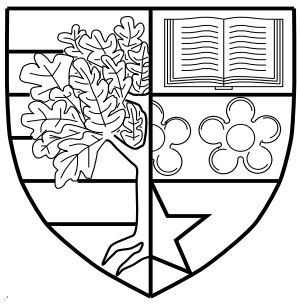
Investigating Hybrids of Evolution and Learning for Real-Parameter Optimization

by

SHE RI GU LENG



Submitted for the Degree of
Doctor of Philosophy
on completion of research in the
Department of Computer Science
School of Mathematical and Computer Sciences
Heriot-Watt University
March 2011

Declaration

I hereby declare that the work presented in this thesis was carried out by myself at Heriot-Watt University, except where due acknowledgment is made, and not been submitted for any other degree.

SHE RI GU LENG(Candidate)

Professor David Wolfe Corne (Supervisor)

Date

Abstract

In recent years, more and more advanced techniques have been developed in the field of hybridizing of evolution and learning, this means that more applications with these techniques can benefit from this progress. One example of these advanced techniques is the Learnable Evolution Model (LEM), which adopts learning as a guide for the general evolutionary search. Despite this trend and the progress in LEM, there are still many ideas and attempts which deserve further investigations and tests. For this purpose, this thesis has developed a number of new algorithms attempting to combine more learning algorithms with evolution in different ways. With these developments, we expect to understand the effects and relations between evolution and learning, and also achieve better performances in solving complex problems.

The machine learning algorithms combined into the standard Genetic Algorithm (GA) are the supervised learning method *k*-nearest-neighbors (KNN), the Entropy-Based Discretization (ED) method, and the decision tree learning algorithm ID3. We test these algorithms on various real-parameter function optimization problems, especially the functions in the special session on CEC 2005 real-parameter function optimization. Additionally, a medical cancer chemotherapy treatment problem is solved in this thesis by some of our hybrid algorithms.

The performances of these algorithms are compared with standard genetic algorithms and other well-known contemporary evolution and learning hybrid algorithms. Some of them are the Covariance Matrix Adaptation Evolution Strategies (CMAES), and variants of the Estimation of Distribution Algorithms (EDA).

Some important results have been derived from our experiments on these developed algorithms. Among them, we found that even some very simple learning methods hybridized properly with evolution procedure can provide significant performance improvement; and when more complex learning algorithms are incorporated with evolution, the resulting algorithms are very promising and compete very well against the state of the art hybrid algorithms both in well-defined real-parameter function optimization problems and a practical evaluation-expensive problem.

to my parents

Bu Ren Te Gu Si

An Shu Fang

Acknowledgements

First, I sincerely thank my supervisor, Professor David Wolfe Corne, for his guidance in the field of naturally inspired computing, and his consistent inspiration, advice, encouragement and support during the whole procedure of my PhD study. Without his help, this PhD thesis could not have been finished.

Thanks are also due to the members of the Intelligent Systems Lab (ISL) at the School of Mathematics and Computer Science, in particular many colleagues for any beneficial discussions during the past years.

I also want to thank the reviewers who read my publications and provided much constructive criticism for the past years, and the examiners who agreed to examine this thesis.

Thanks to the department's Administration and IT support staff for the convenience they provided, especially for allow me to occupy numerous machines in labs to finish the experiments in this thesis.

Finally, I would like to thank my parents and brother for their love!

Contents

List of Tables

List of Figures

Acronyms

BN Bayesian Network. 17

CMAES Covariance Matrix Adaptation Evolution Strategies. 22–25

EC Evolutionary Computation. 6, 7, 11–13

ED Entropy-Based Discretization. 23, 24

EDA Estimation of Distribution Algorithms. 22–25

ES Evolution Strategies. 24

GA Genetic Algorithm. 1, 6, 8, 11–13, 24

GP Genetic Programming. 13

IPOP-CMAES A Restart CMAES With Increasing Population Size. 25

K-PCX A Population-Based, Steady-State Procedure for Real-Parameter Optimization. 25

KNN *k*-Nearest-Neighbors. 17, 22–24

LEM Learnable Evolution Model. 22

LEM(dwKNN) LEM Instantiated with distance-weight KNN algorithm. 22, 24, 25

LEM(ED) LEM Instantiated with ED algorithm. 24

LEM(ID3) LEM Instantiated with ID3 algorithm. 1, 23, 25

LEM(KNN) LEM Instantiated with KNN algorithm. 1, 22–24

LR-CMAES A Local Restart CMAES. 25

MA Memetic Algorithm. 21

MDP Markov Decision Process. 17

ML Machine Learning. 16

MOEA Multi-Objective Evolutionary Algorithm. 12

NN Neural Network. 17

PCA Principle Component Analysis. 22, 24

SVM Support Vector Machine. 17

List of Publications

2010: Guleng Sheri and David Corne. Learning-assisted evolutionary search for scalable function optimization: LEM(ID3). IEEE Congress on Evolutionary Computation, pp. 1-8, IEEE, 2010.

2009: Guleng Sheri and David W. Corne. Evolutionary Optimization Guided by Entropy-Based Discretization. EvoWorkshops, Lecture Notes in Computer Science, Vol. 5484, pp. 695-704, Springer, 2009.

2008: Guleng Sheri and David W. Corne. The Simplest Evolution/Learning Hybrid: LEM with KNN. 2008 IEEE World Congress on Computational Intelligence, IEEE Press, 1-6 June 2008.

Chapter 1

Introduction

1.1 Overview

There are always various problems and tasks in people's daily activities. These problems can be very simple, making the procedure of solving these problems easily ignored; they can also be very complex and even challenging, making the methods of solving these problems become the topic of scientific research. However, regardless of the types and complexities of the problems, solving these problems can be considered as a decision-making procedure of choosing one or several solutions from many alternative solutions. Namely, *to search* for suitable solutions from many solutions is a problem solving procedure, and it exists in many fields such as computer science, engineering, operation research, medicine development, economy and finance. Let us consider the following tasks:

1. Find the quickest route from the current position to the city airport;
2. Find the maximum value for a mathematical function with a complex landscape;
3. Design a new aircraft engine for a new series of commercial planes with the requirements of both safety and speed;
4. Make a smartest play in a Chinese-checker game against the computer;
5. Create an effective treatment plan for a new drug to be applied in treatment periods;
6. Find the best 'model' or method that can predict the performance of a stock index for the future according to historical data.

These problems are universal and challenging. For task 1, firstly, in most cases, we can not arrive at the airport in the quickest way, quickest travel depends on many factors such

as, traffic conditions, vehicles, and road accidents, shortest does not mean quickest. Secondly, if there are many alternative routes with different distances, finding the shortest route may take a long time. Too much time spent on planning a route may reduce the travel time, therefore resulting in a late arrival. Finally, imagine the worst situation, if we are not familiar with the city and no previous knowledge is available, and we cannot get any form of help, we may fail this task completely. For task 2, assume the mathematical function's definition (formula) is given, and the value of the function is decided by a vector of variables. Firstly, the function's shape (landscape) is invisible, and could be very irregular, containing many peaks and troughs, so to decide the vector of variables having the maximum of the function is very difficult. Exploring through the whole variable space or 'search' is very possibly attracted to and lost in one of the local best values and never comes out; secondly, what if the size of the vector of variables is huge? That is, there are many variables involved in deciding the function value. Indicating the relationships between these variables could be important in order to find the maximum efficiently, in other words, these variables need to act congruously. For task 3, there are two aims for this task. If it is the case that higher speed means less safety, then we find that the aims for this task cannot be achieved at the same time. Namely, safety and speed are themselves contradictory to each other in the sense that an increase in the former will inevitably cause a decrease in the latter. Therefore, this problem has to be transformed to find an acceptable compromise between these two aims, safety and speed. When some solutions satisfying all of the aims are derived, the selection from these compromised solutions will depend on many practical considerations from the users. The main difficulty of this task is due to the contradiction feature involved in the task itself. For task 4, the difficulties come from many aspects, for instance, the number of the possible board status and legal next moves available could be huge; more importantly, the success of the current move will not only depend on the performance of the current move but also depend on the following moves. A good move for now and bad moves for all the following moves will also result in losing the game. So, for this task, how to measure a 'smart' move becomes a crucial factor in solving this problem successfully. For task 5, it is probably much easier or 'cheaper' to create a plan than to judge or evaluate a plan. Namely, the evaluation of a cancer chemotherapy treatment plan could be very expensive with regard to safety issues, time and money. It could contain real treatments (injections and observations) on patients or at least a computer-based simulation, both of which may take months for the treatments to take effect. These risky and expensive evaluations cannot be restored and therefore do not allow the method of choosing one solution from many al-

ternative solutions to be fulfilled. For task 6, there are many issues concerned, first, how to build a model which is able to predict, from the given data? Or, how to select the suitable methods to construct the model? Second, how to explore through these models, or how to modify these models from one to another? Third, how to evaluate the quality of these built models? These are all key issues which need to be solved. Finally, the quantity and the quality of the data also matter, the former may affect the evaluation of the built models, while the later may effect the construction procedure for the models.

These tasks are examples of complex problems, they explore many different aspects of complexity. These complexities may come from the partial representation space, huge search space, relationship among the dimensions, multi-modal property for the problem landscape, many conflicting objectives, and challenging measurements of the solutions in real world, etc. In this thesis, we will investigate and construct effective solving methods which can solve problems containing some of these aspects.

All of these complexities place obstacles for the procedure of finding the best solution for these problems, and require considerable efforts both in computation time and space resources. Namely, whenever a method is developed and applied to solve problems, it is always restricted by the time and space resources available. Therefore, we have to consider the balance between the quality of the solutions found and the efficiency of the search method employed. For task 1 again, we may not really care about finding the quickest route to the airport at all, what really concerns us is to arrive at the airport on time. With this aim, it is meaningless and we risk missing the flight if we take a very long time to find the best travel route. For most cases, the best solution for one problem cannot be found simply, people do accept secondary solutions when the best solutions can only be obtained with huge expenses which are not affordable. The only restriction is that these solutions need to be *feasible* solutions, a *feasible* solution should be a solution which is valid and correct to the problem at hand.

Within these limitations, the search methods to be constructed to solve these problems will have to consider the balance between quality of the solutions and efficiency of the methods. Generally, if the time and space resources limitations are not very restrictive, the search can explore more alternative solutions which are quite different, obtaining more *global* knowledge; while the resources limitations are strict, the search should exploit the current solutions to gain more similar solutions, obtaining more *local* knowledge. *Exploration* means the generation of new solutions happen in as yet untested regions of the solution space. Meanwhile, *exploitation* means the search is concentrated in the vicinity of

known good solutions. Therefore, achieving the expected balance between quality of found solutions and the efficiency of the search algorithms requires good design to achieve the balance between exploration and exploitation in the constructed search methods.

So far, our discussion has focused on 'search' as an important problem solving method, search is suitable for many types of complex problems and can be seen as a general problem solving method. However, search is not a universal problem solver. Many types of problems are solved by other problem solving methods. For example, part of task 6 is to construct a 'model' or method that can predict the performance of a stock index for the future according to historical data. This construction task is different from the search procedure. Seemingly, it is a specific method that follows some principles to build up a model, which can deal with some input data and be able to output useful information to make a prediction. This method, in fact, is one of the concrete methods for the general learning based problem solving method. *Learning* is a common concept, it means to improve one's ability to act in the future through accumulation of one's own experiences. As with searching, learning happens throughout people's daily activities as long as some events and decision-making processes take place. For example, task 6 is a common economical activity. The doctor's daily diagnosis treatment is a learning procedure, after a long period of diagnosis on a huge number of patients, the doctor becomes more experienced in treating new patients.

A natural question which arises, as two general problem solving methods, is: what is the relationship between search and learning? And how can these two methods interact to influence each other? We consider this issue in two opposite directions.

How does searching influence learning? Learning is a procedure to improve the ability to act in the future based on past experience. 'The ability to act in the future' needs to be captured and described by an explicit or implicit 'model', to improve the ability means to improve the performance of the model, or equally to construct a better model. This has a very important implication, namely, there exist many models which constitute the so-called 'model space'. Therefore, a search can be considered as a generalized model construction method, while the learning methods can be considered as a specific model construction method. Learning constructs a model by some specific method, while a search constructs a model by modifying one existing model into another one. To this end, a search influences learning by finding a better model than the one constructed by learning. There is one excellent illustration for this influence, the human brain. The brain can be seen as a learning model which is constructed in a way which is not completely known to us, however, what we do know is that, in fact, our brains today are quite different from and are

much more advanced than those of our ancestors. And this improvement is derived by one of the paradigms methods of search, *evolution*. The brain, as a model of learning, is the result of an evolutionary search.

How does learning influence search? Learning can cause useful experience to be obtained, we call such experience *knowledge*. If the learned knowledge is used in the search procedure to guide future search behaviors, then we could expect at least two new results. First, learning incorporates the past learned domain knowledge into the search problem-solving procedure, and makes the choosing of good solutions among many alternative solutions more efficient and accurate. Experienced persons can make better choices than less experienced ones. In this way, learning helps to save time and space resources, while obtaining relatively high quality solutions. Second, learning can help to judge or predict the found solutions. When some new solutions are found by search, learning can estimate the quality of these solutions according to previous knowledge about this particular domain without the expense of real implementation of the quality measurements for these solutions. These measurements could be very expensive and cannot be restored. Experienced persons can predict the results of some events yet to happen and therefore save the resources expended in these events.

1.1.1 Search is a general problem solver

There is no such a method as a universal problem solver for all classes of problems. For some classes of problems which are complex enough, a search is a general problem solver for these problems. Many search methods have been developed for solving various problems, a successful search method depends not only on its advantages (efficiency) but also on its appropriateness for the class of problems it tries to solve. So, the requirements of developing a good search algorithm should be that the algorithm is both general and efficient enough to solve a class of problems. In an ideal situation, we can simply apply a very general search method to attempt to solve all problems. For example, *exhaustive enumeration* can be used to solve all problems only if the computation time and space resources are unlimited, if this is not the case, then exhaustive enumeration will fail easily. This is because exhaustive enumeration is not efficient for many complex problems, which cannot be solved in linear or polynomial time. On the other hand, some local search based algorithms are very efficient only on a small range of problems, which makes these algorithms not general enough.

In general, there are many standards to classify search methods, such as heuristic or non-heuristic search, local search and population-based parallel search. The former standard emphasizes the utility of problem-specific domain knowledge. The later standard emphasizes the difference in search methods. It is the latter standard that will be considered within this thesis. In single point local search based methods, a search is carried out locally and tries to find some good solutions close to the current solution. Therefore, local search methods tend to find good solutions quickly. Some local search techniques iteratively improve upon a solution by searching in its vicinity for better solutions. If better solutions cannot be found, the process terminates; the current solution is taken as a locally optimal solution. For example, for task 1, assume we have found out a feasible travel route, a change or substitution of part of this route with another subroute could result in a better travel route by avoiding an accident. A local search involves the risk that a search is cheated into a local optimization.

Unlike the local search strategy, the population-based search considers many solutions at the same time and works on the whole set of solutions (also called *population*). All of the solutions have an opportunity to be involved in the search procedure, and all of them are possibly modified and substituted. A local search is operated in a one-by-one fashion, while population-based search is carried out in parallel. A local search is not repeatable and it never goes back, the previous visited solutions are not stored for revisiting, instead, they are discarded immediately when they are not used to compare with other solutions. In a population-based search, all solutions are maintained in the current population, as are the modified ones which will form the new population. In this way, the local search is more like a way of 'constructing' a solution, while the population based search is more like a way of 'evolving' a population of solutions. The results of the evolution of a population is that some of the solutions become better and some are worse.

In this thesis, we focus on the population-based search method, especially, a class of such search methods called Evolutionary Computation (EC). Evolutionary search techniques, such as Genetic Algorithms (GAs) have recently gained considerable attention. Evolutionary computation is inspired by Darwin's theory of evolution. For a given environment that can host only a limited number of individuals, the competition of reproduction and survival are inevitable. Each individual is a unique combination of phenotypic traits, representing a solution, the object representing the original problem context are referred to as *phenotypes*, while their encoding, that is, the individuals within EC, are called *genotypes*. Therefore, on the side of the original problem context, solutions or individuals are

used to denote points of the space of possible solutions. This space is commonly called the *phenotype space*. On the side of EC, chromosome or individual can be used for points in the space where the evolutionary search actually takes place. This space is often termed the *genotype space*. These solutions and individuals are evaluated by the environment. Natural selection favors those individuals that fit the environment better, the principle of *survival of the fittest*. The fitter individuals (parents) are more likely to be selected to reproduce new individuals (offspring) which are expected to be better individuals because of the combination of good genes inherited from their parents. There are two main variation operations to generate new individuals, the mutation operates on one individual by randomly changing a part of its genes; while the crossover combines two or more individuals to produce new ones. The new individuals are then evaluated and selected for survival. In this way, evolution continues. Selection and variation operations are the main sources in the evolution procedure for diversity and quality improvement of the population.

To understand how evolutionary computation is used to solve optimization problems. We firstly describe these optimization problems and their main features which make these problems challenging for many optimization algorithms. These understanding about the optimization problems are necessary and beneficial to the understanding of evolutionary algorithms. There is an important tool which can help for this understanding, it is the idea of *landscape*. The individuals or solutions space introduced above can also be described with fitness landscape. Fitness landscape is defined as a triple set: $FitnessLandscape = (\mathcal{S}, \mathcal{V}, f)$, where

1. $\mathcal{S}$ is the set of all potiential solutions;
2. $\mathcal{V}$ is a neigbourhood function, $\mathcal{V} : \mathcal{S} \longrightarrow 2^{\mathcal{S}}$, $\forall x \in \mathcal{S}, \mathcal{V}(x) = \{y \in \mathcal{S} \mid d(x,y) \leq 1\}$;
3. f is a fitness function. $f : \mathcal{S} \longrightarrow \Re$.

Within a landscape, as shown in Figure 1.1, the height dimension belongs to fitness: high altitude stands for high fitness, the other two or more dimensions correspond to individuals' genes. That is the horiental plane holds all possible genes combinations, the vertical values show their fitnesses. Hence, each peak represents a range of successful genes combinations, while troughs belong to less fit combinations. A given population can be plotted as a set of points on this landscape, where each dot is one individual realizing a possible genes combination. Evolution is then the process of gradual advances of the population to high-altitude areas, powered by variation and natural selection.

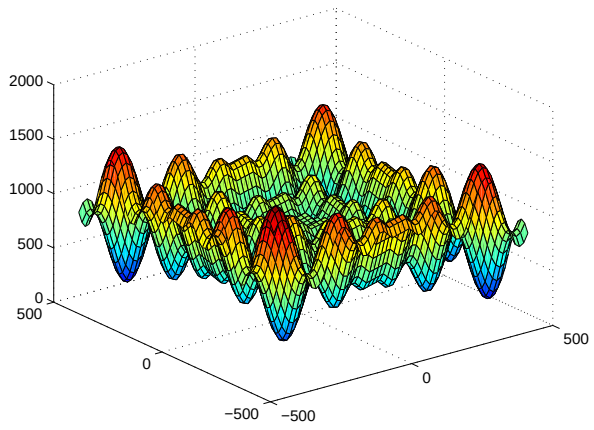


Figure 1.1: An illustrative example of landscape

After the definition and illustration of landscape, we can now explore the main problem features which make the real optimization problems difficult to solve. Those problem features and the problem landscape types correspond to each other accordingly. That is the implict problem features will be reflected in the problem landscapes explicitly. We list and classify the main problem features or landscape types widely known and well-studied in optimization community as follows:

1. Discrete (combinatorial) and Continuous (real) variables

   The discrete optimization problems are the problems whose solutions can be expressed exactly using a finite length string of integer parameters. While, the continuous optimization problems contain one or more continuous parameters and are usually tackled by choosing a finite precision with which to express the parameters. The parameter values may then be represented using chromosomes in which the allele value of each gene reprsents the value of a parameter directly given a precision. Genetic Algorithms are considered as most suitable for binary representation of variables. However, more and more work of applying GA for continuous representation problem have been investigated. There are two important EC paradigm algorithms Evolution Strategies (ES) and Evolutionary Programming (EP) typically operate directly on the continuous decision variables, and thus their operators are particularly suited to these problems.

2. Dimentionality

   Dimensionality refers to the number of dimensions of the parameter space. High-dimensionality problems are more representative of real world problems, in compared

with low-dimensionality problems. Also, high-dimensionality problems are more difficult to solve than low-dimensionality problems, because it is evident that high-dimensions means more variables and therefore bigger search space. Also, high-dimension problems may contain more interactions between different dimensions, these interactions cause more complexity for optimization algorithms.

3. Multimodality

   Multimodal problems are those problems in which there are a number of points that are better than all their neighbouring solutions. Each of these points is a local optimum and denote the highest of these as the global optimum. Problems in which there is only one point that is fitter than all of its neighbours are known as *unimodal problems*.

   Most real-world optimization problems of interest are multimodal, that is they contain more than one optimum. Sometimes, the optima in a multimodal landscape may be of different levels or of the same level. If they are all of the same level then they are all global optima. Finding one of them is usually sufficient to solve the optimization problem exactly, thus multimodality can potentially make a problem easy, as many points are easier to search for than one.

   If the optima are of different levels, then some are not global optima. These local optima can cause difficulties particularly for local-search algorithms such as hill-climbers, because they can become stuck in them, unable to escape to any point of better evaluation. Genetic algorithms and other population-based algorithms are often considered as being particularly suited to searching multimodal landscapes.

4. Discontinuity and Continuity (Non-differentiable and Differentiable)

   There are some real optimization problems whose objective function values are discontinuous, such as the combinatorial problems which are always non-differentiable. When the objective function landscape is continuous, it is possible that gradient methods are more suitable than evolutionary algorithms. Discontinuity is not usually regarded as a main factor of problem difficulty.

5. Epistasis (Non-separability) and Linear separability

   Epistasis is a measure of the degree of interaction between parameters in an objective function. If a problem has no epistasis then all of the parameters can be independently optimized, so that the number of points that must be visited is very small compared

to the whole search space. If the parameters in a problem can be split into groups in such a way that, taking each group separately, the parameter values within that group which give the best evaluation, with the values of all other parameters held constant, are the same as those in the global optimum, then the problem is linearly separable. On the other hand, if in a problem, the contributions of all parameters depend upon all others then the problem has unbounded epistasis and is not linearly-separable. Such a problem is generally difficult to search using an EA or any other general-purpose technique. For this reason, epistasis and epistasis variance have been used as predictors of problem difficulty. It has also been suggested by some that real-world problems exhibit bounded epistasis and this makes it possible to search them efficiently using EAs and other metaheuristics.

6. Unconstrained, Linear constrained and Non-linear constrained

   Constraints are virtually ubiquitous in real world optimization problems, both discrete and continuous, so we should expect that good general-purpose search algorithms can deal with constraints. Constraints can be linear and non-linear. For some problems, the optimum located in different places, particularly on the constraint boundary, and the feasible and infeasible regions have different sizes. It can be argued that population based evolutionary techniques are better suited to constrained optimization problems, because EAs can traverse an infeasible region and less possible to be trapped in suboptimal feasible regions. there are some other different approaches to deal with constraints, including penalty functions, decoders, and repair mechanisms.

7. Neutral fitness landscapes

   The neutral fitness landscape explores another difficulty of many optimization problems. Neutrality results in the search algorithms lossing direction because there will be no enough various fitness information available to reflect the performance difference among solutions. Based on the definition of fitness landscape above, the neutrality of the fitness landscape can be further charactorized with the idea of *test of neutrality*. A test of neutrality is a predicate:

$$isNeutral : \mathcal{S} \times \mathcal{S} \longrightarrow \{true, false\}.$$

   For example, $isNeutral(s_1, s_2)$ is *true* if:

$f(s_1) = f(s_2)$,

$| f(s_1) - f(s_2) | \leq 1/M$, with M is the population size,

$f(s_1) - f(s_2)$ is under the evaluation error.

The *neutral neighborhood* of s is the set of neighbors which have the same fitness $f(s)$

$$\mathcal{V}_{neut}(s) = \{s^{'} \in \mathcal{V}(s) \mid isNeutral(s, s^{'})\}.$$

The *neutral degree* of a solution is the number of its neutral neighbors.

$$nDegree(s) = \#(\mathcal{V}_{neut}(s) - \{s\}).$$

A fitness landscape is *neutral* if there are many solutions with high neutral degree. If we consider applying a evolutionary algorithm to solve a problem with neutral fitness landscape, then we found that the main feature of neutral fitness landscape is that, a considerable number of mutations have no effects on the fitness values.

To this point, we return to those tasks we introduced at the beginning of the chapter, and try to solve them using one of the paradigm algorithms of EC, the GA, without formally introducing it. For task 1, one could simply use a 'search agent' to help with this task. One of the possible implementations for this agent is maintaining a population or set of possible routes given the starting and arriving positions by the user, and repeatedly searching for new and better routes. The routes are evaluated by the measurements of length, traffic conditions, etc. This information could be collected by the agent according to previous statistical data or real-time releasing updates from the city's traffic management center. The search is used to create new alternative routes according to some current good routes. If there are some better routes, then they are kept and some very bad routes are deleted. The newly-generated routes will simply substitute some parts of the current good routes with some other possible subroutes (or roads). Those substitutes could be more beneficial by avoiding a road which is quite often congested, or could worsen the current routes by increasing their lengths. A search can take a certain number of cycles before stopping, and the route returned should have the following features, it is not necessarily the best (which we could not really know) and also is not the shortest, it avoids some serious traffic congestions by traveling on some lanes instead of main roads occasionally, and under the

estimated travel time by the agent, it takes the least traveling time. For task 2, we assume the variables are all real numbers. If the number of variables constitute one input value for the mathematical function is n, and for each variable, there are d equal possible values, then we will have d^n different input solutions. And the set of these solutions form the solution space. The search is then used again to create new alternative solutions based on the current good solution from an initial set of solutions which could be randomly sampled. All of these solutions (set of variables assigned with real numbers) are evaluated by feeding them into the formula and a function value is calculated. Those solutions which have higher function values (maximization) are emphasized and are given a higher probability to be involved in producing a new solution. The new solutions are kept into the next set, where some bad solutions are deleted subject to a pre-fixed set size. As stated before, this task has the difficulty that the invisible landscape of the function could be very irregular. The advantages of the population based search method for this task are that the situations where some solutions are attracted into some local optimum do not apply to other solutions which could represent the global optimizations successfully.

