is to find a point $X_i \in \mathbb{R}^D$ such that $Dist(X_i - X_j) = \delta_{i,j}$, preserving the distance between the points in the new space. In most cases, MDS sets D equal to 2. Here, we use the Pivot MDS method [7].

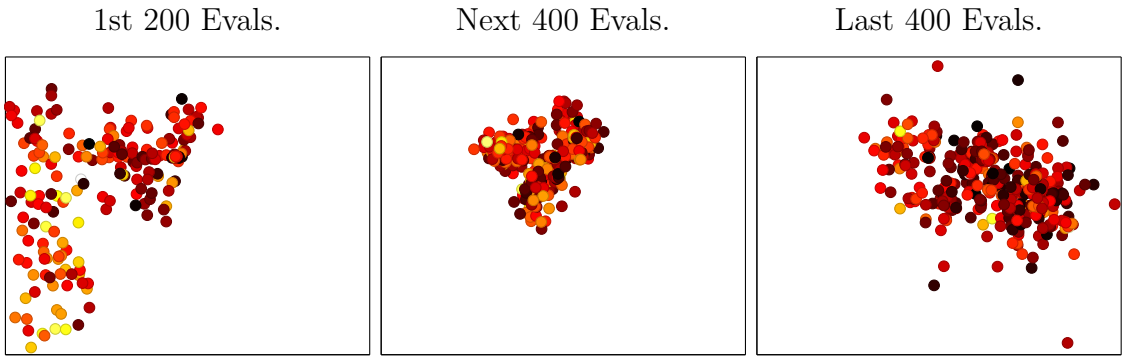


Figure 5.4: Individuals from ICPSO over time optimizing discrete sphere visualized in two dimensions using MDS.

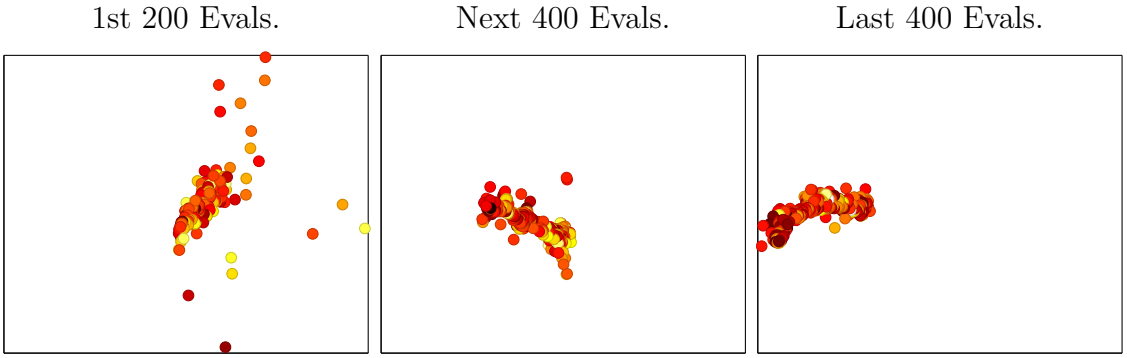


Figure 5.5: Individuals from PPSO over time optimizing discrete sphere visualized in two dimensions using MDS.

For ICPSO, we use Symmetric KL Divergence to measure the distance between individuals.

$$D_{SKL}(P||Q) = D_{KL}(P||Q) + D_{KL}(Q||P)$$

where

$$D_{KL}(P||Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)}.$$

For the other algorithms, we used the Euclidean distance, which is defined as

$$D(X, Y) = \sqrt{\sum_i (X_i - Y_i)^2}$$

The Euclidean metric was chosen for the other discrete PSO algorithms, a gradient is assumed between neighboring states. However, because ICPSO uses a distribution to represent a state, we used the KL divergence because it allows for a better measurement of the differences of two distributions.

We restrict our analysis to BPSO, ICPSO, IPSO, PPSO, VPSO, and RAN and leave out BGPSO and AMPSO since the performance of those algorithms was comparable to BPSO. Figure 5.4 shows the individuals in ICPSO optimizing the discrete sphere over three discretized time slices corresponding to the first 200 evaluations, the next 400 evaluations, and the last 400 evaluations. This allows us to visualize how individuals in ICPSO move throughout time. We present a similar figure for PPSO in Figure 5.5. Both figures are over a single trial. Additionally, darker colors represent better fitness while lighter colors have worse fitness.

The ICPSO algorithm first has all the individuals generated randomly throughout the space. However, in the next 400 evaluations, one can see how the individuals begin to converge to a central space. In the last 400 evaluations, the individuals continue to move towards the right while also continuing to spread apart slowly. In PPSO, we see how the randomly initialized individuals are spread throughout the space, similar to ICPSO. However, the points begin to converge to a tighter region and continue to do so during the last 400 evaluations. Additionally, in both algorithms, one can also see that as the individuals move throughout the space, the fitness gradually improves by the darkening of the points. However, in PPSO, the improvement in color is less than that in ICPSO.

We present the results of all individuals in BPSO, ICPSO, IPSO, PPSO, VPSO, and RAN optimizing the discrete sphere in Figure 5.6. All individuals are from a single trial where each algorithm was run for 1000 iterations. Darker colors represent more fit individuals while lighter colors correspond to individuals with worse fitness.

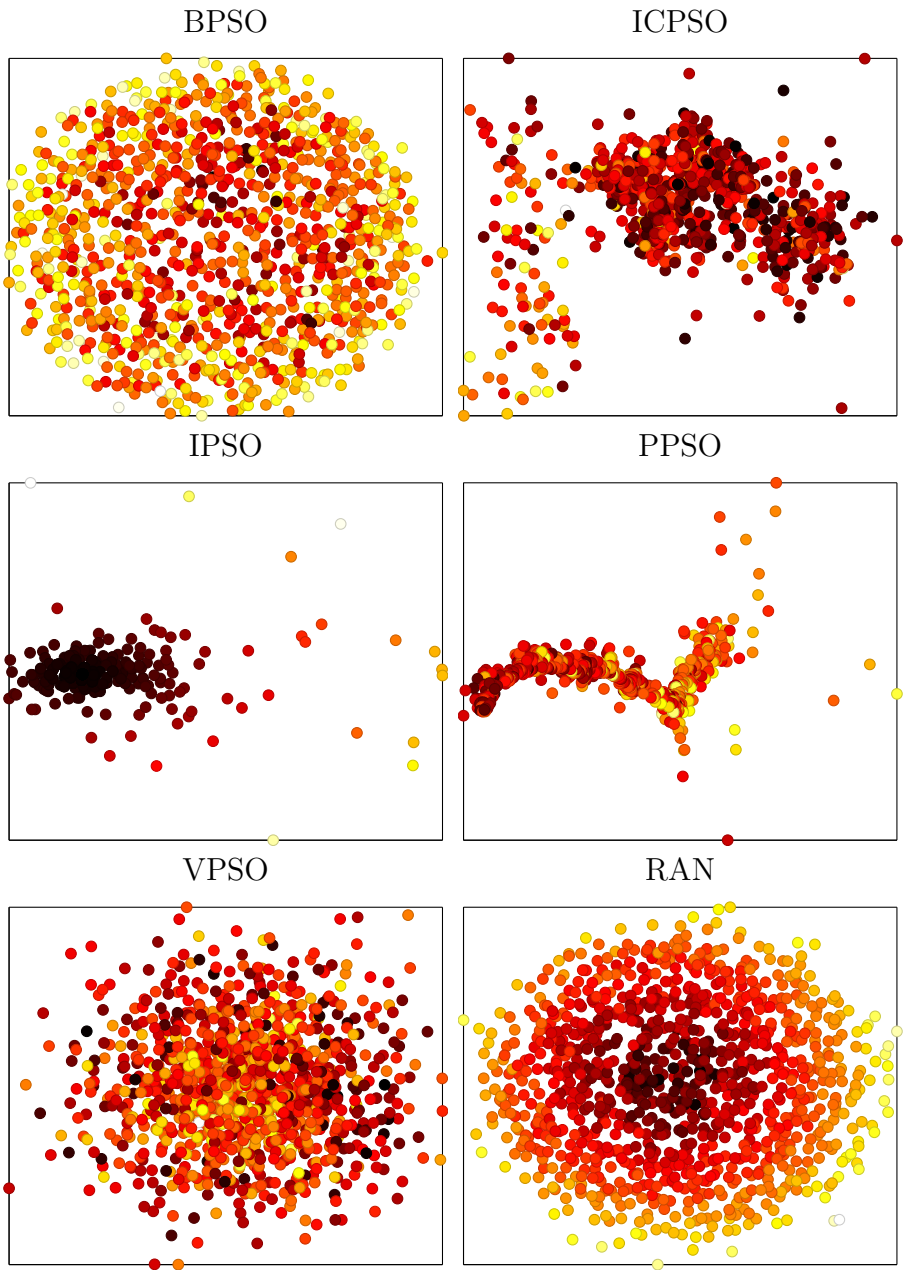


Figure 5.6: Individuals from discrete PSO algorithms optimizing discrete sphere visualized in two dimensions using MDS.

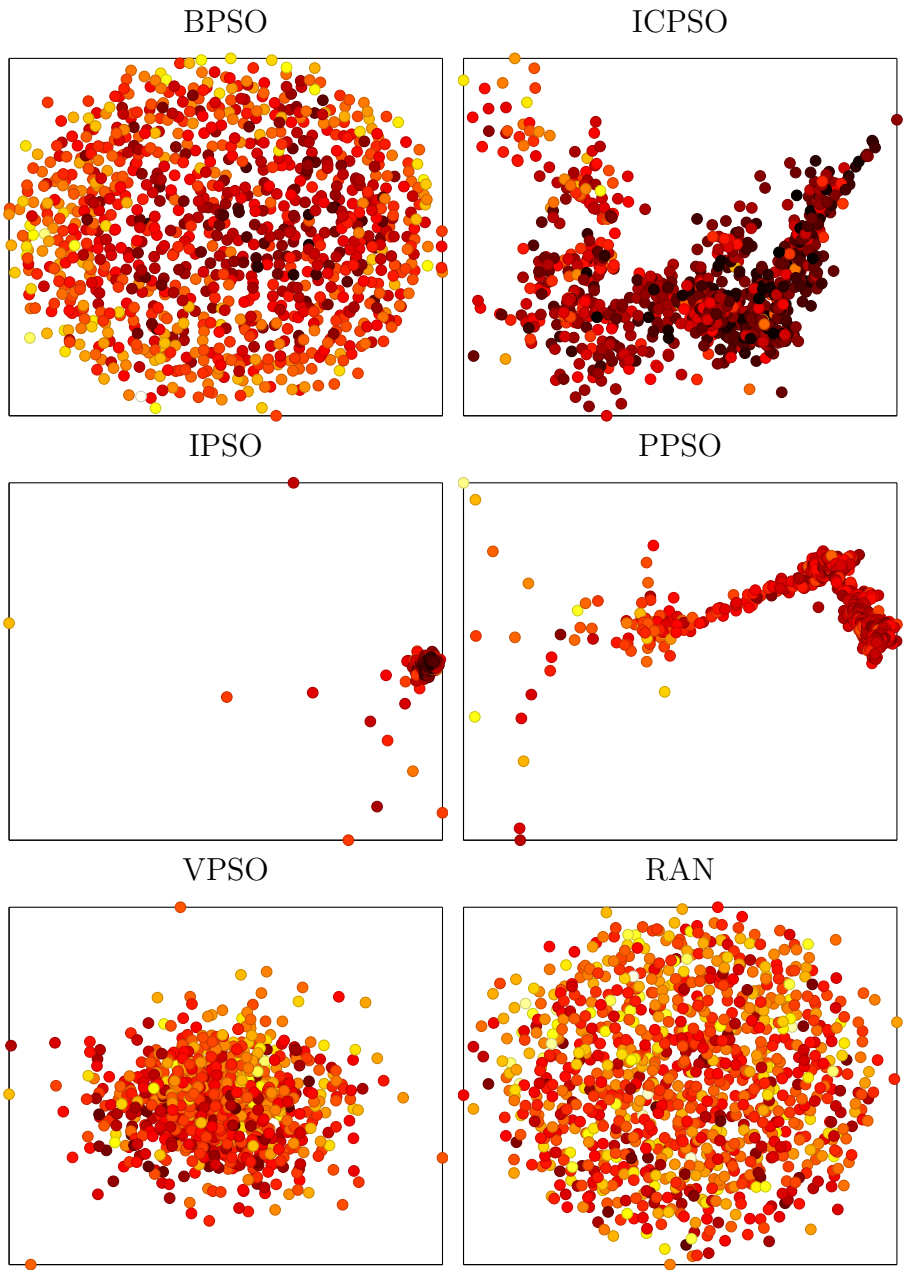


Figure 5.7: Individuals from discrete PSO algorithms optimizing shuffled discrete sphere visualized in two dimensions using MDS.

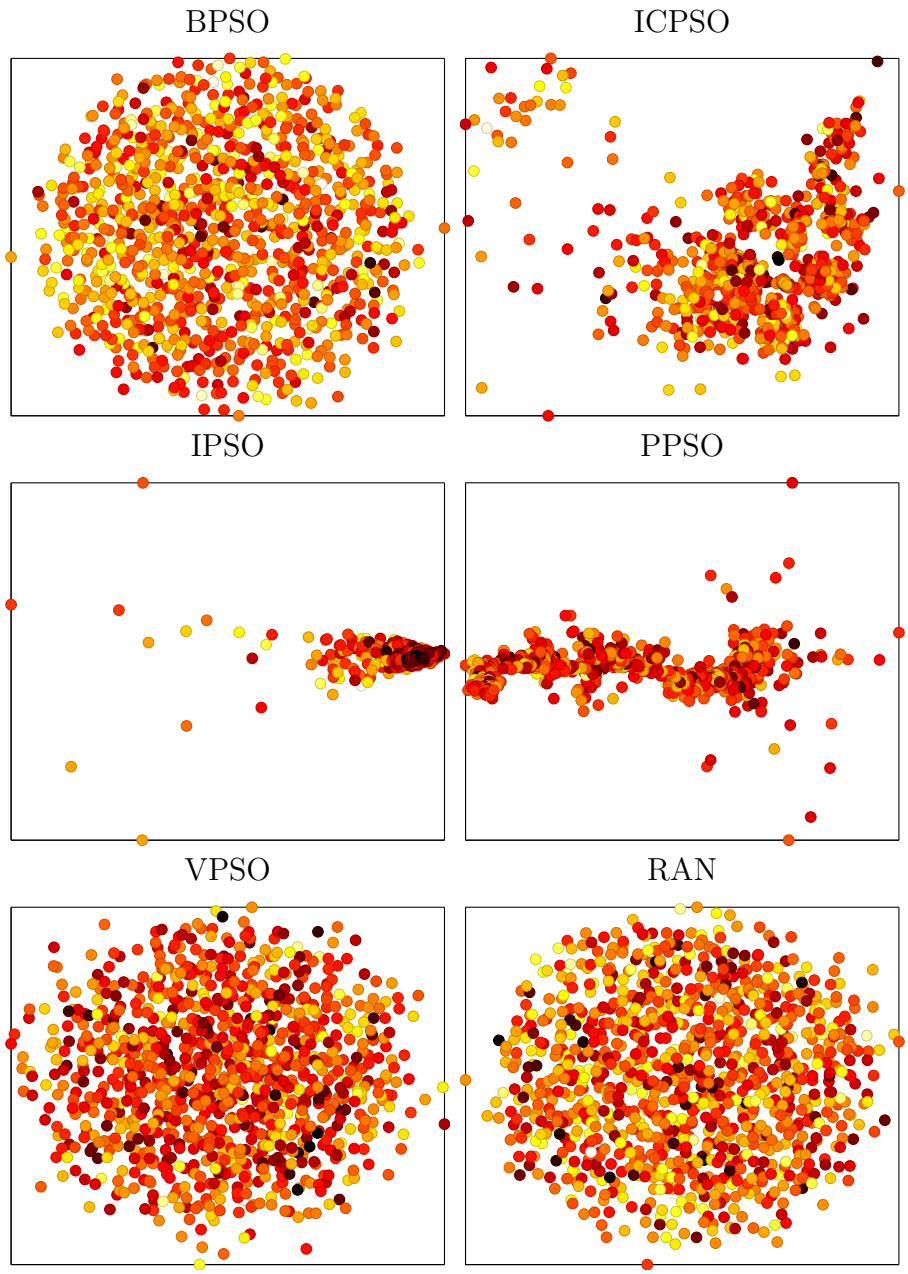


Figure 5.8: Individuals from the discrete PSO algorithms optimizing NK landscapes with $N = 10$, $K = 2$, and $D = 4$ visualized in two dimensions using MDS.

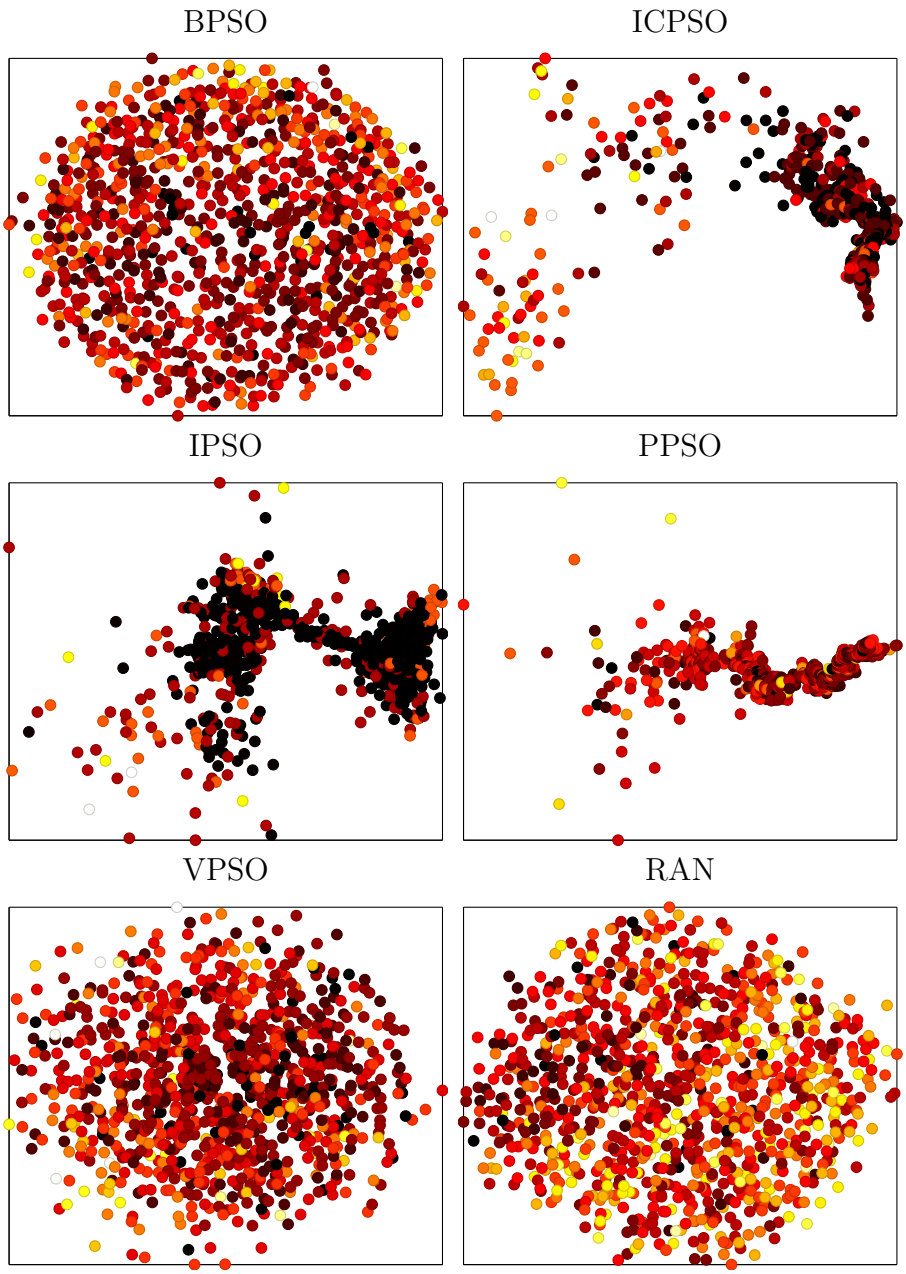


Figure 5.9: Individuals from discrete PSO algorithms performing abductive inference on the Insurance Bayesian network visualized in two dimensions using MDS.

The first thing we notice is how the BPSO, VPSO, and RAN algorithms all have similar structure, that is, the points are randomly distributed throughout the space. One interesting discovery is that the MDS of the individuals in the RAN version is able to reduce the ten dimensional sphere problem into a two dimensional view of the same problem. However, BPSO and VPSO are unable to capture this same relationship. Additionally, none of these algorithms exhibit any structure in their movement throughout the search space.

ICPSO, IPSO, and PPSO on the other hand, have structure to how the individuals move in the space. While the individuals are initialized randomly, they eventually begin to converge to a region in the space. Additionally, this convergence corresponds to a set of points that have high fitness.

Results from performing MDS on individuals of the discrete PSO algorithms on the shuffled discrete sphere problem show a similar phenomenon in Figure 5.7. However, the largest difference is that the structure of the sphere shown in the RAN algorithm is now lost due to the shuffling of the space. Additionally, VPSO shows a tighter clustering of individuals. ICPSO shows similar behavior on the shuffled discrete sphere during the beginning and middle iterations, but has a tighter grouping of the individuals towards the end.

We also present figures of individuals from the different PSO algorithms visualized in 2-dimensions using MDS on NK landscapes where $N = 10$, $K = 2$, and $D = 4$ in Figure 5.8 and the Insurance Bayesian network in Figure 5.9. The MDS figures further demonstrate that only three of the discrete PSO algorithms contain structure to their search process: IPSO, PPSO, and ICPSO. The similarity between the ICPSO and PPSO algorithms can also be seen in these figures as both algorithms show all individuals converging towards a single area. However, based on the results comparing the performance of ICPSO and PPSO, we see that PPSO is not as effective

as ICPSO in locating good solutions. This suggests that while PPSO has similarities to ICPSO, the sampling process in PPSO is not as effective as ICPSO's sampling procedure. Finally, we note that IPSO still contains structure in its exploration of the search space even if there does not exist a gradient between states in a variable.

5.6 FEA with ICPSO

In the previous sections, we proposed a new discrete PSO algorithm called ICPSO and demonstrated its performance over other algorithms on various discrete optimization problems. However, one question is how do FEA versions of the different discrete PSO algorithms perform. Here, we test to see whether ICPSO still outperforms the competing methods. We begin by discussing different ways to use the different discrete PSO algorithms in FEA.

In the previous chapters, all of the optimization algorithms used by FEA, such as GA, VPSO, HC, and DE, maintained the same representation as the underlying problem. However, several of the the discrete PSO algorithms discussed use a different representation of the underlying optimization problem. For example, BPSO used a binary representation of the problem while ICPSO used a probabilistic representation. This raises the question of show to use the various discrete PSO algorithms in FEA.

Here, we discuss two ways to apply the discrete PSO algorithms to FEA. One possible way is first to map the problem to the alternative representation and then create factors on the new problem representation. An alternative method would be to keep the problem representation the same and allow each factor to generate its own representation of its variables.

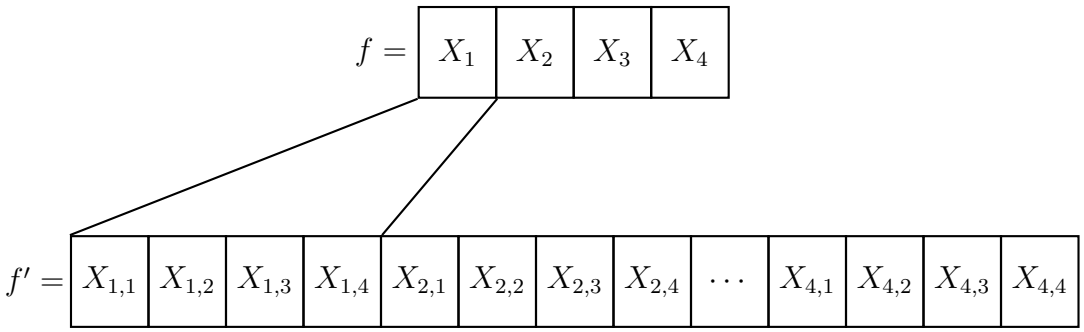


Figure 5.10: Example of factoring a function f by first mapping the function to an alternative representation.