For task 3, how to apply a population based search to solve this task and the relative aspects which need to be considered are beyond the scope of this thesis, but we still emphasize two main aspects which have arisen from this task. First, to keep an evolving population of solutions is crucial to deriving qualified solutions, solutions that are well distributed on the so-called *Pareto front*. Second, there is a research field called Multi-Objective Evolutionary Algorithm (MOEA) [Deb01] in the EC community, which completely contributes to how to apply evolutionary algorithms to solve multi-objective optimization problems and has shown novelty over the traditional weights-based methods. For task 4, to solve this task with GA, the selection of a suitable representation for this problem and how to encode them into an acceptable format for variation operations are important. Each board status is represented as one solution, therefore the whole set of all possible or legal moves (board status) constitute the search space. The search is carried out in a similar way to that stated above, except the evaluation function is much more difficult, which may include some recursive definitions and rewarding mechanisms.

For task 5 in this thesis, chapter 7 is dedicated to the solution of problem 5, the details of the solution procedure and experiment results will be presented until then. For task 6, again the representation of each solution for this task is not easy and will influence the whole GA search procedure being applied. Also, this task extends our understanding of the problem application range for EC. Evolutionary algorithms are used to solve a wide range of prob-

lem styles, in fact, there is another paradigm algorithm called Genetic Programming (GP) [Koz92, Koz94] which is devoted to solving problems like task 6, where a very popular representation for the solutions is a tree. And all the corresponding search operations are defined according to this special tree structure. Again, the discussion about GP will not be included in this thesis, but GP as another main EC dialogue does share many similar features with GA.

Evolutionary algorithms have the advantage that all possible alternative solutions are kept, therefore potential good solutions (solutions that eventually cause global optimum performance) can be retained. Furthermore, if there are many global optimums, all of them are possibly captured within the population, while for local search methods, at best, there is only one global optimum which can be found. Compared with local search methods, population based search methods have a wider view on the entire problem search space, it is therefore less likely to be cheated in the local optimum. Beyond the consideration of EC as a search-based problem solver, it has some other advantageous aspects, first, the idea of 'EC' itself is a fascinating idea, how to realize and simulate this idea in computers and observe its behavior will create considerable interests in both computer science and biological research. Computers can simulate the millions-of- years-long evolutionary process within hours. Second, EC offers an automatic problem solving method for the rapidly growing and demanding problems field. EC is capable, because it deals with demanding problems in a parallel style; EC is automatic, because it selects good solutions from many alternatives and can generate new and better solutions in a progressive cycle. Finally, EC is not a simple search method, or optimizer, it is a powerful natural problem solver. Two noteworthy and beautiful arts produced by evolution probably are the world we live in and the human brain, which we cannot fully understand yet.

Althourgh having such many advantages, EC still has another very outstanding feature which makes it even more attractive and successful problem solver. This feature is called *adaptation*. In fact, all the EC paradigm algorithms have this feature and show adaptation in different aspects and extents. Especially the ES algorithms have adaptation implicitly and are well-known as introducing self-adaptation into the EC field. We introduce adaptation and self-adaptation here only in context of evolutionary based optimization problem solving methods.

Adaptation is the evolutionary process whereby a population becomes better suited to its habitat. This process takes place over many generations, and is one of the basic phenomena of biology. The term 'adaptation' may also refer to a feature which is especially

important for an organism's survival and reproduction. Such adaptations are produced in a variable population by the better suited forms reproducing more successfully, that is by natural selection. Adaptation is, first of all, a process, rather than a physical part of a body. Adaptation is not always a simple matter, where the ideal phenotype evolves for a given external environment. All adaptations help organisms survive in their ecological niches. In this thesis, we discuss adaptation only in the context of evolutionary computation.

In an evolutionary algorithm, usually, adaptive parameters control takes place when there are some forms of feedback from the search that serves as inputs to a mechanism used to determine the direction or magnitude of the change to the strategy parameter. The assignment of the values of the strategy parameters may involve credit assignment, based on the quality of solutions discovered by different operators/ parameters, so that the updating mechanism can distinguish between the merits of competing strategies. Although the subsequent actions of the EA may determine whether or not the new value persists or propagates throughout the population, the important point to note is that the updating mechanism used to control parameter values is externally supplied, rather than being part of the 'standard' evolutionary cycle.

A more advanced idea introduced by evolutionary computation is the self-adaptation of parameters. Here, the parameters to be adapted are encoded into the chromosomes and undergo mutation and recombination. The better values of these encoded parameters lead to better individuals, which in turn are more likely to survive and produce offspring and hence propagate these better parameter values. This is an important distinction between adaptive and self-adaptive schemes: in the latter the mechanisms for the credit assignment and updating of different strategy parameters are entirely implicit, that is they are the selection and variation operators of the evolutionary cycle itself.

Adaptation is an very important concept in evolutionary computation, later in this thesis we will see concrete instance optimization algorithms which have adaptation as their main advantage compared with other optimization algorithms, also some more advanced hybrid optimization algorithms where adaptation is introduced by other techniques like machine learning and statistics. As adaptation does not only belong to evolutionary computation, when machine learning techniques are introduced, we will also discuss the relationship between adaptation and learning.

Despite its advantages, EC also has some features which are not satisfying when used to solve certain type of problems. Such problems may typically have a huge search space and also the measurement for each solution may be very expensive in computational time.

This leads us back to the time and space limitation issue again, that is, for these problems, the expensive measurement requirement is not affordable for the evolutionary search algorithms, due to the fact that evolutionary search methods are stochastic trial-and-error style problem solving methods. The search is completed by slightly modifying the current solutions blindly and at random. The modification and improvement procedure could be very slow, and they heavily depend on the extent of the change, which needs to be parameterized correctly. However, selecting the good parameters is itself a challenging task and can be seen as an optimization problem on its own. If the parameters are not set suitably, then the expected improvement in performance cannot be achieved easily.

1.1.2 Evaluation is expensive

Throughout our discussion so far, we have not yet emphasized an important concept, the evaluation of solutions. Evaluation is an important component in search based problem-solving methods. It is used to measure the quality of the solutions found during the search procedure. Such a measurement will be utilized in many situations during the search procedure. Those situations include, first, selection of solutions as the objects of being improved. For example, in a GA search, a subset of the population needs to be selected to form the parents which will then reproduce to create new individuals. Second, measurement of the quality of solutions are needed when new solutions are generated. For example, in a hill-climber local search method, when a new solution is generated from the current solution, we need to know whether this solution is better than the current one or not, this comparison needs suitable measurements.

However, evaluation could be very expensive, as in some situations, computer-based simulation and real implementation actions are needed. Let us reconsider the tasks at the beginning of this chapter. To evaluate task 1, in fact, we can only estimate these travel route plans using some previously available information, such as the length of the roads, the usual traffic conditions on these roads, etc. Evaluating these travel plans requires actually traveling on these roads which constitute the routes by car or bus, whose expense is not restored. We never evaluate a route by actual traveling, instead we always estimate in advance. The situation for task 2 is much better, because the evaluation for a mathematical function is simply the computation of the function values with the given function formulae. We discuss task 4 first, the measurement of a good move in a current chess board state is not direct, it may depend on many factors, the strategy the player uses, the previous moves

and the next moves can all determine how 'good' this current move is. A not very attacking move does not mean a bad move, and the most aggressive move cannot guarantee the final win of the game.

For the remaining task 3, task 5, and task 6, we group them as the very expensive evaluation tasks. The evaluation of the good design of an aircraft engine contains many stages, such as computer simulated designs and experiments, long time and distance practical flying tests etc. This is obvious technically and financially expensive. In task 5, the evaluation of a medical treatment plan for chemotherapy against cancer may need to run a computer virtual simulation program to simulate the real effects of injecting medicines into the bodies of patients. This simulation also needs to consider many side-effects which may cause damages to the patient's organs. And the entire procedure needs to be finished in a long period of time (usually months) before we can see the effects of treatment. This makes the evaluation for this task not only expensive but also risky. Finally, for task 6, to evaluate the built model, a procedure called *cross-validation* [Koh95] is usually used in an unbiased way, this procedure contains many rounds, each round involves partitioning the available data set into two subsets, the first data set is the training set performing the analysis, the other data set is the validation set or test set validating the analysis. To reduce variability, multiple rounds of cross-validation are performed using different partitions, and the validation results are averaged over the rounds. As we can see, this evaluation procedure is also very expensive in terms of computation time.

As we have discussed, for this class of problems including task 3, task 5, and task 6, the expensive evaluations in the search procedure can mean that the evolutionary search methods struggle to find relevant good solutions efficiently. The huge size of the population will amplify these expensive measurements and make these problems even more expensive.

1.1.3 Learning is useful

As mentioned before, learning is a frequent task in humans' activities, as it is in scientific research. There is a well-known and advanced research field called Machine Learning (ML) [Mit97], which has been one of the cornerstone topic in Artificial Intelligence ever since its invention. It contains many advanced techniques, and is creating consistently increasing research interests and making rapidly progress. Generally, learning is a procedure that works on past experience, and produce patterns or models which can automatically make intelligent decisions for the future.

There are many standards by which to classify machine learning algorithms. Broadly, we can classify learning algorithms as *supervised*, *unsupervised*, and *reinforcement*, or as inductive and analytical, or as lazy and eager etc. Many classic learning algorithms are supervised. For example, the Decision Tree learning method ID3, Bayesian Network (BN) Bayesian learning [Den], Neural Network (NN) learning [Hay99], Support Vector Machine (SVM) [CV95] learning and k-Nearest-Neighbors (KNN) learning. Supervised learning is the machine learning task of inferring a function from labeled and supervised training data. The training data consist of a set of training examples. In supervised learning, each example is a pair consisting of an input object and a desired output value. A supervised learning algorithm analyzes the training data and produces an inferred function, which is often called a classifier or a regression function for discrete and continuous output. The inferred function should predict the correct output value for any valid input object.

Unsupervised learning refers to the problem of trying to find hidden structures in unlabeled data. Since the examples given to the learner are unlabeled, there is no error or reward signal to evaluate a potential solution. Many clustering algorithms are examples of unsupervised learning.

Reinforcement learning concerns with how an agent should take actions in an environment so as to maximize some notion of cumulative reward. In machine learning, the environment is typically formulated as a Markov Decision Process (MDP), and many reinforcement learning algorithms for this context are highly related to dynamic programming techniques. Reinforcement learning differs from standard supervised learning. For reinforcement learning, correct training data are never presented, and the sub-optimal actions are never explicitly corrected. The focus of reinforcement learning is on-line performance, which involves finding a balance between exploration and exploitation. The basic reinforcement learning model consists of:

1. a set of environment states S;
2. a set of actions A;
3. rules of transitioning between states;
4. rules that determine the reward of a transition;
5. rules that describe what the agent observes.

The rules are often stochastic. The observation typically involves the scalar immediate reward associated to the last transition. In many works, the agent is also assumed to observe

the current environmental state, in which case we talk about full observability, whereas in the opposing case we talk about partial observability. Sometimes the set of actions available to the agent is restricted according to different situations.

A reinforcement learning agent interacts with its environment in discrete time steps. At each time t, the agent receives an observation o_t, which typically includes the reward r_t. It then chooses an action a_t from the set of actions available, which is subsequently sent to the environment. The environment moves to a new state s_{t+1} and the reward r_{t+1} associated with the transition (s_t, a_t, s_{t+1}) is determined. The goal of a reinforcement learning agent is to collect as much reward as possible. The agent can choose any action as a function of the history and it can even randomize its action selection.

When the agent's performance is compared to that of an agent which acts optimally from the beginning, the difference in performance gives rise to the notion of *regret*. Note that in order to act near optimally, the agent must reason about the long term consequences of its actions. Thus, reinforcement learning is particularly well suited to problems which include a long-term versus short-term reward trade-off. It has been applied successfully to various problems, including robot control, elevator scheduling, telecommunications, game-playing.

Among these three learning algorithms categories, as we can see, the reinforcement learning algorithms solve the most general or complex problems. In compared, supervised learning methods solve the most specific or well-defined problems.

Meanwhile, many learning algorithms are inductive learning. Inductive learning is to use the training data to induce a set of hypotheses that describe the given training data on the whole data space under the same distribution for both the training and unseen test data. Once the hypotheses are produced, they can be used to predict the target classification of the unseen data. In the same context, in analytical learning, prior knowledge is used to analyze or explain how each observed training example satisfies the target concepts. This explanation is then used to distinguish the relevant features of the training data from the irrelevant features, so that data can be generalized based on logical rather than statistical reasoning. Therefore analysis learning can improve learning efficiency. In most analysis learning methods, in addition to the training data, an extra rule is derived from the training data and is consistent with both the training data and the corresponding domain theory. This rule, when used for classifying the unseen data, will only be able to predict one class, the unsatisfying data will be classified into another class.

In this thesis, we will use the former standard of supervised, unsupervised and reinforce-

ment learning as the classification standard for machine learning algorithms. And all of the learning algorithms applied and discussed in this thesis are supervised learning algorithms. This is due to the fact that, all of our expecting hybrid optimization algorithms should have the capacity of indicating the promising solutions for the current evolving population, this task can be finished by supervised learning methods with the best performance solutions and the worst performance solutions as the training data. Althrough, the unsupervised and the reinforcement learning can all be used to guide optimization in other ways, it is beyond the range of the research conducted in this thesis.

Having talked about the learning algorithms and the classification standards generally, we will discuss the task that can be solved by these learning algorithms, that is the classification or concept learning task. For a typical classification learning task, the experience data consists of positive and negative training data, each of the data item contains a number of data values which characterize a set of attributes. The last attribute is called the target attribute. The task for a learner or classifier is to find or construct a model that correctly classifies the training data according to the target attribute. After constructing such a model, when a new data item is gained in future, its target attribute value should be correctly predicted with this learned model.

Machine learning is not the only method which concerns the classification task, there are similar methods developed in parallel in statistics. For example, for the decision tree induction algorithm, statisticians have done much work on classification and regression trees, which are the similar methods for generating trees from examples. And the use of nearest-neighbor methods for classification is the standard statistical technique that has been extensively adapted by machine learning researchers to improve classification performance.

From the above introduction of a classification learning procedure, we found that learning contains two main functions, the model building function and prediction function on the model. It is due to these functionalities that learning can be used to guide the evolutionary search procedure, and is able to overcome the shortcomings indicated in the evolutionary based search procedure. That is, learning can be used to construct a model which is the description of the current search space. This model then can be used to predict the quality of the unexplored solutions, and to indicate where the unseen promising solutions could lie. If we consider this effect in the balance of exploration and exploitation, the learning is expected to emphasize exploitation by indicating the promising region and generating more new individuals from this region. This adjustment of the balance would benefit the evaluation-expensive problems by exploiting into the promising area of the search space to

gain promising solutions quickly. This is crucial to the success of solving these problems.

We have introduced the concept of adaptation in 1.1.1 when we talk about search based problem solving methods. One of the most important feature of evolutionary search algorithm is that these algorithms have the capacity of adapting to the environment where it is involved in. Also, the more advanced feature of self-adaptation of the evolutionary algorithms have the capacity of learning the correct strategy parameters and therefore are able to adapt the environment more efficiently. In this section, we also have explored the advanced topics about machine learning algorithms, their capacity, explicit construction forms, and classification standards.

An natural question arise, that is what is the relationship between adaptation and learning. Is there any link between these two advanced features from two seemly different problem solving fields. The role of adaptation and learning are becoming increasingly essential and intertwined. The capability of a system to adapt either through modification of its physiological structure or via some revalidation process of internal mechanisms that directly dictate the response or behavior is crucial in many real world applications. Adaptation is the core capacity for most machine learning approaches and whether the learning algorithms are successful very much depends on their adaptation to the problem environment. This kind of adaptation can also be understood as *optimization*. For example, when the ID3 decision tree learning method is applied on the given training data, the resulting decision tree may suffer over-fitting in some extent, many post-pruning based techniques can overcome this difficulty, showing the adaptation of these methods to construct or learn more accurate tree models. Also, the BP algorithm for neural networks learning is to adapt (or optimize) a set of weight set to find the most suitable network structure for a given training data. Finally, in the reinforcement learning, where the whole environment may not be available, the algorithm can select more appropriate actions according to feedback informations to make progressive performance improvement, such as a procedure is an adaptive procedure. Meanwhile, learning is a primary means to effect adaptation in various forms. They usually involve computational processes incorporated within the system that trigger parametric updating and knowledge or model enhancement, giving rise to progressive improvement. We will see more concrete procedures of this kind in the following section and also we develop such optimization algorithms where learning will effect the adaptation procedures.

1.1.4 Hybrid is the trend

We have introduced two general problem solving methods, search and learning. Many search algorithms can be used to solve complex problems with various features by exploring the solution space in different ways. Learning can be used to understand the current search status or progress, and make constructive suggestions or predictions for the future search procedure. Furthermore, we emphasized the shortcomings faced by the search based methods, especially the fact that the expensive evaluations of some complex problems are more problematic. To this point, we raise the question which drives the development of this thesis. Namely, what is the effect of hybridizing search and learning? More specifically, will a learning method influence the evolutionary search methods when it is embedded in the search procedure? If so, how will learning influence search? Will learning help to overcome the shortcomings of evolutionary search? Before we begin this investigation, we claim that we believe that any learned problem-specific knowledge and previous gained experience can benefit the general search problem solving method, if applied properly. However, these benefits come at a price. First, hybridization means more complex algorithm designs and more computational resources and time. Second, the range of the solved problems by the hybrid algorithms will inevitably be reduced, compared with the more general search algorithms. Namely, we are in favor of the *No Free Lunch Theorem* (NFL)[WM97], which states that if we average over the space of all possible problems, then all black box algorithms will exhibit the same performance.

Hybrid is not a new idea in evolutionary computation, many other methods and data structure have been embedded into them. These new hybrid algorithms are very successful in practice, for example, some of these algorithms are called the Memetic Algorithm (MA) [Mos89]. MA are the evolutionary search algorithms that are combined with a local search. The evolutionary search keeps the basic evolutionary features, such as selection, variation and survival selection, however, when the new solutions are generated, they are further improved by the local search methods before they are involved in the survival selection stages. The idea behind memetic algorithms is clearly based on the balance between exploration and exploitation, the evolutionary search is responsible for exploring the search space, while the local search is used to exploit quickly through the new explored local space. So, the key design of a good memetic algorithm will be how to structure the evolutionary and local search more properly. And also, the success of the memetic algorithm also depends on the problems it solves.

Modern hybrid algorithms have more advanced ideas embodying hybridization. They adopt more advanced algorithms from machine learning and statistics. Excellent paradigm algorithms include the Covariance Matrix Adaptation Evolution Strategies (CMAES), the Estimation of Distribution Algorithms (EDA) and the Learnable Evolution Model (LEM). In this thesis, the development of our new hybrid algorithms is expected to follow the same pattern within these algorithms. And the inspiration of our development is also derived from the algorithms.

CMAES as a general optimization algorithm adopt the Principle Component Analysis (PCA) technique to find the covariance relations of the attributes on the selection mutation steps, and therefore is able to learn from past evolution history. This learning capacity makes CMAES have better optimization performance.

EDAs are variants of the standard evolutionary algorithm. As the name of these algorithms suggests, the new individuals are generated according to a probability distribution rather than the variation operators. These probability distributions are inferred from the previous solutions in the search space by statistical inference methods.

LEM is a more explicit hybrid algorithm. It employs a supervised rule based leaning method called AQ learning algorithm, which can learn from the current solutions based on their performance. The learned model, a set of rules, can distinguish the solutions as two groups with different performances. This learned knowledge is then used to guide the following evolutionary search procedure. This basic principle behind LEM inspired many other ideas and also the development of our hybrid algorithms in this thesis.

1.1.5 Contributions

The contributions of this thesis are:

Contribution 1 A simple genetic algorithm combined with *k*-nearest-neighbors learning algorithm, the LEM Instantiated with KNN algorithm (LEM(KNN)), is developed. KNN in this LEM instance algorithm is used as a 'filter' deciding the survival of the new generated individuals. Also, a further refined variation of the LEM(KNN) algorithm, the LEM Instantiated with distance-weight KNN algorithm (LEM(dwKNN)) is developed. LEM(dwKNN) extends LEM(KNN) with the consideration of distance contributions. The performances of these algorithms are compared with the standard genetic algorithms, showing that significant improvements can be achieved by hybridizing even these very simple learning algorithms with the normal evolution

algorithms.

Contribution 2 Simple genetic algorithm combined with Entropy-Based Discretization (ED), ID3 decision tree learning algorithm are developed, respectively. Some of the resulting algorithms including, the LEM Instantiated with ED algorithm (LEM(ED)), and the LEM Instantiated with ID3 algorithm (LEM(ID3)) are all designed under the general LEM framework and are based on the Learning-and-Generating Hypotheses method, showing the flexibility of this framework. With the development of these LEM instance algorithms, we have also investigated different techniques and methods which are important components of the hybrid algorithms and affect the functionalities and performances of the hybrid algorithms.

Contribution 3 The resulting algorithms LEM(KNN), LEM(ID3) and their variant algorithms are compared with other hybrid algorithms, such as CMAES and EDA, on a number of test problems, including the CEC 2005 real-parameter functions optimization suite and the cancer chemotherapy optimization problem. Performance on these problems have shown these LEM instance algorithms are promising, significantly outperform the standard evolutionary search procedure, and compete well against state of the art hybrid algorithms.

1.2 Outline of the thesis

This thesis contains eight chapters, beginning with this introductory chapter. There are two literature review chapters, introducing search, evolution, learning and hybrid of evolution and learning techniques. Chapter 4 introduces the KNN based LEM hybrid algorithms, Chapter 5 is for the Entropy-Based Discretization method for LEM instance algorithm and the resulting LEM(ED) algorithm. Chapter 6 deals with applying decision tree construction algorithm ID3 as the learning component and the resulting LEM(ID3) algorithm. Chapter 7 introduces and solves the optimization problem of cancer chemotherapy treatments. Conclusions of our work are included in Chapter 8. We introduce the details of these chapters as follows:

The review of search and learning methods used in this thesis is in Chapter 2. Search as a general problem solving method to solve optimization problem is introduced. We classify search algorithms according to two broad classes, the local and population-based search methods. With the emphasis on the population-based search methods, the GA and ES are

introduced, respectively. Some main learning algorithms applied in the hybrid algorithms in this thesis are introduced. These algorithms are inductive learning algorithms, statistical methods and probability-based methods. The decision tree learning algorithm ID3 and the covering algorithm AQ are introduced first along with the two main learning strategies behind them. Two important statistical learning methods, the KNN and the PCA are explained next. Finally, the Bayesian network inference and Bayesian learning are introduced.

Chapter 3 mainly introduces hybrid algorithms. We only focus on three modern hybrid algorithms. They are CMAES, EDA and LEM. The main principles behind these hybrid algorithms are explained. All of these algorithms are used to compare with our hybrid algorithms on a number of test optimization problems in this thesis. And also, the LEM framework is the main source of inspiration for our hybrid algorithms.

Chapter 4 introduces our first and simplest learning and evolution hybrid algorithm - LEM(KNN). First, LEM(KNN) is the simple genetic algorithm combined with the supervised and lazy learning method KNN. Second, LEM(KNN) does not follow the original LEM framework principle, where learning is used as a hypothesis generation method for generating new individuals for the next generation. KNN is used as a 'filter', deciding the survival of the new individuals being generated. This is a new idea in the hybridizing of learning and evolution. It extends the original LEM framework and shows the flexibility of this framework. The flexibility comes from the fact that the new learning method cannot only be embedded into the framework, but also the fact that the ways in which learning and evolution interact can vary. A further refined algorithm of the LEM(KNN) algorithm based on distance weights is developed as well. The resulting algorithm, called LEM(dwKNN), is presented, and its advantages over LEM(KNN) are explained. These algorithms are tested on a number of real-parameter function optimization test problems compared with a standard genetic algorithm, their performances are reported.

Chapter 5 introduces the LEM(ED) algorithm, a genetic algorithm combined with ED. LEM(ED) is developed based on our 'cheap' implementation strategy. Based on this strategy and the 'Learning and Instantiation' method in the LEM framework, the simple ED method is employed as the learning component in this LEM instance algorithm. We also compare it with a simple genetic algorithm and the standard version of CMAES on a number of real-parameter function optimization test problems. The performances of these algorithms are also reported.

In Chapter 6, based on the development experiences of the previous two algorithms, we introduce the LEM(ID3) algorithm, which employs the decision tree learning algorithm ID3

as the learning component and a standard genetic algorithm as the evolution component. ID3 uses training data derived from current population to construct a decision tree, which is then transformed into a set of rules representing the learned hypothesis, based on this hypothesis, the new individuals are instantiated. We designed and developed new techniques and methods important in the success of the development of this hybrid algorithm. The performance of LEM(ID3) was tested on the CEC 2005 special session on real-parameters function optimization. And the performance is compared with two variant CMAES algorithms and advanced evolutionary algorithms, the Local Restart CMAES (LR-CMAES) algorithm, the Restart CMAES With Increasing Population Size (IPOP-CMAES) algorithm and the Population-Based, Steady-State Procedure for Real-Parameter Optimization (K-PCX) algorithm, respectively. Through these results, we found out that LEM(ID3) performs very well and is competitive with these general hybrid optimization algorithms.

Chapter 7 introduces an evaluation-expensive problem, optimization of the treatment plan for cancer chemotherapy, where the saving of the evaluation amount could be very crucial to the success of solving this problem. The evaluation for this problem could be very expensive, including a necessary real simulation procedure either in a virtual computer system or on a patient's body. These procedures usually take a long period of time (months) and risk causing side-effects on patients' organs. Our LEM(dwKNN) and LEM(ID3) algorithms are both applied on this cancer chemotherapy problem, and the results are reported, showing outperformance over the traditional genetic algorithms and competitiveness against the CMAES and variant algorithms of EDA.

Chapter 8 concludes this thesis. First, we summarize the work we have done about hybridizing evolution and learning based on the LEM framework. Second, we list the contributions we have achieved during the development of these hybrid algorithms. Finally, we indicate the work remaining to be further investigated in our future research work.

Chapter 2

Methods for Search and Learning

2.1 Overview

There are three core topics in this thesis: search, learning, and hybrid. In this chapter, we explain the first two topics in detail, and the next chapter for the last topic. Search is used to solve a problem optimization task. We give the definition of optimization for problem solving.

Definition An optimization problem requires us to maximize or minimize some measurable function of one or more variables:

$$y = f(\mathbf{x}) \tag{2.1}$$

subject to $\mathbf{x} \in X$ where $\mathbf{x} = \{x_1, x_2, \ldots x_n\}$ is a *decision vector* and its components are called *decision variables*. Decision vectors are also often referred to as *solutions* or *candidate solutions*. The *search space* which is the set of solutions one is going to search over, it may be some subset or superset of X. The function f is known as the *objective function*. If the goal of the search is maximization then f is sometimes called a *fitness functions* or *utility function*, and the value y assigned to a solution is then its *fitness* or *utility*. Conversely, if the goal is to minimize y then f may be called the *cost function* or in the case of constraint satisfaction, the *penalty function*. However, in this thesis, we will not distinguish between these names and will simply use the term *fitness functions*. Also, we consider maximization and minimization as equal optimization tasks, because they can be easily transformed to each other.

To tackle the optimization problems, many search based methods have been developed. We will explore some of these algorithms in the following sections. Our concerns in this

thesis are the evolutionary population-based search methods, for better understanding, we compare them with another important class of search algorithms, the local search. The introduction to the local search will also provide a good explanation of the global search capacity, which is claimed by the evolutionary search methods. Therefore, evolutionary computation is used to solve global optimization problems. Generally, search methods not only concern how to generate candidate solutions, but also whether the solutions satisfy some optimality criteria, that is the *constraint satisfaction* issue. We do not deal with this issue in general, but will discuss it when it is met in the concrete problem.

Learning is used to solve another type of task, classification or conception learning in this thesis. We will give the definitions later when the learning algorithms are examined. These learning algorithms have either been applied in the hybrid algorithms which are involved in our experiment comparison or will be incorporated into our hybrid algorithms. We introduce these learning algorithms with a special emphasis on solving the classification problems which is concerned within our hybrid algorithms.

2.2 Search Algorithms for Optimization

We explore the search algorithms in detail in this section. Before starting this introduction, a word on general purpose problem solver is given first. General purpose problem solver is applicable to any optimization problem. It does not require any problem-specific knowledge and structure, the only requirement is the objective function for the problem. Very often, one does not have any insight into how a problem might be solved, or which strategy should be used. In these cases, it is best to use a more general strategy, often called a *metaheuristic*. Metaheuristic is sometimes also called *black-box* optimization algorithms or simply, *general purpose optimization algorithms*.

2.2.1 Local Search

There is a class of metaheuristic optimization algorithms which are based on a neighbor structure, these metaheuristics are called Local Search (LS) [AK89]. Local search algorithms work by finding a solution maximizing a criterion among a number of candidate solutions. They move from one solution to another based on the neighbor structure in the space of candidate solutions, until a solution deemed optimal according to the criterion is found or a time limit is elapsed. For example, a well-known local search algorithm called Hill Climbing (HC) works by taking a starting solution x, and then searching the candidate

solutions in its neighbors $N(x)$ for one x' that performs better than or equal to x. If such a solution exists, then this is accepted as the new incumbent solution, and the search proceeds by examining the candidate solutions in $N(x')$. Hill climbing is an iterative process of examining the set of points in the neighborhood of the current solution, and replacing it with a better neighbor if one exists. Eventually, this process will lead to the identification of a local optimum: a solution that is superior to all those in its neighborhood. Let us look at the definition for the general HC algorithm:

Definition Let (X, f) be an instance of a combinatorial optimization problem. A neighborhood function is a mapping $\mathcal{N} : X \longrightarrow 2^X$, which defines for each solution $i \in X$ a set $\mathcal{N}(i) \subseteq X$ of solutions that are in some sense close to i. The set $\mathcal{N}(i)$ is the neighborhood of solution i, and each $j \in \mathcal{N}(i)$ is a neighbor of i. We shall assume that $i \in \mathcal{N}(i)$ for all $i \in X$. Roughly speaking, a HC algorithm starts off with an initial solution and then continually tries to find better solutions by searching neighborhoods.

According to the above definition, a very good example of hill climbing algorithm is the Random Mutation Hill Climbing (RMHC), as described in [MHF93]. In RMHC, an initial solution is first generated and evaluated, and this becomes the current solution. Then at each iteration, a copy of the current solution is made, and a random mutation is applied to the copy, producing a new candidate solution. The candidate solution is then evaluated, if it is not worse than the current solution then it becomes the current solution; otherwise, it is discarded. The algorithm may be stopped when a specified number of evaluations have been carried out, or when there has been no improvement in the evaluation of the current solution over a specified number of iterations. The general HC algorithm can be illustrated as Algorithm 1:

Here, the 'local conditions' are some local termination conditions, such conditions could be $count = |n(i)|$, which indicates that the current local search will stop if all the neighbors for the current solution are considered. It can also be changed as $(count = |n(i)|) \parallel (best \neq i)$, which means as long as a better solution is found, the iteration will start again using the better solution as the new initial solution. The iteration conditions are the termination condition for the whole local search algorithm, it is used to decide the depth of the whole search, and could be simply the maximum allowed number of iterations.

However, the drawback of hill climbing based local search algorithms is that, in general, it cannot get out from the local optima, and cannot find the global optimization. And also, the performance of hill climbing algorithms very much depends on the initial solu-

Algorithm 1 pseudo code for the general *hill climbing* algorithm

```
1:  Set best = i;
2:  Set iterations = 0;
3:  repeat
4:      Set count = 0;
5:      repeat
6:          Generate the next neighbor j ∈ n(i);
7:          Set count = count + 1;
8:          if ( f(j) is better than or equal to f(best)) then
9:              Set best = j;
10:         end if
11:     until (Local conditions are satisfied)
12:     Set i = best;
13:     Set iterations = iterations + 1;
14: until (Iteration conditions are satisfied)
```

tions. However, problems frequently exhibit numerous local optima, some of which may be significantly worse than the global optimum, therefore no guarantees can be offered as to the quality of the obtained solutions by local search algorithms. A number of methods have been proposed to get around this problem, in principle, they all try to change the search landscapes in different ways.

Multi-Start Hill Climber

The problem of converging to a local optimum is overcome by restarting the search with new search points within the Multi-Start Hill Climber (MSHC) [YI96] algorithm. It is a modification of the hill climbing search strategy. The multi-start search defines a restart of the algorithm from a new, random initial solution. After each iteration, when there is no improvement in the evaluation of the current solution, search starts from a point very far away from the optimum and no information obtained from previous iterations is reused. If the algorithm is allowed to restart indefinitely according to this criterion, then it will find a global optimum with probability 1.0 on all optimization problems. This is clear since it will eventually search all neighborhoods in the search space. However, the length of time needed to do this will, in general, exceed that needed for a deterministic enumeration of the whole search space.

Variable Neighborhood Search

Another method to overcome the drawback of hill climbing is to change the neighborhood function, the Variable Neighborhood Search (VNS) [HMMP08]. It is a relatively recent metaheuristic which relies on iteratively exploring neighborhoods of growing size to identify better local optima. More precisely, VNS escapes from the current local optimum by initiating other local searches starting from points sampled from a neighborhood of current solutions. In this way, the current point's neighbor size is increased iteratively until a local optimum better than the current one is found. These steps are repeated until a given termination condition is met.

Simulated Annealing

There exists an well-known local search algorithm called Simulated Annealing (SA) [AK89] [KGV83], which is a generic method based on the Markov Chains. It is quite similar to the RMHC algorithm, however it improves upon hill climbing algorithms by occasionally allowing movements to worse solutions and is thus capable of jumping out of local optima. The method draws an analogy from the annealing procedure of metals, where the temperature controls the arrangement of atoms in their lowest energy configuration during the crystallization process. In simulated annealing, moves are accepted or rejected with a certain probability depending on a function of the temperature, such that at higher temperatures, there is greater probability of accepting inferior moves. Temperature is gradually brought down so that the solution converges. This crucial difference with RMHC means that simulated annealing is able to search for a global optimum, and under certain conditions it converges to a globally optimal solution with probability 1.0. In simulated annealing, the probability function for accepting the candidate solution j from the current solution i for a minimization problem is:

$$P_{c_k}\{accept\ j\} = \left\{ \begin{array}{ll} 1.0 & \text{if } f(j) \leq f(i); \\ exp(\frac{f(i)-f(j)}{c_k}) & \text{if } f(j) > f(i). \end{array} \right\} \tag{2.2}$$

where $c_k \in \mathcal{R}^+$ is a control parameter, which is some function of the iteration k of the simulated annealing algorithm. In SA , the value of c_k is set initially high, and is gradually lowered to zero, so that initial transitions to highly inferior solutions are frequently accepted, but later these transitions become extremely unlikely. The regime for controlling c_k is called the *cooling schedule*, and it specifies an initial value of the control parameter c_0, a decrement function for lowering the value of the control parameter, a final value of the

control parameter specified by a stop criterion, a finite number of transitions at each value of the control parameter.

Proofs have been given in [AK89], that the SA algorithm converges to the global optimum with probability 1.0, provided that the sequence of trials (or Markov chains) approximate a stationary distribution. However, this requires that an exponential number of trials are performed, and for some problems, it requires more computation than a complete enumeration of the search space. Despite this extreme complexity, simulated annealing has been practically applied in a large range of applications to solve optimization problems.

Tabu Search

Tabu Search (TS) [Glo96] is a metaheuristic algorithm that uses a local or neighborhood search procedure to iteratively move from a solution x to a solution x' in the neighborhood of x, until some stopping criterion has been satisfied. To explore regions of the search space that would be left unexplored by the local search procedure, tabu search modifies the neighborhood structure of each solution as the search progresses. The solutions admitted to $N^*(x)$, the new neighborhood, are determined through the use of memory structures. The search then progresses by iteratively moving from a solution x to a solution x' in $N^*(x)$.

Perhaps the most important type of memory structure used to determine the solutions admitted to $N^*(x)$ is the tabu list. In its simplest form, a tabu list is a short-term memory which contains the solutions that have been visited in the recent past. Tabu search excludes solutions in the tabu list from $N^*(x)$. A variation of a tabu list prohibits solutions that have certain attributes or prevent certain moves. Selected attributes in solutions recently visited are labeled 'tabu-active'. Solutions that contain tabu-active elements are 'tabu'. This type of short-term memory is also called 'recency-based' memory.

Tabu lists containing attributes can be more effective for some domains, although they raise a new problem. When a single attribute is marked as tabu, this typically results in more than one solution being 'tabu'. Some of these solutions that must now be avoided could be of excellent quality and might not have been visited. To mitigate this problem, 'aspiration criteria' are introduced: these override a solution's tabu state, thereby including the otherwise-excluded solution in the allowed set. A commonly used aspiration criterion is to allow solutions which are better than the currently-known best solution.

Although these and many other algorithms have been developed to tackle the drawbacks of the hill climbing strategy, as we can see, it can only be solved to some extent but not completely. This is due to the key fact that the real focus throughout the search procedure

in the local search and all its variants algorithms is the current solution, which is also why they are referred to as local searches. So, however the local search strategy is modified or improved, it can essentially never overcome this drawback. This is the reason why we need a completely different strategy in the search for optimization.

2.2.2 Genetic Algorithm and Global Optimization

Evolutionary Computation [Bäc96] [TFM99b] [TFM99a] [Mic96] and population based search techniques have recently gained considerable attention. Unlikely the above local search methods, these new search-based problem solving methods work on a whole population of solutions, it is this feature which makes them in principle able to solve the local optimum problem faced by local search. Therefore, they are also called *global search* methods, and the procedure of applying these methods to solve optimization problems is called *global optimization*. Global optimization can also be defined under the neighbors structure used in local search, namely, the global optimum x^* is fitter than all of its neighbors under any neighborhood structures.

Genetic Algorithm

One of the most well-known instance algorithms of the EC family is the Genetic Algorithm [Hol75, Gol89, DJ75], which has been used for solving a wide range of problems including function optimization problems and complex optimization problems, where it is impossible to obtain exact solutions within a reasonable amount of time. GA draws an analogy from the evolution of species in biology. Species evolve by means of genetic operators such as crossover and mutation, and they survive through the mechanism of *survival of the fittest*. In genetic algorithms, this process is simulated by encoding potential solutions (individuals) using a chromosome-like data structure. New individuals (children) are created in the population through reproduction using crossover and mutation operators. These operators ensure that children inherit qualities of parents and they are passed on from one generation to the other. Only a certain number of good quality individuals survive each generation and this ensures that the quality of the population improves with each generation.

The original GA algorithm introduced by John Holland [Hol75] is sometimes known as the 'simple genetic algorithm (SGA)', or the 'canonical GA'. SGA works by generating an initial population of chromosomes or individuals, evaluates the population using a fitness function. Parent selection operation selects an intermediate population based on the fitness

values and put these individuals in mating pool. These individuals are parents waiting to take part in reproduction to generate the next population. Then reproduction operators crossover and mutation are applied on parent individuals to create the next population – the offspring. These offspring are first evaluated, then survival selection is applied on these offspring to decide who will survive to the next generation. The above procedure repeats until some termination conditions are met. The flowchart of SGA is given as Figure 2.1.

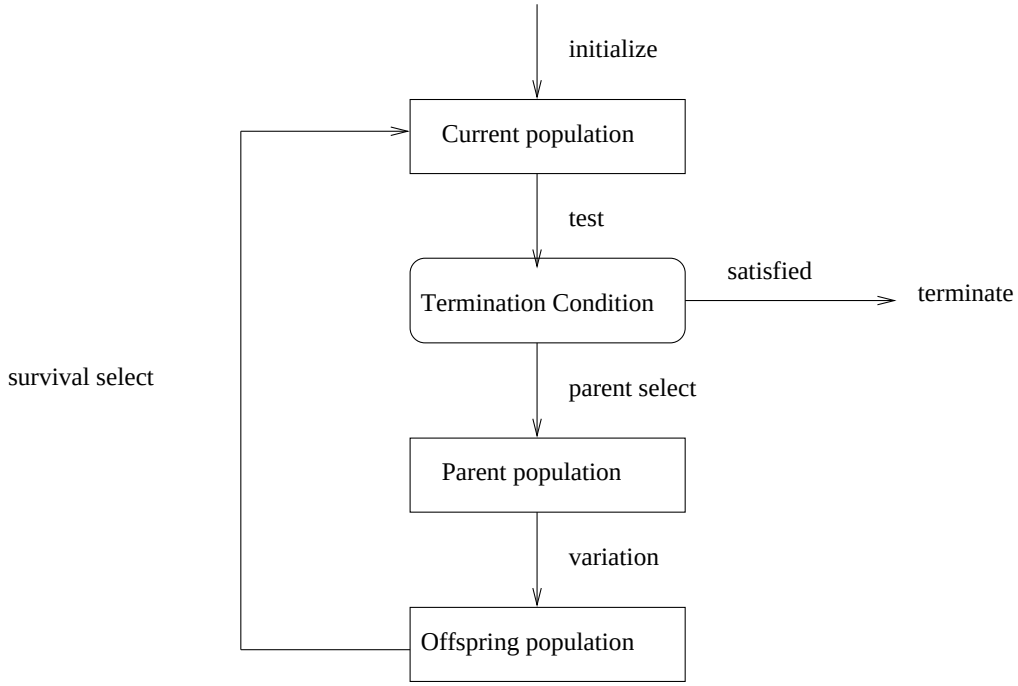


Figure 2.1: Flowchart of the simple genetic algorithm

From the above descriptions and the flowchart of SGA, we notice that population is the basic unit for a GA to operate on. The population is characterized by many properties, such as population size and diversity, which all influence the performance of the GA. The following GA operations we will discuss are all based on population.

The first step in designing a genetic algorithm is to define a suitable representation for the solutions. Namely, to define the representation in the real world, the *phenotypes*, and the representation in GA space, the *genotypes*. A good design for representation is actually to find a good map (encoding) from phenotypes to genotypes. In phenotype space, we use the terms solutions or individuals, in genotypes, we use *chromosome*. Each individual or chromosome consists of a set of *variables*, or *genes*.

Another important component in GA is the *evaluation* or *fitness function*, which is used to assign a quality measurement. This measurement is used to distinguish the individuals within one population according to their quality, and also for the parent and survival selection procedure to act. In optimization problem, where we search to optimize a certain kind

of objective, it is also called *objective function*.

The selection of individuals to be placed in the mating pool is performed according to a probability function that depends on some standards. According to this scheme, individuals having higher fitness might be replicated multiple times to be placed in the mating pool. The most well-known selection methods are fitness proportional selection [Hol75], ranking based selection [Bak87], binary tournament selection [Bäc95, BT96, GD91], all of which can finish the task of selecting some promising individuals as parents.

After the generation of parent population (in the mating pool), the variation operations are applied to generate new offspring. There are two main variation operations, crossover and mutation. According to different representations, the operators act differently. For example, for binary representation, mutating one gene is simply flipping one bit from 1 to 0, or from 0 to 1. For real representation, mutating one gene could be changing the current gene value x_i by adding a new value generated according to a distribution. Meanwhile, for binary representation, crossover is the operation that exchanges parts of the two parent individuals into each other according to one or several predefined exchange points. For real representation, crossover can be implemented as average values of the two parents for all genes. Crossover is applied to randomly selected pairs of individuals according to a probability p_c, called *crossover probability*. Meanwhile, mutation operator is applied with a low probability p_m, the *mutation probability*, on each gene of each individual.

When the new individuals are generated by crossover or mutation, survival selection must be applied in order to decide which individuals from the set of parents and offspring should survive into the next generation given that the resources of the environment are limited, that is, the population size is fixed. There are also many standards for survival section, such as, they can be age-based, the generational model, where the old individuals are all replaced by the new individuals; or they can be fitness based, the steady-state model, which we will mention soon in the following sections.

GA works due to two main operations: variation and selection. Variation is to generate new individuals, it is the main search operator. The capacity of variation operations are crucial to the success of GA, they should be able to generate new individuals different enough from the existing individuals to represent the unexplored search space. Selection operation is also important in that, good individuals should be selected and undergo change to reproduce better individuals. Namely, the combinations of good genes from the promising individuals should be inherited and changed to better combinations of genes, therefore producing better individuals with better fitness values.

We have introduced the idea of balance between exploration and exploitation, and also the balance between them affects the quality and efficiency trade-off issue mentioned in chapter 1. Good design of control strategy for this balance is crucial for the success of the GA search algorithm. Because, if a GA search focuses on too much exploration, it will lead to an inefficient search, wandering around in the search space. If the search focuses on too much exploitation, it will lead to a propensity to focus the search too quickly, causing *Premature Convergence* which is the well-known effect of losing population diversity too quickly and getting trapped in a local optimum.

There are basically two ways to realize this balance, the implicit and explicit controls of the search procedures. For the implicit way, for example, the selection pressure control can be considered an implicit control of the balance between exploration and exploitation. Too much selection pressure results in converging too quickly, too little selection pressure results in random search behaviors. For the explicit way, some parameterized methods can be used to control the process of the search, for example, the parameter C_k in simulated annealing. Good parameters can make the search explore more in the early stages, while it exploits more in the latter stages. And also, any form of operating heuristics or domain knowledge can be used to emphasize exploitation, while reducing the utilization of this information will result in more exploration.

The SGA algorithm is based on a generational model. There is also another model called *steady-state* genetic algorithm, where the whole population is not changed at once, but rather a part of it. In this case, if the given population size is μ, and λ new individuals are generated, then λ old individuals are replaced by λ new offspring, the percentage of the population that is replaced is called the *generational gap* which is equal to λ/μ. An instance of this algorithm is the GENITOR algorithm [WK88], where two parents are selected for reproduction and the offspring is immediately placed into the population, replacing the worst member of the population, therefore the generation gap is $1/\mu$. The steady-state population model can be seen as an example of fitness based survival selection. Namely, the worst individuals will be replaced by new offspring. For several classes of problems, it has been reported that the steady-state genetic algorithm outperforms the simple genetic algorithm [Sys91].

Over the years, several improvements over the original genetic algorithm introduced by John Holland [Hol75] have been suggested. For example, the CHC algorithm developed by Larry Eshelman [Esh91] is a variation on the genetic algorithm. CHC stands for Cross generational elitist selection, Heterogeneous recombination by incest prevention and Cat-

aclysmic mutation. Important features of the algorithm are: after recombination, N best individuals are selected from the parents and offspring to create the next generation, duplicates are removed from the population, individuals are randomly selected for reproduction. However, certain restrictions are imposed on which strings are allowed to mate, strings within a certain hamming distance are not allowed to mate. A form of uniform crossover called HUX is used, where exactly half of the differing bits are swapped. When population converges and starts producing more or less identical strings, cataclysmic mutation is activated, all strings except the best are heavily mutated. Recent evaluations indicate that CHC is generally more efficient than SGA and the steady-state genetic algorithm.

We finish the discussion on this section by giving the definitions and formulations of some well-known crossover and mutation operators, particularly, some of which will be used in the implementation of the algorithms developed in this thesis.

Crossover Operators

Throughout the GA research community, many crossover operators have been developed. In this thesis, we introduce some very typical crossover operators, and also those applied in our GA implementation for experimental comparisons.

For binary representation, given an individual with length l. One-point crossover works by choosing a random number in the whole range (0, l - 1) of the binary characters, and then splitting both parents at this point and creating the two children by exchanging the remaining parts. One-point crossover can be naturally extended to n-point crossover and uniform crossover. In n-point crossover, the individual is divided into several genes segments, and then the offspring are created by taking alternative segments from the two parents. A further generalization of n-point crossover is the uniform crossover, which considers each gene independently and divides the whole range of individual into l genes with (l - 1) points. When new offspring are generated, a set of l random numbers are generated from a uniform distribution over (0,1). In each position of the first offspring, if the value is below a parameter p (say 0.5), the gene is inherited from the first parent; otherwise from the second. The second offspring is generated inversely.

For real parameter representation, there have also been a number of crossover operators available. The simplest one is the *discrete* crossover or also called *naive* crossover, this is analogous to the binary crossover operator. The drawback of this simple mechanism for real number individuals is that it cannot generate new genes values but only new combinations of existing genes, and it will limit the search capacity within the real numbers search space.

In contrast to the discrete crossover, the *arithmetic* or *intermediate* crossover calculates a new value for each gene position according to some formulations with the values from parents. If x_i and y_i are the gene values of the two parents at position i, the new value for the child at position i will be $z_i = \alpha x_i + (1 - \alpha)y_i$, where $\alpha \in (0, 1)$. That is, the new gene value of the child is generated depending on the values from parents.

We have seen some basic crossover operators, the introduction about these operators gives good explanations about the working principle of crossover or recombination operators. However, in practical problems, these operators are not very efficient due to their simplicity. In the following sections, we will see some more advanced and carefully designed crossover operators developed in the GA community. First, a generalized arithmetic crossover operator called Blend Crossover operator [ES93] (the BLX-α crossover) for real-parameter representation is presented, this operator can be seen as an extension of the arithmetic crossover with some adaptive search capacity. For two given parents solutions x_i^1 and x_i^2, the following is an offspring solution generated by BLX-α:

$$x_i^{'} = (1 - \gamma_i)x_i^1 + \gamma_i x_i^2 \tag{2.3}$$

where $\gamma_i = (1 + 2\alpha)\mu_i - \alpha$ and μ_i is a random number between 0.0 and 1.0. If α is zero, this crossover creates a random solution in the range (x_i^1, x_i^2). In a number of test problems, the investigators have reported that BLX-0.5 (with $\alpha = 0.5$) performs better than BLX operators with any other α value. However, it is important to note that the factor γ_i is uniformly distributed for a fixed value of α. However, BLX-α has an interesting property: the location of the offspring depends on the difference of the parent solutions. This will be clear if Equation 2.3 is rewritten as follows:

$$(x_i^{'} - x_i^1) = \gamma_i(x_i^2 - x_i^1) \tag{2.4}$$

If the difference between the parent solutions is small, the difference between the offspring and parent solutions is also small. This property of a search operator allows it to constitute an adaptive search. If the diversity in the parent population is large, an offspring population with a large diversity is expected, and vice versa. Thus, such an operator will allow the search to explore the entire space in the early generations and also allow to maintain a focused search when the population tends to converge in some region in the search space in the later generations.

Some other crossover operators work with the same principle. The arithmetic crossover

uses Equation 2.3 with a fixed value of γ for all decision variables. However, γ is chosen by carefully calculating its maximum allowed value in all decision variables so that the resulting values do not exceed the lower or upper limits. The arithmetic crossover operator can be seen as a specified operator of BLX-α crossover.

There are still some other advanced crossover operators, such as Simulated Binary Crossover and Parent Centered Crossover. We briefly highlight some important properties of these operators and refer the reader to the relevant references for the details of the descriptions of these operators. For these advanced crossover operators, the new generated individuals depend on the parents' values according to some particular distributions rather than random distribution, the difference between the offspring is in proportion to the difference between parent solutions. Namely, near-parent solutions are monotonically more likely to be chosen as offspring than solutions distant from parents. Finally, these operators inevitably introduce new extra parameters in order to achieve the adaptive search capacity.

Mutation Operators

The mutation operator is rather straightforward compared with crossover. Again, we need to discuss mutation both in binary and real parameter situations. For binary representation, mutation is simply flipping the gene values from 0s to 1s with a certain probability. For real representation, random (uniform) mutation and nonuniform mutation are widely used. Random mutation is to generate a new gene value for the offspring individual randomly from the whole range of the gene value. It is more like a re-generating operator and the parent gene value will have no effect on the new gene value for offspring. The nonuniform mutation, on the contrary, works by adding to the parent gene value a new value drawn randomly from a predefined distribution, for example, a Gaussian distribution with mean zero and user-specified standard deviation, as defined in Equation 2.5. In this way, the parent gene value is changed according to a distribution rather than randomly re-generated within the whole gene range. Normal distribution mutation is particularly important in evolutionary computation research, we will see its applications in the next section as we discuss another important EC family algorithm.

$$x_i' = x_i + \sigma \cdot N(0, 1). \tag{2.5}$$

Finally, as for crossover operators, researchers have also designed advanced mutation operators, an example is the Polynomial mutation [Deb01], which has a similar idea to the Simulated-Binary Crossover, a probability distribution is applied to generate a new gene

value for the offspring, and this value very much depends on the parent gene value and the rule that the distribution defines.

2.2.3 Evolution Strategies

Evolution Strategy is another main member of the evolutionary computation family. It was invented by Rechenberg and Schwefel in the early 1960s [Rec73],[Sch95],[BS02]. One of the main contributions of ES to the EC community and the key feature which distinguishes it from the rest of EC members is the *self-adaptation* of the strategy parameters. Self-adaptation means the parameters that decide the evolutionary performance are varied during the runs of the algorithm, we call these parameters the *strategy parameters*, which are different with the *object parameters* representing the chromosomes. During the ES evolution procedure, the strategy parameters are coevolved together with the object parameters. Before any further explanations, we first describe a basic two-membered evolution strategy for the optimization problem of minimizing an n-dimensional function. The outline of this evolution strategy is given as Algorithm 2:

Algorithm 2 pseudo code for an evolution strategy algorithm

1: Set $t = 0$;
2: Create an initial point $\langle x_1^t, \ldots, x_n^t \rangle \in R^n$;
3: **repeat**
4: Draw z_i from a normal distribution $\mathcal{N}(0, \sigma)$, for all $i \in \{1, \ldots, n\}$ independently;
5: $y_i = x_i + z_i$ for all $i \in \{1, \ldots, n\}$;
6: **if** $(f(\vec{x}^t) \leq f(\vec{y}^t))$ **then**
7: $\vec{x}^{t+1} = \vec{x}^t$;
8: **else**
9: $\vec{x}^{t+1} = \vec{y}^t$;
10: **end if**
11: Set $t = t + 1$;
12: **until** (Termination condition is satisfied)

From this simple algorithm, we can find the basic principles and features of ES. First, ES is typically used for real parameter optimization. ES directly operates on the phenotypes space that is the real valued vectors, the problem at hand can be given as an objective function $R^n \rightarrow R$. Second, the mutation operator is the main operation to generate new offspring. Given a current solution $\vec{x}^t$ in the form of a vector of length n, a new candidate

$\overrightarrow{x}^{t+1}$ is created by adding a random number z_i for $i \in \{1, \ldots, n\}$ to each of the n components. A Gaussian or normal distribution is used with zero mean and standard deviation σ for drawing the random numbers, σ is also called the *mutation step size*.