5.6.1 Generate Mapping Before Factoring

The first method we discuss is to generate the mapping of the function to the alternative representation before applying FEA to the modified function. Formally, we can represent this as follows. Assume we are given a $f : \mathbf{D}^N \to \mathbb{R}$ with domain $\mathbf{D}^N$ to be optimized with parameters $\mathbf{X} = \langle X_1, X_2, \ldots, X_N \rangle$. $\mathbf{D} = \langle D_1, D_2, \ldots, D_N \rangle$ and D_i defines a domain for each variable. In the discrete settings, each domain D_i is a set of discrete states. For algorithms like BPSO and ICPSO, a new function $f' : \mathbf{D'}^M \to \mathbb{R}$ is generated with domain $\mathbf{D}^M$ to be optimized with parameters $\mathbf{X'} = \langle X'_1, X'_2, \ldots, X'_M \rangle$. The transformed function requires a mapping g such that $g(\mathbf{X'}) = \mathbf{X}$. $\mathbf{D'}$ is the modified domain $\langle D'_1, D'_2, \ldots, D'_M \rangle$. We can also represent the new modified inputs as $\mathbf{X'} = \langle X_{1,1}, X_{1,2}, \ldots, X_{1,K}, X_{2,1}, X_{2,2}, \ldots, X_{N,K} \rangle$ where $X_{1,1}, X_{1,2}, \ldots, X_{1,K}$ represents the encoding for variable X_1.

This new function is dependent on the the underlying algorithm and the required representation. For example, if BPSO is being used to optimize the function f, the new function f' has binary variables and g maps the binary variables (bits) to integer values. In this example, X_1 is mapped into $\log(|D_1|)$ variables. But in the ICPSO algorithm, f''s variables are probabilities and are mapped into $|D_1|$ variables, which

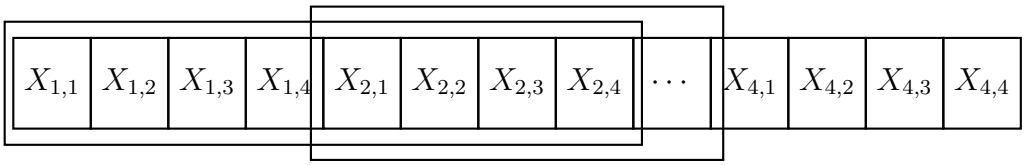


Figure 5.11: Example of creating factors on a transformed function with an alternative representation.

means that g is a function that samples the input to generate a vector of integer values.

A diagram demonstrating the various mappings is shown in Figure 5.10. The top four variables X_1, X_2, X_3, X_4 represent variables from the original problem while the bottom set of variables are the input to the modified problem. The figure demonstrates the mapping of variables in the original function to the new function. In factoring f', the factors are competing over the transformed variables. Assume that all variables have four states $D_1 = V, X, X, Z$. If applying ICPSO, the transformed function f' would have 16 variables where each variable represents the probability of a variable in f' assuming a particular state. We represent the transformed function for ICPSO in Figure 5.11 as $X_{1,1}$ as the probability of variable one state one, V. Given the new transformed function, FEA would generate factors that optimize the probabilities on f'. Figure 5.11 also demonstrates how two overlapping factors compete over the values $X_{2,1}$, $X_{2,2}$, $X_{2,3}$, and $X_{2,4}$.

One of the benefits to this approach is that it allows for competition to be performed over the transformed variables. For example, in ICPSO, competition will determine the probability of variable X_i generating state j. This allows factors to share information between one another on the transformed variables.

However, the drawback to this approach is that it increases the complexity in the competition. Because there are more variables in the transformed space, there are

now more variables that the competition algorithm has to iterate over. Additionally, the competition algorithm may require additional checks to ensure that the new set of values for the variables that correspond to the variable X_i remain in its domain. In ICPSO, this corresponds to ensuring that after each competition, the probability distribution remains valid.

5.6.2 Generate Mappings After Factoring

An alternative method is to allow each of the algorithms to generate the mappings after factors for FEA have been generated. This allows each algorithm to generate its mappings independently of all other factors. From each factors view, all factors are optimizing the function f.

One of the benefits to this approach is that it allows factors to use different types of algorithms. For example, one factor could use a BPSO while another factor could ICPSO. Additionally, competition is performed over the original set of variables $\boldsymbol{X}$, which does not introduce additional complexity or require extra checks to ensure values after competition are valid. However, a drawback to this approach is that it does not allow for a competition between factors on the alternative representation. In ICPSO, generating an alternative representation before factoring allows factors to share and compete over a finer granularity of solutions.

For this set of experiments, we generate the mappings after factoring the functions. There are two reasons for this choice. The first is that creating the alternative mapping after generating the factors requires no adaptation of the FEA compete algorithm. Second, it allows for combining different discrete PSO algorithms, which we discuss as an area of future work.

5.7 Discrete PSO FEA Experiments

In our experiments, we compared FEA using ICPSO with FEA using Binary PSO (BPSO), Integer PSO (IPSO), the PSO proposed by Pugh and Martinoli (PPSO), and the PSO proposed by Veeramachaneni *et al.* (VPSO). Additionally, we tested an FEA version of the Random PSO (RAN) algorithm discussed in the previous section.

Similar to the previous experiments in this chapter, we tested the algorithms on generalized NK landscapes where each variable can assume D states. For our experiments, we used an NK-landscapes with $N = 10, 20$, $K = 2, 4$, and $D = 2, 4, 8$. For each set of NK landscape parameters, we generated 30 different landscapes and ran each algorithm 30 times per landscape.

In addition to testing the algorithms on the NK landscapes, we tested the discrete PSO algorithms on abductive inference in Bayesian networks. We added four different networks (Alarm, Andes, Child, and Water) to the networks used in the previous section (Hailfinder, Hepar2, Insurance, and Win95pts), giving a total of eight different networks to evaluate the different algorithms. Network details are shown Table 4.7. For fitness evaluations, used the loglikelihood and for each network ran each algorithm 30 times per landscape.

Each FEA used factors with a swarm of size five and terminated once the full global solution did not change after ten iterations. Compete and Share were performed every five iterations. We used the Markov blanket factor architecture, and each factor had a population size of five.

All PSO variants used the same set of parameters, in order to make comparisons as consistent as possible. Parameters ϕ_1 and ϕ_2 were set to 1.49618, and $\omega = 0.729$, which has been found to encourage convergent trajectories [26]. This is due to the recommendations of Engelbrecht, which demonstrated that a large swarm,

counterintuitively, may have difficulty exploring the search space [28]. For our approach, we set the scaling factor ϵ to 0.75, and in the VPSO we set σ to the authors' recommended value of 0.2. All algorithms randomly initialized velocity and position vectors. Bold values indicate one algorithm was significantly better than all other algorithms. Two or more bold value indicate that both algorithms outperformed all other algorithms but were not significantly different with each other. If there are no bolded values, than there were more than three algorithms that were significantly equivalent. Significance testing was done using a Paired Student t-Test with $\alpha = 0.05$.

5.7.1 Results

Table 5.5 presents the average fitness for NK landscapes. Standard error is shown in parenthesis. On problems with low epistasis and only two states per variables, there was almost no significant difference between the algorithms. For example, on the $N = 10$, $K = 1$, $D = 2$, there was no significant difference between BPSO-FEA, ICPSO-FEA, PPSO-FEA, and VPSO-FEA. However, IPSO-FEA and RAN-FEA were were both significantly outperformed by all other algorithms.

For problems with high epistasis ($K = 4$), the best FEA discrete PSO algorithm was ICPSO-FEA. Of note, while ICPSO-FEA had a better average fitness than BPSO-FEA on $N = 20$, $K = 4$, $D = 8$, there was no significant difference. Overall, the best performing algorithm was the BPSO-FEA algorithm, as it was significantly better than a majority of the algorithms on six out of the twelve problems. The worst performing algorithm was IPSO-FEA, which had the worst fitness on nine out of the twelve problems. On the remaining three problems, PPSO-FEA performed the worst.

Results for FEA discrete PSO algorithms on abductive inference are shown in Table 5.6. The first thing we note is that there were only two networks in which there was a clear significant difference between algorithms: Child and Water. Of those two

Table 5.5: Discrete PSO FEAs maximizing NK landscapes.

N	K	D	BPSO-FEA	ICPSO-FEA	IPSO-FEA	PPSO-FEA	VPSO-FEA	RAN-FEA
10	2	2	7.44(0.02)	7.41(0.03)	7.19(0.03)	7.43(0.02)	7.43(0.02)	7.25(0.03)
		4	**8.66(0.02)**	8.52(0.02)	8.14(0.02)	8.39(0.02)	8.57(0.02)	8.59(0.02)
		8	9.07(0.01)	8.97(0.01)	8.46(0.02)	8.35(0.02)	9.02(0.01)	9.06(0.01)
	4	2	7.56(0.02)	7.50(0.02)	7.27(0.03)	7.48(0.02)	7.56(0.02)	7.60(0.02)
		4	**8.39(0.01)**	8.33(0.02)	8.02(0.02)	7.56(0.03)	8.29(0.01)	**8.39(0.01)**
		8	8.44(0.01)	**8.63(0.01)**	8.26(0.02)	7.65(0.03)	8.47(0.01)	8.40(0.01)
20	2	2	14.57(0.04)	14.51(0.04)	14.00(0.05)	14.56(0.04)	14.56(0.04)	14.69(0.03)
		4	**17.19(0.02)**	16.91(0.02)	16.15(0.03)	16.91(0.02)	16.97(0.02)	**17.13(0.02)**
		8	**18.20(0.02)**	17.98(0.02)	16.83(0.02)	17.51(0.03)	18.11(0.02)	**18.17(0.02)**
	4	2	15.34(0.03)	15.16(0.03)	14.45(0.04)	15.25(0.03)	15.30(0.03)	15.18(0.03)
		4	**16.97(0.02)**	16.76(0.02)	15.84(0.03)	15.93(0.03)	16.72(0.02)	**16.97(0.02)**
		8	**17.35(0.02)**	**17.44(0.02)**	16.40(0.02)	17.09(0.02)	17.14(0.02)	17.11(0.02)

networks, ICPSO-FEA and V-PSO were both significantly better than the majority of the other algorithms. Additionally, the RAN-FEA tied on the Child network.

On the other six networks, there was no significant difference between four or more of the algorithms. However, in all eight networks, IPSO-FEA performed the worst. While BPSO-FEA performed the best on three out of the eight networks, none of these differences were significant. RAN-FEA performed best on one network while RAN-FEA was the best on two networks, but none of the differences were significant.

In addition to the average fitness, we present the average number of iterations, number of evaluations used by all factors, and number of evaluations used by FEA's Compete. Table 5.7 displays the averages over every abductive inference problem. IPSO-FEA required the fewest iterations while PPSO-FEA required the most. Additionally, PPSO-FEA required the most fitness evaluations. The second best algorithm in terms of iterations and fitness evaluations performed was ICPSO-FEA. Finally, BPSO-FEA, VPSO-FEA, and RAN-FEA, all required a similar number of iterations and evaluations.

Table 5.6: Discrete PSO FEAs performing abductive inference on Bayesian networks.

	Alarm	Andes	Child	Hailfinder
BPSO-FEA	-8.04E+0(3.92E−1)	-6.04E+1(4.53E−1)	-8.17E+0(3.19E−1)	-3.37E+1(2.99E−1)
ICPSO-FEA	-9.61E+0(5.30E−1)	-6.64E+1(4.89E−1)	**-6.70E+0(3.06E−1)**	-3.39E+1(3.95E−1)
IPSO-FEA	-1.49E+1(8.37E−1)	-8.43E+1(1.15E+0)	-9.38E+0(3.43E−1)	-1.63E+2(5.16E+1)
PPSO-FEA	-1.09E+1(6.13E−1)	-6.24E+1(4.10E−1)	-7.93E+0(3.41E−1)	-3.52E+1(3.86E−1)
VPSO-FEA	-9.41E+0(5.01E−1)	-6.04E+1(5.85E−1)	**-6.27E+0(2.69E−1)**	-3.35E+1(2.82E−1)
RAN-FEA	-8.13E+0(5.58E−1)	-6.13E+1(5.31E−1)	**-6.78E+0(3.44E−1)**	-3.29E+1(4.06E−1)

	Hepar2	Insurance	Water	Win95pts
BPSO-FEA	-1.75E+1(2.78E−1)	-1.13E+1(3.61E−1)	-3.61E+2(7.75E+1)	-1.77E+1(7.63E−1)
ICPSO-FEA	-1.87E+1(4.63E−1)	-1.03E+1(4.49E−1)	**-1.09E+2(5.89E+1)**	-1.83E+1(6.01E−1)
IPSO-FEA	-1.88E+1(5.10E−1)	-1.34E+1(5.65E−1)	-6.32E+2(1.38E+2)	-8.65E+1(3.45E+1)
PPSO-FEA	-1.86E+1(3.66E−1)	-1.22E+1(4.26E−1)	-6.25E+2(1.03E+2)	-1.92E+1(7.62E−1)
VPSO-FEA	-1.75E+1(3.36E−1)	-1.06E+1(2.97E−1)	**-8.45E+1(4.14E+1)**	-1.66E+1(6.77E−1)
RAN-FEA	-1.79E+1(3.14E−1)	-1.04E+1(3.48E−1)	-4.60E+2(8.43E+1)	-1.66E+1(8.83E−1)

5.7.2 Analysis

Based on the previous results, we can draw several conclusions. The first is that the Compete function in FEA is able to locate good solutions even if the factors are random. In the previous section, we demonstrated that ICPSO significantly outperformed all other discrete PSO algorithms on both shuffled discrete benchmark functions, NK landscapes, and Bayesian networks. However, the discrete PSO FEA all had very similar performance. In very few instances was there an algorithm that was clearly better than the others. Additionally, in the few cases that an algorithm significantly outperformed another algorithm, the difference was less than the difference between the full versions of the algorithms.

While there was no clear algorithm that performed the best, IPSO-FEA performed the worst the majority of the time. We believe there are two reasons for this. The first is that IPSO lacks any structure to its exploration of the search space, which we demonstrated in the MDS analysis of the full single IPSO. Second, we believe that IPSO is more susceptible to becoming stuck in suboptimal solutions. If IPSO was able to escape suboptimal solutions, its performance, both in terms of fitness and

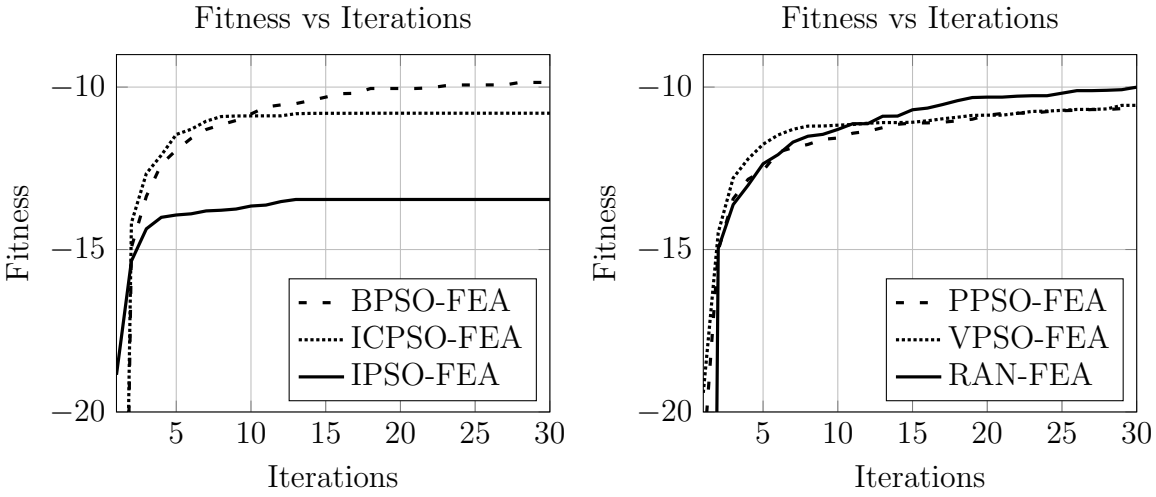


Figure 5.12: Fitness curves for discrete PSO FEAs performing abductive inference on the Insurance network.

number of iterations, would have been closer to that of RAN-FEA. Instead, it required the fewest number of iterations to converge, indicating the particles becoming stuck in suboptimal solutions.

Additionally, IPSO lacks any type of sampling process. BPSO-FEA, ICPSO-FEA, PPSO-FEA, and VPSO-FEA all require a sample to be generated from either the position or velocity vectors. This sampling procedure introduces another level of exploration into the algorithm, which may allow particles to escape local optimum.

To provide more insight into the performance of the different discrete PSO FEAs, we present fitness curves in Figures 5.12. Results are from abductive inference on the Insurance network. The x-axis is the fitness of the full global solution of FEA. BPSO-FEA, PPSO-FEA, and VPSO-FEA, and RAN-FEA all had similar fitness curves while IPSO quickly converged, but only to a poor solution. ICPSO-FEA had a similar fitness curve to that of VPSO-FEA, but failed to make improvements after 20 iterations. This indicates that ICPSO-FEA may suffer from premature convergence.

Table 5.7: Number of iterations and fitness evaluations for FEA versions of discrete PSO algorithms.

	Number Iterations	Number Factor Evals.	Number Compete Evals.
BPSO-FEA	142.38	66763.55	9509.81
ICPSO-FEA	125.58	55281.33	7858.88
IPSO-FEA	87.54	31909.94	4437.57
PPSO-FEA	260.13	241023.86	13185.97
VPSO-FEA	140.83	69475.48	9912.02
RAN-FEA	142.08	65719.96	9360.35

Similar to the previous section, we analyzed the movement of swarms using MDS. Figure 5.13 displays every position evaluated for fitness in two dimensions using MDS. We restrict our analysis to only two neighboring factors on the ICPSO, IPSO, and PPSO algorithms on the NK landscape problem with $N = 10$, $K = 2$, and $D = 2$. All individuals are from a single trial where each algorithm was run for 1000 iterations. Darker colors represent more fit individuals while lighter colors correspond to individuals with worse fitness.

From these figures, we see that the factors for ICPSO-FEA and PPSO-FEA still retain the structure in search that was found in the full single-population versions. Additionally, IPSO still lacks structure in its exploration of the search space. One interesting result is that while Factor 2 on ICPSO and PPSO exhibits structure, it appears that the swarm moves between two different locations in the search space. This is caused by a factor first moving towards a good set of solutions in its subspace. However, after Compete, the factors may be searching in a different subspace, causing the factors to move towards a different set of solutions in the subspace. Finally, we make note that IPSO still lacks structure to its exploration of the search space. Furthermore, while the second factors in ICPSO and PPSO appear to respond to the change in the full global solution after compete, IPSO does not.

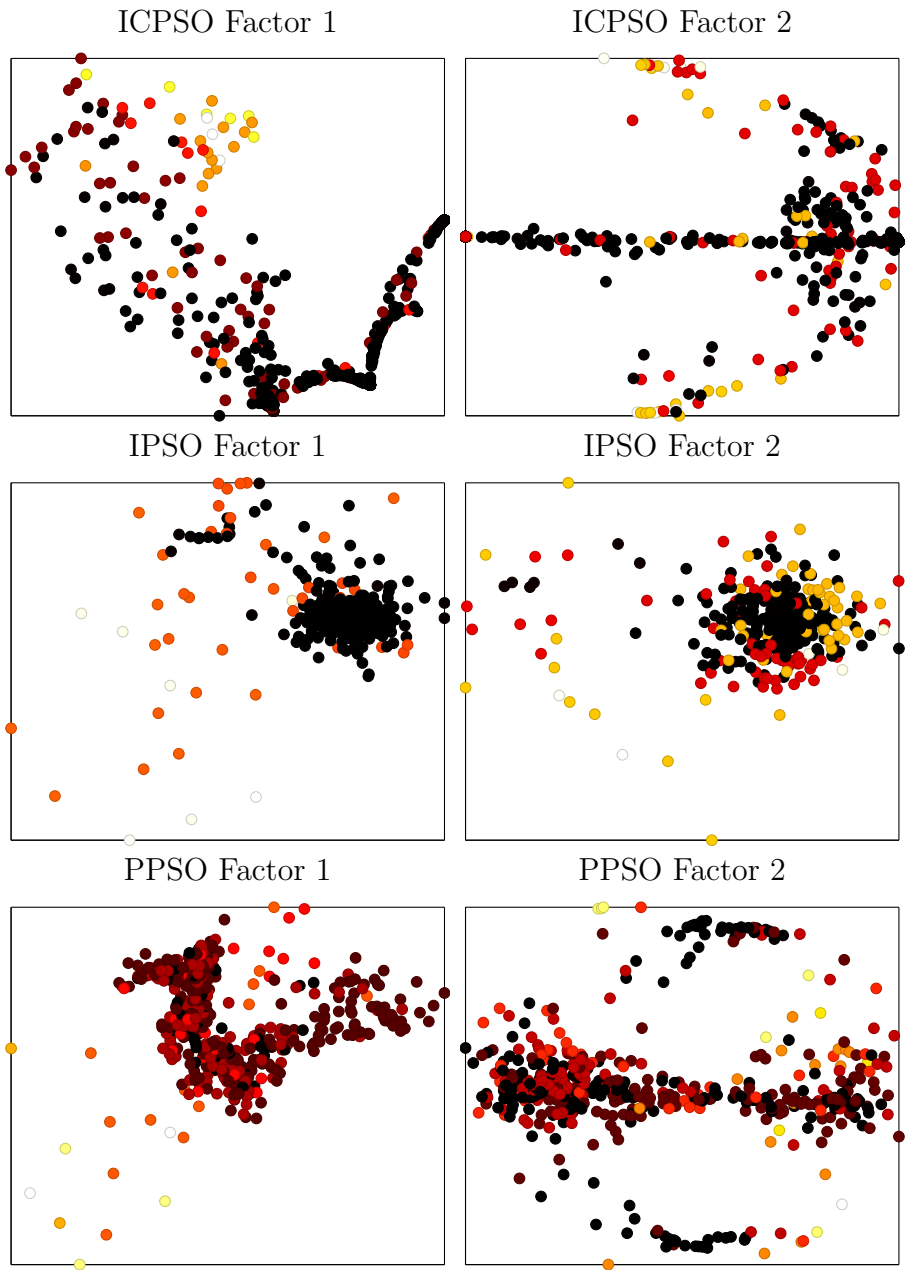


Figure 5.13: Individuals from PSO optimizing NK landscape visualized in two dimensions using MDS.

5.8 Conclusion

In this chapter, we proposed a new discrete PSO algorithm called Integer and Categorical PSO (ICPSO). We then evaluated ICPSO's performance on several different functions and discovered that on functions without a gradient, ICPSO outperforms all other approaches. However, on functions with a gradient, Integer PSO still outperformed discrete PSO algorithms. Next, we compared FEA versions of discrete PSO algorithms and discovered that FEA is able to mask the poor performance. This is because during Compete, the algorithm only the best solution during the entire search process is used. In later chapters, we further investigate the how the Compete step affects the performance of FEA.

CHAPTER SIX

CONVERGANCE OF FEA

One open question concerning FEA is its proof of convergence. Previous work with FEA has found that the full global solution $\mathbf{G}$ always converged to a single solution. However, no one has analyzed under what conditions convergence will occur. Here, we present work showing that the full global solution $\mathbf{G}$ in FEA will converge to a single solution if the individual factors also converge. Additionally, we show that FEA may converge to suboptimal solutions that are not local minima.

6.1 Convergence to Single Solutions

First, we will prove under which conditions the full global solution $\mathbf{G}$ converges to a single solution. Without loss of generality, assumes a minimization problem. We begin with some definitions.

Definition 6.1. Let Δ_i^t be the change in factor $\boldsymbol{\mathcal{P}}_i$'s best position at time t where $\boldsymbol{\mathcal{P}}_i^t = [s_{(i,1)}^t, s_{(i,2)}^t, \ldots, s_{(i,k)}^t]$ is the best position for factor $\boldsymbol{\mathcal{P}}_i^t$ at time t, and each position $s_{(i,j)}$ corresponds to a parameter in the function $f : \mathbf{D}^N \to \mathbb{R}$ with domain $\mathbf{D}^N$ and parameters $\mathbf{X}$. The change in the position for a single factor $\boldsymbol{\mathcal{P}}_i$ is denoted as

$$\Delta_i^t = d(\boldsymbol{\mathcal{P}}_i^{t-1}, \boldsymbol{\mathcal{P}}_i^t)$$

where K is the size of the factor and d is a metric function.