Self-adaptation

As mentioned before, the main feature of ES is self-adaptation, which is reflected in two aspects. For the representation, each individual in ES contains two parts, the first part is the object parameters $(x_1, \ldots, x_n)$ representing the individual itself. The second part is the strategy parameters which contain two sets of values σ and α. The σ values represent the mutation step sizes and their number n_σ is usually either 1 or n. The α values represent interactions between the step sizes used for different variables. So, the general representation of individuals in ES is:

$$\langle x_1, \ldots, x_n, \sigma_1, \ldots, \sigma_{n_\sigma} \alpha_1, \ldots, \alpha_{n_\alpha} \rangle$$

The second aspect where the self-adaptation feature can be reflected is the mutation operator. That is the adaptation of the strategy parameters for the mutation operation during ES runs. There have been two main ways to implement this self-adaptation in the literature, one of these is the covariance matrix adaptation, which determines the probability distribution for mutation, we will talk about this strategy in detail in Chapter 3. For now, we discuss another method, that is the explicit use of self-adaptive control parameters methods [Rec73], [Sch88]. The strategy parameters are explicitly coded along with the decision variables and updated by using predefined update rules in each generation, there are basically three different implementations which are in use.

1. Uncorrelated mutation with one step size, (Isotropic Self-Adaptation).

   In this case of uncorrelated mutation with one step size, the same distribution is used to mutate each x_i, therefore there is only one strategy parameter σ in each individual. This σ is mutated each time step by multiplying it by a term e^{Γ}, with Γ a random variable drawn each time from a normal distribution with mean 0 and standard deviation τ. The mutation mechanism is thus specified by the following formulas:

$$\sigma' = \sigma \cdot e^{\tau \cdot N(0,1)}, \tag{2.6}$$

$$x_i' = x_i + \sigma' \cdot N_i(0, 1). \tag{2.7}$$

In Equation 2.6, $N(0,1)$ denotes a draw from the standard normal distribution, while in Equation 2.7 $N_i(0,1)$ denotes a separate draw from the standard normal distribution for each variable i. The proportionality constant τ is an external parameter to be set by the user. It is usually inversely proportional to the square root of the problem size, $\tau \propto 1/\sqrt{n}$. The parameter τ can be interpreted as a kind of *learning rate*. The reasons that mutating σ by multiplying with a variable with a lognormal distribution are explained in [Bäc96] as, first, smaller modifications should occur more often than large ones; standard deviations have to be greater than 0.0; the median should be 1.0; mutation should be neutral on average, this requires equal likelihood of drawing a certain value and its reciprocal value for all values. Under this scheme, the representation for each individual now has the form $\langle x_1, \ldots, x_n, \sigma \rangle$.

2. Uncorrelated mutation with n step sizes,(Non-Isotropic Self-Adaptation).

   The motivation behind using n step sizes is the wish to treat dimensions differently. In particular, different step sizes are expected to be used for different dimensions $i \in \{1, \ldots, n\}$. The reason for this is the difficulty and complexity that the fitness landscape can have different slopes for each direction on each axis. Therefore, each basic chromosome $\langle x_1, \ldots, x_n \rangle$ is extended with n different step sizes, one for each dimension. The new mutation mechanism is now specified as follows:

$$\sigma_i' = \sigma_i \cdot e^{\tau' \cdot N(0,1) + \tau \cdot N_i(0,1)}, \tag{2.8}$$

$$x_i' = x_i + \sigma_i' \cdot N_i(0,1). \tag{2.9}$$

   where $\tau \propto 1/\sqrt{2n}$, and $\tau' \propto 1/\sqrt{2\sqrt{n}}$. The sum of two normally distributed variables is also normally distributed, hence the resulting distribution is still lognormal. The conceptual motivation is that the common base mutation $e^{\tau' \cdot N(0,1)}$ allows for an overall change of the mutability, guaranteeing the preservation of all degrees of freedom, while the coordinate-specific $e^{\tau \cdot N_i(0,1)}$ provides the flexibility to use different mutation strategies in different directions. Under this scheme the individual is represented as $\langle x_1, \ldots, x_n, \sigma_1, \ldots, \sigma_n \rangle$.

3. Correlated mutations.

   The rationale behind correlated mutations is to allow the variable vector to have any orientation by rotating them with a rotation covariance matrix C. Each entry of this

matrix is decided by the mutation step sizes and the angles between the dimensions. Therefore, the entry of the covariance matrix is $c_{ij(i\neq j)} = 1/2(\sigma_i^2 - \sigma_j^2)\tan(2\alpha_{ij})$, if there is correlationship between the i and j dimensions. The new mechanism is now formulated as:

$$\sigma_i' = \sigma_i \cdot e^{\tau' \cdot N(0,1) + \tau \cdot N_i(0,1)}, \tag{2.10}$$

$$\alpha_j' = \alpha_j + \beta \cdot N(0,1), \tag{2.11}$$

$$\mathbf{x}' = \mathbf{x} + \mathbf{N}(\mathbf{0}, C'). \tag{2.12}$$

where $\tau \propto 1/\sqrt{2n}$, and $\tau' \propto 1/\sqrt{2\sqrt{n}}$. The parameter β is fixed as 0.0873 (or 5^o). The σ_i are mutated in the same way as before in Equation 2.8. The α_j are mutated with an additive, normally distribution variation, similar to mutation of object variables. The object variables $\mathbf{x}$ are now mutated by adding the variance drawn from an n-dimensional normal distribution with covariance matrix C'.

In correlated self-adaptation, in addition to n mutation strengths, at most $n \cdot (n-1)/2$ covariances α are included in each individual solution. So, there are a total of $n \cdot (n+1)/2$ strategy parameters to be updated for each solution. Thus, this type of self-adaptive ES can adapt to problems where decision variables $\mathbf{x}$ are correlated. In a correlated problem, the task is to find all pair-wise coordinate rotations and the spread of solutions in each rotated coordinate system so that the objective function is completely uncorrelated in the new coordinate system. Under this scheme, the individual has the representation of the form in its general form.

Traditionally, evolution strategies do not use any crossover operators. However, ES can be equipped with any form of real coded crossover operators discussed in the GA section. The parent selection operation in ES is not biased by the fitness values. Whenever a parent is needed, it is drawn randomly with uniform distribution from the current whole population. As discussed in the previous GA section, the parent selection is one of the main influences causing the improvement of the average quality of the current population. Evidently, the uniform parent selection operation cannot fulfil this function, this task is finished especially by the survival selection in ES.

Survival Selection

In ES, there are two survivor selection schemes, after creating λ offspring and calculating their fitness values, the best μ of them are chosen deterministically, either from the offspring only, called (μ, λ) selection, or from the union of parents and offspring, called $(\mu + \lambda)$ selection. Both the (μ, λ) and $(\mu + \lambda)$ selection schemes are strictly deterministic and are based on rank rather than the absolute fitness values. The selection scheme that is generally used in evolution strategies is the (μ, λ) selection, which is preferred over the $(\mu + \lambda)$ selection for the following reasons:

- The (μ, λ) discards all parents and is therefore in principle able to leave (small) local optima, so it is advantageous in the case of multimodal landscapes.
- If the fitness function is not fixed, but changes in time, the $(\mu + \lambda)$ selection preserves outdated solutions, so it is not able to follow the moving optimum well.
- $(\mu + \lambda)$ selection hinders the self-adaptation mechanism with respect to strategy parameters to work effectively, because misadapted strategy parameters may survive for a relatively large number of generations when an individual has relatively good object variables and bad strategy parameters.

The selective pressure in ES is generally very high, because the λ value for offspring is much higher than the μ value for parents. Usually, $1/7$ ratio is recommended. We finish the discuss about ES and self-adaptation here, and will come back to this topic in the next chapter when we talk about hybrid optimization algorithms.

2.2.4 Other General Purpose Search Algorithms

The research and development in search based problem solving methods have been flourishing, many other metaheuristic algorithms have been developed to solve optimization problems. The ideas of these algorithms are more or less similar but with a different emphasis on different aspects.

Ant Colony Optimization

The Ant Colony Optimization (ACO) [DBT00] algorithm is one of these well-known metaheuristic algorithms. The ACO algorithm aims to search for an optimal path in a graph, inspired by the behavior of ants seeking a path between their colony and a source of food.

During the construction procedure to find the shortest path with a given graph, the ants incrementally build solutions by moving around in the graph, each ant starts from a randomly selected vertex of the construction graph. Then at each construction step, it moves along the edges of the graph, keeping a memory of its path, and in subsequent steps it chooses from the edges that do not lead to vertices that it has already visited. An ant has constructed a solution once it has visited all the vertices of the graph. This construction process is stochastic and is biased by the *pheromone* value. Pheromone is actually a positive feedback information left by the ant when it is touring through a route, it reflects the attractive strength of this path, when a complete route is found, the pheromone value for this route is calculated according to the quality of this route. That is, the shorter route will have more pheromone value added to it, the longer, the less added. Pheromone evaporates over time, thus reducing its attractive strength, that is, the set of pheromone parameters associated with graph nodes or edges are modified at runtime by the ants. At each construction step, an ant probabilistically chooses the edge to follow to yet unvisited vertices. The probabilistic rule is biased by pheromone values, the higher the pheromone value to an edge, the higher the probability an ant will choose that particular edge. Once all the ants have completed their tour, the pheromone on the edges is updated. Each of the pheromone values is initially decreased by a certain percentage. Each edge then receives an amount of additional pheromone proportional to the quality of the solutions to which it belongs. This procedure is repeatedly applied until a termination criterion is satisfied.

Particle Swarm Optimization

Particle Swarm Optimization (PSO) [KES01] is another novel metaheuristic optimization algorithm, which was first invented to simulate social behavior. PSO optimizes a problem by having a population of candidate solutions, also called a *swarm of particles*, and these particles are moved around in the search-space according to a few simple formulae. The movements of the particles are guided by their own best known positions in the search-space as well as the entire swarm's best known positions which are updated as better positions found by the particles. Namely, when improved positions are being discovered, they will then come to guide the movements of the swarm. The process is repeated and by doing so it is hoped, but not guaranteed, that a satisfactory solution will eventually be discovered.

Differential Evolution

Another similar method is Differential Evolution (DE) [SP95]. DE optimizes a problem by also maintaining a population of candidate solutions and creating new candidate solutions by combining existing ones according to its simple formulae, and then keeping whichever candidate solution has the best fitness.

We have discussed a number of search based optimization algorithms, although they are different in the real optimization procedure, they all have some similarities in common; they are all population based optimization methods, they all utilize concepts of global optimum solution, local optimum, fitness evaluations etc. And also, they all more or less contain a procedure of learning from other solutions in previous generations, that is the cooperational procedure.

2.3 Learning Algorithms

In this section, we start the exciting tour in the field of machine learning, exploring some of the representative and excellent machine learning algorithms. These algorithms will also play very important roles in our development and comparison of hybrid optimization algorithms in the next chapters.

The learning problem we deal with for our supervised learning methods is called the *concept learning* problem, where positive and negative examples of a target concept are given, described with a fixed number of attributes. The goal of the learning task is to discover a description for the target concept in some explicit forms which are able to correctly recognize instances of the target concept and discriminate them from objects that do not belong to the target concept. We now formally give the definition of the concept learning task which is one of the most common problem tasks in machine learning literature.

Definition The set of items over which the concept is defined is called the set of *instances*, *data*, or *examples*, which we denote by X. Each of the items is represented by some attributes. The concept or function to be learned is called the *target concept*, which is denoted by c. In general, c can be any boolean valued function defined over the instances set X; that is, $c : X \rightarrow \{0, 1\}$. Within the training examples D, each instance x from X is presented with the attribute values along with its target concept value $c(x)$. Instances for which $c(x) = 1$ are called *positive examples*, Instances for which $c(x) = 0$ are called *negative examples*. The positive examples are also called members of the target concept.

Given a set of training examples of the target concept c, concept learning task is to hypothesize or estimate the target concept c. In general, under a representation scheme, each hypothesis h in the set of all possible hypotheses H, represents a boolean-valued function defined over X, that is, $h : X \rightarrow \{0, 1\}$. The concept learning task is the task to find a hypothesis h such that $h(x) = c(x), \forall x \in X$.

The concept learning task is sometimes also called *classification*. It belongs to the supervised learning method, where the training examples are used to guide the generation of model for prediction of future instances. In the machine learning community, this problem has been studied very well. Generally speaking, the machine learning community has developed a number of learning paradigms which can all solve this concept learning task. Among them, rule-based inductive learning methods output the set of rules as the explicit form of recognizing and discriminating future instances. Instance-based learning algorithms use the concept of similarity to decide the classifications of instances, in this sense, no explicit model exists. Neural network based learning adaptively adjusts a set of weights for the network connection model which predicts the new instances.

More specifically, there are many learning algorithms which can solve this attribute-value based concept learning problem in each learning paradigm. For example, for the inductive rule-based learning paradigm, there are attributes rule-sets based covering learning algorithm for classification. Ordered rule-sets, also called decision lists, are a generalized variant of concept learning problems for multi-class problems. Problems with continuous class variables can be solved by learning regression rules. Inductive logic programming has a richer representation language by inducing logic programs for classification or for predicting output values in functional relations. Finally, the classic decision tree learning algorithm also belongs to this inductive rule-based learning paradigm.

How can one be sure that one's learning algorithm has produced a theory that will correctly predict the future? In formal terms, how do we know that the hypothesis h is close to the target function f, if we don't know what f is? Generally, according to the computational learning theory, any hypothesis that is consistent with a sufficiently large set of training examples is unlikely to be seriously wrong. Namely, it must be Probably Approximately Correct (the PAC learning framework). Any learning algorithm that returns hypotheses that are probably approximately correct is called a PAC-learning algorithm. The key assumption is that the training and test sets are drawn randomly and independently from the same population of examples with the same probability distribution, the *stationary assumption*.

Both decision tree learning and an instance of the covering learning algorithm called AQ learning algorithm are introduced in this section, they are all inductive, rule-based, supervised concept learning methods. The decision tree learning will be applied in the development in Chapter 6. Another learning algorithm which is inductive, supervised and instance-based is introduced, this is the KNN and its generalization. The difference between KNN algorithm and inductive rule learning is that, there is no rules as output for the learning algorithm, KNN is a lazy learning method, that is classification is delayed until a new instance needs to be classified. KNN is for the hybrid algorithm developed in Chapter 4. As we discussed before, learning and statistics are two field sharing many similar ideas. We also illustrate one powerful and successfully applied statistical learning method, the PCA, concerning multi-variable classification in the subfield of statistics called *Multivariate Statistics*. Also, another successful and one of the members of the classic machine learning methods based on Bayesian probability theorem is introduced, which can also solve the above concept learning problem. The AQ learning algorithm, principal components analysis and Bayesian learning are the main algorithms involved in the hybrid algorithms introduced in Chapter 3. We will explore all these algorithms in more detail in the following sections.

2.3.1 Decision Tree Learning

Decision tree learning is one of the most widely used and practical methods for inductive inference learning. It was invented by Quinlan as a method for approximating discrete valued target functions, the learned output is represented as a decision tree which can be translated to sets of if-then rules to improve human readability. The decision tree constructing algorithms, ID3 [Qui86] and C4.5 [Qui93] search for or construct a decision tree solution in the complete hypothesis space.

ID3 Decision Tree Construction Algorithm

For a given set of training data consisting of positive and negative training examples, ID3 constructs a decision tree in a top-down style, sorting instances down the tree from the root to some leaf nodes, which provides the classification of the instances. Each node in the tree specifies a test of an attribute of the instances, with the question “which attribute should be tested at the current node of the tree”. In order to answer this question, each instance attribute is evaluated using a statistical test to determine how well it alone classifies the

current training examples. The best attribute is selected and used as the test at the current node of the tree. A descendant of the root node is then created for each possible value of this attribute, each branch descending from that node corresponds to one of the possible values for this attribute. And then the training examples are sorted to the appropriate descendant node. The whole process is then repeated using the training examples associated with each descendant node to select the best attribute to test at that node in the tree.

The above procedure forms a greedy search for an acceptable decision tree, in which the algorithm never backtracks to reconsider earlier choices. Once a decision tree is constructed, it can be used to classify the unseen instances. An instance is classified by starting at the root node of the tree, testing the attribute specified by this node, then moving down the tree branch corresponding to the value of the attribute in the given instance. This process is then repeated for the subtree rooted at the new node. A constructed decision tree example for illustration is given in Figure 2.2:

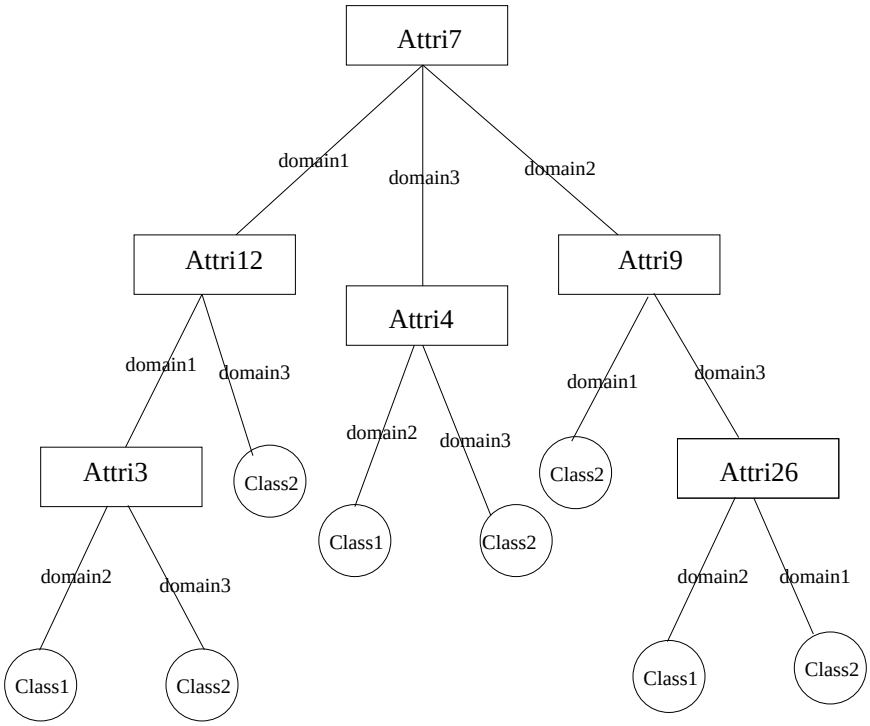


Figure 2.2: An illustrative example of a decision tree

Still, the key question in the above algorithm remains, that is how to find the best attribute for classification at each node. That is, the attribute that is most efficient for classifying examples should be selected. This quantitative measure in ID3 is a statistical property called *information gain*[Sha01], which measures how well a given attribute separates the training examples according to their target classification. ID3 uses this information gain measure to select from the candidate attributes at each step while making the tree grow. In-

formation gain is defined on the concept called *entropy*, which characterizes the (im)purity of an arbitrary collection of examples. Given a collection S of positive and negative examples of a target concept, the entropy of S relative to this classification (here, we only consider the case that the target concept has two values) is:

$$Entropy(S) = -p_{\oplus} \log_2 p_{\oplus} - p_{\ominus} log_2 p_{\ominus} \tag{2.13}$$

Here, $p_{\oplus}$ and $p_{\ominus}$ are the percentages of the positive and negative examples in S. Information gain is simply the expected reduction in entropy caused by partitioning the examples according to this attribute. More precisely, the information gain, $Gain(S, A)$ of an attribute A, relative to a collection of examples S, is defined as:

$$Gain(S, A) = Entropy(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} Entropy(S_v) \tag{2.14}$$

where $Values(A)$ is the set of all possible values for attribute A, and S_v is the subset of S for which attribute A has value v. The first term in Equation 2.14 is just the entropy of the original collection S, and the second term is the expected value of the entropy after S is partitioned using attribute A. The expected entropy described by this second term is simply the sum of the entropies of each subset S_v, weighted by the fraction of examples $\frac{|S_v|}{|S|}$ that belong to S_v. $Gain(S, A)$ is therefore the expected reduction in entropy caused by knowing the value of attribute A. Namely, $Gain(S, A)$ is the information provided about the target function value, given the value of some other attribute A. Information gain is precisely the measure used by ID3 to select the best attribute at each step in making the tree grow.

Issues in Decision Trees

There are also some practical issues in the decision tree learning method, these issues are important considerations which will influence the performance of the constructed decision tree significantly. Among of them, the problem of *overfitting* has attracted most attentions.

Overfitting means that a hypothesis overfits the training examples if some other hypothesis that fits the training examples less well actually performs better over the entire distribution of instances including the instances beyond the training set. Overfitting can happen in the following situations, when ID3 grows each branch of the tree just deeply enough to perfectly classify the training examples; when there is noise or errors in the data, when the number of training examples is too small to produce a representative sample of the true target function; and also the situation of *coincidental regularities*, where some at-

tributes happen to classify the examples very well even if they are not related well with the target function. In either of these cases, the decision tree construction algorithm can produce trees that overfit the training examples.

There are many methods to avoid overfitting, they can be grouped into two classes. First, approaches that stop making the tree grow earlier, before it reaches the node where it classifies the training data perfectly. For this method, it is not very practical, since deciding when to stop the growing is a very difficult task. The other method is the post-pruning method. Post-pruning is the approach that allows the tree to overfit the data first, and then post-prunes the tree.

There are also many ways to implement post-pruning. One of them is the rule post-pruning method [Qui93] applied in C4.5 algorithm. According to the name, rule post-pruning first infers the decision tree from the training set, making the tree grow until the training data fits as well as possible and allowing overfitting to occur. Secondly, it transforms the learned tree into an equivalent set of rules by creating one rule for each path from the root node to a leaf node. Thirdly, it prunes each rule by removing any preconditions that result in improving its estimated accuracy. Finally, it sorts the pruned rules by their estimated accuracy, and considers them in this sequence when classifying subsequent instances.

There are also many other issues in decision tree learning, such as dealing with continuous valued attributes, alternative measure for information gain, etc. For the first issue, it can deal with continuous valued attributes by selecting a suitable discretization method. For the second issue, when there are some attributes which have too many domains, these attributes will have very high gain values over other attributes according to the definition of information gain. Very possibly, the resulting decision tree is constructed by these attributes and is able to classify training data very well, but will be very bad at predicting the future data instances. One way to overcome this difficulty is to define another measure which also considers the broadness and uniformity of the attribute in splitting the data. Such an alternative for information gain is the *gain ratio*[Qui86], which depends on a term called *split information*:

$$SplitInformation(S,A) = -\sum_{i=1}^{c} \frac{|S_i|}{S} \log_2 \frac{|S_i|}{S}. \tag{2.15}$$

therefore, the gain ratio is defined by the information gain and the split information as:

$$GainRatio(S,A) = \frac{Gain(S,A)}{SplitInformation(S,A)}. \tag{2.16}$$

The decision tree learning algorithm has an advantage which is that the constructed decision tree can be transformed into a set of rules which can be read and understood more easily. This transformation procedure is also applied in our development of hybrid algorithms, we will discuss this topic in more detail in the corresponding chapter.

2.3.2 AQ Learning

The decision tree learning algorithm is a classic inductive rule-based learning algorithm, in this section, we introduce another inductive rule learning algorithm. The main difference between these two algorithms is not only because of the algorithms' constructing principles, but also the learning strategies employed by these algorithms. We will see these two strategies first, and then introduce the AQ learning algorithm.

Two General Strategies

Generally, for the rule-based inductive learning algorithms, there are two well-known strategies for solving the classification problems, the divide-and-conquer strategy[Qui86] and the separate-and-conquer strategy [PH90].

We have seen an example algorithm for the divide-and-conquer strategy, the decision tree learning algorithm in Section 2.3.1. As the name of this strategy suggests, the divide-and-conquer strategy is based on the idea that, at each stage of learning the algorithm seeks an attribute that splits the training examples best among the classes, and then the algorithm processes the following divided sub-training examples recursively according to the same criterion. This recursive method naturally results in a decision tree.

Quite different from the divide-and-conquer strategy, the separate-and-conquer strategy is based on the idea of considering each class in turn and looking for a way of covering all instances in this class, at the same time excluding all instances not in the class. Namely, to identify a rule that covers some of the instances at each stage. Due to this nature, it directly leads to a set of rules. The learning algorithms based on this strategy are also called *covering algorithms* [Mic69].

Specifically, this strategy learns or searches for a rule that explains (covers) a part of its training instances, removes the covered examples from the training set (the separate part), separates these examples, and recursively conquers the remaining examples by learning

more rules that cover some remaining examples until no examples remain. This ensures that each instance of the original training set is covered by at least one rule. For all of the covering algorithms, this strategy plays a top level loop, which is invariant for all algorithms. However, the specific methods to learn one rule are hugely different from algorithm to algorithm. Let us first explore a simple covering algorithm to solve the concept learning task:

Algorithm 3 pseudo code for a simple covering algorithm

```
1:  Set rule_set = ∅;
2:  Set covered_set = ∅;
3:  Declare rule;
4:  Initialize training_examples_set;
5:  while (positive examples exist in training_examples_set) do
6:      rule = {true};
7:      while (negative examples exist in covered_set) do
8:          for all (Condition ∈ Conditions) do
9:              Find the best_condition with highest correct rate;
10:         end for
11:         rule = rule ⋃ best_condition;
12:         covered_set = examples satisfying rule;
13:     end while
14:     rule_set = rule_set ⋃ rule;
15:     training_examples_set = training_examples_set \ covered_set;
16: end while
17: Return rule_set;
```

Algorithm 3 is a simple covering algorithm, the algorithm begins with an empty rule-set and successively adds rules to it until all positive examples are covered. The learning of a single rule starts with a rule whose body is always true. As long as it still covers negative examples the current rule is specialized by adding conditions to its body. Possible conditions are tests on the presence of certain values of various attributes. In order to move towards the goal of finding a rule that covers no negative examples, the algorithm selects a test that optimizes the purity of the rule, that is, a test that maximizes the percentage of positive examples among all covered examples. When a rule has been found that only covers positive examples, all of these covered examples will be removed and another rule

will be learned from the remaining examples. This is repeated until no positive examples remain. Thus, it is ensured that the learned rules together cover all of the given positive examples, the *completeness*, but none of the negative examples, the *consistency*. Almost all of the separate-and-conquer algorithms share the same structure of this algorithm, but are different in how to construct the single rule. We will see another example algorithm for this strategy shortly, but for now, we simply summarize and compare the covering algorithms with decision tree learning.

As we have seen that both of these algorithms are supervised learning algorithms and can be used to solve concept learning tasks. Also, the input for both algorithms is the same, that is the training data set, which is a set of data instances, consisting of a vector of attribute-value pairs. And the output for both algorithms has the same form, a set of rules. For the case of the decision tree, the constructed tree can be translated into a set of rules. The only important difference between these two learning algorithms is the middle learning procedures.

AQ Learning Algorithm

Although there are many separate-and-conquer based rule learning algorithms, this strategy has its roots in the covering algorithm called AQ learning algorithm by Michalski[Mic69]. The representation language used in AQ is called the *Attributional Calculus*, which is a simple-to-implement but highly expressive description language. It has well-defined syntax and semantics, and its representational power is between propositional logic and first-order predicate logic. Its most important construction is an attributional rule, which has its form as follows[Mic00]:

$$Condition \Longrightarrow Decision$$

where *Condition* is a conjunction of attributional conditions, and *Decision* is an elementary attributional condition. An attributional condition is in the form:

$$Left\ relation\ Right$$

where *Left* is an set of attributes joined by $\wedge$ or $\vee$, called internal conjunction and disjunction, respectively. *Right* is a list of values from the domain of attributes in *Left*, joined by the symbol $\vee$, or a pair of values joined by '...' (called *range*). *relation* is a relational symbol from the set $\{=, \neq, \geq, >, <, \leq\}$.

An attributional condition '*Left relation Right*' is true (or satisfied) if *Left* is in relation to *Right*. A condition is called *elementary* if *Left* is a single attribute, *relation* is not $\neq$, and *Right* is a single value; otherwise it is called *composite*. Here are examples of attributional conditions and their interpretations, (*Shape* = *rectangle*), (*Density* > 39) are elementary condition, while (*Optimization Method* = $GA \vee ES \vee PSO$), ($Salary \wedge Bonus = 3000 \ldots 5000$) are composite conditions.

Attributional calculus can be seen as the description language for the AQ learning algorithms, the training data and the output rules can all be described with this language. With this powerful tool, the AQ learning algorithm can be constructed now. Given a set of positive and negative training examples of a decision class, an AQ learner generates a set of attributional rules (a ruleset) characterizing this class. Training examples are in the form of attributional events, which are vectors of attribute values. Events in the decision class for which a ruleset is generated are considered positive examples, and events in all other classes are considered negative examples. The description of a simple form of the AQ learning algorithm is given below as stated in [Mic00]:

1. Seed selection: Select randomly a positive example and call it a *seed*.

2. Star generation: Generate a star of the seed, defined as a set of general attributional rules that cover the seed and any other positive examples, but do not cover negative examples. In the general case, a rule can cover negative examples if it optimizes a description quality criterion.

3. Rule selection: Select the highest quality rule from the star according to a given description quality criterion. Such a criterion can be tailored to the requirements of the problem domain. For example, a quality criterion may require selecting the rule that covers the largest number of positive examples, covering no negative examples, and has the lowest computational cost among other equivalent rules in the star.

4. Coverage update: Remove examples covered by the rule from the set of positive examples and select a new seed from the remaining positive examples. If there are no positive examples left, return the generated set of rules. Otherwise, go to Step 1.

This algorithm has the same top-level structure as the simple covering algorithm 3, but the generation of rule is different. The most important step of the algorithm is star generation (Step 2), which involves a repetitive application of the extend-against generalization operator, and logically multiplying out the resulting collections of partial rules. If properly

implemented, such a process can be executed highly efficiently. For example, recent implementations of AQ-type learning have been effectively applied to problems with hundreds of attributes and tens of thousands of examples.

AQ learning algorithms, such as AQ15 [JKES95] and AQ18 [KM99], have several special features. AQ15 includes the ability to learn a range of different types of attributional rulesets, such as intersecting, disjoint, ordered, characteristic, and discriminate; to adapt inductive reasoning to different types of attributes, including nominal, rank, cyclic, numerical, and structured; to learn from noisy and/or inconsistent data; to learn incrementally; and to match rules with examples using a strict or flexible matching method. AQ18 includes several additional features, such as the ability to discover strong patterns in data, thus it can optimize a multi-criterion measure of description quality, and automatic constructive induction. The latter feature enables the program to automatically search for a better representation space when the original one is found to be inadequate.

AQ learning algorithm is the learning algorithm applied in a learning and evolution hybrid algorithm, which is very important in our study of the LEM hybrid algorithms in this thesis, we will come back to AQ learning in the next chapter for hybrid algorithms.

2.3.3 *K* Nearest Neighbors (KNN) Learning

In this section, we introduce a new learning paradigm, which is called *instance-based* learning and a well-known instance learning algorithm from this paradigm, the KNN [CH67] learning method.

In contrast to learning methods that construct a general, explicit description or model of the target function when training examples are provided, instance-based learning methods simply store the presented training examples. Generalizing beyond these examples is postponed until a new instance must be classified. Each time a new query instance is encountered, its relationship to the previously stored set of similar related instances is examined in order to assign a target function value for the new instance, that is to classify the new query instance.

Instance-based methods are sometimes referred to as ‘lazy’ learning methods, because they delay processing until a new instance must be classified. In this sense, the previous inductive learning algorithms, decision tree learning, AQ learning are called ‘eager’ learning methods. One key difference between lazy and eager learning methods is that the former can construct a different approximation to the target function for each distinct query

instance that must be classified. In fact, many techniques construct only a local approximation to the target function that applies in the neighborhood of the new query instance, and never construct an approximation designed to perform well over the entire instance space. This has significant advantages when the target function is very complex, but can still be described by a collection of less complex local approximations.

The most well-known instance-based learning method are KNN, local weighted regression, and case-based reasoning, the difference between these methods are the representation forms for instances.

KNN Algorithm

The KNN assumes all instances correspond to points in n-dimensional space. The nearest neighbors of an instance are defined in terms of the standard Euclidean Distance. More precisely, let an arbitrary instance x be described by the feature vector:

$$\langle a_1(x), a_2(x), \ldots a_n(x) \rangle \tag{2.17}$$

where $a_r(x)$ denotes the value of the rth attribute of instance x. Then the distance between two instances x_i and x_j is defined to be $d(x_i, x_j)$, where

$$d(x_i, x_j) = \sqrt{\sum_{r=1}^{n} (a_r(x_i) - a_r(x_j))^2} \tag{2.18}$$

In nearest-neighbors learning the target function may be either discrete-valued or real-valued. Let us first consider learning discrete-valued target functions of the form $f : R^n \rightarrow V$, where V is the finite set $\{v_l, v_2, \ldots v_s\}$. The KNN algorithm for approximating a discrete-valued target function is given as Algorithm 4:

Algorithm 4 pseudo code for KNN

1: All training examples are stored in the *training_data* vector.

2: For each query instance x_q that will be classified.

3: Find the k nearest instances $x_1, x_2, \ldots x_k$ in *training_data*.

4: Return

$$\hat{f}(x_q) = \max_{v \in V} \sum_{i=1}^{k} \delta(v, f(x_i))$$

5: where

$$\delta(a, b) = \begin{cases} 1 & \text{if(a = b) ;} \\ 0 & \text{otherwise.} \end{cases}$$

As shown above, the value $\hat{f}(x_q)$ returned by this algorithm as its estimate of $f(x_q)$ is just the most common value of f among the k training examples nearest to x_q.

The KNN algorithm is easily adapted to approximating continuous-valued target functions. This can be accomplished by calculating the mean value of the k nearest training examples rather than calculating their most common value. More precisely, to approximate a real-valued target function $f : R^n \rightarrow R$, step 4 of Algorithm 4 is replaced with Equation 2.19:

$$\hat{f}(x_q) = \frac{\sum_{i=1}^{k} f(x_i)}{k} \tag{2.19}$$

Distance-Weighted Nearest Neighbors Algorithm

One obvious refinement to the KNN algorithm is to weight the contribution of each of the k neighbors according to their distance to the query point x_q, giving greater weight to closer neighbors. Namely, in Algorithm 4, which approximates discrete-valued target functions, we might weight the vote of each neighbor according to the inverse square of its distance from x_q. This can be accomplished by replacing step 4 of Algorithm 4 with Equation 2.20:

$$\hat{f}(x_q) = \max_{v \in V} \sum_{i=1}^{k} w_i \delta(v, f(x_i)) \tag{2.20}$$

where $w_i = \frac{1}{d(x_q, x_i)^2}$. We can distance-weight the instances for real-valued target functions in a similar fashion, replacing step 4 of Algorithm 4 in this case with Equation 2.21:

$$\hat{f}(x_q) = \frac{\sum_{i=1}^{k} w_i f(x_i)}{\sum_{i=1}^{k} w_i} \tag{2.21}$$

Due to distance weighting, there is really no harm in allowing all training examples to have an influence on the classification of the x_q, because very distant examples will have very little effect on $\hat{f}(x_q)$. Of course, this will result in running more slowly. If all training examples are considered when classifying a new query instance, we call the algorithm a global method. If only the nearest training examples are considered, we call it a local method. One main advantage about KNN is that it is robust to noisy training data and quite effective when it is provided a sufficiently large set of training data.

However, some disadvantages which cause practical implementation issues do exist. One of them is that the cost of classifying new instances can be high. This is due to the fact that nearly all computation takes place at classification time rather than when the training examples are first encountered. Therefore, techniques for efficiently indexing training

examples are needed to reduce computation required at query time. Various methods have been developed for indexing the stored training examples so that the nearest neighbors can be identified more efficiently at some additional cost in memory. One such indexing method is the kd-tree [Ben75], in which instances are stored at the leaves of a tree, with nearby instances stored at the same or nearby nodes. The internal nodes of the tree sort the new query x_q, to the relevant leaf by testing selected attributes of x_q.

A second disadvantage of many instance-based approaches, especially nearest neighbor approaches, is that they typically consider all attributes of the instances when attempting to retrieve similar training examples from memory. If the target concept depends on only a few of the many available attributes, then the instances that are truly most 'similar' may well be a large distance apart. Namely, the distance between neighbors will be dominated by the large number of irrelevant attributes. This difficulty, which arises when many irrelevant attributes are present, is sometimes referred to as the *curse of dimensionality*. Nearest-neighbors approaches are especially sensitive to this problem. There are some methods to overcome this problem, we consider one of them by formulating the problem as solving the following task. For a given set of attributes with size n in training examples, the task is to find a subset out of this set of attributes, satisfying this subset can classify the unseen examples with the highest accuracy. This task can be viewed alternatively as finding a weight set with size n, and each of value of the weight could be with the range of $(0.0 \ldots 1.0)$ indicating the relevance of the corresponding attribute. When the weight value is 0.0, it is completely eliminated from future classification. These weight subsets can be optimized by classification on a cross-validation set. Another excellent work on this topic about feature selection can be found in [RIG+00], where a genetic algorithm is used as the optimizer.

Although disadvantages are inevitable for KNN as for any other learning paradigms, we start our investigation in learning and evolution hybrid algorithm with the KNN algorithms due to their efficiency in classification, robustness for noisy training data, and simplicity in implementation (the only application-specific demand is a suitable distance measure, this is in contrast to other learning algorithms). We will come back to the KNN algorithm in Chapter 4, where our first hybrid algorithm is developed.

2.3.4 Principal Components Analysis

Principal Components Analysis (PCA) was invented in 1901 by Karl Pearson [Pea01] as a mathematical procedure that transforms a number of possibly correlated variables into

a smaller number of uncorrelated variables called *principal components*. The first principal component accounts for as much of the variability in the data as possible, and each succeeding component accounts for as much of the remaining variability as possible.

PCA is the simplest of the true eigenvector-based multivariate analyses. Often, its operation can be thought of as revealing the internal structure of the data in a way which best explains the variance in the data. If a multivariate dataset is visualized as a set of coordinates in a high-dimensional data space (one axis per variable), PCA supplies the user with a lower-dimensional picture, a 'shadow' of this object when viewed from its (in some sense) most informative viewpoint. PCA is mathematically defined as an orthogonal linear transformation that transforms the data to a new coordinate system such that the greatest variance by any projection of the data comes to lie on the first coordinate, the first principal component, the second greatest variance on the second coordinate, and so on.

Given a set of points in Euclidean space, the first principal component (the eigenvector with the largest eigenvalue) corresponds to a line that passes through the mean and minimizes sum squared error with those points. The second principal component corresponds to the same concept after all correlation with the first principal component has been subtracted out from the points. Each eigenvalue indicates the portion of the variance that is correlated with each eigenvector. Thus, the sum of all the eigenvalues is equal to the sum squared distance of the points with their mean divided by the number of dimensions. PCA essentially rotates the set of points around their mean in order to align with the first few principal components. This moves as much of the variance as possible (using a linear transformation) into the first few dimensions. The values in the remaining dimensions, therefore, tend to be highly correlated and may be dropped with minimal loss of information.

PCA is often used in this manner for dimensionality reduction. It is mostly used as a tool in exploratory data analysis, finding patterns in data of high dimension, and for making predictive models. And it is a useful statistical technique that has found application in fields such as face recognition and image compression.

Covariance Matrix

We introduce PCA in more detail by following the main steps needed to compute it with the covariance method. PCA involves the calculation of the eigenvalue decomposition of a data covariance matrix or singular value decomposition of a data matrix, usually after mean centering the data for each attribute. For a given set of training examples, we first calculate the *variance* s for one attribute, given the mean x and standard deviation s, variance is

simply the standard deviation squared, it represents the extent of data spread:

$$s^2 = \frac{\sum_{i=1}^{n}(x_i - x)^2}{(n-1)} \tag{2.22}$$

Standard deviation and variance only operate on one dimension, so that only the standard deviation for each dimension of the data set independently of the other dimensions can be calculated. However, it is useful to have a similar measurement to find out the relationship between these dimensions and how much the dimensions vary from the mean with respect to each other. *Covariance* is such a measure, it is always measured between two dimensions. If we calculate the covariance between one dimension and itself, we get the variance. So, if we had a 3-dimensional data set (x, y, z), then we could measure the covariance between the x and y dimensions, the x and y dimensions, and the y and z dimensions. The formula for covariance is very similar to the formula for variance:

$$cov(x, y) = \frac{\sum_{i=1}^{n}(x_i - x_{mean})(y_i - y_{mean})}{(n-1)} \tag{2.23}$$

Covariance is between two dimensions, and variance is about one dimension. If the value of covariance is positive, two dimensions change together. If the value is negative, two dimensions change differently. If the value is zero, two dimensions are independent to each other.

A *Covariance matrix* is a matrix that stores all the possible covariance values between all the different dimensions for the given many dimensional data set. A covariance matrix is a matrix for an n-dimensions data set:

$$C^{n\times n} = (c_{i,j}, c_{i,j} = cov(Dim_i, Dim_j)) \tag{2.24}$$

where $C^{n\times n}$ is a matrix with n rows and n columns, and Dim_x is the xth dimension. For example, for 3-dimensional data set, the covariance matrix is calculated as the 3×3 matrix

$$C = \begin{pmatrix} cov(x,x) & cov(x,y) & cov(x,z) \\ cov(y,x) & cov(y,y) & cov(y,z) \\ cov(z,x) & cov(z,y) & cov(z,z) \end{pmatrix} \tag{2.25}$$

When the covariance matrix is formed, we can calculate the *eigenvectors* and *eigenvalues* of the covariance matrix. First, the eigenvectors and eigenvalues give important information about the matrix, they appear in pair for square matrix. It is beyond the scope of this thesis to discuss the method of calculating eigenvectors and eigenvalues, all we will

say is that those are complicated iterative methods especially for many large size matrices. Before introducing the meanings and usability of eigenvectors and eigenvalues in PCA, we give an example of eigenvectors and eigenvalues:

$$\begin{pmatrix} 2 & 3 \\ 2 & 1 \end{pmatrix} \times \begin{pmatrix} 6 \\ 4 \end{pmatrix} = 4 \times \begin{pmatrix} 6 \\ 4 \end{pmatrix} \tag{2.26}$$

In this example, the square matrix can be thought of as a transformation matrix. If we multiply this matrix on the left of a vector, the result is another vector that is transformed from its original position. It is the nature of the transformation that the eigenvectors arise from. A transformation matrix that, when multiplied on the left, reflected vectors in the line $y = x$. Then we can see that if there was a vector that lay on the line $y = x$, its reflection is itself. This vector would be an eigenvector of that transformation matrix. The vector $\begin{pmatrix} 6 \\ 4 \end{pmatrix}$ can be seen as an eigenvector of the square matrix. And the eigenvalue is 4.

Feature Vector

Once the eigenvectors and eigenvalues are derived, we can begin to chose components and form a *feature vector*. This feature vector is an important concept in PCA, it is used to produce dimensionality reduction. As we have found out, the eigenvectors are used to indicate the patterns of the variables and eigenvalues are used to indicate the significance of these patterns. Namely, the eigenvectors with the highest eigenvalues are the principal components of the data set. In general, once eigenvectors are found from the covariance matrix, the next step is to order them by eigenvalues, highest to lowest. This gives the components in order of significance and indicates whether the patterns are strong or weak compared with other patterns. If one pattern is less significant, it can be deleted by removing the corresponding eigenvector and eigenvalue from the n eigenvectors and eigenvalues list. The new reduced eigenvectors set is called a *feature vector*, which is now used together with the original data set to calculate a new data set, where the variables number or dimensions are now reduced. In such a transformation procedure, we lose some information which are less important. So, the feature vector simply consists of the remaining eigenvectors, that is the ones with highest eigenvalues, as columns.

$$FeatureVector = (eigenvector_1, eigenvector_2, eigenvector_3, \ldots, eigenvector_n) \tag{2.27}$$

Once we have chosen the components (eigenvectors) that we wish to keep in our data

and formed a feature vector, we simply take the transpose of the vector and multiply it on the left of the original data set. When the original data are restored, we can see more clearly about the strong patterns, and the weak patterns are deleted. Therefore, PCA is a way of identifying patterns in data, and expressing the data in such a way as to highlight their similarities and differences. Since patterns in data can be hard to find in data of high dimension, where the luxury of graphical representation is not available, PCA is a powerful tool for analyzing data. We will come back to PCA in the next Chapter 3 again, when we discuss an important hybrid optimization algorithm which applies the PCA method.

2.3.5 Bayesian Network and Bayesian Learning

In this section, we introduce another important and popular machine learning paradigm in the machine learning community. *Bayesian inference* is a statistical inference method among many hypotheses (the *hypothesis space*), where some kind of evidence or observations are used to calculate the probabilities of these hypothesis, or else to update their previously-calculated probabilities. *Probability* comes naturally from the world or environment which is full of uncertain knowledge. In practice, we are never completely sure about the statements of the environment we are interested in. For example, assume we want to construct a rule to describe the following knowledge:

$$\forall(s) FailedIn(s, Exams) \Rightarrow \neg WorkHard(s)$$

Unfortunately, this is not a correct rule. First, *not WorkHard* is not the only reason that students will fail in the exams, there are many other reasons for failure in exams. For example, not feeling well, coming late, etc. There could be an infinite list of reasons. Second, *not WorkHard* does not necessarily mean students will fail in exams, many students can pass the exams without working hard. So, rule-based knowledge representation system simply fail to represent uncertain knowledge. It is due to either to the fact that we cannot list all rules to capture the uncertainty, or to the fact that we do not have complete knowledge about a particular domain, or we will never have complete information about an instance.

In fact, much knowledge about the world is suitable to be provided with a *degree of belief*. Namely, they are better to be interpreted in the *probability theory*, which assigns to each statement about knowledge a numerical degree of belief between the range of $(0 \ldots 1)$. For example, we can assign to the above rule 0.8, meaning if a student has failed his examination, he/she has an 80% probability that he was not working hard in preparation.

Probability provides a way of summarizing the uncertainty that comes from our laziness and ignorance. We may not know for sure about one statement, but we can believe that in what percentage of probability that statement will happen. This belief can be derived from statistical data, or some general rules, or from a combination of sources of evidence. We distinguish the degree of belief discussed here with the degree of truth which is used in another uncertain handling method called *fuzzy logic*.

In the following discussions, we assume the knowledge of basic concepts and theorems in probability theory, otherwise, a brief introduction to probability theory is given in Appendix A. We will introduce the well-known general bayesian network inference model for uncertainty knowledge base and the bayesian learning method, also called naive bayesian classifier, which is a simple probabilistic classifier based on applying Bayes' theorem with strong independence assumptions.

Bayesian Network

A Bayesian Network (BN) is a data structure that represents the dependencies among variables and gives a concise specification of any full joint probability distribution. A Bayesian network is a directed graph in which each node is annotated with quantitative probability information. The full specification is as follows:

1. A set of random variables makes up the nodes of the network, variables may be discrete or continuous.

2. A set of directed links or arrows connects pairs of nodes, if there is an arrow from node X to node Y, X is said to be a parent of Y.

3. Each node X_i has a conditional probability distribution $P(X_i|Parents(X_i))$ that quantifies the effect of the parents on the node.

4. The graph has no directed cycles, that is, it is a directed, acyclic graph, or DAG.

The topology of the network, the set of nodes and links, specifies the conditional independence relationships that hold in the domain. The intuitive meaning of an arrow in a properly constructed network is usually that X has a direct influence on Y. Once the topology of the bayesian network is laid out, we need only specify a conditional probability distribution for each variable, given its parents. The semantic of bayesian network is that the bayesian network can be used to represent the full joint distribution:

$$P(x_1, \ldots, x_n) = P(x_i|parents(X_i)) \tag{2.28}$$

Once a BN is built up, it can be used to make inferences efficiently, good methods have been developed, such as *exact inferences* and *approximate inferences*. They require a well-constructed network to exist. Therefore, we need to discuss how to construct the bayesian network, this will include two important aspects, one is parameter learning, and the other is structure learning. In this section, we only talk about a parameter learning method called *maximum-likelihood* parameter learning. We talk about structure learning in the next chapter.

Bayesian Learning

To induce bayesian networks correctly, as we known from the previous section, there are two important components need to be learned correctly. One is the parameters for random variables and the other is the structures representing the (in)dependence relations between the random variables. We start to introduce the parameters learning method with *Bayes's rule* (or *Bayes's theorem*):

$$P(a \wedge b) = P(b|a)P(a) \tag{2.29}$$

$$P(a \wedge b) = P(a|b)P(b) \tag{2.30}$$

Equating the two right-hand sides and dividing by $P(a)$, we get

$$P(b|a) = \frac{P(a|b)P(b)}{P(a)} \tag{2.31}$$

$$P(Y|X) = \frac{P(X|Y)P(Y)}{P(X)} \tag{2.32}$$

$$P(Y|X) = \alpha P(X|Y)P(Y) \tag{2.33}$$

Bayes's theorem underlies all modern AI systems for probabilistic inference. Explicitly, it requires a conditional probability and two unconditional probabilities to calculate one conditional probability. In practice, there are many situations which match this formula very well. This makes Bayes's rule very popular and useful in realistic problems.

So, what is a bayesian learning problem? Bayesian learning can calculate the probability of each hypothesis in the hypothesis space, given the experienced data, and makes predictions on that basis. In the context of bayesian learning, learning is in fact a probabilistic inference problem. Formally, given a random variable H for hypothesis space, with the possible values h_i. Let D represent all the data, D_i is also a random variable with possible values v_1 and v_2, with observed value d, then the probability of each hypothesis is obtained by Bayes's rule:

$$P(h_i|d) = \alpha P(d|h_i)P(h_i) \tag{2.34}$$

$$P(d|h_i) = \prod_j P(d_j|h_i) \tag{2.35}$$

The key quantities in the bayesian approach are the prior hypothesis, $P(h_i)$ for each hypothesis, which is some pre-fixed value according to experiences, and the likelihood of the data under each hypothesis, $P(d_j|h_i)$, which is described in each h_i, all of them are known in advance. In bayesian learning, we are now interested in looking for the most probable hypothesis h in H, given the observed data d, we can find this hypothesis $P(h_i|d)$, according to Equations 2.35 and 2.34. Any such maximally probable hypothesis is called a *maximum a posterior* (MAP) hypothesis, h_{MAP}.

A simplification for bayesian and MAP learning is that, assuming a uniform prior probability of all hypotheses, that is $P(h_i)$ are all equivalent. And any hypothesis which maximizes $P(d|h_i)$ is called a *maximum-likelihood* hypothesis, h_{ML}. According to Equation 2.34, if all $P(h_i)$ are equivalent, then the maximized $P(h_i|d)$ is equal to $P(d|h_i)$, so this simplification only requires us to find hypothesis h_θ, where θ is the maximum-likelihood parameter, which is the proportion of appearance times in previous experiments for the random variable's one domain value, therefore for the other domain value, the appearance times is $1-\theta$.

We assume that N events have happened, of which n_1 is the times for v_1 and $n_2(n_2 = N - n_1)$ is for v_2. According to Equation 2.35, the likelihood of this particular data set is:

$$P(d|h_\theta) = \prod_{j=1}^{N} P(d_j|h_\theta) = \theta^{n_1} \times (1-\theta)^{n_2} \tag{2.36}$$

The maximum-likelihood hypothesis is given by the value of θ that maximizes this expression. The same value is obtained by maximizing the log likelihood:

$$L(d|h_\theta) = logP(d|h_\theta) = \sum_{j=1}^{N} logP(d_j|h_\theta) = n_1 log\theta + n_2 log(1-\theta) \tag{2.37}$$

By taking this algorithm, we reduce the product to a sum over the data, which is usually easier to maximize. To find the maximum-likelihood value of θ, we differentiate L with respect to θ and set the resulting expression to zero:

$$\frac{dL(d|h_\theta)}{d\theta} = \frac{n1}{\theta} - \frac{n2}{1-\theta} = 0 \tag{2.38}$$

$$\theta = \frac{n_1}{n_1 + n_2} = \frac{n_1}{N} \tag{2.39}$$

In this way, we have constructed a method for bayesian network parameter learning, where there is only one random variable D_i and its probability distribution information is θ. Although the resulting bayesian network contains only one node for this variable, it is also applicable to networks with many variables and dependence relationships.

Bayesian Classifier

Finally, we state the most common bayesian network model used in machine learning, the *naive bayesian model* or *naive bayesian classifier*. It is often used in cases where the attributes variables are conditionally independent given the class variable. The full joint distribution for this simplified bayesian network model can be written as:

$$P(Cause, Effect_1, \ldots, Effect_n) = P(Cause) \prod_i P(Effect_i|Cause) \tag{2.40}$$

This model shows a simple but very common pattern in which a single cause directly influences a number of effects, all of which are conditionally independent, given the cause. As a simplified bayesian network model, naive bayesian learning scales well to very large problems: with n boolean attributes, there are just $2n + 1$ parameters, and no search is required to find h_{ML}, the maximum-likelihood naive bayesian hypothesis. The bayesian classifier can be seen as a specified instance of the bayesian network inference, and also inference can be seen as a more general concept for learning.