Definition 6.2. Let $df(\boldsymbol{\mathcal{P}}_i)^t$ be the change in fitness in factor $\boldsymbol{\mathcal{P}}_i$'s at time t, where

$$df(\boldsymbol{\mathcal{P}}_i)^t = f(\boldsymbol{\mathcal{P}}_i^{t-1}) - f(\boldsymbol{\mathcal{P}}_i^t).$$

Because $\boldsymbol{\mathcal{P}}_i$ is only updated if the fitness is strictly less than the previous fitness, we know $df(\boldsymbol{\mathcal{P}}_i)^t \geq 0$.

Definition 6.3. Let $\Delta^t_{\mathbf{G}}$ be the change in position for $\mathbf{G}$ where

$$\Delta^t_{\mathbf{G}} = d(\mathbf{G}^{t-1}, \mathbf{G}^t)$$

is a metric function. Additionally, $\mathbf{G}^t$ is equal to $[g^t_1, g^t_2, \ldots, g^t_N]$.

Definition 6.4. Let $df(\mathbf{G})^t$ be the change in fitness of $\mathbf{G}$ at time t where

$$df(\mathbf{G})^t = f(\mathbf{G}^{t-1}) - f(\mathbf{G}^t)$$

Definition 6.5. Let $\mathbf{D}_f$ represent the search space for the function f. Similarly, let $\mathbf{D}^t_{Comp}$ be the search space for the competition algorithm at time t. Note that $\mathbf{D}^t_{Comp}$ is a discrete and finite set of points and is given by the set of best positions from all factors,

$$\mathbf{D}^t_{Comp} = [D^t_1, D^t_2, \ldots, D^t_N]$$

where D^t_i is the set of values a variable X_i in $\mathbf{G}$ can assume at time t,

$$D^t_i = [s^t_{(1,i)}, s^t_{(2,i)}, \ldots, s^t_{(K,i)}]$$

and $s^t_{(1,i)}$ is the best position at time t for the first factor $\boldsymbol{\mathcal{P}}_1$ that optimizes parameter X_i.

Definition 6.6. Let $\mathbf{C}^t$ be the set of all points that $\mathbf{G}$ can assume at time t. The set $\mathbf{C}^t$ is determined by $\mathbf{D}^t_{Comp}$ and consists of

$$\mathbf{C}^t = \{D^t_1 \times D^t_2 \times \cdots \times D^t_N\},$$

where

$$C_i^t = [s_{(a,1)}^t, s_{(b,2)}^t, \ldots, s_{(z,N)}^t]$$

and $s_{(a,1)}^t$ is a value for variable X_1 from a factor a. We denote the size of $\mathbf{C}^t$ as L.

Definition 6.7. A factor $\boldsymbol{\mathcal{P}}_i$ converges when

$$\lim_{t\to\infty} \Delta_i^t = 0.$$

Similarly, $\mathbf{G}$ is said to have converged when

$$\lim_{t\to\infty} \Delta_{\mathbf{G}}^t = 0.$$

Lemma 6.1.1. Assume that at time $t-1$, $\mathbf{G}$ is at a local minimum in $\mathbf{D}_{Comp}^{t-1}$. If $df(\boldsymbol{\mathcal{P}}_i)^t = 0$ for all factors i, then $\Delta_{\mathbf{G}}^t = 0$ and $df(\mathbf{G})^t = 0$.

Proof. If $df(\boldsymbol{\mathcal{P}}_i)^t = 0$, then no factors were updated and $\Delta_i^t = 0$ for all i. This indicates that the search space remain unchanged from $t-1$ to t and $\mathbf{C}^{t-1} = \mathbf{C}^t$. Since $\mathbf{G}$ was at a local minimum in $\mathbf{C}^{t-1}$, $\mathbf{G}$ is also at a local minimum in $\mathbf{C}^t$. Because there are no single changes that the competition algorithm can make to improve the fitness of $\mathbf{G}$, $\Delta_{\mathbf{G}}^t = 0$. Finally, because $\mathbf{G}$ remains unchanged, so does its fitness $df(\mathbf{G})^t = 0$. □

The above lemma shows that if the full global solution is locally optimal and the search space $\mathbf{D}_{Comp}^t$ does not change, then there are no changes that the competition algorithm can make to improve the fitness of $\mathbf{G}$. This leads us to the next Theorem, which relates the convergence of factors to the convergence of $\mathbf{G}$.

Theorem 6.1.2. If $\mathbf{G}$ is at a local minimum in $\mathbf{D}_{Comp}^t$ and all factors have converged, then $\mathbf{G}$ has also converged.

Proof. By definition of convergence for a factor $\boldsymbol{\mathcal{P}}_i$,

$$d(\boldsymbol{\mathcal{P}}_i^{t-1}, \boldsymbol{\mathcal{P}}_i^t) = 0$$
$$\boldsymbol{\mathcal{P}}_i^{t-1} = \boldsymbol{\mathcal{P}}_i^t.$$

Therefore,

$$f(\boldsymbol{\mathcal{P}}_i^{t-1}) = f(\boldsymbol{\mathcal{P}}_i^t)$$
$$f(\boldsymbol{\mathcal{P}}_i^{t-1}) - f(\boldsymbol{\mathcal{P}}_i^t) = 0$$
$$df(\boldsymbol{\mathcal{P}}_i)^t = 0$$

for all factors i. By Lemma 6.1.1, $\mathbf{G}$ will not change and therefore has converged. □

The above theorem requires that all of the factors have already converged to guarantee the full global solution also converges. We relax the constraint in the next Theorem by only assuming that, at some point in time, the factors converge.

Theorem 6.1.3. If all the factors $\boldsymbol{\mathcal{P}}_i$ in FEA converge at some point in time during FEA's optimization of f, then the full global solution $\mathbf{G}$ will also converge.

Proof. By definition of convergence for each factor $\boldsymbol{\mathcal{P}}_i$,

$$\lim_{t\to\infty} \Delta_{\boldsymbol{\mathcal{P}}_i}^t = 0$$
$$\lim_{t\to\infty} d(\boldsymbol{\mathcal{P}}_i^{t-1}, \boldsymbol{\mathcal{P}}_i^t) = 0$$

Because d is a metric function,

$$\lim_{t\to\infty} {s_{i,j}}^{t-1} - {s_{i,j}}^{t} = 0$$

for all j in i. Let $\Delta_{\mathbf{C}}^{t}$ be the change for all points in $\mathbf{C}$ at time t,

$$\Delta_{\mathbf{C}}^{t} = \frac{\sum_{i=1}^{L} d(C_i^{t-1}, C_i^{t})}{L}$$

where C_i^t is the ith point in $\mathbf{C}$ at time t, and d is the difference. Because $C_{i,j}^{t-1}$ is factor k's value for variable j, we know that the sequence of values will converge

$$\lim_{t\to\infty} ({s_{k,j}}^{t-1} - {s_{k,j}}^{t}) = 0$$

$$\lim_{t\to\infty} ({s_{k,j}}^{t-1} = {s_{k,j}}^{t})$$

and therefore

$$\lim_{t\to\infty} d(C_i^{t-1}, C_i^{t}) = 0$$

for all factors i. Putting this together gives us

$$\lim_{t\to\infty} \frac{\sum_{i=1}^{M} d(\mathbf{C}_i^{t-1}, \mathbf{C}_i^{t})}{M} = 0$$

$$\lim_{t\to\infty} \Delta_C^t = 0$$

This shows that the search space $\mathbf{D}_{Comp}^{t}$ also converges to a set of discrete points. Remember that this is a discrete and finite set of points for the competition algorithm to explore. Eventually, the competition algorithm, which is a Greedy local search

algorithm, will either reach a local minimum or hit all points. Therefore, $\mathbf{G}$ will converge. □

The above theorem requires that all of the factors in FEA have converged. However, it is possible for the full global solution in FEA to converge even if not all of the factors have converged. Instead, we only require that $\mathbf{G}$ is a local minimum in all subsequent domains, which allows for some of the factors to not converge.

Theorem 6.1.4. If $\mathbf{G}$ is a local optimum in all spaces $\mathbf{D}^t_{Comp}$ $\forall t > t_0$, then $\mathbf{G}$ has converged.

Proof. Since $\mathbf{G}$ was at a local minimum in $\mathbf{C}^t$ for all future time past t_0, there are no single changes that the competition algorithm can make to improve the fitness of $\mathbf{G}$ during each FEA iteration. Therefore, $\Delta^t_{\mathbf{G}} = 0$, which by definition is the convergence of $\mathbf{G}$. □

6.2 Pseudominimum Convergence

The previous section showed that if the search space for the competition algorithm converges or if $\mathbf{G}$ is at a local optimum, then the full global solution $\mathbf{G}$ will converge to a single point. However, it is unknown to what kind of solution the full global solution will converge to. Additionally, while $\mathbf{G}$ may be located at a local minimum in $\mathbf{D}^t_{Comp}$, it may not be a local minimum in $\mathbf{D}^t_f$. The next section proves that $\mathbf{G}$ may become stuck in suboptimal points in the search space that are not locally minimal.

Definition 6.8. A *local minimum* is a point

$$\mathbf{x}^* = (x_1, x_2, \ldots, x_N)$$

such that, for some $\epsilon > 0$, $f(\mathbf{x}^*) < f(\mathbf{x})$ for all points within an ϵ of $\mathbf{x}^*$.

Definition 6.9. Given a subspace $\mathcal{S}$ in $\mathbb{R}^K$ where $K < N$, a *pseudominimum* is a point

$$\mathbf{x}_p = (x_1, x_2, \ldots, x_N)$$

such that $\mathbf{x}_p$ is a local minimum in the subspace $\mathcal{S}$ but is not a local minimum for f.

Definition 6.10. Given a subspace $\mathcal{S}$ in $\mathbb{R}^K$ where $K < N$, a *global pseudominimum* is a point

$$\mathbf{x}_p = (x_1, x_2, \ldots, x_N)$$

such that $\mathbf{x}_p$ is a global minimum in the subspace $\mathcal{S}$ but is not a local or global minimum for f.

Example 7. An example of a global pseudominimum is the point $(0, 0)$ for the function

$$f(\mathbf{X}) = g(h(\mathbf{X})) \tag{6.1}$$

where

$$g(\mathbf{X}) \quad = \quad X_1^2 \; + \; X_2^2 \; - \; (\tanh(10X_1) + 1)\,(\tanh(-10X_2) + 1) \exp\left(\frac{X_1 + X_2}{3}\right)$$

and $\mathbf{h}(\mathbf{X}, \theta)$ is a rotation operator defined as

$$\mathbf{h}(\mathbf{X}, \theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \end{bmatrix}.$$

To highlight the shape of the function, we plot the inverse in Figure 6.1. The point $(0, 0)$ is a global pseudominimum in the subspace defined by the y axis at $X = 0$ because moving in any direction in the y-axis, the function increases in value.

However, the point is not a local minimum since the function can be decreased in the x and y axes simultaneously.

Note that not all pseudominimum are saddle points, but all saddle points are pseudominimum. The next proof gives an example of a pseudominimum that is not a saddle point. Finally, we define a special case of a pseudominimum.

Definition 6.11. Given a subspace $\mathcal{S}$ in $\mathbb{R}^K$ where $K < N$, a *maximal pseudominimum* is a point

$$\mathbf{x}_p = (x_1, x_2, \ldots, x_N)$$

such that $\mathbf{x}_p$ is a local minimum in the subspace $\mathcal{S}$ but is not a local minimum for $\mathcal{S} \cup X_i$ for all variables in $\mathbb{R}^N$.

Definition 6.11 differs from Definition 6.9 in a key way. A pseudominimum in subspace $\mathcal{S}$ may also be a pseudominimum $\mathcal{S} \cup X_i$ where X_i is some dimension not in $\mathcal{S}$. The definition of a pseudominimum only requires that a point not be a local minimum for all dimensions. For example, a point may be a pseudominimum in two dimensions X_1 and X_2 for a function with five variables X_1, X_2, X_3, X_4, X_5. But it may also be a pseudominimum in the dimensions X_1, X_2, and X_3. If the point is not a pseudominimum in the subspaces X_1, X_2, X_3, X_4 and X_1, X_2, X_3, X_5, then it is a maximal pseudominimum.

With these definitions, we first present a theorem showing that there exist global pseudominimas that will trap FEA. Second, we will generalize the existence theorem by showing under what conditions FEA will become trapped by global pseudominima.

Theorem 6.2.1. There exist global pseudominima that will trap FEA.

Proof. Because this is an existence theorem, we prove this by example based on the proof sketch used by Van den Bergh [112, 114]. Assume we are given a function

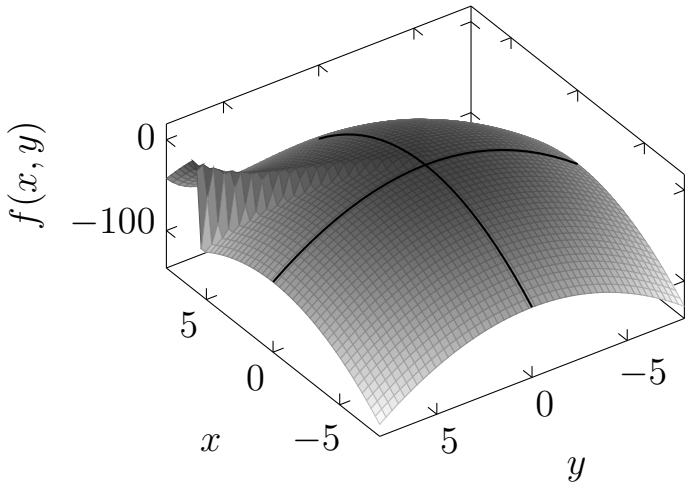


Figure 6.1: An inverse plot of Equation 6.1. Point $(0, 0)$ is an example of a global pseudominimum.

$f : \mathbf{D}^N \to \mathbb{R}$ with domain $\mathbf{D}^N$, that the output of the function is greater than 0, and that it is strictly increasing when moving in any direction from the origin Additionally, there exists a simplex with a point at the origin such that the rest of the points that define the simplex are strictly greater than 0. At the tip of simplex, the output of f is equal to the value at the origin, and within the simplex, the fitness decreases from all of the points that define the simplex to a local minimum $\mathbf{X}^* = (X_1^*, X_2^*, \ldots, X_N^*)$.

An example function with three dimensions is shown in Figure 6.2 with the sides of the triangle region projected onto the primary planes. In this example, the function for three variables X, Y and Z, is equal to $f(\boldsymbol{X}) = ||\boldsymbol{X}||$. The only exception is in a triangle shaped region where $f(\boldsymbol{X}) < 0$. Let B be the simplex region. We represent the function as

$$f(\boldsymbol{X}) = \begin{cases} g(\boldsymbol{X}) & \text{if } B \text{ contains } \boldsymbol{X} \\ \sqrt{X^2 + Y^2 + Z^2} & \text{else} \end{cases}$$

where

$$g(\boldsymbol{X}) = (X - X^*)^2 + (Y - Y^*)^2 + (Z - Z^*)^2 - C.$$

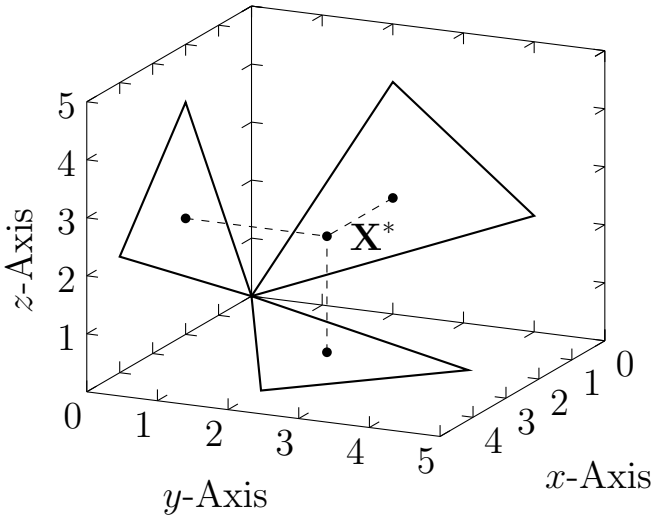


Figure 6.2: Diagram of an example function used in the proof of Theorem 6.2.1.

C is a constant value to ensure that all values of the function g within the simplex are negative.

An FEA is applied with factors $\boldsymbol{\mathcal{P}}_1$, $\boldsymbol{\mathcal{P}}_2$, $\ldots \boldsymbol{\mathcal{P}}_M$ where each factor S_i optimizes over a pair of variables X_j, X_k. Note that the origin $(0, 0 \ldots, 0)$ is a global pseudominimum defined by the FEA's factors since moving in only two directions causes f to increase in value. However, the origin is not a local minimum since f decreases in value by moving in all N directions simultaneously, thereby moving into the simplex with negative values.

Suppose the FEA has a full global solution G at the origin $(0, 0, \ldots, 0)$. If during factor $\boldsymbol{\mathcal{P}}_1$'s Update step it evaluates the point (X_1^*, X_2^*), the fitness will be some value L. By definition of f, the output is strictly increasing when moving away from the origin, and therefore, L is greater than 0. This is because FEA uses the full global solution $\mathbf{G}$ to evaluate the values (X_1^*, X_2^*).

In the example shown in Figure 6.2, this would equate to three factors: $\boldsymbol{\mathcal{P}}_1, \boldsymbol{\mathcal{P}}_2$, and $\boldsymbol{\mathcal{P}}_3$, which optimize over variables xy, yz, and xz, respectively. Additionally,

$\mathbf{G} = (0, 0, 0)$. Factor $\boldsymbol{\mathcal{P}}_1$ would evaluate the point X^*, Y^* as

$$f(X^*, Y^*, 0) = \sqrt{(X^*)^2 + (Y^*)^2}.$$

Note that this point is not within the simplex because all points that define the simplex are strictly greater than 0.

Because the fitness of (X^*, Y^*) is greater than 0 and the current best fitness of the factor, the factor discards this point and (X_1^*, X_2^*) will not be used in the competition and sharing steps of FEA. Consequently, the full global solution is unable to move from the origin to $(X^*, Y^*, 0, \ldots, 0)$. This same phenomenon will also occur for the other factors; therefore, FEA will be stuck at the global pseudominimum. □

We note that the point $\mathbf{p}$ in the subspace x in Figure 6.2 is not a maximal pseudominimum because the point is also a pseudominimum in the subspace xy. However, the point $\mathbf{p}$ is a maximal pseudominimum in the subspace xy.

Theorem 6.2.2. Given an FEA, let $\boldsymbol{\mathcal{S}}$ be a set of subspaces $\mathbb{R}^K$ with $K < N$ as defined by the set of factors in the FEA. If FEA's full global solution reaches a point $\mathbf{p}$ that is a global pseudominimum in all subspaces $\mathcal{S}_i \in \boldsymbol{\mathcal{S}}$, then FEA will be unable to escape $\mathbf{p}$.

Proof. Assume we are given a function $f : \mathbf{D}^N \to \mathbb{R}$ with domain $\mathbf{D}^N$ and that the FEA has a set of factors $\boldsymbol{\mathcal{P}}_1$, $\boldsymbol{\mathcal{P}}_2$, $\ldots \boldsymbol{\mathcal{P}}_M$. Also, assume that the full global solution $\mathbf{G} = (X_1^*, X_2^*, \ldots X_N^*)$ is at a point $\mathbf{p}$ that is a global pseudominimum. By definition of a global pseudominimum, each factor $\boldsymbol{\mathcal{P}}_i$ will be unable to move to a better position because all other points in the subspace that factor $\boldsymbol{\mathcal{P}}_i$ is searching over will have fitness greater than the current fitness of $\boldsymbol{\mathcal{P}}_i$ at $\mathbf{p}$. Since each factor will be unable to locate a position with better fitness than its current position, the factor will not

use other values different than those in **G** during the competition and sharing steps of FEA. Because no other values are used during competition, the full global solution is unable to move from the global pseudominimum; therefore, FEA will be unable to escape from the global pseudominimum. □

While this shows that FEA may become stuck, the factors must be optimizing over the variables that induce the pseudominimum. For example, if a factor is optimizing over all three variables in Figure 6.2, then the factor using hill climbing as the algorithm will not be trapped by the point (0,0,0). This suggests that if it is known where a pseudominimum occurs in an optimization problem, then there should exist a factor that optimizes over a superset of variables that induce the pseudominimum.

Another consequence of this result is that it may provide another explanation of results in the previous chapter as to why certain factor architectures outperform others. In those experiments, we showed that the Markov architecture performed better than other architectures because it minimized hitchhiking. However, the performance of the better factor architecture may also be due to fact that the factors have less chance of becoming trapped in pseudominima.

6.3 Hybrid FEA

In the previous section, we showed that the full global solution **G** in FEA will converge to a single solution. However, similar to CPSO, FEA may converge to suboptimal solutions called pseudominima that regular EA are able to escape. As described earlier, van den Bergh and Engelbrecht proposed an extension to the CPSO algorithm called Hybrid Cooperative Particle Swarm Optimization (CPSO-H). CPSO-H combined the benefits of CPSO and a regular PSO by performing a number of iterations using CPSO followed by running a regular PSO. By performing several

Algorithm 6.11: Hybrid Factored Evolutionary Algorithm

Input: Function f to optimize, optimization algorithm A
Output: Full solution $\boldsymbol{G}$

```
1: S ← initializeFactors(f, X, A)
2: G ← initializeFullGlobal(S)
3: repeat
4:    for all S_i ∈ S do
5:       repeat
6:          S_i.updateIndividuals()
7:       until Termination criterion is met
8:    end for
9:    G ← Compete(f, S)
10:   Share(G, S)
11:   Full.seed(G)
12:   repeat
13:      Full.updateIndividuals()
14:   until Termination criterion is met
15:   if Full.bestFitness() is better than f(G) then
16:      G ← Full.bestSolution()
17:   end if
18: until Termination criterion is met
19: return G
```

iterations with a full PSO, CPSO-H is able to escape pseudominima because an individual is able to modify all variables in one round of updates. Here, we present a similar extension to FEA called FEA-H (Algorithm 6.11).

FEA-H works as follows. First, it performs the same set of operations as FEA—Update, Compete, and Share—in lines 1–10. Next, FEA-H performs a set of updates to the full EA population, which is denoted as $Full$. First, the position of the individual with the worst fitness from $Full$ is set to the full global solution $\mathbf{G}$ (line 11). In line 14, the full population updates its individuals until some stopping criteria is satisfied. Finally, the fitness of the best solution in $Full$ is compared with $\mathbf{G}$. If

the fitness of the best individual from the full population is better than the full global solution, then the algorithm sets **G** to solution with the better fitness (line 16). However, if the fitness is not better, then no changes are made to **G**.

By running a set of iterations with a full EA, FEA-H is should be able to escape pseudominima. This is because when *Full* updates its individuals, each individual has the opportunity to change every variable simultaneously. For example, in Figure 6.1, the full population is able to move individuals from the origin to the optimal solution by following the incline on the ridge. However, if the factor architecture subsumes the pseudominima in the fitness landscape, then FEA will not become stuck at suboptimal solutions, and FEA-H will provide little to no benefit over FEA.

6.4 Empirical Analysis of Pseudominima in FEA

As shown in the previous section, FEA is still susceptible to pseudominima. However, FEA only becomes stuck at these points if the factors are optimizing over a subset of variables in $\mathbb{R}^k$. We hypothesize that for certain factor architectures in FEA, the probability of pseudominima becomes low. To test this hypothesis, we compare versions of CPSO and FEA with the hybrid version of CPSO-H presented by Van de Bergh and Englebrecht [114]. In addition, we compare all algorithms with FEA-H presented in the previous section. If FEA does become stuck in pseudominima, then one would expect FEA-H to outperform FEA because the full population in FEA-H allows the algorithm to escape pseudominima. However, if FEA does not become stuck in pseudominima, FEA-H will provide little to no benefit over FEA.

For the test problems we chose NK landscapes, abductive inference in Bayesian Networks, and some common benchmark optimization problems. NK landscapes were included because they represent commonly used functions for evaluating the performance of evolutionary algorithms applied to epistatic functions. We included

abductive inference in Bayesian Networks because they are a practical combinatorial optimization problem. On the NK landscapes and abductive inference, we used a modified version of PSO since both problems are functions with a discrete input. To handle these problems, we used the ICPSO algorithm as the underlying search algorithm [107].

6.4.1 Test Problems

For the test problems we again chose NK landscapes, abductive inference in Bayesian Networks and some common benchmark optimization problems. We generated NK landscapes with parameters $N = 25, 40, 50$, and $K = 5$. For each set of parameters, we created 30 random landscapes and ran each algorithm 30 times. On the abductive inference, we used the Alarm, Andres, Child, Hailfinder, Hepar2, Insurance, and Win95pts Bayesian networks from the Bayesian Network Repository [93]. For the benchmark functions, we chose the following: Ackley's, Dixon Price, Exponential, Griewank, Rastrigin, Rosenbrock, Schwefel 1.2, and Sphere with 30 dimensions. [49].