Bayesian inference methods play important roles in a class of learning and evolution hybrid algorithms raised in the EC community recently, the EDA methods, which we will discuss in the next chapter.

Chapter 3

Hybrids of Learning and Evolution

3.1 Overview

Many existing search and learning methods have been particularly explored in chapter 2. We have seen that search methods, as a general problem solver, can be used to solve complex optimization problems without the need for any domain-specific knowledge. The only requirements for search based methods are suitable representations for the problems and the measurement or evaluation functions for these problems. Meanwhile, the learning methods can be used to learn useful hypotheses, classify instances, and predict based on these learned output, to gain beneficial insights into the problem space.

After the introduction of search and learning techniques, we begin to explore the core topic for this chapter, which is the hybrid of learning and evolution algorithms. Modern hybrid algorithms utilize the advantages from both learning and evolution. Hybrid algorithms take the feature of evolutionary search algorithms as general optimizers, which are robust to local optima, and also take the advantage of learning algorithms for creating hypotheses that indicate promising solutions efficiently. Due to these advantages of these two techniques, the aim of combining these two methods is to find relatively promising solutions while keeping enough efficiency. This is again the aim we stated in chapter 1, the trade-off between the quality of the solutions and the time and space resources expended on finding these solutions, because such a trade-off is crucial to the success of many practical application problems, especially evaluation-expensive problems.

Hybrid optimization paradigm algorithms are being developed rapidly in the evolutionary computation community. In this thesis, we consider three representative methods, which have attracted considerable attention in this research field. Based on one of these methods, we developed our new hybrid algorithms, and compare the performance of these

algorithms on a number of test problems of which the results are analyzed. These three methods are discussed in the following three sections.

3.2 Covariance Matrix Adaptation Evolution Strategies

CMAES [HO96, HO97] is an Evolution Strategy adapting the covariance matrix of the normal mutation search distribution. Basically, it records the population history for a certain number of iterations to calculate covariance and variance information among the object variables, the following search effort is influenced by these variance values. Compared to other evolutionary algorithms, an important property of the CMAES is its invariance against linear transformations of the search space. Namely, it exhibits the same performances for a given objective function $f : x \in R^n \rightarrow f(x) \in R$, where $n \in N$, or for the same function where a linear transformation is applied, $f_R : x \in R^n \rightarrow f(Rx) \in R$, where R denotes a linear transformation. This is true only if a corresponding transformation of the strategy parameters is made. In fact, this transformation is learned by the CMA with the application of the principle of the principal components analysis introduced in Chapter 2.

As we discussed before, as a member of the evolutionary computation family, the evolution strategy is a stochastic search algorithm that can be used in the search for optimization problems. The mechanism behind ES search is the stochastic variation operator, mutation, on the current individuals. The mutation is usually carried out by adding a realization of a normally distributed random vector, and the parameters of the normal distribution play an essential role for the performance of the search algorithm. Therefore, the correct adaptation of the parameters for the normal mutation distribution becomes crucial. There are two types of parameters, one is the object parameters that define the individuals or search points in search space, the other is the strategy parameters that characterize the mutation distribution. The essential feature of ES is the self-adaptation of the mutation distribution, that is, adapts strategy parameters during the search process.

In Chapter 2, we have already seen some attempts at this automatic adaptation of normal mutation distribution in ES. The search based on the uncorrelated mutation with one mutation step forms a hyper-sphere with equal probability density on the surface. This global step size is further generalized, each coordinate axis is assigned as different variance, that is the uncorrelated mutation with n mutation steps. An even further generalization adapts the orthogonal coordinate system, where each coordinate axis is assigned a different variance, any normal distribution with zero mean can be produced. However, this most generalized

method depends on the orientation and permutation of the coordinate axes and therefore will perform very badly on the quadratic functions which are badly scaled and not axis parallel oriented.

For these reasons, the covariance matrix adaptation method is developed. CMAES contains a generalized individual step size control, which is independent of the given coordinate system. First, we give the details of the $(\mu/\mu_I, \lambda)$-CMAES algorithm as it is defined in [HO97] for completeness.

3.2.1 $(\mu/\mu_I, \lambda)$-CMAES algorithm

Every new object parameter vector $\mathbf{x}_k^{(g+1)}$, $k = 1 \ldots \lambda$, of generation $g+1$ is generated by adding a realization of a $\mathcal{N}\left(\mathbf{0}, \delta^{(g)^2}\mathbf{C}^{(g)}\right)$ distributed random vector. The vector is generated by linear transformation of $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$, where $\mathbf{I}$ is the identity matrix. For $k = 1 \ldots \lambda$, it yields

$$\mathbf{x}_k^{(g+1)} = \langle \mathbf{x} \rangle_\mu^{(g)} + \delta^{(g)}\mathbf{B}^{(g)}\mathbf{D}^{(g)}\mathbf{z}_k \tag{3.1}$$

where $\mathbf{x}_k^{(g+1)} \in \mathbb{R}^n$. Object variable vector of k^{th} individual at generation $g+1$.

$\langle \mathbf{x} \rangle_\mu^{(g)} = \frac{1}{\mu}\sum_{j \in \mathbf{I}_{sel}}^{(g)} \mathbf{x}_j^{(g)}$. Center of mass of the μ selected (best) individuals of generation g. $\mathbf{I}_{sel}^{(g)}$ is the set of indices of the selected individuals at generation g, $|\mathbf{I}_{sel}| = \mu$.

$\delta^{(g)}$ Step size.

$\mathbf{B}^{(g)}$ Orthogonal $n \times n$-matrix, which linearly transforms $\mathbf{D}^{(g)}\mathbf{z}$. Columns of $\mathbf{B}^{(g)}$ are eigenvectors of the covariance matrix $\mathbf{C}^{(g)}$. For any two columns $\mathbf{b}_i$ and $\mathbf{b}_j$, $i \neq j$, of $\mathbf{B}$ holds $\| \mathbf{b}_i \| = 1$ and $\langle \mathbf{b}_i, \mathbf{b}_j \rangle = 0$ and therefore $\mathbf{B}^{-1} = \mathbf{B}^{\mathbf{T}}$.

$\mathbf{D}^{(g)}$ Diagonal $n \times n$-matrix. The diagonal element $d_{ii}^{(g)}$ is the square root of an eigenvalue of the covariance matrix $\mathbf{C}^{(g)}$. The corresponding eigenvector is the i^{th} column of $\mathbf{B}^{(g)}$. That is, for any column $\mathbf{b}_i^{(g)}$ of $\mathbf{B}^{(g)}$ holds $\mathbf{C}^{(g)}\mathbf{b}_i^{(g)} = d_{ii}^{(g)^2}\mathbf{b}_i^{(g)}$.

$\mathbf{z}_k \in \mathbb{R}^n$. $k = 1 \ldots \lambda$ realizations of a $\mathcal{N}(\mathbf{0}, \mathbf{I})$ distributed random vector, i.e. components of $\mathbf{z}$ are independent identically (0,1)-normally distributed.

$\mathbf{D}$ scales the axes of the distribution; isodensity lines of $\mathbf{Dz}$ are coordinate axes parallel (hyper-)ellipsoids. $\mathbf{B}$ determines the new orientation of this ellipsoid. The covariance matrix $\mathbf{C}$ determines $\mathbf{B}$ and $\mathbf{D}$, and is adapted by means of a so called evolution path, denoted by $\mathbf{s}$.

$$\mathbf{s}^{(g+1)} = (1-c) \cdot \mathbf{s}^{(g)} + c_\mu \cdot \frac{\sqrt{\mu}}{\delta^{(g)}}\left(\langle \mathbf{x} \rangle_\mu^{(g+1)} - \langle \mathbf{x} \rangle_\mu^{(g)}\right) \tag{3.2}$$

$$\mathbf{C}^{(g+1)} = (1 - c_{cov}) \cdot \mathbf{C}^{(g)} + c_{cov} \cdot \mathbf{s}^{(g+1)} \left(\mathbf{s}^{(g+1)}\right)^{\mathbf{T}} \tag{3.3}$$

where

$\mathbf{s} \in \mathbb{R}^n$. Sum of weighted center of mass differences. $\mathbf{s}$ represents the evolution path of the strategy.

$c \in [0; 1]$. $1/c$ corresponds to the accumulation time for $\mathbf{s}$. For $c = 1$, $\mathbf{s}^{(g+1)}$ only depends on object parameter vectors of generation g and $g + 1$.

$c_u = \sqrt{c \cdot (2 - c)}$ normalizes the variance of $\mathbf{s}$ because $1^2 = (1 - c)^2 + c_u^2$.

$\mathbf{C}^{(g)}$ Symmetric $n \times n$-matrix, which is the covariance matrix of the normally distributed random vector $\mathbf{B}^{(g)}\mathbf{D}^{(g)}\mathbf{z}$, where $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$. $\mathbf{C}^{(g)}$ determines $\mathbf{B}^{(g)}$ and $\mathbf{D}^{(g)}$ and $\mathbf{C}^{(g)} = \mathbf{B}^{(g)}\mathbf{D}^{(g)}(\mathbf{B}^{(g)}\mathbf{D}^{(g)})^{\mathbf{T}}$.

$c_{cov} \in [0; 1]$. $1/c_{cov}$ corresponds to the averaging time for the covariance matrix.

The step size δ is adapted separately, because changes of overall variance should be made on a much shorter time scale than the adaptation of the covariance matrix. For step size adaptation $\langle \mathbf{x} \rangle_\mu^{(g+1)} - \langle \mathbf{x} \rangle_\mu^{(g)}$is transformed to reverse the scaling by $\mathbf{D}$, done in Equation 3.1. This allows to calculate the expected length of $\mathbf{s}_\delta$.

$$\mathbf{s}_\delta^{(g+1)} = (1 - c) \cdot \mathbf{s}_\delta^{(g)} + c_\mu \cdot \mathbf{B}^{(g)} \left(\mathbf{D}^{(g)}\right)^{-1} \left(\mathbf{B}^{(g)}\right)^{-1} \frac{\sqrt{\mu}}{\delta^{(g)}} \left(\langle \mathbf{x} \rangle_\mu^{(g+1)} - \langle \mathbf{x} \rangle_\mu^{(g)}\right) \tag{3.4}$$

$$\delta^{(g+1)} = \delta^{(g)} \cdot \exp\left(D \frac{\|\mathbf{s}_\delta^{(g+1)}\| - \hat{X}_n}{\hat{X}_n}\right) \tag{3.5}$$

where $\mathbf{s}_\delta \in R^n$ represents an evolution path, which is not scaled by $\mathbf{D}$.

$\mathbf{D}^{-1}$ can easily be calculated by inverting the diagonal elements of $\mathbf{D}$ individually.

$\mathbf{B}^{-1} = \mathbf{B}^T$.

$D \in [0, 1]$. Parameter for damping the step size variation.

$\hat{X}_n = \sqrt{n}(1 - \frac{1}{4n} + \frac{1}{21n^2})$ estimates the expected length of $\mathbf{s}_\lambda$ under random selection, which is then $\mathcal{N}(\mathbf{0}, \mathbf{I})$ distributed.

In the above steps, some important points need to be emphasized. First, the mutation steps history is recorded in the covariance matrix $\mathbf{C}$, which is then used to calculate the corresponding eigenvector $\mathbf{B}$ and eigenvalue $\mathbf{D}$ based on the multi-variates statistical method PCA. Second, after the analysis of the mutation history, the relationship between different variables and the significance of these relationships are indicated by $\mathbf{B}$ and $\mathbf{D}$, respectively. In PCA, these are the components (eigenvectors) and their significance (eigenvalues). There

is no feature vector formed here, all the components are considered. Third, the k^{th} new normal distribution vector $\mathbf{z}_k$ is now influenced by the eigenvector **B** and eigenvalue **D**, multiplied by **D** gives the vector a new scale, changing its length; multiplied by **B** gives the vector a new direction, changing its direction. In this way, the new normal distribution vector is not arbitrary, it is guided by the evolution history data, having any length and direction in the search space. Finally, the mutation step size σ is changed or evolved separately.

Essentially, the covariance matrix adaptation implements the idea of improving the probability of emphasizing the mutation steps that can create promising solutions. Namely, the covariance matrix of the mutation distribution is changed in order to increase the probability of producing the selected mutation steps again. And also, the rate of change is adjusted according to the number of strategy parameters to be adapted. The adaptation mechanism is inherently independent of the given coordinate system. Finally, the CMA implements a principal component analysis of the previous selected mutation steps to determine the new mutation distribution.

As we have noticed that the CMAES algorithm needs to set a number of relevant parameters, those are recommended according to the authors' experiences, we refer to further details and discussions for CMAES to [HO97, HO01]. In the following chapters, especially in Chapter 6, we will also introduce two CMAES variant algorithms for experimental comparison with our hybrid algorithms.

3.3 Estimation of Distribution Algorithms

A new evolutionary computation paradigm algorithm has recently received a lot of attention, it is the hybrid optimization algorithm called Estimation of Distribution Algorithms (EDA) [MP96, LELP99, PGL99]. EDA can be seen as an outgrowth of genetic algorithms, where a population of candidate solutions are maintained as part of the search for an optimum solution. This population is typically represented explicitly as an array of objects. Depending on the problems, the objects might be bit strings or vectors of real numbers representation. In an EDA, this explicit representation of the population is replaced with a probability distribution over the choices available at each position in the vector that represents a population member.

The most important difference between GAs and EDAs are, in the latter, there they are neither crossover nor mutation operators, instead, EDAs generalize GAs by replacing the crossover and mutation operators with learning and sampling the probability distribution of

the best individuals of the population at each iteration of the algorithm. That is, the new population of individuals is sampled from a probability distribution, which is estimated from a data set containing selected individuals from the previous generation. Working in such a way, the relationships between the variables involved in the problem domain are explicitly and effectively captured and exploited through the joint probability distribution associated with the individuals selected at each iteration. In evolutionary computation heuristics, on the other hand, the interrelations between the different variables representing the individuals are kept implicitly in population. Before exploring each concrete EDA algorithm, we introduce a general EDA algorithm first as Algorithm 5:

Algorithm 5 pseudo code for a general EDA algorithm

1: Generate the initial population at random with M individuals;
2: **while** (the stopping criterion is not met) **do**
3: Select ($N \leq M$) individuals from current population according to a selection method;
4: Estimate the probability distribution of an individual within the selected individuals;
5: Sample M new individuals from this probability distribution;
6: **end while**

As stated before, the EDA algorithms do not have crossover and mutation operators. When the initial population is generated, a subset of current individuals are selected as the best individuals according to a fitness or ranking based selection method. Then the probability distribution for each variable of each individual is estimated. The new population is then generated according to this probability distribution. As we mentioned, this is the general EDA algorithm, it can be seen as a framework for various concrete EDA algorithms, and the most important feature that distinguishes different EDA algorithms is the method that estimates the probability distribution, before we see those specific estimations methods, we will give an illustrative example of a simplest EDA algorithm by simulating the creation of the initial few populations.

3.3.1 Example Illustration

For the problem of optimizing (minimizing) the function $f(\mathbf{x}) = \sin(x)$ with binary variable x_i, for $i = (1 \ldots 5)$. The initial population is obtained at random by sampling the following probability distribution: $p_0((x_i = 1) = 0.5)$ for $i = (1 \ldots 5)$. According to this probability,

Table 3.1: Initial population, P_0

index	X_1	X_2	X_3	X_4	X_5	$f(X)$
1	1	1	1	0	1	0.290285
2	1	1	1	0	0	0.382683
3	0	0	1	0	1	0.471397
4	0	0	1	0	1	0.471397
5	0	1	0	0	0	0.707107
6	1	0	1	0	1	0.881921
7	1	0	0	1	1	0.95694
8	1	0	0	1	0	0.980785

Table 3.2: Selected population

index	X_1	X_2	X_3	X_4	X_5	$f(X)$
1	1	1	1	0	1	0.290285
1	1	1	1	0	1	0.290285
4	0	0	1	0	1	0.471397
5	0	1	0	0	0	0.707107

for each variable x_i, the probabilities of generating 0 or 1 are equivalent. P_0 is the initial population (Table 3.1), with the average fitness value of 0.642814.

In the selected population (Table 3.2) with half of the initial population size, it is possible to emphasize the same individual twice. We estimate the probability distributions for this selected population, the probabilities are: p(X_1 = 1) = 0.5, p(X_2 = 1) = 0.75, p(X_3 = 1) = 0.75, p(X_4 = 1) = 0.0, p(X_5 = 1) = 0.75. According to this probability distribution, the new population is generated as in (Table 3.3).

We can see that the new generated population also have 8 individuals with the average fitness value, 0.529666, compared with the initial average value, 0.642814, the average fitness value is optimized now. This finishes an iteration of the simplest EDA algorithm. In this simplified version of EDA, we ignore the method of creating the probabilities distribution. Also, the above problem is univariate, the variables are independent to each other, so the probability distributions are univariate marginal distributions. However, in many other problems, the variables are not independent, the interdependencies relation could be complex, in these cases, in EDA algorithms, the bayesian network model is used to represent

Table 3.3: New generated population

index	X_1	X_2	X_3	X_4	X_5	$f(X)$
1	1	1	1	0	1	0.290285
2	1	1	1	0	1	0.290285
3	1	1	1	0	0	0.382683
4	0	0	1	0	1	0.471397
5	0	0	1	0	1	0.471397
6	1	1	0	0	1	0.634393
7	0	1	0	0	1	0.77301
8	0	1	1	0	0	0.92388

these dependence relations, which have to be constructed first and the relevant probability distributions need to be calculated.

3.3.2 Structure Learning Methods

As discussed in Chapter 2, Section 2.3.5, the induction of a bayesian network includes two important components, one is the parameters learning component, the other is the structure learning component. We have discussed how to learn the parameters component, here we simply discuss the structure learning method. There are generally two wide methods, one is *detecting conditional dependencies*, the other is *search and score method*.

The PC algorithm [SG91] is one of the examples of detecting conditional dependencies algorithms. It starts by forming the complete undirected graph, then 'thins' that graph by removing edges with any conditional independence relations, after all such conditional independences are all removed, a conditional dependence directed acycled graph is derived, the bayesian network.

The search and score method is to search for a good bayesian network from a huge feasible networks space. To be able to do this, recall from the genetic algorithm section, we need to define suitable measurements of the candidate networks. Once such an evaluation method is available, any heuristic search algorithms can be used to implement searching, we have seen such a search method in Chapter 2, for example, local search algorithms and genetic algorithms are all applicable. And the modification of one network structure could be adding or deleting one arc of the current structure. Finally, the measurement method can depend on the maximum likelihood measurement which we have discussed in Chapter

2 in the parameters learning part. That is, for some observed data set D and a bayesian network, the maximum likelihood estimate, θ, can be used as a measurement of the success of the candidate structure to describe the observed data D. However, it seems that the more complex structure has a bigger likelihood, while complexity is not preferred. So, some suitable penalty functions also need to be defined.

3.3.3 Concrete EDA Algorithms

Under the general EDA algorithm principle, many concrete EDA algorithms have been developed. The main differences of these algorithms are the probability distribution methods applied for sampling new solutions. However, these differences on probability distributions are also due to the features of the problems which these EDAs try to solve. Also, these problem features define the criterion for classifying these EDA algorithms. Before introducing the most commonly used EDA algorithms, we give the classification standards for these algorithms. First, they can be grouped according to the problem types, discrete value (combinatorial) and continuous-value. And then, EDA algorithms can be classified by the complexity of the probabilistic models used to learn the interdependencies between the variables from the data set of selected individuals. Therefore, EDAs can be classified as non-dependencies, bivariate dependencies, multivariate dependencies. We introduce the EDA algorithms in the order of these classification standards. The first of them is the Univariate Marginal Distribution Algorithm (UMDA) introduced by Mühlenbein[Müh97], detailed as Algorithm 6.

Algorithm 6 pseudo code for Univariate Marginal Distribution Algorithm (UMDA)

1: generate the initial population at random with M individuals;
2: **while** (the stopping criterion is not met) **do**
3: select $N \leq M$ individuals from current population according to a selection method;
4: Estimate the joint probability distribution with $p(x) = p(x|D) = \prod_{i=1}^{n} p(x_i) = \prod_{i=1}^{n} \frac{\sum_{j=1}^{N} \delta_j(X_i=x_i|D)}{N}$
5: sample M new individuals from this probability distribution;
6: **end while**

The model used by UMDA to estimate the joint probability distribution of the selected individuals at each generation, $p(x)$, is very simple. Each univariate marginal distribution is estimated from marginal frequencies:

$$p(x_i) = \frac{\sum_{j=1}^{N} \delta_j(X_i = x_i|D)}{N}$$

where

$$\delta_j(X_i = x_i|D) = \begin{cases} 1 & \text{if in the } j^{th} \text{ case of D, } X_i = x_i; \\ 0 & \text{otherwise.} \end{cases}$$

Another EDA algorithm which considers multiple dependencies is the algorithm called the Estimation of Bayesian Networks Algorithm (EBNA) introduced in [LELP00]. This and its variant algorithms are typical algorithms that apply the bayesian networks as the probability distribution estimating method. In these algorithms, the parameters learning for the networks is implemented by learning the factorization of the joint probability distribution encoded by a bayesian network from the selected data set. The structures of the bayesian networks are learned from the following steps. The first generation of the networks is generated throughout the networks space. And then either of the following options can be chosen, test on conditional independences between variables, applying the PC algorithm; or some simple search algorithms can be employed to search for a good network structure and some evaluating methods for guiding the search algorithm for good network structures are applied, among these methods, the K2 algorithm combined with penalty function or BIC are applied, each of the options gives different instances of the EBNA algorithms, as shown as Algorithm 7.

Algorithm 7 pseudo code for $EBNA_{PC}$, $EBNAK2 + pen$, $EBNA_{BIC}$ algorithms

1: generate the initial population at random with M individuals;
2: **while** (the stopping criterion is not met) **do**
3: select $N \leq M$ individuals from current population according to a selection method;
4: conditional (in)dependence tests $\rightarrow EBNA_{PC}$
5: penalized Bayesian score+search $\rightarrow EBNA_{K2+pen}$
6: penalized maximum likelihood + search $\rightarrow EBNA_{BIC}$
7: sample M new individuals from this probability distribution;
8: **end while**

The $EMNA_{global}$ algorithm [LnLB01] is an approach based on the estimation of a multivariate normal density function at each generation. As described in Algorithm 8, at each generation, we estimate the vector of means, $\mu = (\mu_1, \mu_2, \ldots, \mu_n)$, and the variance-covariance matrix, Σ, whose elements are denoted by σ^2_{ij} with $i, j = 1, \ldots, n$. This means

Algorithm 8 pseudo code for $EMNA_{global}$ algorithms

1: generate the initial population at random with M individuals;
2: **while** (the stopping criterion is not met) **do**
3: select $N \leq M$ individuals from current population according to a selection method;
4: $f(x) = f(x|D) = \mathcal{N}(x; \mu, \sum)$ Estimate the multivariate normal density function from the selected individuals.
5: sample M new individuals from this probability distribution;
6: **end while**

that we need to estimate m means, n variances and $n \cdot (n-1)/2$ covariances. These parameters estimations use their maximum likelihood estimates in the following way:

$$\mu_i = X_i = \frac{1}{N} \sum_{r=1} N x_{i,r} i = 1, \ldots, n$$

$$\sigma_i^2 = \frac{1}{N} \sum_{r=1} N (x_{i,r} - \mu_i)^2 i = 1, \ldots, n$$

We finish our introduction on EDA concrete algorithms for now. However, in Chapter 7, another EDA algorithm called Population Based Incremental Learning (PBIL) will be used to solve and make comparisons with our hybrid algorithms on optimizing the cancer chemotherapy treatments problem which is a practical evaluation-expensive optimization problem. We will delay the introduction of PBIL until then.

3.4 Learnable Evolution Model (LEM)

The LEM was introduced by Michalski in 2000[Mic00]. LEM is a highly generalized hybrid approach for optimization, which involves interleaved bouts of evolution and learning. The overall idea of LEM is to run repeated stretches of evolution and learning in series, where the next 'evolution' stretch is informed in some way by the previous 'learning' stretch, which in turn learned about the mapping between genotype and fitness from previous populations. Namely, to infer relationships between gene values and fitness.

3.4.1 LEM(AQ)

Before we introduce the general LEM framework, we explain a LEM instance algorithm, Learnable Evolution Model with AQ learning algorithm (LEM(AQ)), as described in [Mic00]. Firstly, in LEM(AQ), the initial population is generated and evaluated. It is then divided

into high-performance (H-group) and low-performance (L-group) groups according to the initial individuals' fitness values. These two groups are then used as the positive and negative training examples for the AQ learning algorithm, which has been discussed in Chapter 2. The outcome of the AQ learning algorithm is a set of rules expressing inductive hypotheses (in terms of intervals of gene values) for the positive and negative examples, and can be used to predict the class information (i.e. H-group or L-group) for future unseen examples. LEM(AQ) then proceeds with an otherwise normal evolutionary algorithm, except that the operators are designed so that new individuals are generated only with gene values within the ranges of values sanctioned by the recently learned inductive hypotheses. LEM(AQ) then continues for a specified number of generations, and then pauses for more learning based on the current population. This in turn feeds into the next stage of evolution, and so on. There are additional complications and sophistications in LEM(AQ) that mediate the transitions between learning and evolution.

3.4.2 LEM Framework

With the introduction of the LEM(AQ) algorithm, we are ready now to introduce the general LEM framework. The general LEM framework is very important in our investigations in the hybrid optimization algorithms in this thesis. All of the hybrid algorithms developed in this thesis, and also the LEM(AQ), are based on this framework, or more precisely, are instantiations of this general framework. Our hybrid algorithms are inspired by this basic and general LEM framework, to which we have shown more flexibility and creativity by incorporating new learning components and new interaction methods. We give this framework first in Figure 3.1:

As we can see from the LEM framework, first, many standard evolutionary computation components and operations are applied in the LEM framework. Second, the way in which learning and evolution interact are flexible and depends on different situations and on the progress of the optimization procedure. Finally, LEM(AQ) is only one of the possible instantiations for the wider LEM framework.