6.4.2 Setup

For the FEA algorithm on the NK landscapes and Bayesian networks, we used the Markov blanket factor architecture since this was shown in previous Chapters to outperform all competing factor architectures. On the Benchmark functions, we used the Simple Centered (SC) architecture proposed in Chapter 4 since SC architecture had the most consistent performance over all functions [106]. For the CPSO algorithms, we had each subswarm optimize over two variables in the problem. This subswarm size was found to have the most consistent performance during the tuning of the algorithms.

Table 6.1: Results from comparing regular and hybrid versions of CPSO and FEA on NK landscapes.

	CPSO	CPSO-H	FEA	FEA-H
N=25, K = 5	1.78E+01(6.06E−02)	**1.83E+01(3.80E−02)**	**1.91E+01(3.15E−02)**	1.89E+01(3.19E−02)
N=40, K = 5	2.81E+01(1.09E−01)	2.86E+01(7.00E−02)	**3.05E+01(4.72E−02)**	3.01E+01(5.08E−02)
N=50, K = 5	**3.47E+01(1.31E−01)**	3.47E+01(6.74E−02)	**3.81E+01(3.87E−02)**	3.65E+01(5.11E−02)

Each algorithm was given a total of 400 individuals to divide between their subswarms. For the hybrid algorithms CPSO-H and FEA-H, an additional 10 individuals were used for the full algorithm step of the algorithms. These values were found to perform well for all algorithms during tuning. On the NK landscapes and abductive inference, both versions of CPSO and FEA used ICPSO as the underlying search algorithm. On the benchmark problems, we used canonical PSO. For both PSOs, the ω parameter was set to 0.729, and ϕ_1 and ϕ_2 were both set to 1.49618. In ICPSO, the scaling value ϵ was set to 0.75. These values were found to perform well for all algorithms on all problems during tuning of the algorithms.

6.4.3 Results

Table 6.1 shows the results comparing CPSO, CPSO-H, FEA, and FEA-H on maximizing NK landscapes. Results from abductive inference on Bayesian networks are shown in Table 6.2 Note that both these problems are maximization. Results comparing CPSO, CPSO-H, FEA, and FEA-H on minimizing the benchmark functions are in Table 6.3. All results are expressed as means over 30 trials with standard errors in parentheses. Bold values indicate a significant difference between the regular CPSO or FEA algorithms with the hybrid versions using Mann-Whitney U test with $\alpha = 0.05$.

As we can see in the NK landscape results, CPSO-H outperformed CPSO on two out of the three landscapes, but was only significantly better on $N = 25, K = 5$.

Table 6.2: Results from comparing regular and hybrid versions of CPSO and FEA on Bayesian networks.

	CPSO	CPSO-H	FEA	FEA-H
Alarm	−1.99E+01(2.01E+00)	−1.59E+01(1.49E+00)	−9.12E+00(5.44E−01)	−9.87E+00(6.45E−01)
Andes	−2.01E+02(8.72E+01)	**−1.72E+02(7.12E+00)**	**−7.37E+01(8.34E−01)**	−8.80E+01(1.30E+00)
Child	−9.54E+00(5.08E−01)	−9.06E+00(4.48E−01)	−6.61E+00(3.03E−01)	−6.57E+00(2.98E−01)
Hailfinder	−7.39E+02(1.60E+02)	**−2.28E+02(6.92E+01)**	**−3.43E+01(3.61E−01)**	−3.57E+01(3.18E−01)
Hepar2	**−1.86E+01(4.48E−01)**	−2.14E+01(7.96E−01)	−1.81E+01(4.35E−01)	−1.76E+01(3.04E−01)
Insurance	−3.26E+02(8.56E+01)	**−1.57E+01(9.07E−01)**	−9.65E+00(4.13E−01)	−1.04E+01(2.91E−01)
Win95pts	−2.65E+02(1.10E+02)	**−1.17E+02(3.46E+01)**	**−1.82E+01(5.90E−01)**	−3.26E+01(1.16E+00)

Table 6.3: Results from comparing regular and hybrid versions of CPSO and FEA on benchmark functions.

	CPSO	CPSO-H	FEA	FEA-H
Ackleys	**6.47E−06(3.86E−07)**	5.61E−05(6.24E−06)	**6.12E−06(7.39E−07)**	9.90E−05(1.20E−05)
Dixon Price	**2.22E−02(2.22E−02)**	2.17E−01(1.52E−01)	**4.44E−02(3.09E−02)**	1.56E−01(5.23E−02)
Exponential	**−1.00E+00(2.95E−10)**	−1.00E+00(1.02E−08)	**−1.00E+00(1.47E−10)**	−1.00E+00(7.34E−08)
Griewank	3.17E−02(1.95E−02)	**3.11E−03(1.50E−03)**	4.77E−02(8.32E−03)	**5.33E−03(1.99E−03)**
Rastrigin	**4.34E+00(3.83E−01)**	5.27E+00(3.76E−01)	**3.38E+00(2.96E−01)**	4.51E+00(4.49E−01)
Rosenbrock	2.81E+00(5.77E−01)	2.28E+01(1.40E+01)	**9.29E+00(3.07E+00)**	4.80E+01(7.22E+00)
Schwefel	2.86E+05(5.31E+04)	**1.42E+03(5.08E+02)**	**6.06E+02(5.63E+01)**	1.00E+03(6.96E+01)
Sphere	**2.15E−08(6.64E−09)**	7.48E−07(1.68E−07)	**8.97E−09(1.20E−09)**	3.21E−06(1.15E−06)

Only on the larger problems ($N = 50$) did CPSO-H tie with CPSO-H. However, when looking at the FEA, the hybrid version of FEA was always outperformed by regular FEA, so hybridization provided no benefit and appears to have hurt performance.

The Bayesian network results show a similar trend in comparing regular and hybrid versions of CPSO and FEA. On the Andes, Hailfinder, Insurance, and Win95pts networks, CPSO-H outperformed CPSO significantly. Additionally, it was not outperformed by CPSO on the Alarm and Child networks. Only on the Hepar2 networks did CPSO significantly outperform CPSO-H. FEA outperformed FEA-H significantly on the Andes, Hailfinder, and Win95pts networks. Furthermore, it was only outperformed by FEA-H on the Child and Hepar2 networks. Note that in both of these cases, there was no significant difference between FEA and FEA-H.

On the benchmark functions, CPSO-H outperforms CPSO significantly on Griewank and Schwefel whereas, CPSO performs significantly better than CPSO-H on the Ackleys, Dixon Price, Exponential, Rastrigin and Sphere functions. FEA outperformed FEA-H by a significant margin on all functions except the Griewank function, where FEA-H outperformed FEA. We also note that the trends in the differences between CPSO and CPSO-H are similar to that of FEA and FEA-H. Only on the Schwefel function was there a significant difference in the trends of CPSO and CPSO-H with that of FEA and FEA-H.

6.4.4 Analysis

From the NK landscape, Bayesian network, and benchmark results, we see that the full PSO steps used by FEA-H provided a performance gain only a few times. In particular, FEA-H only significantly outperformed FEA on the Griewank function. Additionally, in many of the problems, the full population steps in FEA-H hurt the performance.

While CPSO-H significantly outperformed CPSO on the majority of the NK landscapes and Bayesian networks, CPSO performed significantly better than CPSO-H on the majority of the benchmark functions. There are two possible reasons for this result. One is that in the majority of the functions, there are fewer pseudominima that trap CPSO; therefore, CPO-H provides fewer benefits than regular CPSO. The other possible reason is that the creation of the subswarms for CPSO leads to the ability to avoid the pseudominima in the search space on the majority of the functions.

Another result we would like to make note of is the similarities between CPSO and FEA on the benchmark functions. We believe that the similarity of these results is due to the subswarm (factor) size for both CPSO and FEA being set to two thus not adequately capturing all of the variable interactions. Even with these similarities, on

the Rosenbrock and Schwefel functions, CPSO did not outperform CPSO-H whereas FEA outperformed FEA-H on both these functions. Specifically, with Rosenbrock and Schwefel functions the overlap in FEA appears to allow the subpopulations to capture the majority of the of the variable interactions that CPSO is unable to capture. Again, this is because CPSO subpoulations optimize only disjoint sets of variables.

For the NK landscape and Bayesian networks, we believe that the performance of FEA over FEA-H is because the factors in FEA are less susceptible to pseudominima than the subswarms in CPSO. In the benchmark functions, CPSO was able to escape the majority of the pseudominima. But on the NK landscapes and Bayesian networks, CPSO had a higher liklihood of becoming trapped in pseudominima, which explains why CPSO-H often outperformed CPSO. Meanwhile, the factors in FEA are less prone to get stuck in pseudominima because they are optimizing over larger groups of subspaces that induce the pseudominima; therefore, the full PSO steps are not needed.

We explored this hypothesis further by running an experiment where, during FEA, we checked to see if the factors' best solution were at a pseudominimum after the Compete step. Because the Bayesian networks and NK landscapes are discrete problems, we are able to look at all neighboring states of a factor. For a given factor $\boldsymbol{\mathcal{P}}_i$, if there does not exist a neighboring state with better fitness, then $\boldsymbol{\mathcal{P}}_i$ could be at a pseudominimum. However, this point could also be a true local minimum. To see if $\boldsymbol{\mathcal{P}}_i$ is in fact at a pseudominimum, we checked to see if any neighboring points of $\boldsymbol{\mathcal{P}}_i \cup \mathbf{R}_i$ had better fitness. If there does exist a neighboring point of $\boldsymbol{\mathcal{P}}_i \cup \mathbf{R}_i$ with better fitness, then $\boldsymbol{\mathcal{P}}_i$ is a true pseudominimum. However, if there are no neighboring points of $\boldsymbol{\mathcal{P}}_i \cup \mathbf{R}_i$ with better fitness, then $\boldsymbol{\mathcal{P}}_i$ is at a local minimum and not a pseudominimum. If a factor was not at a pseudominima or local minima, it was ignored in the calculation. This is because a factor not at a local or pseudominimum

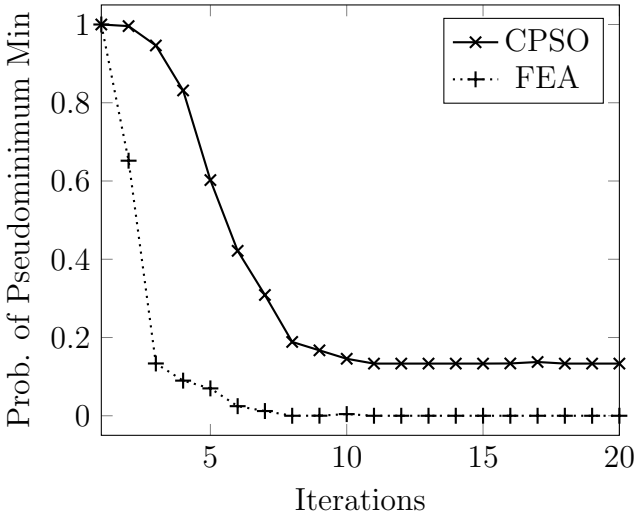


Figure 6.3: Probability of pseudominimum for NK landscape $N = 25$, $K = 5$.

suggests that the factor is still in the process of moving towards a better area in the search space.

We ran these experiments on the NK landscape with $N = 25$, $K = 5$, and on the Insurance network. Results for both CPSO and FEA are shown in Figures 6.3 and 6.4 respectively. In both of these figures, the x-axis represents iterations of FEA. The y-axis gives the probability of a pseudominimum at each iteration and is calculated as

$$\frac{Pm}{Pm + Lm}$$

where Pm is the number of factors at a pseudominimum and Lm is the number of factors at a local minimum. A value of 1 means that all subswarms and factors are located at pseudominima while 0 indicates that all factors are located at a local minimum. Note that there is a possibility that the probability of a pseudominimum may become undefined if none of the factors are at a local minimum or pseudominimum. However, those instances were never encountered in these experiments.

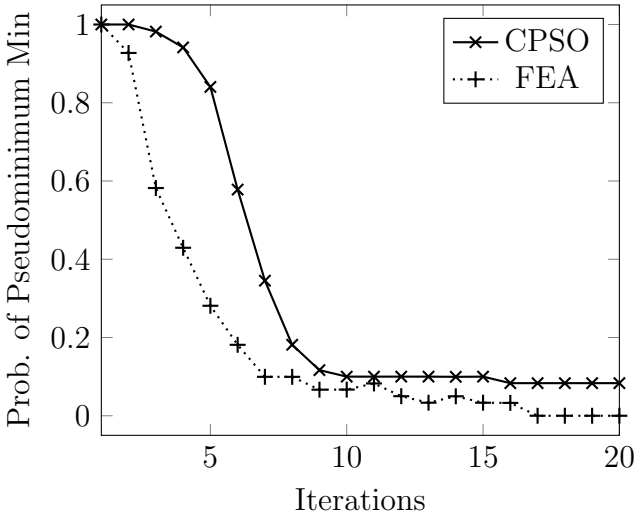


Figure 6.4: Probability of pseudominimum for Insurance network.

As we can see in the two figures, both CPSO and FEA begin with a high probability of being located at pseudominima. This is likely due to the fact that the subswarms have just begun to locate good areas in the search space and are still moving towards those areas. However, we can see that the probability of factors being located at pseudominima in FEA decreases much faster than CPSO. Additionally, as the number of iterations increase, the probability for FEA becomes closer to zero. While the probability of CPSO does decrease over time, there is approximately a 20% probability of a subswarm being located at a pseudominimum. Finally, we note that the probability of pseudominima never increases because the definition of pseudominimum excludes a local minimum. Once a factor reaches a local minimum, it becomes more difficult for the factor to escape the local minimum, thus reducing the likelihood of a factor moving from a local minimum to a pseudominimum.

These results highlight why CPSO-H sees greater performance gains over CPSO than FEA-H does over FEA. On the NK landscapes and Bayesian networks, CPSO becomes trapped in pseudominima. CPSO-H is able to escape these pseudominima

and continue searching towards better solutions. FEA, on the other hand, has a smaller probability of becoming stuck in a pseudominima; therefore, the full swarm steps in FEA-H provide less benefit because the algorithm is already able to move towards good locations in the search space.

While the results suggest that FEA is less prone to pseudominima, the benchmark results suggest that the architecture is not the best for all functions. This is demonstrated by FEA-H significantly outperforming FEA on the Griewank function. We believe that main cause is that the factor architecture used for the Griewank function is suboptimal, and given a better factor architecture, we may see FEA outperform FEA-H. However, despite a suboptimal factor architectures, FEA is still competitive with both FEA-H and both versions of CPSO.

6.5 Conclusion

In this paper, we proved that the full global solution $\boldsymbol{G}$ found by FEA will converge to a single point if the individual factors also converge. Even so, we also proved that FEA is still susceptible to pseudominima. Despite the fact that FEA can become stuck at pseudominima, we demonstrated that, when using certain factor architectures, the probability of factors becoming stuck at pseudominimum approaches zero. To test this, we compared hybrid versions of CPSO and FEA and demonstrated that FEA-H did not provide significant performance gains on discrete problems. Additionally, we showed that over time, FEA has a lower probability of pseudominimum than CPSO.

CHAPTER SEVEN

FEA ON UNITATION AND DECEPTIVE FUNCTIONS

In Chapter 4, we demonstrated that the best-performing factor architecture also maintained better diversity between the individuals. However, it is unknown how the increased diversity corresponds with improved performance. Furthermore, the question still remains as to whether FEA's performance is most influenced by the creation of subpopulations through the factoring of the function or the competition and collaboration between factors in the Compete stage. For example, if a factor architecture is poorly constructed, will Compete in FEA still be able to locate good solutions?

There are three main areas we will explore in this chapter. The first is what types of problems an individual factor is able to solve. Second, we examine how effective Compete is at combining solutions from overlapping factors. Finally, we investigate if there exists guidelines for creating factors for problems without intuitive methods for factoring the function.

To investigate these different areas of research, we explore how FEA performs on two sets of functions with binary variables. The first set are *unitation* functions, which are functions whose fitness is based on the number of zeros and ones in the input vector. Even though the functions themselves do not have complex definitions, they are still able to define complex search spaces [88]. Additionally, the functions contain properties found in real-world problems. The second set of functions are the Royal Road functions, which were proposed by Mitchell *et al.* to explore the building block hypothesis in GAs [66].

While we analyzed the number of iterations in previous chapters, the optimal solutions in the Bayesian networks and NK landscapes were unknown. By dealing with unitation functions, we are able to analyze the number of evaluations FEA requires to directly find the optimal solution. This also allows us to observe how FEA handles deceptive problems.

7.1 Unitation Functions

In our discussion of unitation functions, we split them in two groups: Base functions and Royal Road functions. Base functions are functions in which the fitness is based upon counting the number of 1's. Royal Road functions are variations of the base unitation functions in that, in addition to the fitness being based upon the number of 1's, a specific structure is imposed upon the fitness landscape. We begin by discussing the base unitation functions.

7.1.1 Base Functions

A unitation function is a fitness function whose output is based upon the number of ones in a binary array. We can formally define the unitation function $u(\mathbf{X}) \rightarrow \mathbb{R}^+$ of a binary array as

$$u(\mathbf{X}) = \sum_{i=0}^{N} X_i$$

where X_i is the ith binary variable. The simplest unitation problem is called *One Max*, which is defined as

$$f(\mathbf{X}) = u(\mathbf{X})$$

The optimal solution is all 1's. Figure 7.1 gives a ten dimension example of the fitness of the One Max function where the x-axis is the unitation (number of 1's) and the y-axis is the fitness.

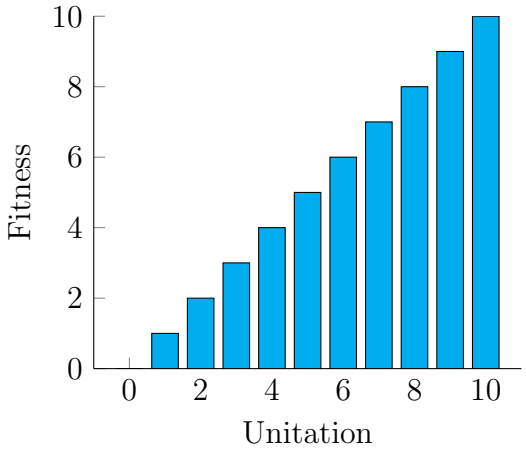


Figure 7.1: Fitness of OneMax function.

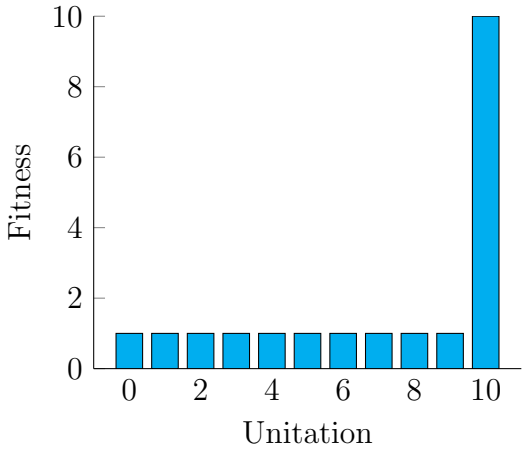


Figure 7.2: Fitness of Needle function.

Two of the simplest variations of the One Max problem are the *Needle* and *BiNeedle* problems. The Needle function contains a search space with a large flat basin and a narrow optimal solution when the input contains all 1's. In some ways, the Needle function is reminiscent of a pseudominimum from Chapter 6, where a search algorithm may become stuck in a suboptimal location that it is unable to escape because only K variables can be changed. The function is defined as

$$f(\mathbf{X}) = \begin{cases} 1+\alpha & u(\mathbf{X}) = N \\ 1 & \text{otherwise.} \end{cases}$$

The BiNeedle, on the other hand, contains two narrow optimal solutions: one with all 0's and one with all 1's. The function is defined as

$$f(\mathbf{X}) = \begin{cases} 1+\alpha & u(\mathbf{X}) = 0 \\ 1+\alpha & u(\mathbf{X}) = N \\ 1 & \text{otherwise.} \end{cases}$$

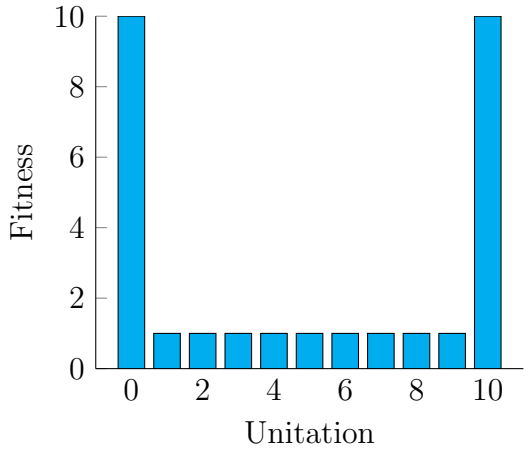


Figure 7.3: Fitness of BiNeedle function.

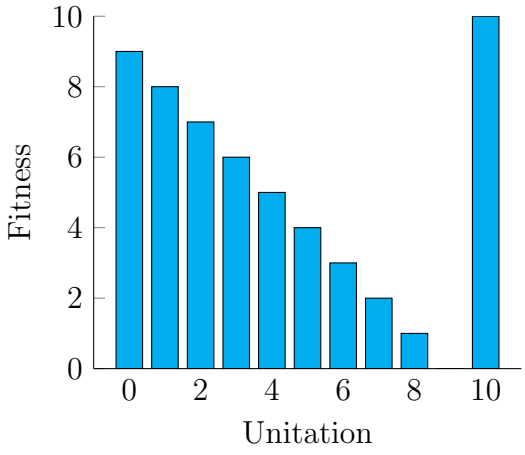


Figure 7.4: Fitness of DecTrap function.

A ten dimension example of the Needle function is shown in Figure 7.2 and Figure 7.3 contains an example of the BiNeedle problem.

Three more-deceptive versions of the Needle and BiNeedle functions exists: *DecTrap*, *TwoTrap*, and *DecTwoTrap*. A deceptive function is one in which there are multiple paths to local optimum solutions; however, one of more of those paths lead to a local optimum.

DecTrap contains one global optimum with all 1's and one suboptimal solution all 0's. Additionally, the function slope is defined such that the search will be biased towards the suboptimal solution. An example of DecTrap is shown in Figure 7.4. Formally, it is defined as

$$f(\mathbf{X}) = \begin{cases} N & u(\mathbf{X}) = N \\ N - 1 - u(\mathbf{X}) & \text{otherwise.} \end{cases}$$

From the figure, we observe how the function is deceptive to a search algorithm. A search algorithm will be led in one of two directions: either toward all 0's or all 1's. However, the solution with all 0's is only a local optimum.

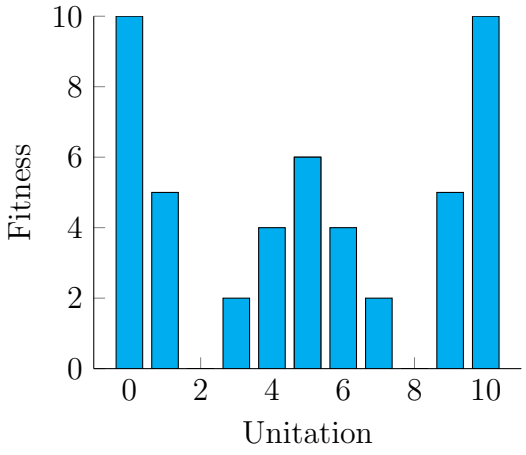


Figure 7.5: Fitness of TwoTrap function.

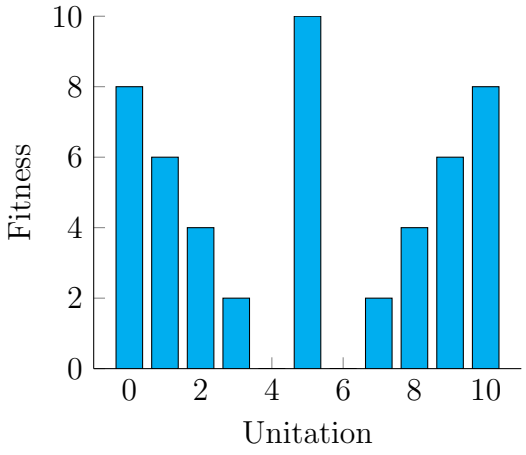


Figure 7.6: Fitness of DecTwoTrap function.

TwoTrap is an extension of the DecTrap function that contains two optimal solutions: all 0's and all 1's. However, the suboptimal solution is centered in the middle of the search space: $N/2$ 1's. Similar to DecTrap, much of the search space guides the search algorithm toward the suboptimal solution. The function is defined as

$$f(\mathbf{X}) = \begin{cases} N - \frac{N}{2}u(\mathbf{X}) & u(\mathbf{X}) < \frac{N}{5} \\ 2u(\mathbf{X}) - \frac{2}{5}N & \frac{N}{5} \leq u(\mathbf{X}) \leq \frac{N}{2} \\ \frac{8}{5}N - 2u(\mathbf{X}) & \frac{N}{2} < u(\mathbf{X}) \leq \frac{4}{5}N \\ \frac{N}{2}u(\mathbf{X}) - \frac{2}{5}N^2 & \frac{4}{5}N < u(\mathbf{X}). \end{cases}$$

DecTwoTrap is the opposite of the TwoTrap function; it has one optimal solution and one suboptimal solution and is defined as

$$f(\mathbf{X}) = \begin{cases} N & u(\mathbf{X}) = \frac{N}{2} \\ -2u(\mathbf{X}) + N - 2 & u(\mathbf{X}) < \frac{N}{2} \\ 2u(\mathbf{X}) - N - 2 & \frac{N}{2} < u(\mathbf{X}). \end{cases}$$

Figures 7.5 and 7.6 show an example of the TwoTrap and DecTwoTrap functions, respectively.

7.1.2 Royal Road

The Royal Road is an optimization problem that was original designed to be easy for GAs to solve [66, 88]. The function is defined as

$$f(\mathbf{X}) = \sum_{s \in S} c_s \sigma_s(\mathbf{X})$$

where $\mathbf{X}$ is a bit string and S defines a set of schemata. A schema is a string that that denotes a pattern in a function. Additionally, a schema allows variables to be wild cards [66]. For example, a schema of $*1$ is a pattern that matches any string where the second variable is 1 and the first variable is either 0 or 1. If the bit string $\mathbf{X}$ contains a schema s, then $\sigma_s(\mathbf{X})$ returns 1; otherwise it returns 0. c_s defines a cost or weight for schema s.

In the first presentation of Royal Road, Mitchell *et al.* proposed two different versions: R1 and R2. Schemata for R1 are shown in Figure 7.7. The inspiration for R1 the generation of a function that was thought to be easy for a GA to solve. It was believed that intermediate schemata ($s_9 - s_{14}$) would help guide the search process for the GA. Mitchell *et al.* also proposed the R2 Royal Road, which eliminates the intermediate schema. However, the authors discovered that intermediate schemas slowed down the GA as it searched for the optimal solution of all 1's.

We propose an extension of R1 that combines concepts from the deceptive unitation functions with those concepts in the Royal Road. This is accomplished by introducing deceptive schemata into set as shown in Figure 7.9. The motivation for this is the introduction of groups of variables that are purposefully designed to mislead FEA. Previous results on factor architectures suggest a factor should be

$s_1 =$ 11111111**, $c_1 = 8$
$s_2 =$ ********11111111**, $c_2 = 8$
$s_3 =$ ****************11111111**, $c_3 = 8$
$s_4 =$ ************************11111111********************************, $c_4 = 8$
$s_5 =$ ********************************11111111************************, $c_5 = 8$
$s_6 =$ **11111111****************, $c_6 = 8$
$s_7 =$ **11111111********, $c_7 = 8$
$s_8 =$ **11111111, $c_8 = 8$
$s_9 =$ 1111111111111111**, $c_9 = 16$
$s_{10} =$ ****************1111111111111111********************************, $c_{10} = 16$
$s_{11} =$ ********************************1111111111111111****************, $c_{11} = 16$
$s_{12} =$ **1111111111111111, $c_{12} = 16$
$s_{13} =$ 11111111111111111111111111111111********************************, $c_{13} = 32$
$s_{14} =$ ********************************11111111111111111111111111111111, $c_{14} = 32$
$s_{15} =$ 1111111111111111111111111111111111111111111111111111111111111111, $c_{15} = 64$

Figure 7.7: The R1 Royal Road function.

$s_1 =$ 11111111**, $c_1 = 8$
$s_2 =$ ********11111111**, $c_2 = 8$
$s_3 =$ ****************11111111**, $c_3 = 8$
$s_4 =$ ************************11111111********************************, $c_4 = 8$
$s_5 =$ ********************************11111111************************, $c_5 = 8$
$s_6 =$ **11111111****************, $c_6 = 8$
$s_7 =$ **11111111********, $c_7 = 8$
$s_8 =$ **11111111, $c_8 = 8$
$s_9 =$ 1111111111111111111111111111111111111111111111111111111111111111, $c_9 = 64$

Figure 7.8: The R2 Royal Road function.

s_{16} = ****00000000**********, $c_{16} = 8$
s_{17} = **********00000000**********, $c_{17} = 8$
s_{18} = **********00000000**********, $c_{18} = 8$
s_{19} = **********00000000**********, $c_{19} = 8$
s_{20} = **********00000000**********, $c_{20} = 8$
s_{21} = **********00000000**********, $c_{21} = 8$
s_{22} = **********00000000****, $c_{22} = 8$
s_{23} = ********0000000000000000**********, $c_{23} = 16$
s_{24} = **********0000000000000000**********, $c_{24} = 16$
s_{25} = **********0000000000000000********, $c_{25} = 16$
s_{26} = **********00000000000000000000000000000000**********, $c_{26} = 32$

Figure 7.9: The R2 Royal Road function. The schema above are added to the R1 schema, comprising the the R3 Royal Road function.

generated for interactive groups of variables. In the R3 function, the negative schema represents groups of related variables; however, those groups of related variable are likely to lead to suboptimal solutions.

7.2 FEA on Unitation Functions

As discussed at the beginning of the chapter, two of the main questions we wish to answer are what kinds of problems can an individual factor solve and how effective is the Compete function at combining solutions from an individual factor. In this section, we set out to answer these questions by factoring ten variable base unitation functions. By dealing with smaller problems, we are able to perform a more detailed analysis of the search process. Additionally, the ten dimensional functions can be thought of as being part of a much larger function, which allows us to examine the types of problems smaller factors perform well on.

One of the first goals in evaluating FEA on the unitation functions is how to generate an effective factor architecture. Based on the results in previous chapters, we discovered that the optimal way to factor a function is probably based on the Markov blanket of the variable's factor graphs. In the base unitation functions, there is no intuitive way to group the variables. Based on the OneMax function, which forms the basis for all the unitation functions, we hypothesize that all variables are independent of one another. However, in the Needle or BiNeedle functions, a relationship is created between the variables because the optimal solution cannot be located unless all variables are at a specific value. We begin by exploring how varying the size and overlap between factors affects the performance of FEA on the base unitation functions.

7.2.1 Varying Factor Size

The first thing we test is how the factor size with no overlap affects the performance of FEA on the base unitation functions. To do so, we generate factors of decreasing size, starting with a full GA and decreasing the factor size by one. Because the focus is strictly on size and not overlap, we do not allow factors to overlap with one another. For example, the second largest factor architecture generates one factor of size nine and one factor of size one, whereas the next largest factor has one factor of size eight and one of size two. When possible, we generate factors of equal size, such as two factors of size $N/2$. In the case where factors optimize over four variables, we create two factors of size four and one of size two. Following this process gives us the set of factor architectures shown in Table 7.1.

In all experiments, we use a GA as the underlying optimization algorithm. The GA used in our experiments used tournament selection and uniform crossover with a mutation rate set to 0.1. In all algorithms, we used a budget of 250 individuals to

Table 7.1: Set of factor architectures varying factor size and with no overlap between factors.

Name	Number of Factors (M) and Size (N): (N, M)
F	(1,10)
S_9	(1,9), (1,1)
S_8	(1,8), (1,2)
S_7	(1,7), (1,3)
S_6	(1,6), (1,4)
S_5	(2,5)
S_4	(2,4), (1,2)
S_3	(3,3), (1,1)
S_2	(5,2)

distribute evenly between all the factors. We also present results for a full, single-population GA.

All problem had ten variables. Each algorithm was run until the optimal solution was found. However, each algorithm was given a budget of 35,000 evaluations before terminating the algorithm. All results are from 200 trials. In FEA, factors performed one round of updates before Compete and Share were performed.

7.2.1.1 Results Figure 7.10 presents the number of successful trials to find the optimal solutions, whereas Figure 7.11 presents the average number of fitness evaluations, excluding the failures required for each factor architecture to locate the optimal solution. In Figure 7.10, the y-axis corresponds to the number of successful trials, whereas in Figure 7.11 the y-axis on the corresponds to the number of fitness evaluations. The x-axis lists the the different factor architectures, starting with the full GA (F) and then moving down the architectures S_9, S_8, ..., S_2. For the sake of readability, labels for S_9, S_7, S_5, and S_3 are not shown. The average number of fitness evaluations is calculated only from trials that were successful in locating the optimal solution. Additionally, the standard error is shown for the number of fitness

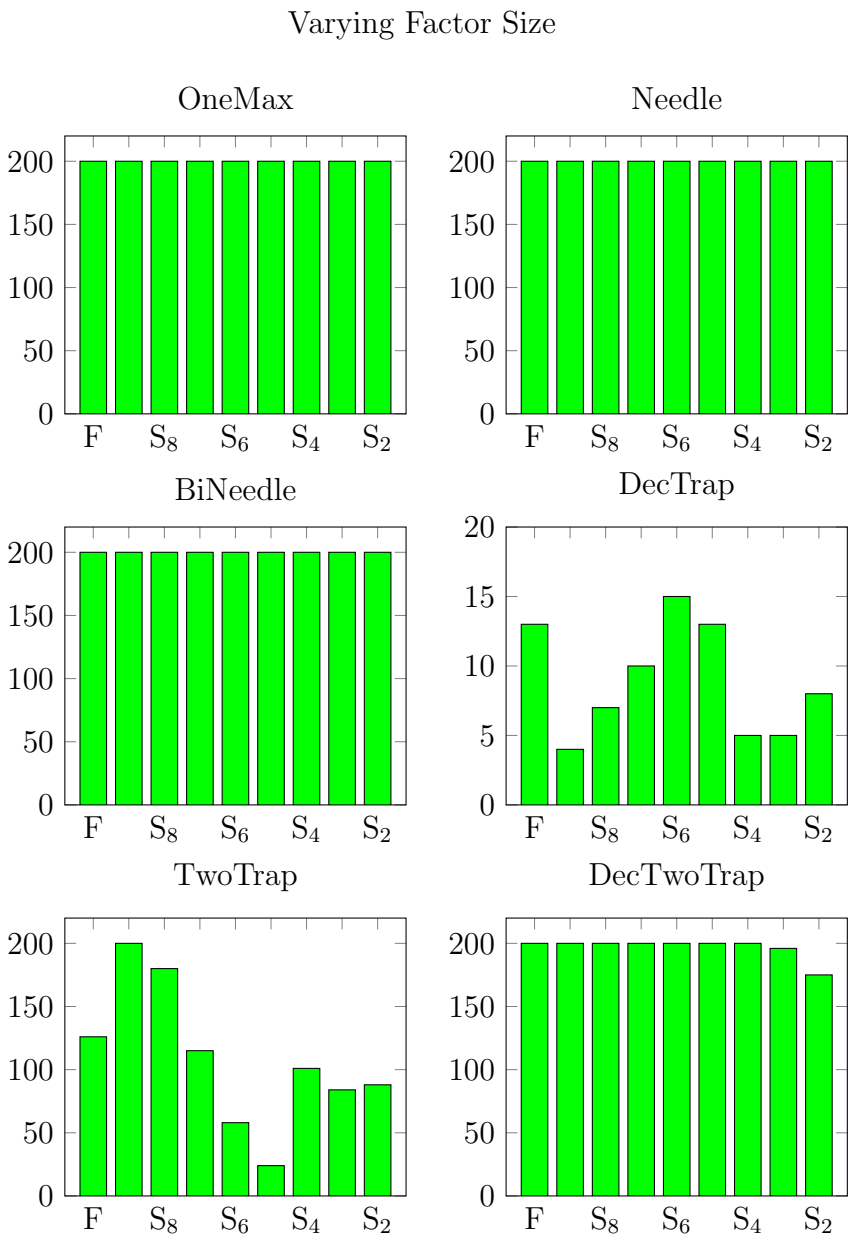


Figure 7.10: Number of successful trials of FEA with factors of decreasing size on unitation functions.

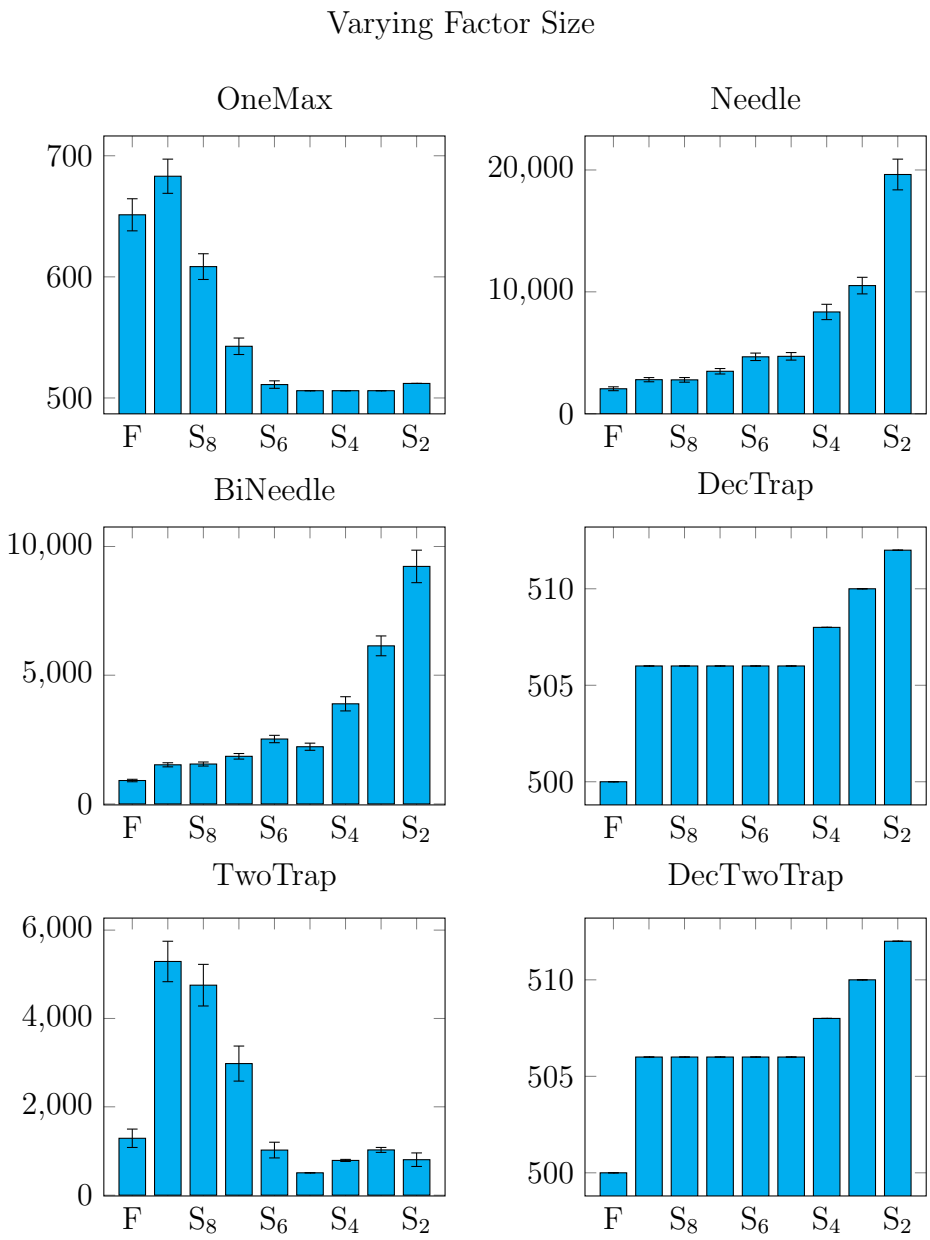


Figure 7.11: Average number of fitness evaluations of FEA with factors of decreasing size on unitation functions.

evaluations. Note that in many cases, such as for the full GA (F) on the Needle function, the standard error is very small and may not be visible in the graph.

In the OneMax function, both the full GA and every factor architecture were able to find the optimal solution every time. However, S_5, S_4, and S_3 were able to do so in the fewest iterations, requiring on average only one iteration. The worst factor architecture was S_8, requiring approximately 180 more fitness evaluations than S_3, equating to almost one full iteration.

Conversely, on the Needle and BiNeedle functions, the larger factors performed best. The full GA performed the best, while the best FEA factor architecture was S_9. S_2 performed the worst, requiring seven times more fitness evaluations than S_9. However, all factor architectures were able to locate the optimal solution 100% of the time. Notably, out of all the base unitation functions, the Needle function required the most fitness evaluations.

On the DecTrap function, the algorithms were able to locate the optimal solution only 5% of the time. In terms of locating the best solution, S_6 performed the best, finding the optimal solution 15 times. The worst architectures were S_3 and S_4, which found the optimal solution only five times. When the full GA or FEA found the optimal solution, it was always in a single iteration. Additionally, the variety in the number of fitness evaluations for this function is caused by the extra fitness evaluations FEA performs during initialization of the algorithm.

The best architecture on TwoTrap in terms of successful trials was S_9, which found the optimal solution 100% of the time. However, it required the most fitness evaluations, roughly nine times the number of evaluations required needed by S_5. The downside to S_5 is that it had the lowest success rate out of all architectures. As the factor size decreased from S_9 to S_5, both the number of successful trials and number of evaluations steadily decreased. From S_5 to S_2, the number of successful

trials and fitness evaluations increased, with the largest increase occurring from S_5 to S_4 followed by a slight decrease from S_4 to S_3.

Finally, on the DecTwoTrap function, the larger factor architectures performed the best. The only two architectures that were unable to locate the optimal solution 100% of the time were S_3, and S_2, which found the optimal solutions only 98% and 87.5% of the time. Similar to the DecTrap function, FEA and the full GA were always able to locate the optimal solution in one iteration with the the differences in the number of fitness evaluations being caused by extra fitness evaluations during FEA's initialization.

7.2.1.2 Analysis In the OneMax function, the factors of smaller size performed better than the larger factors. This supports our hypothesis that the variables in the OneMax functions are independent of one another. Conversely, the factor architectures with large factors outperformed those with smaller factors on all other functions. This suggests that even though all functions are based upon the number of 0's and 1's, by requiring specific values from all or some of the variables, a relationship is created between the variables.

Another result is that on most functions, the best algorithm was the full GA. Only on the OneMax and TwoTrap functions did FEA outperform the single-population. FEA's superior performance on the OneMax is due to the Compete step, a Greedy algorithm that is able to join together the best solutions from individual factors quickly. The probability of a solution of with ten 1's being generated is $\frac{1}{2^{10}} = 0.097\%$, whereas the probability of a subsolution with two 1's is $\frac{1}{4} = 25\%$. Given these odds, there is a high likelihood that each of the factors will contain the optimal subsolution, which Compete is then able to combine together into the optimal

solution. Conversely, the full GA can rely only on crossover and mutation to locate the optimal solution.

The TwoTrap function demonstrates the drawback of using a greedy method for the Compete algorithm. As the population size decreases, a greater emphasis is placed on Compete being able to assemble good solutions. Because the majority of the search space leads to the suboptimal solution, the likelihood of Compete finding the optimal solution is small. Additionally, once Compete locates the suboptimal solution, the full global solution is unable to leave the suboptimal solution because finding a better solution requires changing more than one bit. Only when the full global solution is initialized in a region that leads to the optimal solution are the smaller factors for FEA able to find the optimum. This explains why the factor architectures with smaller factors found the optimal solutions the least number of times but were able to do so in the fewest iterations.

Meanwhile, FEA with the S_9 factor architecture is able to locate the optimal solution more often than the full GA is because FEA has a smaller search space to explore to locate a solution in the optimal region. Conversely, S_8 has less chance of success because it requires the both of the variables in the smaller factor to be 0s or 1s, instead of just one variable being 0 or 1, as is the case in S_9.

From these results, we conclude that in the majority of the functions, the larger factors perform better because of the relationship between all the variables. However, when a relationship does not exist between all variables, such as in OneMax, FEA performs better than the single-populations.

7.2.2 Varying the Number of Overlapping Factors

While we have shown that larger factors perform better than smaller factors on the majority of the basic unitation functions, the question remains, how do the

Table 7.2: Set of factor architectures varying the number of overlapping factors.

Name	Number of Factors (M) and Size (N): (M, N)
F	(1,10)
F_2	(2,10)
F_3	(3,10)
F_4	(4,10)
F_5	(5,10)
F_6	(6,10)

number of overlapping factors affect the performance of FEA? Furthermore, in the previous section, there was no competition between factors because there was no overlap. Compete was responsible only for assembling the full global solution. Here, we are able to directly examine how effective Compete is at determining optimal values for the full global solution.

In these experiments we investigate how increasing the number of overlapping factors affects the performance of FEA. Starting with two factors optimizing over all ten variables, we increase the number of factors from two to five. By increasing the number of factors, we are able to evaluate how FEA is able to utilize multiple values for each variable during Compete. Table 7.2 presents the different factor architectures along with each architecture's name. We also include results of a single-population GA for reference.

As in the previous experiments, we used a Genetic Algorithm (GA) as the underlying optimization algorithm with tournament selection and uniform crossover with a mutation rate of 0.1. All algorithm architectures used a budget of 250 individuals distributed evenly between all the factors. Each function was of ten variables. FEA and the single-population GA were run until the optimal solution was found. However, each algorithm was given a budget of fitness evaluations for performing Update and Compete. All results are averages from 200 trials.

7.2.2.1 Results Figure 7.12 presents the number of successful trials required to find the optimal solution, whereas Figure 7.13 presents the average number of fitness evaluations required for each factor architecture to locate the optimal solution. The y-axis in Figure 7.12 corresponds to the number of successful trials, whereas the y-axis in Figure 7.13 corresponds to the number of fitness evaluations. The x-axis lists the the different factor architectures, starting with the full GA (F) and increasing the number of overlapping factors, F_1, F_2, ..., F_6. The average number of fitness evaluations is calculated only from trials that were successful in locating the optimal solution. Additionally, the standard error is shown for the number of fitness evaluations.

The first result we make note of is that the best architecture on the OneMax was F_4, which required the fewest number of fitness evaluations. On the other functions, FEA was often outperformed by the single-population GA in terms of number of fitness evaluations required to find the optimal solution. However, this difference was not always significant. For example, on the Needle and BiNeedle functions, F_2 required more evaluations than the full GA, but both values were within the standard error. Only with the DecTrap and DecTwoTrap is there a clear difference between F and F_2.

Another result is that there appears to be a correlation between the increase in both the number of factors and the number of fitness evaluations. However, there are a few exceptions, such as F_4 in the Needle function or F_3 and F_5 on the BiNeedle function. The only function that does not display this correlation is the DecTrap function, where there is an increase from F to F_2, but then the number of evaluations levels out.