The standard evolutionary computation components such as generating initial population, evaluating individuals, and parent selection etc are parts of the evolutionary component for the LEM framework. They can all be implemented in the normal way as described in the evolutionary computation section. The way learning and evolution interact need to be decided by the select actions stage, where each action corresponds to one mode of opera-

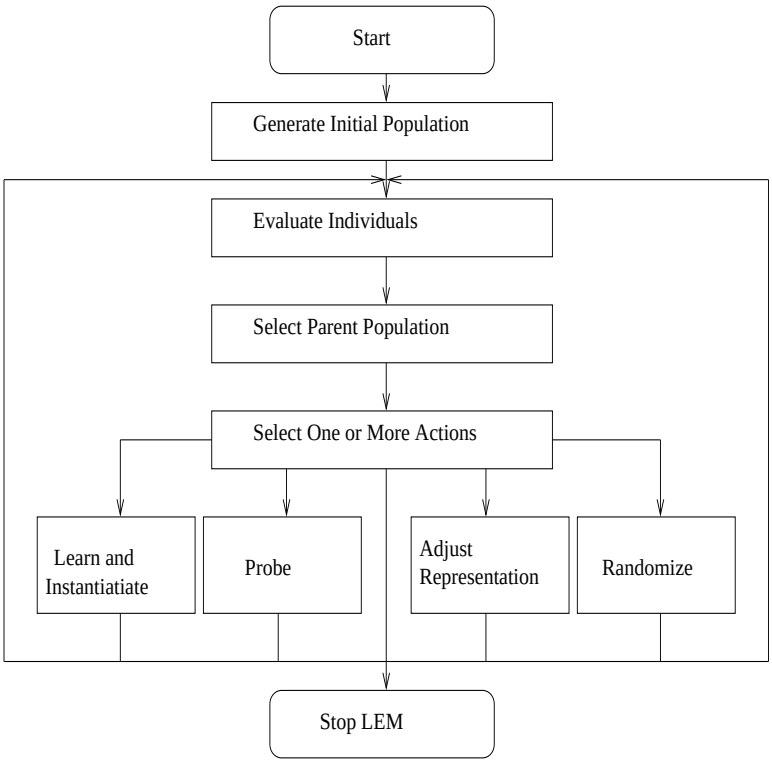


Figure 3.1: The general LEM framework

tion. LEM can also select one or more actions in parallel, which is controlled by the Action Profile Function (APF). There are basically four modes in LEM, they are learning mode, probe mode, change representation, and randomization.

Learning mode is the main operation in LEM, it contains three operations, which are the training examples selection, hypothesis generation, and hypothesis instantiation. The training examples selection stage can be implemented in many ways, such as ranking based and fitness based. For both methods, a threshold is needed, in the ranking based method, all of the individuals that are in the high group defined by the threshold are selected as the positive data, the same principle for the negative data. In the fitness based method, precisely those fitness values which are in the top and low groups defined by the threshold are selected as positive and negative data.

When the training data are selected, the AQ learning algorithm is used to generate the hypothesis, a rule-set describing the training data. After this, the instantiation procedures are used to generate new individuals for the next generation. The successful implementation of instantiation is crucial to the success of LEM implementation, even the learning algorithm has successfully indicated the promising districts of the search space, inefficient implementation of instantiation also cannot lead to good performance. This needs to develop efficient instantiation procedures to utilize the learnt information effectively.

The probing mode executes evolutionary computation operations in order to generate new individuals. The two operators implemented in LEM are crossover and mutation. One important issue in applying these genetic algorithm operators is that, they are not used to generate promising solutions but rather to maintain the diversity of the population. This is due to the fact that as the learning and evolution procedure continues, the population will soon converge to some narrow districts involving local or global optima, meanwhile, the diversity of the population will disappear very quickly. This is a quite normal situation which occurs in an evolutionary search, however, it causes a particular difficulty for the LEM method due to the application of learning in this framework. Namely, lack of diversity for the population and therefore the problem of not having enough training data will cause the learning algorithms to be unable to generate useful and representative hypotheses and rule-sets.

The discretization mode in LEM framework raises the requirement for any adaptive discretization methods. These discretization methods are necessary for the LEM framework based optimization procedures and can increase the precision of the discretized real variables in the most promising areas, or neighborhoods of the fittest individuals.

In addition to the probing mode which can generate random individuals, randomization mode further adds randomly generated individuals to a population in order to introduce more diversity, or replaces the entire population in a start-over process. This mode is beneficial when the learning mode leads the search procedure into a wrong direction or local optima, and has no hope of restoring from the wrong directions easily.

The switch between these modes and actions described above is controlled by the Action Profiling Function (APF). In the LEM framework, APF controls two important aspects first, it can adaptively decide the number of individuals that will be generated by each mode, this is done by defining parameters like *average-learning-fitness*, *average-probing-fitness*. If the former is bigger than the latter, then the number of individuals generated in the learning mode should be increased. Another aspect is the no-progress parameters, indicating within a number of iterations that the program is making no progress in terms of values of the fitness function. This situation can be identified through the use of two program parameters, learn-probe and learn-threshold. Learn-probe defines the maximum number of iterations that are performed even if there is no satisfactory progress, and the learn-threshold defines the minimal acceptable increase of fitness of the best individual. In such a situation, the no-progress condition is met, and the available actions will be triggered in a pre-defined order.

3.4.3 Relations with EDAs

Meanwhile, while LEM was initially published only in the machine learning community, at around the same time Estimation of Distribution Algorithms started to shoot to prominence in the evolutionary computation community [LL02]. EDAs can also be viewed as learning/evolution hybrids, with the emphasis on building and maintaining models of fit chromosomes. Both LEM and EDA techniques now have several published variants (particularly EDA variants), and it is interesting to consider what the definitive differences are. It seems correct to suggest that while EDAs focus on modeling as the key force behind search activity (i.e. search is guided closely by statistical models, with new sample points generated directly from the model), in LEM the evolutionary component is more responsible for the search (i.e. new points are sampled mainly in the familiar way by using genetic operators on a population of chromosomes), with guidance from learning processes.

Most interestingly, recent results from the LEM team compare EDAs and LEM3 directly [WM06]. They report using various EDA implementations from [BMLL02], with best results of these on Rosenbrook and Griewank functions, found by EMNA_*global* [LLB02]. Comparison of LEM3(AQ) and EMNA_*global* on these functions showed LEM3 is between 15 and 230 times faster in achieving its best value, which in turn was always better than that achieved by EMNA_*global*. Finally, it must be pointed out that *hybrids* of EDA and GAs, or of EDAs and other search methods, have started to appear since at least 2003 [ZSTF03, ZSTF06, PRL+04]. When contrasting the LEM framework with the EDA framework, it is perhaps clearest to say that LEM is similar in style to a hybrid EDA/GA, and this seems to be reflected in the relative success that has so far been shown for EDA/GA hybrids.

3.4.4 Applications of LEM

The performances of LEM(AQ) have been reported as very promising, with improvements both in solution quality and dramatic speedup when compared to the 'without learning' equivalent EA. The developers of the LEM framework are continually updating the 'AQ15' version [JKES95] for the AQ learning algorithm and continuing to report impressive results, albeit on a limited suite of test functions. In application-oriented work, a multiobjective LEM-based approach, using C4.5 [Qui93] as the learning method, was found to significantly speed up and improve solution quality for large-scale problems in water distribution networks [JCSW05]. Meanwhile the team that developed LEM has updated the framework [WM05] and continued to obtain impressive results [WM06].

The design and application of LEM clearly merits considerably more research. The speedup derived by applying LEM is reported in several papers, that is, the reduction in the number of fitness evaluations needed to reach high quality results, this improvement is of particular interest for many important applications in which fitness evaluations are costly. In such applications, time savings can make the difference between the problem being solvable or not at all. With an interest in a clearer understanding of the LEM framework and its performance, we investigated in this thesis a number of LEM instance algorithms and the performance improvements obtained by using these algorithms are reported in the following chapters.

Chapter 4

KNN Based LEM Hybrid Algorithms

4.1 Overview

In Chapter 3, we have seen some modern hybrid optimization algorithms. Among them, we are particularly interested in the general LEM framework and the LEM instance algorithm LEM(AQ). In this chapter, we start the expedition of investigating more new LEM instance algorithms. This further research into LEM methods is due to the following three main reasons.

The first one is the scientific research interest. As we can see, there exist many learning methods in the machine learning community. What will happen if we replace the AQ learning algorithm with the other well-known learning methods? Will the resulting LEM algorithms perform equally well? That is, can they still achieve the same performance improvements over the same set of problems as the original LEM(AQ) does? The feasibility of this investigation is based on the fact that, although the learning methods are varied in many aspects, many of them do share many similarities. For example, for the supervised learning methods, all of them need some training data as input, and output some forms of model or hypothesis which take the form of either trees, or rules, or the training data themselves. So, the general forms are the same, apart from the induction details. More precisely, we want to make the LEM framework more flexible and extendable to any learning methods, and the application or choice of a particular learning method will depend on the problems at hand or user's preference. Ultimately, this aim can also be understood as offering a user-friendly interface, where before the run of the LEM framework (which is a huge collection of various learning and evolution algorithms), a set of optional parameters can be chosen, or during the run of the LEM framework, the suitable learning methods can be selected adaptively according to the progress of optimization.

The second reason is that we want to clarify how learning and evolution interact. LEM(AQ) has shown a good way to interleave the learning and evolution procedures, they can be carried out in series or in parallel or the output of one procedure can be used as the input of another procedure. Learning methods generally contain several functions, for example, the classification and prediction functions. Therefore, the question arises, are there any other ways in which learning and evolution can interact? To answer this question, that is to find out another new way for learning and evolution to interact will be a very interesting and challenging task, and also will further show the flexibility of the LEM framework. Therefore, this is an important investigation direction for LEM research.

Finally, another important reason for investigating LEM is application-oriented. The promise shown in the work on original LEM for considerable speedup for the optimization of many evaluation-expensive problems also clearly merits considerably more research into the design and application of LEM. This led to an investigation of an LEM variant algorithm on the large scale water distribution network problem [JCSW05]. In these and many other problems where fitness function evaluations take considerable time, time savings are precious, and can easily make the difference between the problem being solvable or not. This forms another important reason for investigating LEM, we want to see how successful LEM is in achieving speedup for evaluation-expensive problems.

Based on these reasons, in this chapter, we start this expedition by investigating LEM in its (we think) simplest form, using KNN (Section 2.3.3) as the 'learning' mechanism. The resulting algorithm is called LEM(KNN) [SC08]. In LEM(KNN), the way learning is used is quite different from the way it is used in LEM(AQ). KNN learning is used to predict the new individuals generated by the evolution mode and can be seen as a survival selection method for selecting the newly generated individuals. More precisely, learning is used as a 'filter' which can predict the 'fitness' of these newly generated individuals in some way prior to the evaluation of these individuals. If one individual is predicted as fit enough, it will then survive and be evaluated, otherwise, it will be discarded. Evidently, a new learning and evolution interaction mechanism is created in LEM(KNN). Finally, if such predictions by KNN are correct to a certain extent, then a suitable substitute for the survival selection operation is found, which allows a huge amount of evaluations to be avoided and saved.

We will present LEM(KNN) in complete detail in the following sections and evaluate the performance of this LEM instance algorithm. We test LEM(KNN) on the same set of problems that were used in the original LEM paper. A further refined LEM(KNN) algorithm called LEM(dwKNN) is also developed for reasons we will indicate later. LEM(dwKNN)

incorporates the distances contribution to the KNN algorithm and is able to obtain better optimization performance. Both algorithms are tested on a set of test functions widely used in the optimization community.

In the remainder, we continue as follows. Section 4.2 provides complete detail of our LEM(KNN) algorithm, (also denoted as, GA hybridized with KNN algorithm (KNNGA)), and presents the experiments and results. Section 4.3 provides complete details of the refined and generalized LEM(KNN) algorithm, LEM(dwKNN), (sometimes we also denote as the GA hybridized with distance-weight KNN algorithm (dwKNNGA)), and presents the experiments and results. We conclude and discuss in Section 4.4.

4.2 LEM(KNN) – KNNGA

The LEM(KNN) algorithm has its evolution component as the standard genetic algorithm and its learning component as the KNN method. This algorithm is inspired by the original LEM method and can be viewed in a number of different ways. It belongs to the general LEM framework, because it replaces the AQ learning method with KNN; it shows the flexibility of LEM framework by adding a new interaction relationship between learning and evolution; it uses a new survival selection mechanism in the context of a standard GA. In the following discussion, we use both the terms KNNGA and LEM(KNN), first, they are completely equal terms. LEM(KNN) is used when we emphasize it as part of the LEM framework; KNNGA is used when we emphasize its similarity with GA. The same term conventions apply for dwKNNGA and LEM(dwKNN).

4.2.1 KNNGA Algorithm

There is a big difference between LEM(AQ) and our KNNGA in how learning influences evolution, which is quite simplified in KNNGA. In LEM(AQ), the generation of new individuals are instantiating of the description (set of rules) of the H-group or L-group. However, in KNNGA, new individuals are still generated by the common GA mutation and crossover operators, KNN is applied as a particular form of survival selection operator which judges an individual according to its neighbors. A detailed description of our KNNGA algorithm is given below, in which we assume a maximization problem is being considered.

As with LEM(AQ), KNNGA divides the population into high-performance (H-group) and low-performance (L-group) groups according to their fitness values and a given *thresh-*

old (say, 30% – that is, the fittest 30% form the H-group and the worst 30% form the L-group). This is then saved as the *learning population*. Individuals of the H-group and L-group in the *learning population* form the training examples used by the KNN algorithm. Effectively, 'learning' here corresponds entirely to the process of classification into these groups based on fitness, and hence is one of the simplest learning schemes conceivable. However, this goes hand in hand with the use of the *learning population* in predicting the quality of newly generated individuals, which goes as follows.

The common mutation and crossover operators are used to generate new individuals in the normal way. Once a new individual is generated, KNN is used to predict if this individual is 'good' or 'bad' according to the *learning population* which is the training examples. First, we find the k nearest neighbors for this new individual; if the majority of these neighbors are in the H-group, then this individual is predicted as 'good', otherwise this individual is predicted as 'bad'. The 'good' individuals are retained to form the new population for the next generation, and are evaluated in the normal way in GA. The 'bad' individuals are discarded without evaluation. This continues until sufficient new individuals are generated in (or, predicted to be in) the H-group to form a new population. When a fixed number of generations (we indicate this as *learning gap* (LG)) are generated, the *learning population* is updated by the current population. Again, the *learning population* is classified into the H-group and L-group. This is repeated until a termination condition is reached. Now we try to ensure a replicable explication with pseudo-code. 'Overview' pseudo-code for KNNGA is as follows:

1. Set parameters: Set values for *population size*, parameters for mutation (mutation step size, mutation probability), parameters for crossover (crossover probability) and set elite-preserve operator option. Set k (indicating the number of neighbors in KNN algorithm), *learning gap* (indicating the interval before one learning population is updated by another) and the *threshold*.

2. Generate initial population: Choose a method to create the initial population with *population size* and evaluate this population.

3. Derive extrema: Copy the *current population* as the *learning population* from which create the high fitness group (H-group) and low fitness group (L-group), according to fitness values and *threshold*. These two groups could have a joint set, or their union could be a subset of the whole population set or even equals to the whole population set. These two groups are stored for KNN algorithm.

4. Generate new generations: After selecting parents based on the *current population*, apply the mutation, crossover operators to generate new individuals. Once a new offspring is generated (it is not evaluated and is not placed in the mating pool immediately), KNN is applied to find its k nearest neighbors with regard to H-group and L-group (not the whole *learning population*). For these k nearest neighbors of this offspring, KNNGA judges the majority according to their fitness values, there will be two cases:

   i) if the majority is high (that is, most of this offspring's k neighbors are members of H-group), then this offspring is retained into the newly created population and evaluated.

   ii) if the majority is low (that is, most of this offspring's k neighbors are members of L-group), then this offspring is aborted.

   The generating procedure continues until this new population is filled with such newly generated individuals. This finishes the generation of one generation.

5. Update H-group and L-group: When the *learning gap* is reached, the *learning population* is replaced by the *current population*. The H-group and L-group are therefore recalculated according to the current *learning population* and the same *threshold*. The new H-group and L-group are again stored for KNN.

6. Termination condition: The above steps 4 and 5 repeat until some termination conditions are satisfied:

   i) the optimal (if known) is reached; or

   ii) the maximum number of generations allowed is reached; or

   iii) the best fitness value has not been improving for a certain number of generations.

The pseudo-code for our specific instantiation of KNNGA is set out as algorithm 9. The idea behind KNNGA is that, instead of using the traditional survival selection operation, we can utilize the prediction capacity which is almost available to all machine learning methods. The only requirement for these learning methods is a set of suitable training data which should satisfy some certain criterion of quality and quantity. Predicting the fitness of each new generated individual does the same job as the survival selection does in principle. The former is 'guessing' according to the performance of previous individuals, and the

Algorithm 9 pseudo-code for KNNGA

1: $population\_size = 100, i = 0$;
2: $generation\_number = 0, max\_generation\_number = 500$;
3: $k = 5; learning\_gap = 1, threshold = 0.3$;
4: Initialize a new population with *population_size*, and evaluate it;
5: **repeat**
6: Select parents based on *current population*;
7: **if** (*generation_number*%*learning_gap* == 0) **then**
8: Copy *current population* into *learning population*;
9: Calculate the *H*-group and *L*-group according to *threshold*;
10: **end if**
11: **while** ($i <$ *population_size*) **do**
12: Mutate a parent individual to generate a new child;
13: Calculate the *k* nearest neighbors for this child;
14: **if** (the majority of this child's *k* neighbors are nearer to *H*-Group) **then**
15: Evaluate and place it into the next generation;
16: *i*++;
17: **else**
18: Child is aborted;
19: **end if**
20: Apply crossover on two parent individuals in the *current population* to generate two new children;
21: For each of these two children, repeat steps 13-19;
22: **end while**
23: *generation_number*++;
24: $i = 0$;
25: **until** (*generation_number* == *max_generation_number*)

later is exactly ‘evaluating’ or ‘working out’ the accurate fitness value for each individual, which is very expensive for evaluation-expensive problems. We highlight the advantage of KNNGA by comparing it with the standard GA, and we find that the main and only difference between these two algorithms is the substitute of survival selection operator with KNN learning method, as seen in Figure 4.1 and Figure 2.1, showing the similarity and difference between the KNNGA and a normal GA.

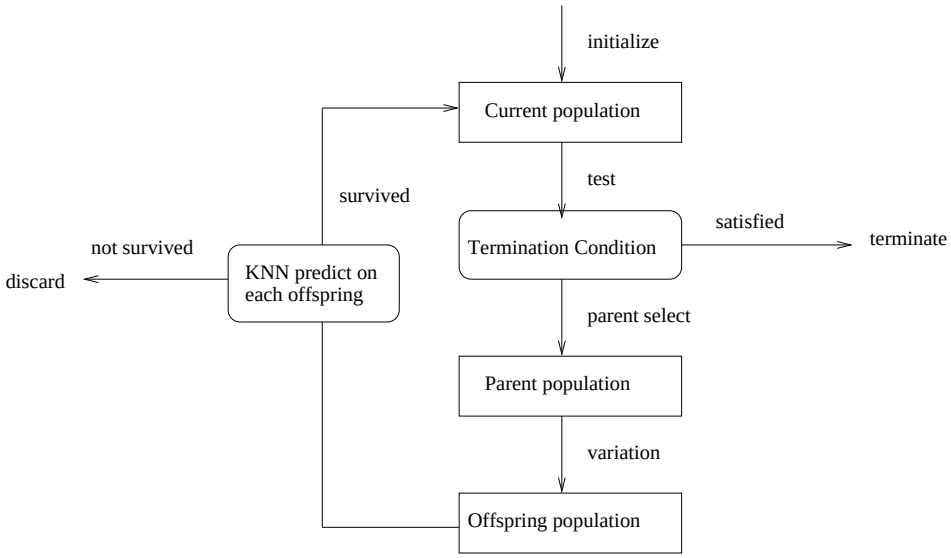


Figure 4.1: Flowchart of the KNNGA algorithm

KNNGA Execution Time Analysis

One of our main motivations for investigating LEM-based methods is their promise of speedup on large-scale optimization problems. That is, achieving good results with relatively few evaluations, which is particularly important when a single evaluation is time-comsuming. We therefore provide this simple analysis of execution time for completeness, in order to better understand how the number of evaluations depends on other aspects of the algorithms studied.

We assume for both KNNGA and GA that the population size is p, the maximum allowed number of generations is M, the time for evaluating one single individual is t_{eval}, and the new population is generated either by crossover operation or mutation operation. Meanwhile, t_{search} represents the time spent on searching for a satisfying individual. For the general GAs, the time spent on the whole evolutionary process T_{GA} is calculated by:

$$T_{GA} = (p + M \cdot p) \cdot t_{eval} \tag{4.1}$$

There are p evaluations for the initial population, and p evaluations for each of the following M generations. The time spent on the evolution/learning process T_{KNNGA} is calculated by:

$$T_{KNNGA} = (p + M' \cdot p) \cdot t_{eval} + M' \cdot p \cdot t_{search} \tag{4.2}$$

Again, p evaluations are needed for the initial population, and p evaluations in each of the following M' generations, the generations spent by KNNGA when the same qualified solution is found as GA does. In addition to the evaluation time, KNNGA needs search time $p \cdot t_{search}$ in each M' generations. Finally, the time difference T_D between T_{GA} and T_{KNNGA} is:

$$T_D = (M - M') \cdot t_{eval} - M' \cdot t_{search} \tag{4.3}$$

The most important point is that, generally, the search time spent in the problem representation space t_{search} is proportional to the properties that define the problem representation space, such as discrete or continuous, attributes or dimension number, and domain number for each attribute. However, the time spent on evaluating each solution t_{eval} depends on the problem definition and the problem complexity. So, in the evaluation-expensive problems, the evaluation time could be much longer than the search time for a qualifying solution. That is, we have $t_{eval} >> t_{search}$ in Equation 4.3 for evaluation-expensive problems. And also, the development of the LEM(KNN) algorithm and the claim made by the original LEM authors, there should be speedup in evaluation number for LEM(KNN) over the normal GA algorithm, that is, $M > M'$. Therefore, the saved computation time T_S should be expected and calculated approximately as:

$$T_S \approx (M - M') \cdot t_{eval} \tag{4.4}$$

As we can see from the Equation 4.4, the t_{search} time is omitted, as it is actually a relatively fixed time expense for the given problems, it only depends on the problem representation and dimensions. On the other hand, the t_{eval} could be very different from problem to problem and much more expensive than the search time especially for expensive-evaluation problems. The term $M - M'$ represents the expected time savings that we want to achieve by developing new LEM hybrid algorithm, it should be a positive integer and as big as possible. In the following experiment sections, we will try to verify this anticipation.

Picturing the KNNGA evolution procedure

We use a simple linear function maximization problem as an example problem to illustrate how KNNGA operates. The problem has a two-dimensional population space, therefore each individual consists of two genes (attributes). As KNNGA is running, the sequential populations will be occupied by the individuals nearer to the H-group in the current population.

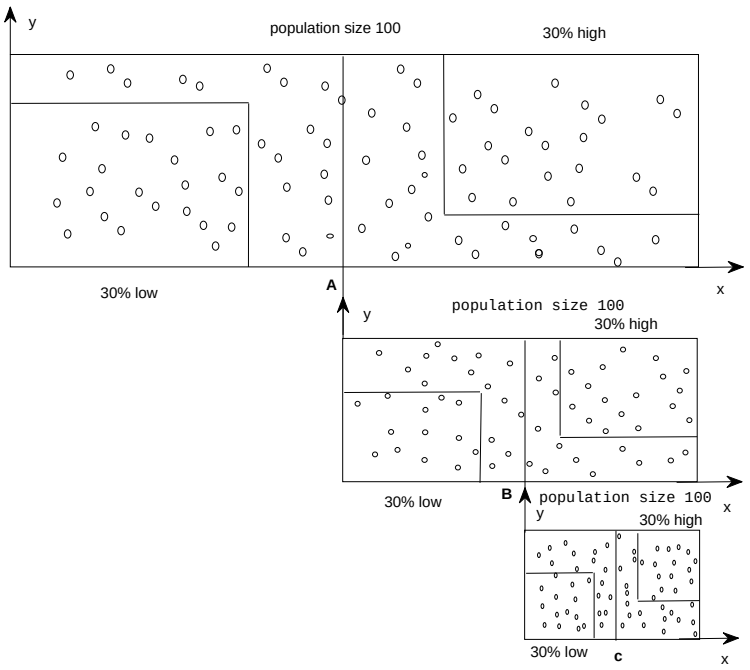


Figure 4.2: An illustrative flowchart for the KNNGA algorithm evolution procedure

Figure 4.2A shows the first generation, the initial individuals are evenly distributed within the whole population space, and for a given *threshold* (eg. 30%) the H-group and L-group are formed.

Figure 4.2B shows the second generation derived by LEM(KNN), this population space is crowded by individuals that are within the high fitness half of the first population. Since the degree to which the individuals are now spreading out in genotype space is around half what it was previously, the density in genotype space is roughly doubled. This population now undergoes classification into H-group and L-group, resulting in Figure 4.2C.

Figure 4.2C shows the third generation of the population, and we see continued reduction in the 'spread' of the population. Clearly, the current whole population has focused on a region increasingly defined by the H-group individuals of the first and second generations.

An obvious and perhaps important aspect of LEM(KNN) (and LEM methods in general, is this strongly defined movement of the population between generations, which is clearly

guided (by the results of learning) and less randomized and exploratory than a normal GA. Naturally this has potential drawbacks; we could expect the learning process to misguide the population on certain landscapes, and become stuck in poor regions. Whether or not this generally happens on problems of interest and importance, and (if so) whether the deceptive nature of the landscape is equally deceptive for normal GAs in such cases, are moot points. Empirical evidence to date is suggestive that this general strategy is certainly more often effective than not.

We implement the LEM(KNN) algorithm in later sections, before we do that, we will investigate LEM(KNN) with more ideas. These ideas are some trivial modifications based on the LEM(KNN) algorithm, and one of the main purposes of these modifications is to better evaluate the capacity of the LEM(KNN) algorithm. We see one such idea which is to verify the quality of the individuals who survived the KNN method in LEM(KNN) algorithm in the next section.

4.2.2 KNNGA 'with verification'

We have designed the LEM(KNN) algorithm. Before we test its performance, there are many questions and ideas about KNNGA which deserve more discussion. Among them, some questions interest us. In KNNGA, when a new individual is generated, the fitness of the neighbors of this individual from the *learning population* are checked, and this guides whether or not it enters the population in the next generation. The key difference between KNNGA (a 'learning-guided' search) and GA (a pure black box search) is that, in this way, a newly generated individual is discarded before evaluation if we predict that it will not be good enough. The flip-side of this, of course, is that we may well admit new individuals into the population that pass this test, but ultimately they prove to be unfit. That is, it could be that the prediction provided by KNN is wrong, and as a supervised learning method, this case happens normally. The prediction accuracy depends on many factors as we discussed in Chapter 2, and cannot always be high as long as the learning algorithm satisfies the PAC-learning theory. Therefore, our concern and question in mind are: how often will this situation happen? And will these wrong predictions influence the performance of our LEM(KNN) algorithm? If so, how serious could this influence be?