Finally, we note that each architecture was able to locate the optimal solution almost 100% of the time. Only with the DecTrap function did FEA and the full GA

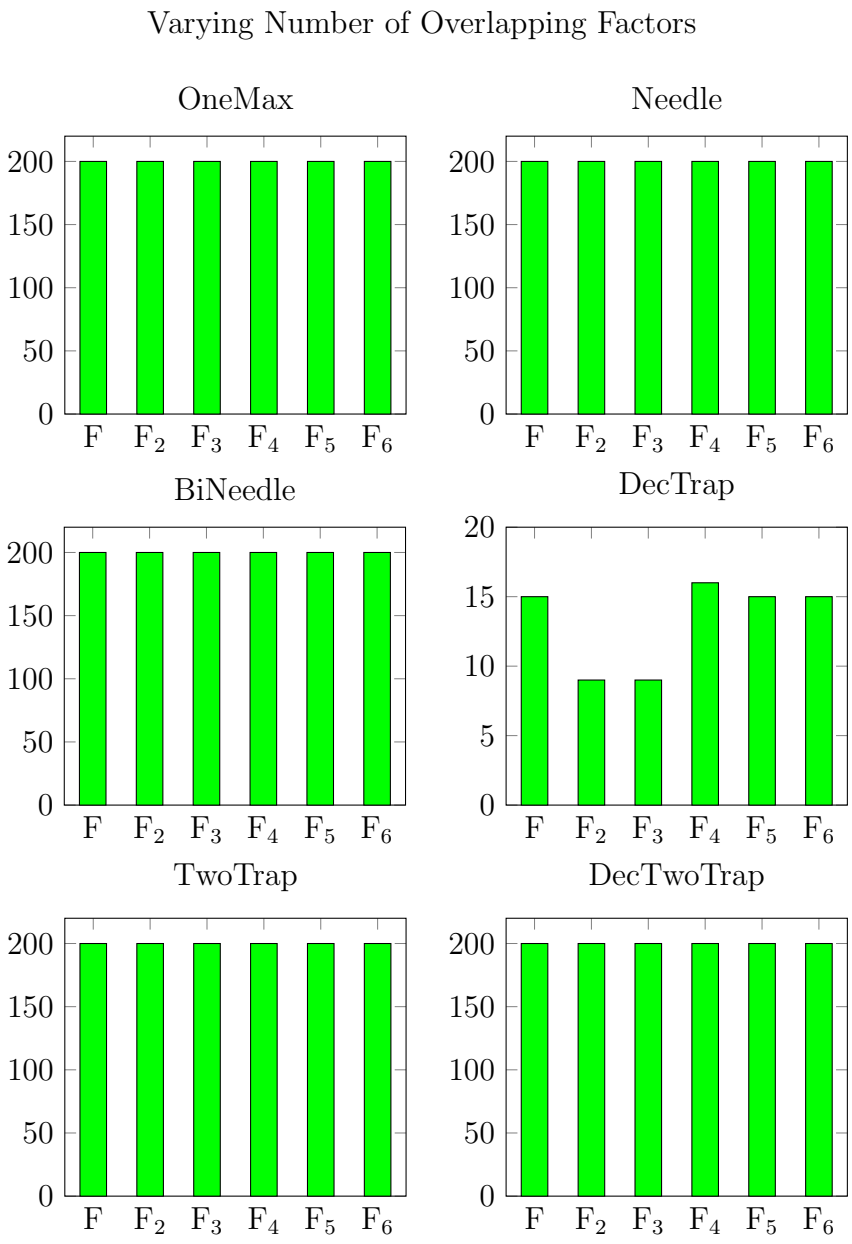


Figure 7.12: Number of successful trials of FEA with factors of decreasing overlap on unitation functions.

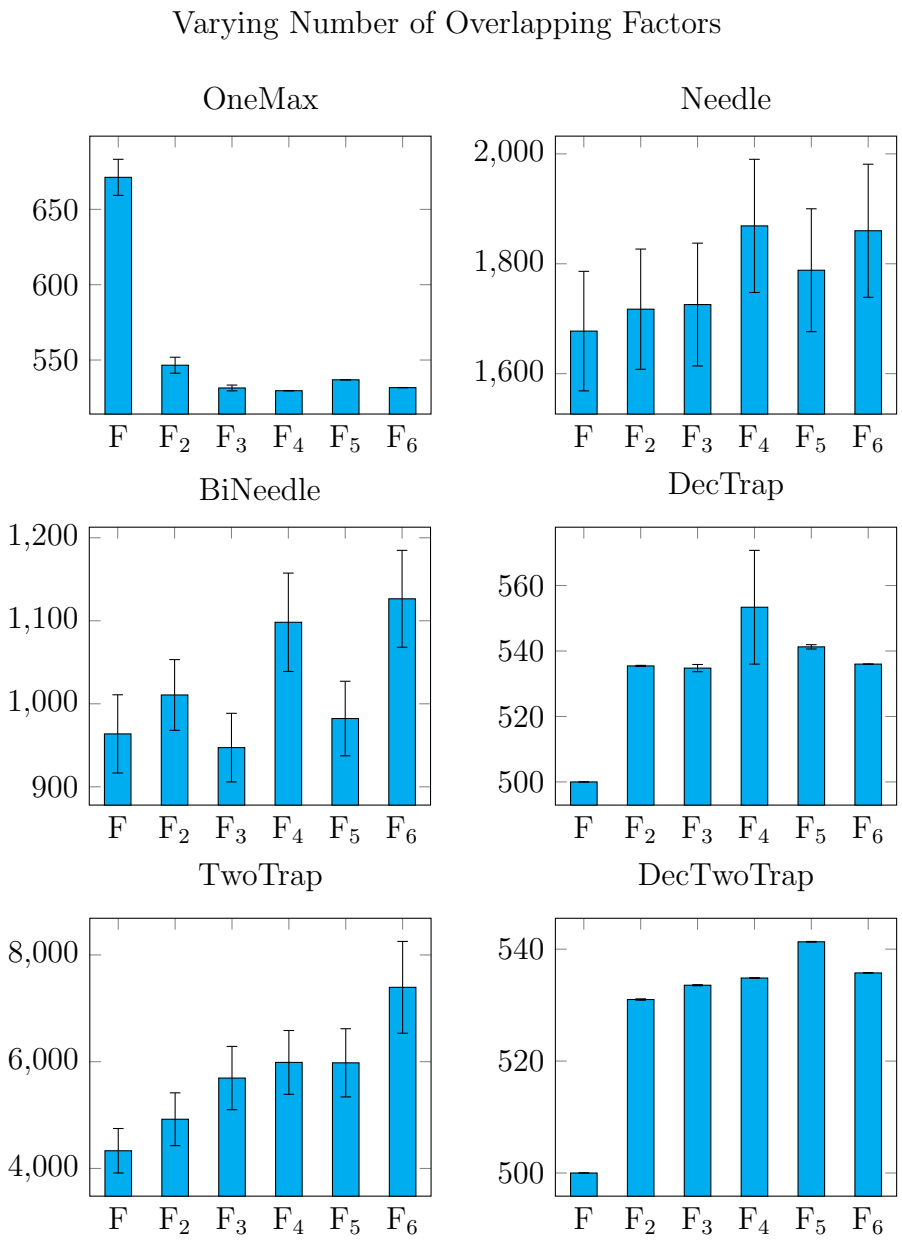


Figure 7.13: Average number of fitness evaluations of FEA with factors of decreasing overlap on unitation functions.

struggle to find the optimal solution, where F_4 was the best architecture for locating the optimal solution.

7.2.2.2 Analysis From the results varying the number of overlapping factors, we discovered that increasing the number of factors does not increase the performance of FEA on most of the base unitation function. Only when there exists a clear gradient to the optimal solution does adding additional overlapping factors increase FEA's performance.

For the remaining functions, the algorithms are either misled to suboptimal solutions or are forced to randomly explore a flat basin to locate a gradient. There are two possible ways for FEA to locate the optimal solution: through an individual factor locating the solution independent of all factors or by Compete piecing together good values from individual factors. This leads us to two possible explanations for FEA's poor performance on the unitation functions. The first is that by distributing 250 individuals between the factors, the individual factors are not as efficient as the full GA at finding the optimal solutions. Ideally, we would expect FEA to be able to overcome the difference between the factor sizes and the full global solution by having factors collaborating with one another. FEA performs this collaboration through Compete, which leads us to the our second explanation for the poor performance of FEA: Compete is unable to efficiently combine values from overlapping factors.

The greedy search in Compete makes it impossible for the full global solution to escape the suboptimal solution when more than one variable must be changed simultaneously. Therefore, if the full global solution becomes located at a suboptimal point, the only way for FEA to locate the optimal solution is if a factor locates the optimal solution on its own. But because each factor has fewer individuals than a single-population GA, the likelihood of FEA finding the optimal solution decreases.

In reviewing logs from the individual trials, we discovered that in the vast majority of trials, the optimal solution was located by an individual factor in FEA.

From these results, we conclude that Compete is only able to efficiently combine values from overlapping factors if there exists a gradient for Compete to follow. Additionally, these results demonstrate that Compete is susceptible to becoming trapped in suboptimal solutions. When the full global solution becomes trapped at a suboptimal point, the only way for FEA to escape is for an individual factor to locate a better point.

7.2.3 Varying Factor Overlap and Size

While the previous experiment varied the number of overlapping factors, we did not vary the amount of overlap between the two factors. Additionally, because each factor optimized over all ten variables, the factors were unaffected by changes to the full global solution after Compete. In these experiments, we vary the size of factors and the amount of overlap between factors.

To do so, we use two factors of equal size and vary the number of variables each factor is optimizing and the amount of overlap between factors. Starting with two factors of size nine, we position each factor at both ends of the ordered variables. In this case, factor one optimizes variables one - nine while factor two optimizes variables two-eight, which gives an overlap of eight. We then decrease the size of each factor, starting with nine and moving down to five, where there is no overlap between the factors. Table 7.3 presents the name and structure of the five different architectures.

All experiments used a GA with tournament selection, uniform crossover with a mutation rate of 0.1, and 250 individuals. FEA was also given 250 individuals that were distributed between the two factors. Both the GA and FEA were allowed 35,000 fitness evaluations to find the optimal solution.

Table 7.3: Set of factor architectures varying factor size and overlap between two factors

Name	Number of Factors (M) and Size (N): (M, N)	Overlap
F	(1,10)	—
T_9	(2,9)	8
T_8	(2,8)	6
T_7	(2,7)	4
T_6	(2,6)	2
T_5	(2,5)	0

7.2.3.1 Results Figure 7.14 presents the number of successful trials for the various FEA factor architectures over each of the unitation functions and Figure 7.15 shows the average number of fitness evaluations for successful trials. The y-axis in Figure 7.14 corresponds to the number of successful trials, and the y-axis in Figure 7.15 corresponds to the number of fitness evaluations. The x-axis lists the the different factor architectures, starting with the full GA (F) and increasing the number of overlapping factors, T_1, T_2, ..., T_6. The average number of fitness evaluations is calculated only from the trials (out of 200) that were successful in locating the optimal solution. Additionally, the standard error is shown for the number of fitness evaluations.

In the OneMax function, the factor architecture T_6 performed the best, requiring the fewest number of fitness evaluations. On the Needle and BiNeedle functions, T_9 and T_8 tied with the full GA (F). However, as the factor size and overlap decreased from T_8 to T_5 for the Needle and BiNeedle, the number of fitness evaluations also increased.

The DecTrap function was the most difficult function to optimize. On average, each factor architecture was able to locate the optimal solution only approximately 5% of the time. However, when FEA and the full single-population GA found the

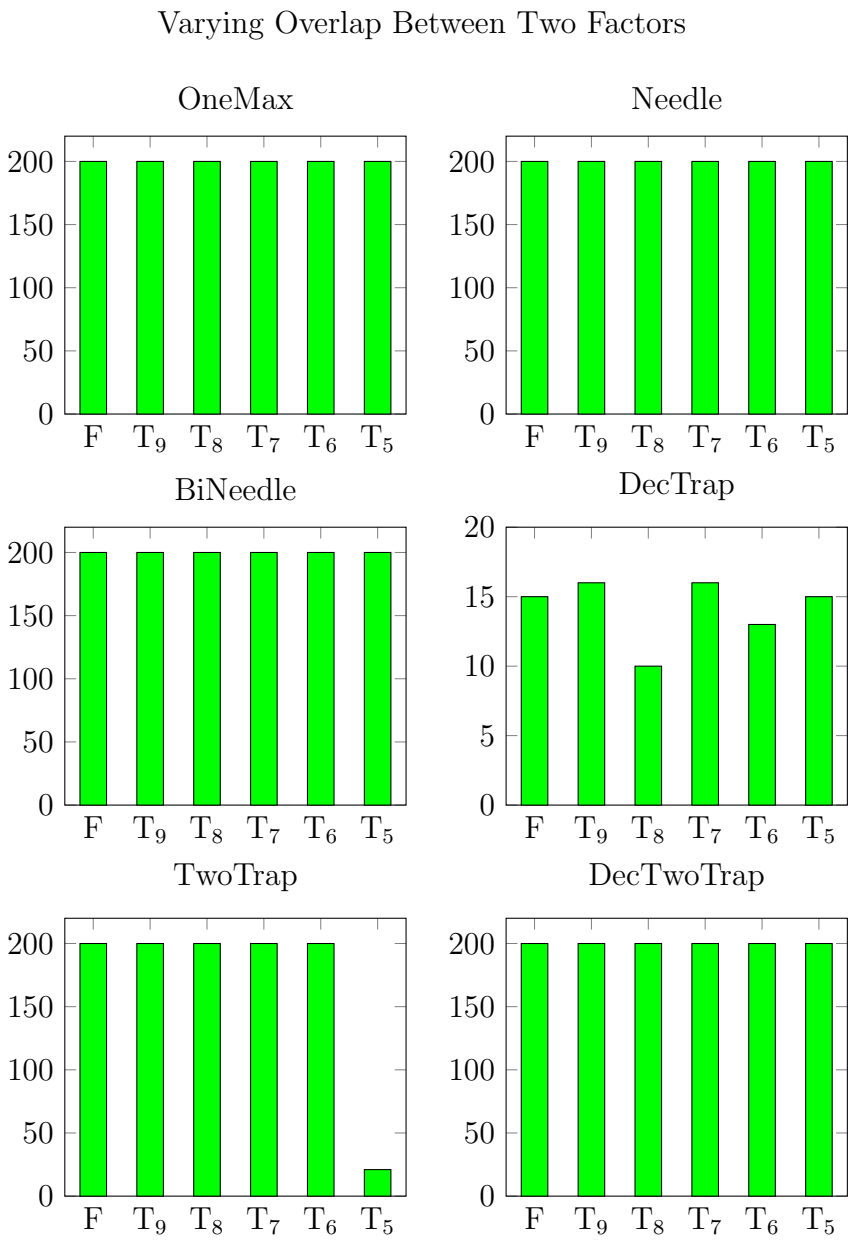


Figure 7.14: Number of successful trials of FEA with two factors of decreasing overlap on unitation functions.

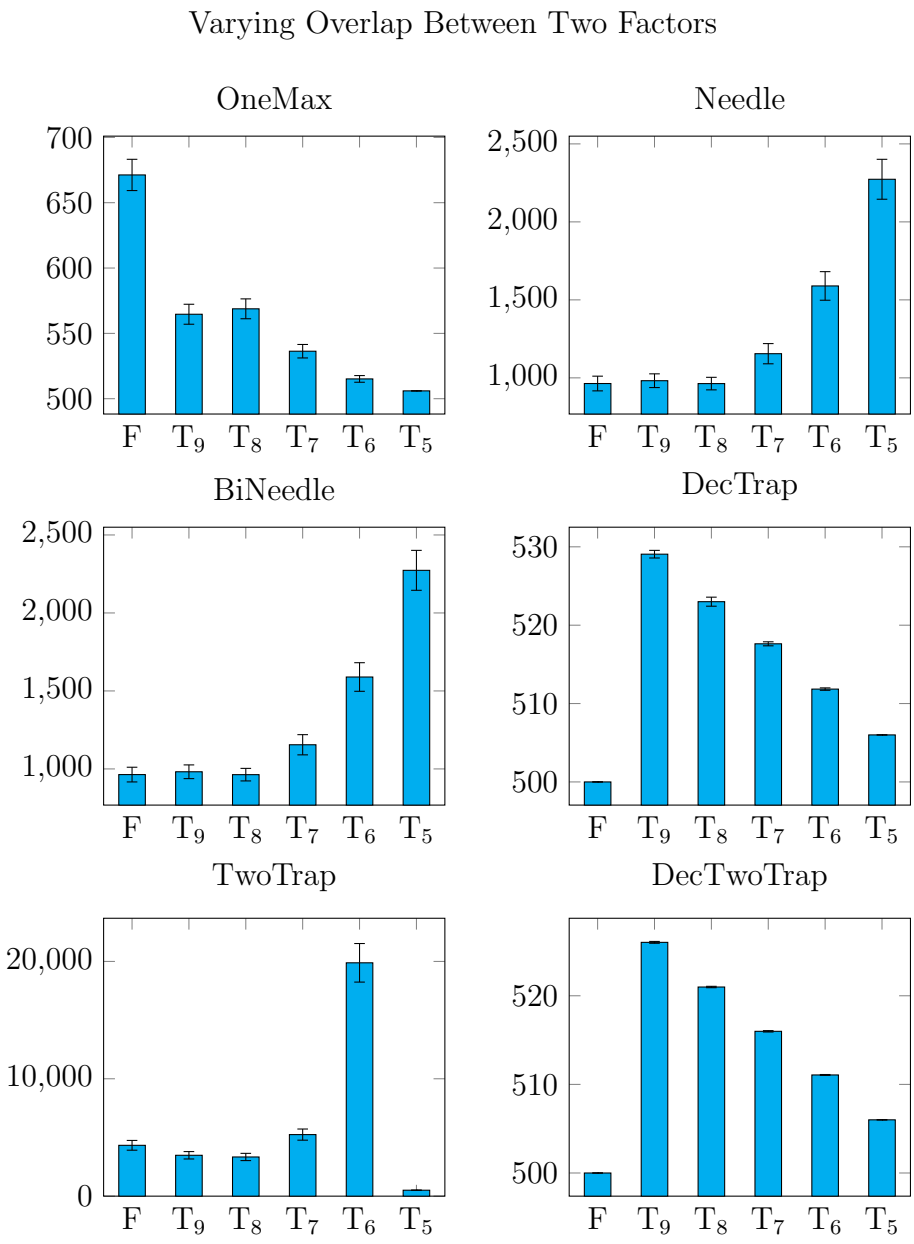


Figure 7.15: Average number of fitness evaluations of FEA with two factors of decreasing overlap on unitation functions.

optimal solution, it was done in only one iteration. The differences between the number of fitness evaluations is due to Compete evaluating different values from the factors. A similar result can be seen in the DecTwoTraps, where each architecture was able to locate the optimal solution within one iteration. The difference, however, is that each architecture was able to locate the optimal solution every time.

Finally, on the TwoTrap function, the best architectures were T_8 and T_9, which required the fewest number of fitness evaluations, taking only 11.5 iterations to find the optimal solution. As the amount of overlap decreases, the performance begins to decrease as well. T_5 was the worst factor architecture, which was able to find the optimal solution only 21 times. However, when it did so, it required only one iteration.

7.2.3.2 Analysis From the previous results, the best factor architectures has two large factors with a large amount of overlap, which reaffirms the results from the previous sections. In the majority of the functions, T_9 required the fewest fitness evaluations. Only on the OneMax was T_9 outperformed by a significant margin. While T_9 required the most evaluations on DecTrap and DecTwoTrap, the architecture was still able to find the optimal solution within one iteration. Additionally, the total number of evaluations was relatively small.

We believe that this is because the larger factors are able to balance the benefits of having a global view of the entire function while decreasing the number of variables to optimize. For example, in the Needle function for T_9, the last factor only has to find a solution that has a 1 for the last variable in order for the first factor to able locate the optimal solution. This is because the second factor is the only factor optimizing over the last variable. Likewise, the first factor only has to find a solution with a 1

for the first variable in order for the second factor to be able to locate the optimal solution.

Additionally, these larger factors remove the burden from Compete to piece together a full global solution that places the factors in a good subregion in the search space. In T_9, there was only one variable that each of the swarms were not optimizing; therefore, all Compete had to do is choose a good value for the last variable. But because there was only one factor optimizing the last variable, Compete used the best value found by the other factor.

From these results, we conclude that the Compete algorithm is not effective at piecing together good solutions in unitation functions that have a high degree of variable interaction. This is because Compete is able to change only one variable at a time; therefore, if multiple variables need to be changed simultaneously in order for the full global solution to move from a suboptimal solution, Compete is unable to do so. Finally, we determined that smaller factors are effective only when the problem is decomposable. When high interaction occurs between variables, the smaller variables are unable to efficiently interact with one another to locate good solutions. This possibly explains why the Parents factor architecture for abductive inference is less effective than the Markov architecture. The smaller factors are likely getting stuck in flat regions or a local minimum.

7.3 FEA on Royal Road

From the previous results, we were able to show that Compete is often unable to effectively create optimal solutions on the base unitation functions. This places a greater emphasis on generating good factor architectures. In this section, we examine what comprises a good factor architecture and why. To do so, we evaluate the

performance of FEA on the Royal Road functions. Furthermore, this will also help explore other functions that Compete is more effective at optimizing.

For the Royal Road problems, we used R1, R2, and R3 defined in Figures 7.7, 7.8, and 7.9. Each algorithm was run until the optimal solution was found; however, each algorithm was given a budget of 500,000 evaluations before terminating the algorithm. All results are over 30 trials. In FEA, factors performed one round of updates before Compete and Share were performed. Factor sizes were set to 2, 4, 8, and 16 with an overlap of 1, 2, 4, and 8, respectively. We also include results where there is no overlap between factors, allowing us to investigate the previous results by Ochoa *et al.* [71].

In all of these experiments, we use a GA as the underlying optimization algorithm. The GA used in our experiments used tournament selection and uniform crossover with a mutation rate of 0.015. In all algorithms, we used a budget of 250 individuals distributed evenly between all the factors. We also present results for a full, single-population GA.

7.3.1 Results

Table 7.4 presents the results for the different factor architectures on the Royal Road functions. "Success" is the number of times the algorithm was able to locate the optimal solution. "Evals" is the average number of evaluations required to find the optimal solution with the standard error shown in parentheses. A bold value indicates the algorithm was significantly better than all other algorithms using a Paired Student t-Test with $\alpha = 0.05$. If more than one algorithm is in bold, then there was no significant difference between algorithms, but they were still statistically better than all other algorithms. Note that the mean and standard error were calculated only for the trials that were able to successfully locate the global optimums, which is why

Table 7.4: Average number of evaluations for FEA to find optimal solution on Royal Road functions.

	R1		R2		R3	
	Success	Evals	Success	Evals	Success	Evals
Full	17	4.28E+4(7.99E+3)	30	3.24E+4(3.64E+3)	2	4.50E+4(3.34E+4)
Size = 2 Overlap = 0	29	2.54E+5(1.65E+4)	28	2.36E+5(2.09E+4)	1	5.36E+4(NA)
Size = 4 Overlap = 0	30	9.77E+4(7.70E+3)	30	1.08E+5(9.69E+3)	4	9.41E+4(4.24E+4)
Size = 8 Overlap = 0	30	**3.39E+4(3.41E+3)**	30	4.26E+4(3.99E+3)	1	8.34E+3(NA)
Size = 16 Overlap = 0	30	**2.28E+4(2.56E+3)**	30	**2.26E+4(2.12E+3)**	4	2.12E+4(6.05E+3)
Size = 2 Overlap = 1	27	3.27E+5(2.81E+4)	27	3.64E+5(2.16E+4)	1	2.48E+5(NA)
Size = 4 Overlap = 2	30	1.59E+5(1.48E+4)	30	1.36E+5(1.47E+4)	2	1.58E+5(1.77E+3)
Size = 8 Overlap = 4	30	7.27E+4(8.87E+3)	30	5.88E+4(5.19E+3)	1	1.05E+5(NA)
Size = 16 Overlap = 8	30	**4.56E+4(5.22E+3)**	30	3.57E+4(4.15E+3)	4	1.92E+4(5.73E+3)

certain algorithms have a standard erorr of "NA" on R3. In FEA, this also includes evaluations used during the competition phase. We note that each algorithm required 250 fitness evaluations during the initialization of the population.

In R1, the full single-population GA (Full) was less effective than every factor architecture, because it was only able to locate the optimal solution 17 times. The best architectures had size = 8, overlap = 0; size = 16, overlap = 0;and size = 16, overlap = 8, which required the fewest number of fitness evaluations by a significant margin. The worst architecture was size = 2, overlap 1, which was able to find the optimal solution only 27 of 30 times. Additionally, it required the most fitness evaluations to do so.

On the R2 function, the single-population GA had a much easier time locating the optimal solution, because it was able to locate the optimal solution 100% of the time. Size = 2, overlap = 1 was still the worst factor architecture, locating the optimal

solution only 27 times and requiring the most fitness evaluations to do so. The best architecture had size = 16, overlap = 0, requiring the fewest number of evaluations.

The R3 function, on the other hand, presented a much tougher challenge for the Full and FEA versions of GA. Note that because several of the architectures were able to locate the optimal solution only once, hypothesis testing was not performed between the architectures. In terms of locating the optimal solution, both size =16, overlap = 8 and size =16, overlap = 0 had the best performance, as they successfully located the optimal solution four times. Additionally, when they did locate the optimal solution, they required the second-fewest number of fitness evaluations. We make note that the difference between the number of evaluations for these architectures was not significant. In terms of finding the best solution, the size = 8, overlap = 0 architecture performed the best, requiring only 8400 fitness evaluations. However, it only did so once. The worst performing architecture was size = 2, overlap = 0, as it only located the optimal solution once and required the most evaluations to do so.

To further help interpret the results, particularly on R3, we also present the average fitness of the Full and FEA versions of GA on the Royal Road in Table 7.5, which allows us to see the average performance of the algorithms over the 30 trials. From this table, we can see that the best architecture was size = 16, overlap = 8 followed by size = 8, overlap = 0. While most architectures achieved perfect performance on R1 and R2, these two architectures had the best performance in terms of fitness for R3. Meanwhile, size = 8, overlap = 4 had the worst performance in terms of average fitness.

Table 7.5: Average fitness of FEA on Royal Road functions.

	Full	Size = 2 Overlap = 0	Size = 4 Overlap = 0	Size = 8 Overlap = 0	Size = 16 Overlap = 0	Size = 2 Overlap = 1	Size = 4 Overlap = 2	Size = 8 Overlap = 4	Size = 16 Overlap = 8
R1	204.00	252.00	256.00	256.00	256.00	244.00	256.00	256.00	256.00
R2	128.00	123.20	128.00	128.00	128.00	120.80	128.00	128.00	128.00
R3	139.47	122.13	127.47	130.67	125.33	118.40	126.67	109.33	137.60

7.3.2 Analysis

From these results, we can make several observations. The first is that the larger factor architectures performed the best on all Royal Road functions. We believe this is because these larger factors map more closely to the basic building blocks in the Royal Road functions. For example, in size = 2, overlap = 0, an individual factor must rely on six other factors locating all 1's at the same time in order to satisfy the smallest building block. Conversely, the larger factors are able to find the good schema without requiring that other factors be located in specific positions.

This also verifies our results from the base unitation functions: Compete is effective at finding good solutions only when single variable changes have a direct affect on the fitness. For example, to locate the smallest schema in the size = 2, overlap = 1, Compete has to explore a flat basin until eight 1's are located, similar to the Needle function. As demonstrated in the results of FEA on the Needle function, Compete struggles at being able to piece together values when there is not a gradient to follow.

Another observation is that the results in Table 7.4 differ from those of Ochoa *et al.* [71]. In that work, the authors found that the optimal architecture was size = 4, overlap = 0. However, we found that the best non-overlapping architecture was size = 16. As earlier stated, we believe that the larger factor sizes allow the individual factors to have a better global picture of the fitness landscape. Previous chapters analyzing the factor architectures demonstrated that while smaller factors

do decrease the probability of hitchhiking, they do not necessarily maintain better diversity. The results presented in this chapter suggest that smaller factor sizes do not allow a large enough picture of the search space.

While we have shown that the best factor architecture for Royal Road is one that maps factors to base schema, this assumes knowledge of the schemata in the function. However, the schemata are not usually known, which is why much of the work in this dissertation and other related work on CCEA's is focused on testing factor architectures. This also offers an explanation for why the Markov blanket factor architecture on the probabilistic graphical models outperformed other architectures: the Markov blanket acts as a building block in a Bayesian network.

On comparing the overlapping factors versus the non-overlapping factors, we discovered that overlap was almost always a detriment to the performance of FEA on the Royal Road functions. The only time overlap did not hurt performance was for size = 16, overlap = 8. This is because the overlapping factors are attempting to locate the two halves of the different schema.

R3 further demonstrates that having factors optimize over two halves negatively impacts the performance. This function contained groups of "negative" schemata that caused all of the architectures to be misled. In size = 8, overlap = 4, this means that there are factors optimizing over the variables within the deceptive schema. A factor optimizing over those variables is more likely to locate a schema of 0's instead of one with all 1's, because the factor is able to see an increase in fitness independent of the values in the full global solution.

This discovery introduces another design decision when creating factor architectures. The architecture must not only generate factors that optimize over groups of related variables, but it should do so only for variables that are positively correlated. This is one possible explanation as to why the Markov blanket architecture in previous

chapters did not perform well on the Rosenbrock function. While a relation in the Rosenbrock exists between variables X_i and X_j, by searching over both variables simultaneously, the factors were misled into suboptimal solutions.

7.4 Conclusion

In this chapter, we evaluated the peformance of FEA on two sets of binary problems: unitation and Royal Road functions. From the unitation functions, we discovered that there are several instances where the Compete step in FEA struggles to build good solutions. In the Royal Road functions, we demonstrated for the regular Royal Road functions, the optimal factor architecture is one that maps factors directly to schemata. However, we also discovered scenarios where mapping factors directly to schemata decreased the performance FEA. These negative schemata represent another design decision to consider when generating a factor architecture.

CHAPTER EIGHT

CONCLUSIONS

In this chapter, we review the work presented in this dissertation along with how each result contributes to the field of Computer Science. Additionally, we present several areas of future work.

8.1 Contributions

In this dissertation, we introduced Factored Evolutionary Algorithms (FEA) and explored various properties of the algorithm. FEA is unique in that it factors the function being optimized by creating subpopulations that optimize over a subset of dimensions of the function. However, unlike other optimization techniques that subdivide optimization problems, FEA encourages subpopulations to overlap with one another. This allows the subpopulations to Compete and Share information.

In Chapter 2, we reviewed the necessary background in order to understand the contributions of this dissertation along with related work. Next, we presented a framework for defining multiple populations EAs in Chapter 3. We then demonstrated how several of the approaches discussed in the relation work can be mapped to this framework. This contributes towards to the field of evolutionary computation by establishing the relationship between multi-population algorithms, regardless of the underlying optimization algorithm or number of variables a subpopulation is optimizing over. We also presented a formal definition of FEA and a complexity analysis of each of its subfunctions, which provides a better understanding of FEA by showing the functions of FEA that have the largest influence on its runtime. Finally,

we demonstrated the generality of FEA by demonstrating how FEA outperforms the single-population and CCEA versions of Hill Climbing, Genetic Algorithms, Differential Evolution, and Particle Swarm Optimization. This contributes to a better understanding that the benefits of FEA are not specific to only one optimization method.

Chapter 4 explored different factor architectures for FEA and found that the optimal architecture for FEA might be based upon the Markov blanket in Bayesian networks. These results advance the knowledge of how to create subpopulations for multi-population algorithms efficiently. We then formalized this relationship in by proving that optimizing certain fitness functions is equal to abductive inference in factor graphs. Additionally, we further empirically explored why the Markov factor architecture outperforms other architectures. These results further our understanding of the relationship between optimization problems and probabilistic graphical models.

Chapter 5 examined the performance of different discrete PSO algorithms and proposed a new discrete PSO algorithm. ICPSO provides a better understanding of discrete PSO algorithms and general guidelines on how the different algorithms should be used. Additionally, we performed experiments analyzing the performance of FEA versions of the different discrete PSO algorithms. These results further improve our understanding of FEA's performance advantages.

In Chapter 6 we explored various convergence properties of FEA and found that, although FEA is susceptible to pseudominima, the likelihood of becoming stuck in a pseudominima is often small. These results further our understanding of the impact of an overlapping factor architecture on the performance of FEA. Finally, we examined the performance of FEA on unitation and Royal Road functions and discovered instances where the Compete stage in FEA is unable to provide benefit. These results provide a better understanding of functions for which Compete can and

cannot provide benefits. In addition, these results provide a relationship between Bayesian networks and Royal Road functions.

8.2 Future Work

We have targeted several different areas for future research. The first area is learning the factor architecture. In all of the experiments in this dissertation, we assumed that we were able to directly observe the fitness function and its variables. However, there are often cases where the function is a black box. For these cases, it will be necessary to learn a probabilistic graphical model that can then be used as a factor architecture.

We propose two two to accomplish this. The first uses a pre-processing step to generate samples from the fitness function and then uses those samples to learn the structure of a probabilistic graphical model. Another approach modifies FEA to use dynamic factors that are modified during the search process. This is similar to the work done by Qureshi and Sheppard, who proposed an OSI algorithm for training a Neural Network that created and deleted factors based on randoms paths through the neural network [85]. However, this sampling was done randomly and lacked any structure for creating and deleting the factors. Additionally, the main focus of the work was the reduction of the computational burden that was encountered by Ganesan Pillai and Sheppard [80].

One possible way to dynamically learn a factor architecture while optimizing the function would be to modify the algorithm by Qureshi and Sheppard so that the factors would be mutated based on measuring variable correlation. Variables with low correlation would be separated while variables with high correlation would be combined into a single factor. This concept is similar to EDA's, where fit samples

generated from a probabilistic graphical model are used to update the structure and parameters of the network [78].

We would like to explore modeling the search process of FEA. One of the most common methods for modeling evolutionary search algorithms is Markov chain analysis, which models the probability of all states of a random variable over time. Because FEA contains multiple populations, each population would need to be modeled by its own Markov chains. Additionally, although there are periods where the factors update their individuals independent of one another, the Compete and Share steps in FEA will also affect the probability of the individual Markov chains.

A Dynamic Bayesian Network (DBN) is an extension of a Markov chain that allows for more complex representations of random variables over time. One way that the search process of FEA can be modeled is by allowing each factor to represent a random variable in the DBN. Additionally, a random variable could be added for the Compete and Share functions. Links would then be added between the random variables for factors and the random variables for Compete and Share.

Given this structure, the conditional probability distributions are then set based upon the underlying search algorithms and the factor sizes and overlap. Once the network has been parameterized, inference can then be performed to evaluate the probability of FEA to locate a particular solution. This provides another framework to compare different architectures. For example, suppose the function being optimized is R1. The DBN for the factor architecture of size = 2, overlap = 0 is going to be different that the DBN for size = 8, overlap = 0. Given these two DBNs, one could calculate the expected number of iterations required for each of the DBNs to reach a goal state. Additionally, this could be used to prove that specific architectures are superior.

A third possible area for future research is the investigation of how FEA is able to avoid pseudominima. In Chapter 8, we demonstrated that the Markov architecture is able to avoid pseudominima. However, no formal analysis was performed to understand why this is the case. By using a DBN, we could evaluate the probability of FEA becoming trapped at a pseudominima and examine the conditions that caused FEA to become stuck.

Finally, we would like to explore different algorithms for performing competition. As shown in Chapter 9, the Compete step in FEA may be trapped when only one variable is changed and there is no gradient to follow. There are several different changes that could be made to Compete. For example, instead of using the best known solution from a factor, competition could be performed over the current best individual or a randomly selected individual. Additionally, Compete could select the value for inclusion into the full global solution based on either roulette wheel or tournament selection instead of the greedy selection that is currently used.

However, all of these changes are based upon Compete performing competition one variable at a time and may not address the issues highlighted in Chapter 9. One possible way to address the poor performance of Compete on the unitation functions would be to allow competition on multiple variables simultaneously. For example, Compete could evaluate every combination of values for two or more variables. In the extreme case, Compete could use a GA or PSO for a limited number of iterations to select values for the full global solution.

REFERENCES CITED

[1] P. Abbeel, D. Koller, and A. Y. Ng. Learning factor graphs in polynomial time and sample complexity. *The Journal of Machine Learning Research*, 7:1743–1788, 2006.

[2] H. Aguirre and K. Tanaka. A study on the behavior of genetic algorithms on NK-landscapes: Effects of selection, drift, mutation, and recombination. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 86(9):2270–2279, 2003.

[3] S. Baluja. Population-based incremental learning: A method for integrating genetic search based function optimization and competitive learning. Technical Report ADA282654, DTIC Document, 1994.

[4] S. Baluja and S. Davies. Using optimal dependency-trees for combinatorial optimization: Learning the structure of the search space. In *Proceedings of the Fourteenth International Conference on Machine Learning*, 1997.

[5] T. C. Belding. The distributed genetic algorithm revisited. In *Proceedings of the 6th International Conference on Genetic Algorithms*, pages 114–121. Morgan Kaufmann Publishers Inc., 1995.

[6] E. Bengoetxea and P. Larrañaga. EDA-PSO: a hybrid paradigm combining estimation of distribution algorithms and particle swarm optimization. In *Swarm Intelligence*, pages 416–423. Springer, 2010.

[7] U. Brandes and C. Pich. Eigensolver methods for progressive multidimensional scaling of large data. In *International Symposium on Graph Drawing*, pages 42–53. Springer, 2006.

[8] A. E. Brownlee, J. A. McCall, and Q. Zhang. Fitness modeling with Markov networks. *IEEE Transactions on Evolutionary Computation*, 17(6):862–879, 2013.

[9] E. K. Burke, S. Gustafson, and G. Kendall. Diversity in genetic programming: An analysis of measures and correlation with fitness. *IEEE Transactions on Evolutionary Computation*, 8(1):47–62, 2004.

[10] S. Butcher, S. Strasser, J. Hoole, B. Demeo, and J. Sheppard. Relaxing consensus in distributed factored evolutionary algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pages 5–12. ACM, 2016.

[11] E. Cantú-Paz. A survey of parallel genetic algorithms. *Calculateurs paralleles, reseaux et systems repartis*, 10(2):141–171, 1998.

[12] S. Chen, J. Montgomery, and A. Bolufé-Röhler. Measuring the curse of dimensionality and its effects on particle swarm optimization and differential evolution. *Applied Intelligence*, 42(3):514–526, 2015.

[13] W.-N. Chen, J. Zhang, H. S. Chung, W.-L. Zhong, W.-G. Wu, and Y.-H. Shi. A novel set-based particle swarm optimization method for discrete optimization problems. *IEEE Transactions on Evolutionary Computation*, 14(2):278–300, 2010.

[14] C. Chow and C. Liu. Approximating discrete probability distributions with dependence trees. *IEEE Transactions on Information Theory*, 14(3):462–467, 1968.

[15] M. Clerc. Discrete particle swarm optimization, illustrated by the traveling salesman problem. In *New optimization techniques in engineering*, pages 219–239. Springer, 2004.

[16] M. Clerc and J. Kennedy. The particle swarm-explosion, stability, and convergence in a multidimensional complex space. *IEEE transactions on Evolutionary Computation*, 6(1):58–73, 2002.

[17] G. F. Cooper. The computational complexity of probabilistic inference using bayesian belief networks. *Artificial intelligence*, 42(2-3):393–405, 1990.

[18] B. Coppin. *Artificial intelligence illuminated*. Jones & Bartlett Learning, 2004.

[19] P. Dagum and M. Luby. Approximating probabilistic inference in bayesian belief networks is np-hard. *Artificial intelligence*, 60(1):141–153, 1993.

[20] S. Das and P. N. Suganthan. Differential evolution: A survey of the state-of-the-art. *IEEE Transactions on Evolutionary Computation*, 15(1):4–31, 2011.

[21] J. S. De Bonet, C. L. Isbell, P. Viola, et al. Mimic: Finding optima by estimating probability densities. *Advances in Neural Information Processing Systems*, pages 424–430, 1997.

[22] K. A. De Jong. *Evolutionary computation: a unified approach*. MIT press, 2006.

[23] R. Dechter. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence*, 113(1):41–85, 1999.

[24] R. Dechter and I. Rish. Mini-buckets: A general scheme for bounded inference. *Journal of the ACM (JACM)*, 50(2):107–153, 2003.

[25] M. Dorigo, V. Maniezzo, and A. Colorni. Ant system: optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, 26(1):29–41, 1996.

[26] R. C. Eberhart and Y. Shi. Comparing inertia weights and constriction factors in particle swarm optimization. In *Proceedings of the Congress on Evolutionary Computation (CEC)*, volume 1, pages 84–88. IEEE, 2000.

[27] M. El-Abd and M. S. Kamel. Black-box optimization benchmarking for noiseless function testbed using an EDA and PSO hybrid. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO): Late Breaking Papers*, pages 2263–2268, 2009.

[28] A. Engelbrecht. Fitness function evaluations: A fair stopping condition? In *Proceedings of the IEEE Swarm Intelligence Symposium (SIS)*, pages 1–8. IEEE, 2014.

[29] A. P. Engelbrecht. *Computational intelligence: An introduction*. John Wiley & Sons, 2007.

[30] R. Etxeberria and P. Larranaga. Global optimization using Bayesian networks. In *Proceedings of the Second Symposium on Artificial Intelligence (CIMAF-99)*, pages 332–339. Habana, Cuba, 1999.

[31] N. Fortier, J. Sheppard, and K. G. Pillai. Bayesian abductive inference using overlapping swarm intelligence. In *Proceedings of the IEEE Swarm Intelligence Symposium (SIS)*, pages 263–270. IEEE, 2013.

[32] N. Fortier, J. Sheppard, and S. Strasser. Learning Bayesian classifiers using overlapping swarm intelligence. In *Proceedings of the IEEE Swarm Intelligence Symposium (SIS)*, pages 1–8. IEEE, 2014.

[33] N. Fortier, J. Sheppard, and S. Strasser. Abductive inference in bayesian networks using distributed overlapping swarm intelligence. *Soft Computing*, 19(4):981–1001, 2015.

[34] N. Fortier, J. Sheppard, and S. Strasser. Parameter estimation in Bayesian networks using overlapping swarm intelligence. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pages 9–16. ACM, 2015.

[35] N. Fortier, J. W. Sheppard, and K. Pillai. DOSI: training artificial neural networks using overlapping swarm intelligence with local credit assignment. In *Proceedings of the Joint 6th International Conference on Soft Computing and Intelligent Systems (SCIS) and 13th International Symposium on Advanced Intelligent Systems (ISIS)*, pages 1420–1425. IEEE, 2012.

[36] N. L. Fortier. *Inference and learning in Bayesian networks using overlapping swarm intelligence*. PhD thesis, Montana State University, 2015.

[37] B. Giffler and G. L. Thompson. Algorithms for solving production-scheduling problems. *Operations research*, 8(4):487–503, 1960.

[38] Y.-J. Gong, W.-N. Chen, Z.-H. Zhan, J. Zhang, Y. Li, Q. Zhang, and J.-J. Li. Distributed evolutionary algorithms and their models: A survey of the state-of-the-art. *Applied Soft Computing*, 34:286–300, 2015.

[39] P. B. Grosso. *Computer Simulations of Genetic Adaptation: Parallel Subcomponent Interaction in a Multilocus Model*. PhD thesis, 1985.

[40] A. Gupta, Y.-S. Ong, and L. Feng. Multifactorial evolution: toward evolutionary multitasking. *IEEE Transactions on Evolutionary Computation*, 20(3):343–357, 2016.

[41] B. K. Haberman and J. W. Sheppard. Overlapping particle swarms for energy-efficient routing in sensor networks. *Wireless Networks*, 18(4):351–363, 2012.

[42] G. Harik. Linkage learning via probabilistic modeling in the ECGA. Technical Report 61, Illinois Genetic Algorithms Laboratory, 1999.

[43] D. Heckerman, D. Geiger, and D. M. Chickering. Learning Bayesian networks: The combination of knowledge and statistical data. *Machine Learning*, 20(3):197–243, 1995.

[44] A. M. Helal and A. M. Abdelbar. Incorporating domain-specific heuristics in a particle swarm optimization approach to the quadratic assignment problem. *Memetic Computing*, 6(4):241–254, 2014.

[45] Y.-C. Ho. On the numerical solutions of stochastic optimization problem. *IEEE Transactions on Automatic Control*, 42(5):727–729, 1997.

[46] J. H. Holland. *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence.* U Michigan Press, 1975.

[47] T. Ishimizu and K. Tagawa. A structured differential evolution for various network topologies. *International Journal of Computers and Communications*, 4(1):2–8, 2010.

[48] K. James and E. Russell. Particle swarm optimization. In *Proceedings of the IEEE International Conference on Neural Networks*, pages 1942–1948, 1995.

[49] M. Jamil and X.-S. Yang. A literature survey of benchmark functions for global optimisation problems. *International Journal of Mathematical Modelling and Numerical Optimisation*, 4(2):150–194, 2013.

[50] A. W. Johnson and S. H. Jacobson. On the convergence of generalized hill climbing algorithms. *Discrete Applied Mathematics*, 119(1):37–57, 2002.

[51] D. S. Johnson, C. H. Papadimitriou, and M. Yannakakis. How easy is local search? *Journal of Computer and System Sciences*, 37(1):79–100, 1988.

[52] S. A. Kauffman. *The origins of order: Self-organization and selection in evolution*. Oxford university press, 1993.

[53] J. Kennedy and R. C. Eberhart. A discrete binary version of the particle swarm algorithm. In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*, volume 5, pages 4104–4108, 1997.

[54] D. Koller and N. Friedman. *Probabilistic Graphical Models - Principles and Techniques*. MIT Press, 2009.

[55] J. R. Koza. *Genetic programming II: automatic discovery of reusable programs*. California, 1994.

[56] J. B. Kruskal. Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis. *Psychometrika*, 29(1):1–27, 1964.

[57] J. B. Kruskal. Nonmetric multidimensional scaling: a numerical method. *Psychometrika*, 29(2):115–129, 1964.

[58] R. V. Kulkarni and G. K. Venayagamoorthy. An estimation of distribution improved particle swarm optimization algorithm. In *Proceedings of the 3rd International Conference on Intelligent Sensors, Sensor Networks and Information (ISSNIP)*, pages 539–544, 2007.

[59] J. Lampinen and I. Zelinka. Mixed integer-discrete-continuous optimization by differential evolution. In *Proceedings of the 5th International Conference on Soft Computing*, pages 77–81, 1999.

[60] P. Larranaga and J. A. Lozano. *Estimation of distribution algorithms: A new tool for evolutionary computation*, volume 2. Springer Science & Business Media, 2002.

[61] R. Li, M. T. Emmerich, J. Eggermont, E. G. Bovenkamp, T. Bäck, J. Dijkstra, and J. H. Reiber. Mixed-integer NK landscapes. In *Parallel Problem Solving from Nature-PPSN IX*, pages 42–51. Springer, 2006.

[62] X. Li and X. Yao. Cooperatively coevolving particle swarms for large scale optimization. *IEEE Transactions on Evolutionary Computation*, 16(2):210–224, 2012.

[63] J. Liang, B. Qu, P. Suganthan, and A. G. Hernández-Díaz. Problem definitions and evaluation criteria for the cec 2013 special session on real-parameter optimization. *Computational Intelligence Laboratory, Zhengzhou University, Zhengzhou, China and Nanyang Technological University, Singapore, Technical Report*, 2013.

[64] Z. Michalewicz. *Genetic algorithms + data structures = evolution programs*. Springer Science & Business Media, 1996.

[65] M. Mitchell. *An introduction to genetic algorithms*. MIT press, 1998.

[66] M. Mitchell, S. Forrest, and J. H. Holland. The royal road for genetic algorithms: Fitness landscapes and ga performance. In *Proceedings of the First European Conference on Artificial Life*, pages 245–254. Cambridge: The MIT Press, 1992.

[67] M. Mitchell, J. H. Holland, and S. Forrest. When will a genetic algorithm outperform hill climbing? In *Advances in Neural Information Processing Systems 6*, 1993.

[68] H. Mühlenbein and T. Mahnig. FDA-A scalable evolutionary algorithm for the optimization of additively decomposed functions. *Evolutionary Computation*, 7(4):353–376, 1999.

[69] H. Mühlenbein, T. Mahnig, and G.-F. Informationstechnik. Convergence theory and applications of the factorized distribution algorithm. *Journal of Computing and Information Theory*, 7(1):19–32, 1999.

[70] H. Mühlenbein and G. Paass. From recombination of genes to the estimation of distributions i. binary parameters. In *Proceedings of the International Conference on Parallel Problem Solving from Nature*, pages 178–187. Springer, 1996.

[71] G. Ochoa, E. Lutton, and E. Burke. The cooperative royal road: avoiding hitchhiking. In *Proceedings of the International Conference on Artificial Evolution (Evolution Artificielle)*, pages 184–195. Springer, 2007.

[72] O. Olorunda and A. P. Engelbrecht. Differential evolution in high-dimensional search spaces. In *2007 IEEE Congress on Evolutionary Computation*, pages 1934–1941. IEEE, 2007.

[73] G. Pampara, N. Franken, and A. P. Engelbrecht. Combining particle swarm optimisation with angle modulation to solve binary problems. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC)*, volume 1, pages 89–96, 2005.

[74] L. Panait. Theoretical convergence guarantees for cooperative coevolutionary algorithms. *Evolutionary computation*, 18(4):581–615, 2010.

[75] L. Panait, R. P. Wiegand, and S. Luke. A visual demonstration of convergence properties of cooperative coevolution. In *Proceedings of the International Conference on Parallel Problem Solving from Nature*, pages 892–901. Springer, 2004.

[76] J. Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann, 1988.

[77] M. Pelikan, D. E. Goldberg, and E. Cantu-Paz. Linkage problem, distribution estimation, and Bayesian networks. *Evolutionary Computation*, 8(3):311–340, 2000.

[78] M. Pelikan, D. E. Goldberg, and F. G. Lobo. A survey of optimization by building and using probabilistic models. *Computational Optimization and Applications*, 21(1):5–20, 2002.

[79] M. Pelikan and H. Mühlenbein. The bivariate marginal distribution algorithm. In *Advances in Soft Computing*, pages 521–535. Springer, 1999.

[80] K. G. Pillai and J. Sheppard. Overlapping swarm intelligence for training artificial neural networks. In *Proceedings of the IEEE Swarm Intelligence Symposium (SIS)*, pages 1–8. IEEE, 2011.

[81] M. A. Potter. *The design and analysis of a computational model of cooperative coevolution*. PhD thesis, 1997.

[82] M. A. Potter and K. A. De Jong. A cooperative coevolutionary approach to function optimization. In *Proceedings of the International Conference on Parallel Problem Solving from Nature*, pages 249–257. Springer, 1994.

[83] M. A. Potter and K. A. De Jong. Cooperative coevolution: An architecture for evolving coadapted subcomponents. *Evolutionary Computation*, 8(1):1–29, 2000.

[84] J. Pugh and A. Martinoli. Discrete multi-valued particle swarm optimization. In *Proceedings of IEEE Swarm Intelligence Symposium (SIS)*, pages 103–110, 2006.

[85] S. Qureshi and J. W. Sheppard. Dynamic sampling in training artificial neural networks with overlapping swarm intelligence. In *Proceedings of the Congress on Evolutionary Computation (CEC)*, pages 440 – 446. IEEE, 2016.

[86] I. Rechenberg. Cybernetic solution path of an experimental problem. 1965.

[87] A. P. Reynolds, A. Abdollahzadeh, D. W. Corne, M. Christie, B. Davies, and G. Williams. A parallel BOA-PSO hybrid algorithm for history matching. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC)*, pages 894–901, 2011.

[88] J. N. Richter. *On mutation and crossover in the theory of evolutionary algorithms*. PhD thesis, Montana State University, 2010.

[89] S. J. Russell and P. Norvig. *Artificial intelligence: A modern approach*. Prentice Hall, 3rd edition, 2009.

[90] M. Safe, J. Carballido, I. Ponzoni, and N. Brignole. On stopping criteria for genetic algorithms. In *Advances in Artificial Intelligence–SBIA*, pages 405–413. Springer, 2004.

[91] A. Salman, I. Ahmad, and S. Al-Madani. Particle swarm optimization for task assignment problem. *Microprocessors and Microsystems*, 26(8):363–371, 2002.

[92] V. Santucci and A. Milani. Particle swarm optimization in the EDA framework. In *Soft Computing in Industrial Applications*, pages 87–96. Springer, 2011.

[93] M. Scutari. Bayesian network repository. http://www.bnlearn.com/bnrepository, 2012.

[94] D. Sha and C.-Y. Hsu. A hybrid particle swarm optimization for job shop scheduling problem. *Computers & Industrial Engineering*, 51(4):791–808, 2006.

[95] S. Shakya, A. Brownlee, J. McCall, F. Fournier, and G. Owusu. A fully multivariate DEUM algorithm. In *Proceedings of the Congress on Evolutionary Computation (CEC)*, pages 479–486. IEEE, 2009.

[96] S. Shakya and J. McCall. Optimization by estimation of distribution with DEUM framework based on Markov random fields. *International Journal of Automation and Computing*, 4(3):262–272, 2007.

[97] S. K. Shakya. *DEUM: A framework for an Estimation of Distribution Algorithm based on Markov Random Fields*. PhD thesis, The Robert Gordon University, 2006.

[98] Y. Shi and R. C. Eberhart. Parameter selection in particle swarm optimization. In *International Conference on Evolutionary Programming*, pages 591–600. Springer, 1998.

[99] Y.-j. Shi, H.-f. Teng, and Z.-q. Li. Cooperative co-evolutionary differential evolution for function optimization. In *Advances in natural computation*, pages 1080–1088. Springer, 2005.

[100] H. A. Simon. The structure of ill structured problems. *Artificial Intelligence*, 4(3-4):181–201, 1973.

[101] Z. Skolicki and K. De Jong. Improving evolutionary algorithms with multi-representation island models. In *International Conference on Parallel Problem Solving from Nature*, pages 420–429. Springer, 2004.

[102] F. J. Solis and R. J.-B. Wets. Minimization by random search techniques. *Mathematics of operations research*, 6(1):19–30, 1981.

[103] J. C. Spall. *Introduction to Stochastic Search and Optimization: Estimation, Simulation, and Control*. John Wiley & Sons, 2005.

[104] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

[105] R. Storn and K. Price. Differential evolution: A simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization*, 11(4):341–359, 1997.

[106] S. Strasser, N. Fortier, J. Sheppard, and R. Goodman. Factored evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, PP(99):1–1, 2016.

[107] S. Strasser, R. Goodman, J. Sheppard, and S. Butcher. A new discrete particle swarm optimization algorithm. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pages 53–60. ACM, 2016.

[108] P. N. Suganthan, N. Hansen, J. J. Liang, K. Deb, Y.-P. Chen, A. Auger, and S. Tiwari. Problem definitions and evaluation criteria for the cec 2005 special session on real-parameter optimization. *KanGAL report*.

[109] G. B. Thomas, M. D. Weir, J. R. Hass, and F. R. Giordano. *Thomas' Calculus Early Transcendentals (11th Edition) (Thomas Series)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[110] H. Tosun. *Efficient Machine Learning using Partitioned Restricted Boltzmann Machines*. PhD thesis, Montana State University, 2016.

[111] H. Tosun and J. W. Sheppard. Training restricted boltzmann machines with overlapping partitions. In *Proceedings of the Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 195–208. Springer, 2014.

[112] F. Van Den Bergh. *An analysis of particle swarm optimizers*. PhD thesis, University of Pretoria, 2006.

[113] F. Van den Bergh and A. P. Engelbrecht. Cooperative learning in neural networks using particle swarm optimizers. *South African Computer Journal*, (26):p–84, 2000.

[114] F. Van den Bergh and A. P. Engelbrecht. A cooperative approach to particle swarm optimization. *IEEE Transactions on Evolutionary Computation*, 8(3):225–239, 2004.

[115] K. Veeramachaneni, L. Osadciw, and G. Kamath. Probabilistically driven particle swarms for optimization of multi valued discrete problems: Design and analysis. In *Proceedings of the IEEE Swarm Intelligence Symposium (SIS)*, pages 141–149, 2007.

[116] E. D. Weinberger. NP completeness of Kauffman's NK model, a tuneable rugged fitness landscape. Technical report, 1996.

[117] D. Whitley, S. Rana, and R. B. Heckendorn. The island model genetic algorithm: On separability, population size and convergence. *Journal of Computing and Information Technology*, 7:33–48, 1999.

[118] D. Whitley and T. Starkweather. Genitor ii: A distributed genetic algorithm. *Journal of Experimental & Theoretical Artificial Intelligence*, 2(3):189–214, 1990.

[119] R. P. Wiegand, W. C. Liles, and K. A. De Jong. An empirical analysis of collaboration methods in cooperative coevolutionary algorithms. In *Proceedings of the genetic and evolutionary computation conference (GECCO)*, volume 2611, pages 1235–1245, 2001.

[120] R. P. Wiegand, W. C. Liles, and K. A. De Jong. Modeling variation in cooperative coevolution using evolutionary game theory. In *Proceedings of the Conference on Foundations of Genetic Algorithms (FOGA)*, pages 203–220, 2002.

[121] R. P. Wiegand and M. A. Potter. Robustness in cooperative coevolution. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pages 369–376. ACM, 2006.

[122] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.

[123] Z. Yang, K. Tang, and X. Yao. Large scale evolutionary optimization using cooperative coevolution. *Information Sciences*, 178(15):2985–2999, 2008.

[124] Y. Zhou, J. Wang, and J. Yin. A discrete estimation of distribution particle swarm optimization for combinatorial optimization problems. In *Proceedings of the Third International Conference on Natural Computation (ICNC)*, volume 4, pages 80–84, 2007.

APPENDIX A

BENCHMARK FITNESS FUNCTIONS

In this section, we present the benchmark functions used in this dissertation. Note that almost all of these functions are part of the CEC benchmark sets [63, 108]. When possible, we present a graph of each function with two variables to give the reader a general idea on the shape of the function.

A.1 Ackley's

$$f(\mathbf{X}) = -20 \times \exp\left(-0.2\sqrt{\frac{1}{N}\sum_{i=1}^{N} X_i^2}\right) - \exp\left(\frac{1}{N}\sum_{i=1}^{N}\cos(2\pi X_i)\right) + 20 + e$$

$$-32 \leq X_i \leq 32$$

$$\text{minimum at } f(0, \cdots, 0) = 0$$

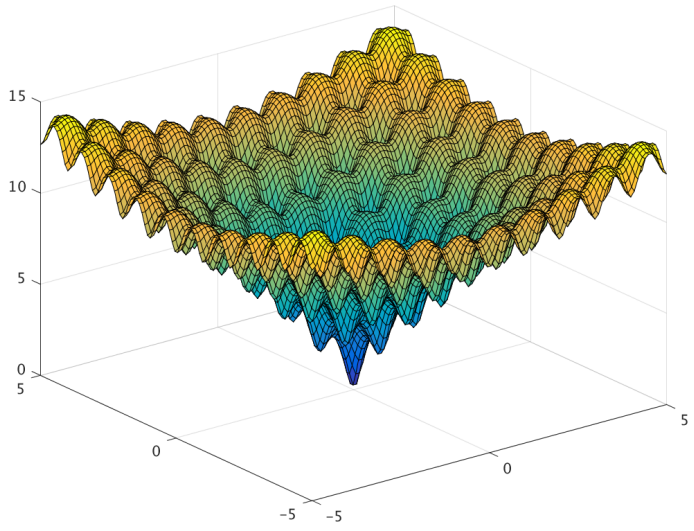


Figure A.1: Ackley's function in two dimensions.

A.2 Brown

$$f(\mathbf{X}) = \sum_{i=1}^{N-1} (X_i^2)^{(X_{i+1}^2+1)} + (X_{i+1}^2)^{(X_i^2+1)}$$

$$-1 \leq X_i \leq 4$$

$$\text{minimum at } f(0, \cdots, 0) = 0$$

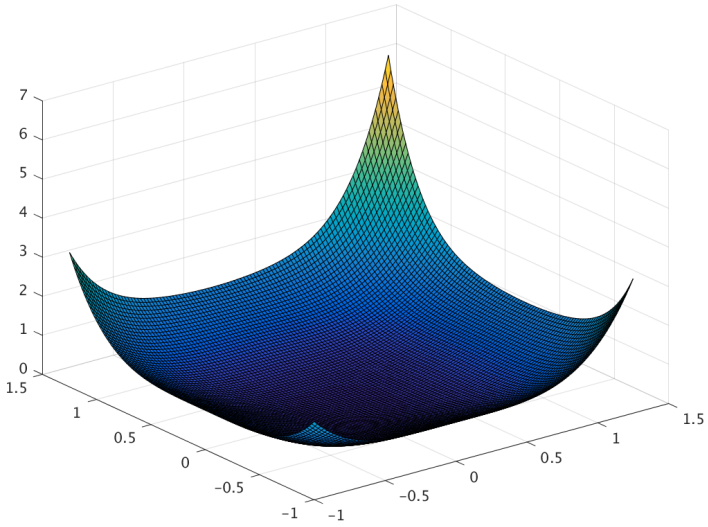


Figure A.2: Brown function in two dimensions.

A.3 Dixon-Price

$$f(\mathbf{X}) = (X_1 - 1)^2 + \sum_{i=2}^{N} i(2X_i^2 - X_{i-1})^2$$

$$-10 \leq X_i \leq 10$$

$$\text{minimum at } f\left(2\frac{2^i - 2}{2^i}, \cdots, 2\frac{2^i - 2}{2^i}\right) = 0$$

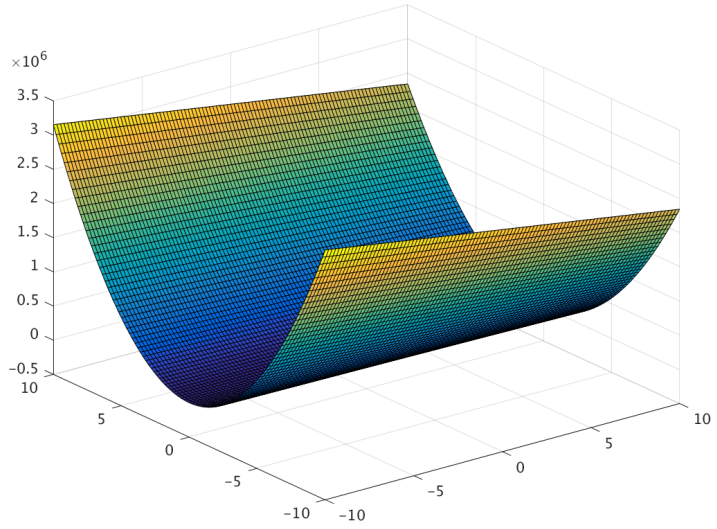


Figure A.3: Dixon-Price function in two dimensions.

A.4 Exponential

$$f(\mathbf{X}) = -\exp\left(-0.5\sum_{i=1}^{N} X_i^2\right)$$

$$-1 \leq X_i \leq 1$$

$$\text{minimum at } f(0,\cdots,0) = 0$$

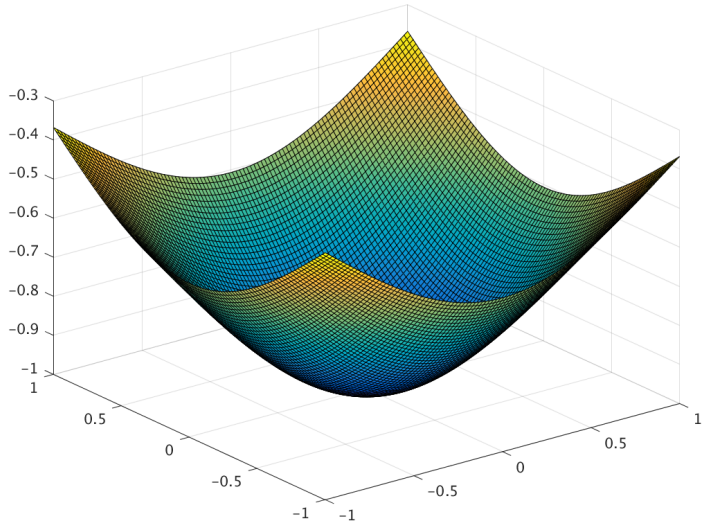


Figure A.4: Exponential function in two dimensions.

A.5 Griewank

$$f(\mathbf{X}) = 1 + \frac{1}{4000}\sum_{i=1}^{N} X_i^2 - \prod_{i=1}^{N} \cos\left(\frac{X_i}{\sqrt{i}}\right)$$

$$-100 \leq X_i \leq 100$$

$$\text{minimum at } f(0, \cdots, 0) = 0$$

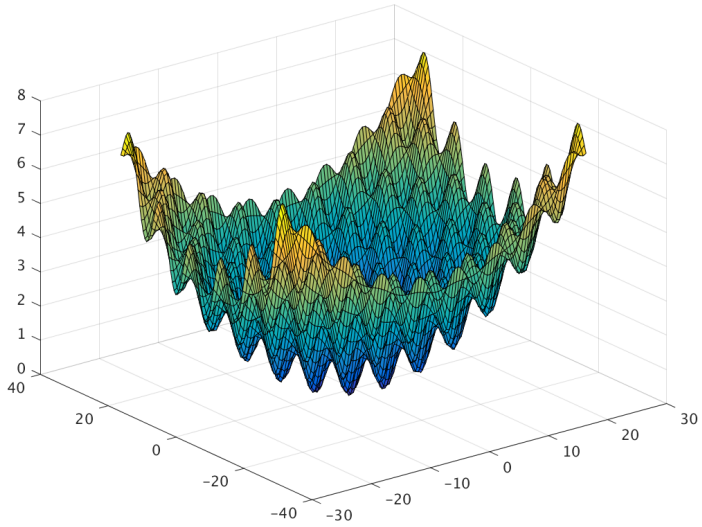


Figure A.5: Griewank function in two dimensions.

A.6 Powell Singular

$$f(\mathbf{X}) = \sum_{i=1}^{N/4} (X_{4i-3} + 10X_{4i-2})^2 + (5X_{4i-1} - X_{4i})^2 + (X_{4i-2} - X_{4i-1})^4 + 10(X_{4i-3} - X_{4i})^4$$

$$-4 \leq X_i \leq 5$$

$$\text{minimum at } f(3, -1, 0, 1, \ldots, 3, -1, 0, 1) = 0$$

Note that the function requires at least four variables; therefore, a two-dimensional plot is not possible.

A.7 Rana

$$f(\mathbf{X}) = \sum_{i=1}^{N-1} (X_{i+1} + 1) \times \cos(t_2) \sin(t_1) + X_i \times \cos(t_1) \sin(t_2)$$

$$\text{where } t_1 = \sqrt{|X_{i+1} + X_i + 1|} \text{ and } t_2 = \sqrt{|X_{i+1} - X_i + 1|}$$

$$-500 \leq X_i \leq 500$$

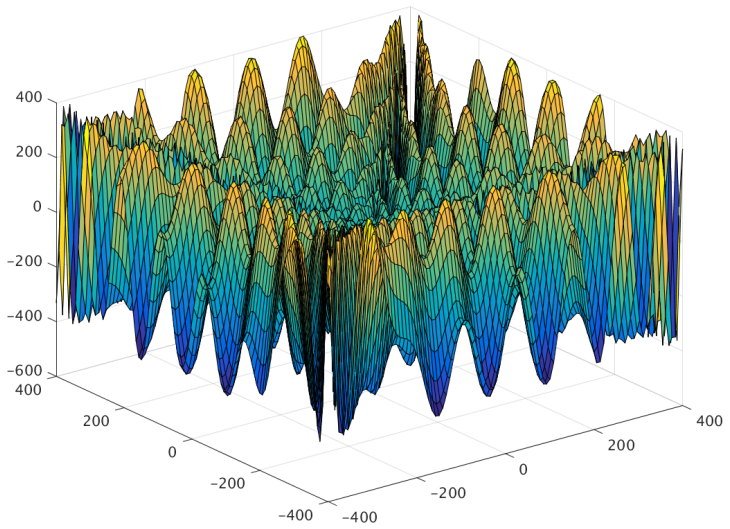


Figure A.6: Rana function in two dimensions.

A.8 Rastrigin

$$f(\mathbf{X}) = 10N + \sum_{i=1}^{N}(X_i^2 - 10\cos(2\pi X_i))$$

$$-5.12 \leq X_i \leq 5.12$$

$$\text{minimum at } f(0, \cdots, 0) = 0$$

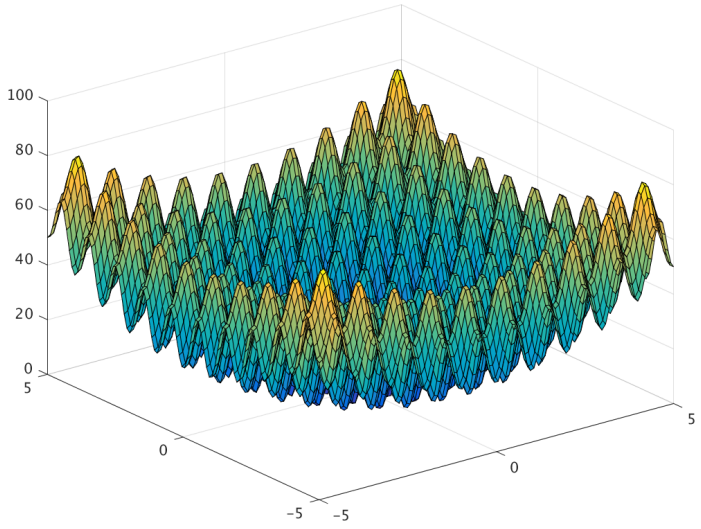


Figure A.7: Rastrigin function in two dimensions.

A.9 Rosenbrock

$$f(\mathbf{X}) = \sum_{i=1}^{N-1} (100(X_i^2 - X_{i+1})^2 + (1 - X_i)^2)$$

$$-2.048 \leq X_i \leq 2.048$$

$$\text{minimum at } f(1, 1, \cdots, 1) = 0$$

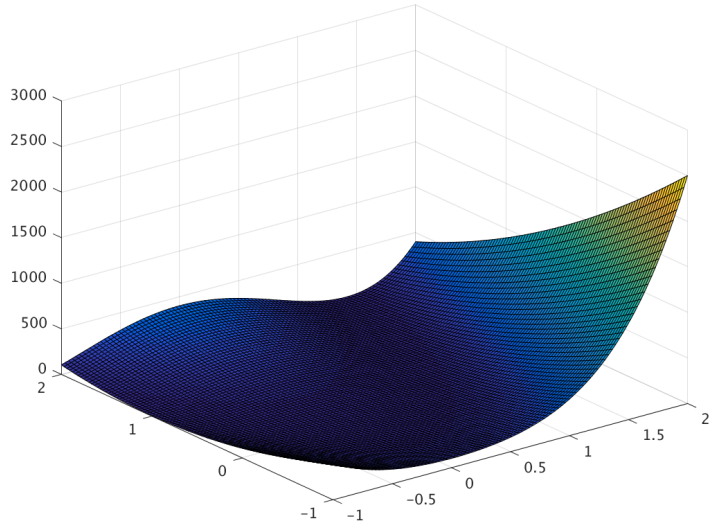


Figure A.8: Rosenbrock function in two dimensions.

A.10 Schwefel

$$f(\mathbf{X}) = 418.9829 \times N - \sum_{i=1}^{N} X_i \sin(\sqrt{|X_i|})$$

$$-512 \leq X_i \leq 512$$

minimum at $f(420.968746, 420.968746, \cdots, 420.968746) = 0$

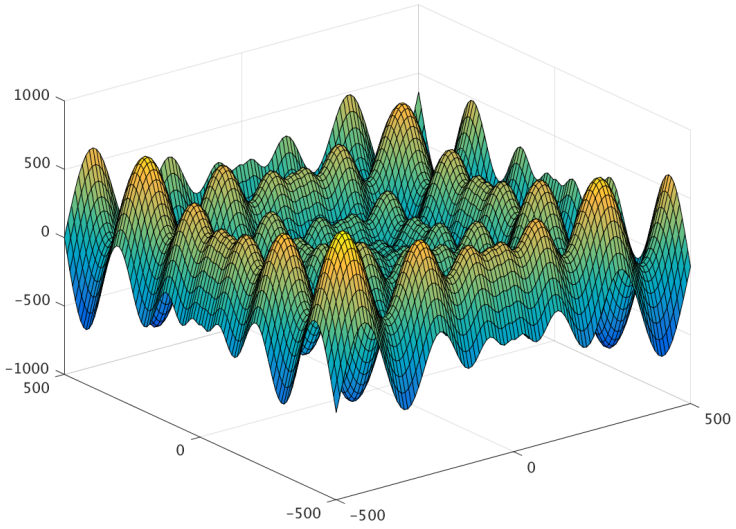


Figure A.9: Schwefel function in two dimensions.

A.11 Sphere

$$f(\mathbf{X}) = \sum_{i=1}^{N} X_i^2$$

$$-5.12 \leq X_i \leq 5.12$$

minimum at $f(0, \cdots, 0) = 0$

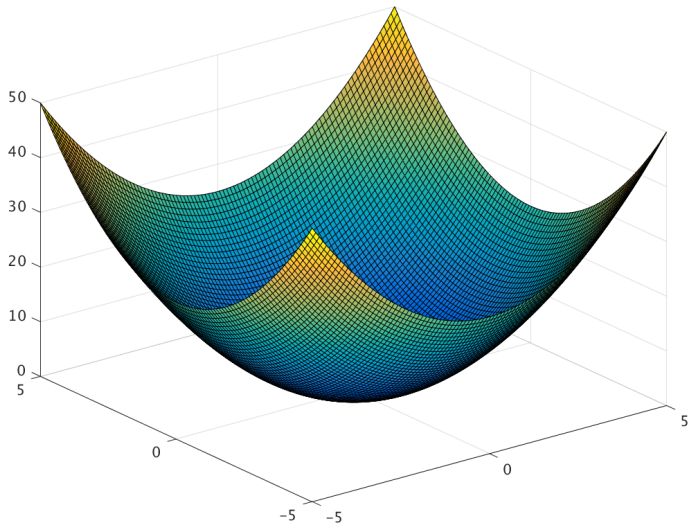


Figure A.10: Sphere function in two dimensions.