To understand the degree to which this happens, we also test a modified version of the KNNGA algorithm which includes a step of 'verifying' the correctness of the prediction. When an individual is generated by a mutation or crossover operator, as before, KNNGA

calculates its *k* nearest neighbors, again there are two possible cases. For the second case, where the majority of its *k* nearest neighbors are in L-group (for maximization problem), this individual is aborted without evaluation; for the first case where the majority of its *k* nearest neighbors are in H-group, this individual is further tested instead of being immediately placed into the next generation. That is, after being evaluated, it is compared with a pre-selected value (eg the worst fitness value in the current population), if this individual's fitness value is higher than this value, then it survives into the next generation; otherwise, it is still aborted. We call this modified version of KNNGA as KNNGA [SC08]. Compared with KNNGA, KNNGA with verification (KNNGA(V)) adds one more condition restricting the new individuals' ability to survive. Namely, in order to survive into the next generation, the new individual should not only survive the KNN filter, but also should be better than the worst individual in the current generation. The predictions made by KNN are verified as 'correct' or not in this sense. The corresponding KNNGA algorithm should also be modified. KNNGA(V) algorithm is the same as KNNGA except that Algorithm 9's 14-19 lines are replaced by Algorithm 10.

Algorithm 10 part pseudo-code for KNNGA(V)

```
1:  if (the majority of this child's k neighbors are nearer to H-Group) then
2:     Evaluate this child with the fitness function;
3:     Find the worst_fitness value of the current population;
4:     if (fitness(child) > worst_fitness ) then
5:        Places this child into the next generation;
6:        i++;
7:     else
8:        Child is aborted;
9:     end if
10: else
11:    Child is aborted;
12: end if
```

This 'with-verification' variant does not at first sight seem well-suited to the goal, in problems with time-consuming fitness functions, of reducing the number of evaluations as much as possible. However, we were interested in any trade-off there may be between the increase in computation time and the quality of solutions obtained. We will come back to this topic in later sections in this chapter when we introduce the refined LEM(KNN) algo-

rithm. For now, we continue our investigation of the KNNGA algorithm by experimenting on both the KNNGA and KNNGA(V) algorithms.

4.2.3 Experiments and Results

This section describes the experiments and results of the comparison between KNNGA algorithms and the corresponding GA, which is the evolutionary algorithm identical to our KNNGA implementation in all respects other than the use of KNN.

Test Functions

The test problems used here are those used originally in [Mic00] to evaluate the performance of LEM(AQ). In that work, the author reported on two problems from the De Jong's suite [DJ75], and variants are tested with different numbers of dimensions. They also reported that similar findings were achieved with the other De Jong problems in [MZ00]. An additional problem tested in [Mic00] is also tested here, this is from the domain of parameters estimation in nonlinear digital filter design, which is simulated using equations gleaned from [YS94]. The problems test suite we used in the section is named as 'test suite 1' for convenience, which consists of five functions.

1. Problem 1 : Find the maximum of function f_1 with five variables.

$$f_1(x_1, x_2, x_3, x_4, x_5) = \sum_{i=1}^{5} integer(x_i) \quad -5.12 \leq x_i \leq 5.12 \tag{4.5}$$

$$Maximum : 25.$$

2. Problem 2 : Find the maximum of the function f_2 of 30 continuous variables with Gaussian noise:

$$f_2(x_1, x_2, x_3, \ldots, x_{30}) = \sum_{i=1}^{30} ix_i^4 + Gauss(0, 1) \quad -1.28 \leq x_i \leq 1.28 \tag{4.6}$$

$$Maximum : approximately\ 1248.225.$$

3. Problem 3 : Determine optimal parameters of nonlinear filters defined by the equation:

$$y(k) = \left[\frac{3 - 0.3y(k-1)u(k-2)}{5 + 0.4y(k-2)u^2(k-1)}\right]^2 + (1.25u^2(k-1) - 2.5u^2(k)) \times \ln(|1.25u^2(k-2) - 2.5u^2(k)|) + n(k) \tag{4.7}$$

where k is the sample index or time, $n()$ is a noise component ranging from -0.25 to 0.25, and $u()$ is an inserted function (sin, step, random). The coefficients -0.3, 0.4, 1.25, and -2.5 are assumed as variables which will be optimized and can be seen as the genes of individuals. The problem is to find their correct values using samples $\{\langle vector_i, y(vector_i)\rangle\}$, where $vector_i$ is a specific assignment of values to variables and $y(vector_i)$ is the value of the equation for this assignment. When substituted in the equation, individuals generate a value of y that is compared with the value computed when correct coefficients are used in the equation. The fitness of an individual is defined as in [YS94] as the reciprocal of the mean-square error over 200 sample window:

$$Fitness(Vector) = \frac{1}{MeanSquareError} = \frac{200}{\sum_{200}(Vector - KnownValue)^2} \tag{4.8}$$

4. Problem 4 : Find the maximum of function f_4 with 100 variables.

$$f_4(x_i) = \sum_{i=1}^{100} integer(x_i) \quad -5.12 \leq x_i \leq 5.12 \tag{4.9}$$

$$Maximum : 500.$$

5. Problem 5 : Find the maximum of the function f_5 of 100 continuous variables with a Gaussian noise:

$$f_5(x_1, x_2, x_3, \ldots, x_{100}) = \sum_{i=1}^{100} ix_i^4 + Gauss(0,1) \quad -1.28 \leq x_i \leq 1.28 \tag{4.10}$$

Maximum : 13556.

We were interested in the basic performance of KNNGA vs GA, so that we could sample the degree to which (if any) the LEM framework could be successful when using the simplest possible learning scheme. However, we also took the opportunity to contrast with- and without-crossover versions for both the GA and KNNGA. Thus we use notation such as 'GA(m)' (the GA with mutation only) and 'KNNGA(c,m)' (KNNGA with both crossover and mutation).

Parameter Settings

In all cases, the encoding was a vector of real-valued genes each encoding numbers within a specified interval. We used binary tournament selection [Bäc95, BT96, GD91], elitism (the next generation's population is always initialized with the best of the previous generation), and uniform crossover [Sys89]. Mutation is implemented by randomly adding or subtracting a small value to one gene. For different problems, the values for *k*, the *learning gap* may be different. For each problem, KNNGA and GA use the same initialization method to generate the initial population. For all cases, the population size is 100. We summarize all the parameter settings for GA, KNNGA and KNNGA(V) in Tables 4.1, 4.2.

Summary of Results

All experiments are repeated 100 times independently to provide sufficient evidence for claims of statistical significance. For statistical analysis, we use randomization testing [Edg86], which is relatively free of assumptions about the true distributions of the samples involved. As it turns out, the differences in performance as suggested by the plots shown were all confirmed significant at a confidence level of 99.9%, except in those cases where the best two are clearly close (usually KNNGA(c,m) and KNNGA(c,m)(V)), in which case the difference in performance was inconclusive at this confidence level. Finally, it is worth pointing out again that all algorithms began with the same initial population. It sometimes appears from the graphs (e.g. see Figure 4.4) that the LEM(KNN) variants began with an advantage, however they did not. The LEM(KNN) variants tended to achieve very rapid improvement in fitness in the first few generations, which is horizontally compressed to almost nothing in the plots.

Table 4.1: Parameters settings for GA(c,m) and GA(m)

	Problem 1	Problem 2	Problem 3	Problem 4	Problem 5
Initialization	Randomly generate				
Representation	Real numbers				
Crossover	Uniform Crossover				
Crossover Probability	0.1				
Mutation	Random mutation				
Mutation step size	0.1	0.005	0.1	0.1	0.005
Mutation Probability	0.2	0.03	0.25	0.01	0.01
Parent selection	Binary Tournament Selection				
Survival selection	Generation selection				
Population size	100				
Number of offspring	100				
Termination condition	500 Evaluations	2000 E	15000 E	500000 E	600000 E

Table 4.2: Parameters settings for KNNGA and KNNGA(V)

Threshold	0.3
k value	5
Learning gap	1
Distance function	Euclidean distance
GA applied	GA(share all GA settings if applied)

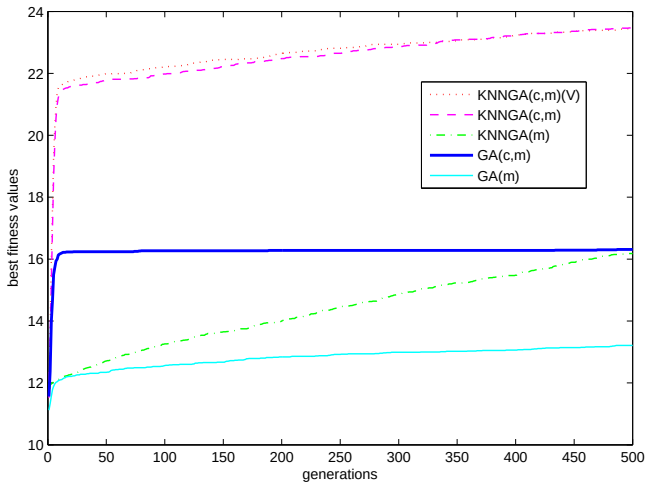


Figure 4.3: Results of running 5 algorithms to maximize problem 1

Figure 4.3 shows the results of running KNNGA(c,m), KNNGA(m), KNNGA(c,m)(V), GA(c,m) and GA(m) on problem 1. For both KNNGA and GA, mutation step size is 0.1, mutation rate is 0.2, and crossover is implemented with 100 parents and 10 children. For KNNGA, *k* is 5, *threshold* is 30%, and *learning gap* is 1. For Problem 1, all KNNGA variants outperform the GA variants. Within 500 generations, GA(m) only reaches the best fitness value of 13.21, and GA (c,m) reaches 16.31. In contrast, within the same number of generations, KNNGA(m) and KNNGA(c,m) achieve the best fitness values 16.18 and 23.47, respectively. KNNGA(c,m)(V) achieves best fitness value 23.0. It is interesting that the extra evaluation step of KNNGA(c,m)(V) does not yield any advantage in solution quality.

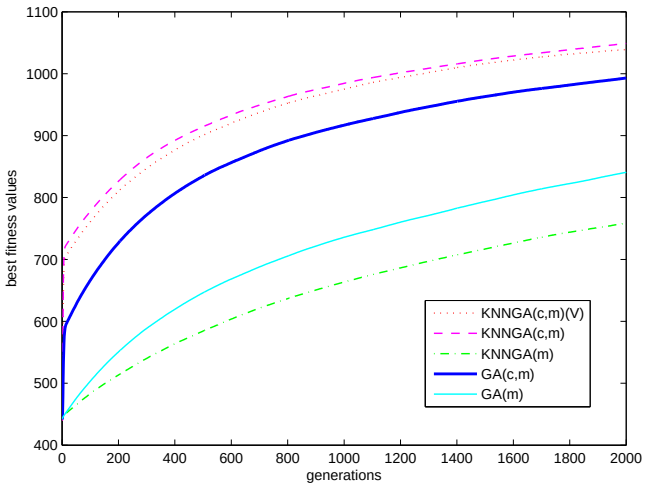


Figure 4.4: Results of running 5 algorithms to maximize problem 2

For this problem 2, the number of optimum increases as the number of variables scales up. Figure 4.4 shows the results of running the five KNNGAs and GAs algorithms on problem 2. For both KNNGA and GA algorithms, the mutation value is 0.005 due to the smaller variables range (-1.28, 1.28) and the mutation rate is 1/30. Crossover is implemented with 100 parents and 10 children. For KNNGA, *k* is 5, *threshold* is 30%, and *learning gap* is 1. Within 2000 generations, KNNGA(m) and KNNGA(c,m) reach the best fitness values 758.3 and 1048.7, GA(m) and GA(c,m) can reach 840.7 and 993.1, respectively. KNNGA(c,m)(V) achieves best fitness value 1039.2.

In this study for problem 3, we test KNNGA algorithm on the problem of parameter estimation for digital filter design. The fitness function was defined by equations specifying linear and nonlinear filters presented in [YS94]. For this minimization problem, the fitness landscape is not clear even for the low variables cases. Figures 4.5 and 4.6 show the results of running the five KNNGAs and GAs on problem 3. For both KNNGA and GA, the

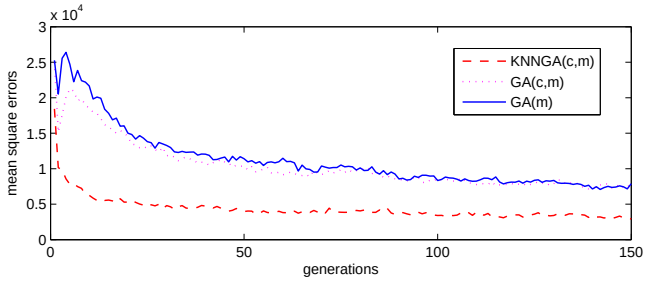


Figure 4.5: Results of running GA(m),GA(c,m),KNNGA(c,m) to minimize problem 3

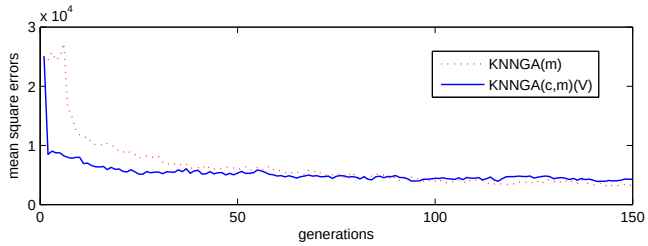


Figure 4.6: Results of running KNNGA(m),KNNGA(c,m)(V) to minimize problem 3

mutation value is 0.1 and the mutation rate is 1/4. Crossover is implemented with 100 parents and 10 children. For KNNGA, *k* is 5, *threshold* is 30%, and *learning gap* is 1. The reduction in mean square errors achieved by KNNGA over GA is evident. Within 150 generations, KNNGA(m) and KNNGA(c,m) reduce the mean square errors to 3426.3 and 2896.7, GA(m) and GA(c,m) can reduce the mean square error values to 7886.5 and 7588.2, respectively. KNNGA(c,m)(V) reaches mean square error to 4301.5.

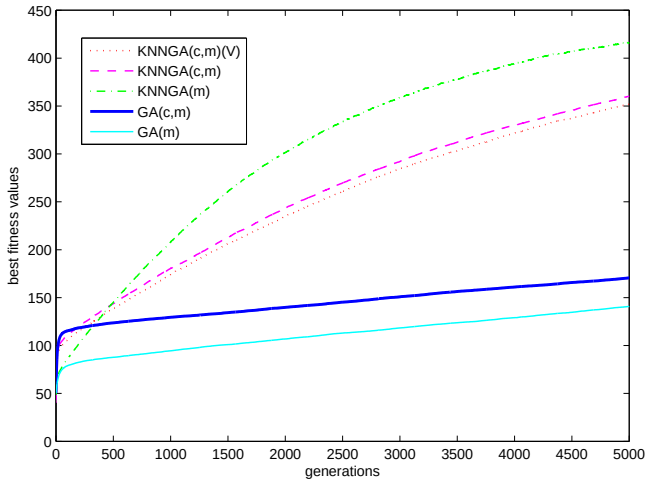


Figure 4.7: Results of running 5 algorithms to maximize problem 4

Problem 4 is the same problem as problem 1, but with more variables (100 variables). Figure 4.7 shows the results of running the five KNNGAs and GAs algorithms on problem 4. The mutation value is 0.1, the mutation rate is (1/100 = 0.01). Crossover is implemented

with 100 parents and 10 children. For KNNGA, *k* is 5, *threshold* is 30%, and *learning gap* is 5. The improvement achieved by KNNGA over GA is evident. Within 5000 generations. KNNGA(m) and KNNGA(c,m) reach the best fitness values 415.7 and 360.1, GA(m) and GA(c,m) reach the best fitness values 140.8 and 170.6, respectively. KNNGA(c,m)(V) achieves 351.87.

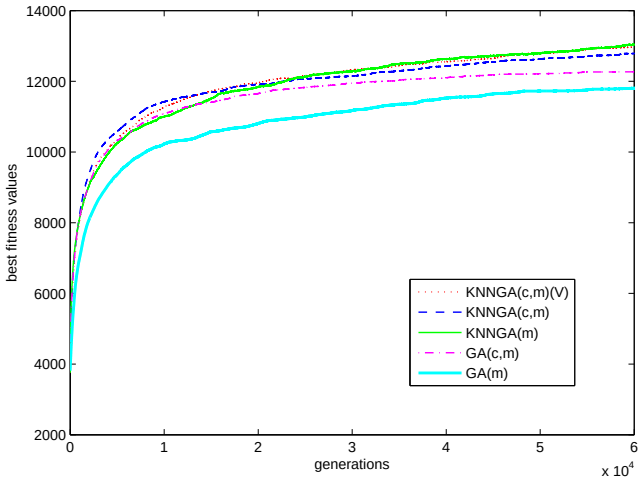


Figure 4.8: Results of running 5 algorithms to maximize problem 5

Problem 5 is the same problem with problem 2, but with 100 variables. The optima are approached with more difficulty than in problem 2. Figure 4.8 shows the results of running the five KNNGAs and GAs on problem 5. The mutation value is 0.05, the mutation rate is 0.01. Crossover is implemented with 100 parents and 10 children. For KNNGA, *k* is 5, *threshold* is 30%, and *learning gap* is 5. The improvement achieved by KNNGA over GA is evident within 60000 generations. KNNGA(m) and KNNGA(c,m) reach the best fitness values 13042.7 and 12787.3, GA(m) and GA(c, m) can only reach 11805.4 and 12269.7, respectively. KNNGA(c,m)(V) reaches 12971.5.

The results on this suite of problems show clear and very significant superiority for the KNN based LEM hybrid algorithms over the standard genetic algorithms. Either KNNGA(c,m) or KNNGA(m) were in top place on each problem, and always better than the non-KNN versions. The typical result is that the LEM(KNN) variants show a significant acceleration in fitness in the very early generations, followed by steady further improvement, leaving the ordinary versions far back in their wake. These findings reflect those of [MZ00, Mic00] and other recent LEM works that used more sophisticated learning mechanisms and interaction between the learner and the underlying genetic algorithms. Since the only application-specific demand of KNN is a suitable distance measure (in that way it is more generally applicable than many other learning mechanisms), it seems fair to say

that LEM methods using KNN are clearly recommended for trial in the case of large-scale optimization tasks in which savings in evaluation time are necessary. Far more work needs to be done to establish this properly, however even if LEM methods have not been tried on large-scale real-world problems so far, their promise has indeed been realized in [JCSW05].

Meanwhile, the performance of the 'with-verification' version of KNNGA was generally not significantly different from that of KNNGA(c,m), which suggests that the 'without' verification version is preferable, simply because it is faster. More interestingly, the lack of a major difference in performance between these two suggests that KNN's predictions, at least in the cases of the problems tested here, are generally not misleading. However, this could be problem-dependent, for other types of problems or more complex problems, we are not sure about the predication capacity of the KNN algorithms. We will leave the current conclusion about neglecting the 'with verification' case for the moment, and will do more investigation on this mechanism in our further research. Finally, it is clear that the differences between GA(m) and GA(c,m) were generally reflected in the differences between KNNGA(m) and KNNGA(c,m).

4.3 LEM(dwKNN) – dwKNNGA

We have investigated LEM(KNN), perhaps the simplest possible LEM variant algorithm. In LEM(KNN), the algorithm operates almost identically to the operation of its (ordinary) underlying evolutionary algorithm (EA), however, KNN is applied as a particular form of survival selection operator, which judges an individual according to the fitness of its neighbors. Also, we introduced the idea of 'verification' and therefore the resulting algorithm KNNGA(V). In this section, we reconsider the idea on 'verification' for LEM(KNN). To clarify this idea, we repeat our initial and main goal for developing LEM(KNN) algorithm, we want to apply KNN algorithm's prediction capacity to achieve the goal of reducing useless evaluations, (those evaluations spent on the individuals which disappear in the next generation in the normal evolution procedure). Based on this goal, we find that the LEM(KNN) algorithm cannot avoid some situations where too many unfit individuals survived the KNN learning method and occupy the next generation (we will show how we find this and the relative experiments later in this section). This is due to the fact that KNN as a learning method, like all other learning methods, cannot make predictions or classifications completely correct (100% accuracy) for the reasons discussed in Chapter 2. Namely, it is possible that certain individuals which survived (those predicted as 'good' by KNN)

are actually very unfit individuals. The percentage for these 'actual unfit' individuals in the following generation concerns, because if the percentage is too big, it will affect the optimization performance. It is due to this drawback that we introduce the idea of 'verification' into LEM(KNN).

However, 'verification' does not seem a good solution, at least from our experiments in the last section. And we think this could be for the following reasons, first, it is difficult to choose a suitable *survival fitness* (we simply choose the worst fitness value in the current population) for verification to make a comparison with. Worse and more importantly, 'verification' requires more evaluations, it requires the individuals that survived the KNN algorithm to be evaluated first before they are further tested with the *survival fitness* to decide whether they can be retained into the next generation or not. Such a procedure clearly needs more evaluations and makes 'verification' contradictory to the main goal of our LEM(KNN) algorithm which is to reduce the number of evaluations.

To overcome the drawbacks of the KNN algorithm and to find a good substitution for verification, we find a nice solution which is the distance-weight KNN (dwKNN), and it allows us to explore a more intelligent version of LEM(KNN) which uses distance-weighted KNN, the resulting algorithm is called LEM(dwKNN). In LEM(dwKNN), we replace straightforward KKN with distance-weighted KNN, which considers the classification of a given inquiry instance according not only to its neighbors but also their distances to the enquire instance. This generalization and refinement of the KNN algorithm have two main advantages, firstly, there is no essential difference between k and all training data due to the consideration of the distance contribution for each instance during the classifying. Secondly and most importantly, the classification for the enquire instance is not decided by the majority class information of its neighbors, but instead an estimated function value. It is this estimated function value in dwKNN which makes an essential different from the LEM(KNN) algorithm to the LEM(dwKNN) algorithm - not only can it be used to predict the newly generated individuals, but it can also make a direct comparison with the survival fitness value without evaluating these individuals. Namely, the estimated function does the same job as the verification, but without any extra expenses on fitness evaluations which are needed for the verification version.

In this section, we will explain how LEM(dwKNN) achieves the goal of LEM(KNN), reducing evaluation with KNN as survival selection, and avoids the drawback of LEM(KNN), unfit individuals are possibly able to survive into the next generation. And, more importantly, LEM(dwKNN) does these without the expenses on extra evaluations. Also, in the

following experiment part, we also try to unravel the question of although attempts to use simpler learning strategies within the LEM framework, such as LEM(KNN), have shown outperformance over the underlying evolutionary algorithm, whether these LEM(KNN) algorithms can seriously challenge the state-of-the-art optimization hybrid algorithms? Since LEM(KNN) produced significant benefits, we hypothesized that LEM(dwKNN) may show further benefits, and perhaps begin to rival state-of-the-art optimization methods such as CMAES, especially in terms of solution quality over reduced-evaluation number.

4.3.1 Distance-Weighted K Nearest Neighbors Algorithm

As we have introduced in Chapter 2 Section 2.3.3, Distance-Weighted Nearest K Neighbors Algorithm (dwKNN) is a refinement and generalization of the original KNN algorithm. The main refinement of dwKNN is that it weights the contribution of each of the k nearest neighbors according to their distances to the new query candidate during classification. Namely, distance-weight is to give greater influence to the closer neighbors, while reducing the influence of further neighbours. This refinement and generalization is worth and believed to bring more precise classification results. For example, when we consider a 3-NN (k = 3) algorithm for classifying the new query candidate x_q, if 2 of 3 nearest-neighbors of x_q are in the H-group, for the original KNN algorithm, x_q will be predicted as 'good' (individuals in H-group are labeled as 'good' for our optimization problems); however it could well be the case that the nearest neighbor to x_q out of this 3 neighbors is in the L-group, while the other two are a considerably greater distance away. In this case, it should be reasonable to consider more influence of this nearest neighbor by giving it more 'credit' (the weight), than the other two. Finally, the consideration of giving weights to neighbors according to their distances to query candidate in dwKNN may or may not change the classification results.

A way to take distance into account in the KNN algorithm is to directly assign a predicted or estimated function value to the new candidate, based upon the distance-weighted average of its neighbors. Namely, for the new candidate x_q, we may approximate its fitness as weighting the contribution of each of the k neighbors according to the inverse square of its distance from x_q, as in Equation 2.21 in Chapter 2, here, we list this equation again for completeness.

$$\hat{f}(x_q) = \frac{\sum_{i=1}^{k} w_i f(x_i)}{\sum_{i=1}^{k} w_i} \tag{4.11}$$

$$w_i = \frac{1}{d(x_q, x_i)^2} \tag{4.12}$$

4.3.2 dwKNNGA Algorithm

In the resulting LEM(dwKNN) algorithm, the GA algorithm hybridized with the dwKNN classification algorithm or LEM(KNN) extended with distance-weight, the idea is therefore simply to operate in the same way as LEM(KNN), but using instead the distance-weighted approach to predict the group membership of a new candidate. Namely, we replace straightforward KNN with dwKNN. DwKNN essentially predicts or estimates the fitness of a new candidate as a weighted sum of the fitness of its neighbors among already-evaluated solutions. A candidate is rejected before evaluating if its predicted or estimated fitness falls below a *survival fitness*. Note, if an individual survives this test, it means firstly that this individual is classified as 'good' class, second, its fitness value is 'estimated' to be better than the survival fitness without any real evaluations. The 'verification' idea uses the real fitness value to compare with the survival fitness, which requires real evaluating on that individual.

There are several possibilities for how we determine whether a candidate survives or not based on the *survival fitness*. We have not explored variations on this yet, but our default approach used is to allow survival if the estimated fitness is better than the worst in the current H-group.

Finally, given distance weighting, it is reasonable to allow all (or many) of the training examples to influence classification, since all influences are moderated by distance. Therefore, for the dwKNNGA algorithm, there is no need to indicate the parameter k, or it can be simply indicated as the size of the training data. This actually is another advantage of dwKNNGA algorithm, that is, it reduces the necessity of optimizing the algorithm parameter k, which could be important in the optimization performance and is not done in the LEM(KNN) algorithm. To ensure a replicable explication, detailed pseudo-code for our specific instantiation of LEM(dwKNN) is set out.

1. Set Parameters: Set values for *population size*, parameters for mutation (mutation probability, mutation step size), parameters for crossover (crossover probability) and set elite-preserve operator option. Set k (indicating the number of neighbors in dwKNN algorithm), *learning gap* (indicating the interval before one learning population is updated by another) and the *threshold*.

2. Generate initial population: Choose a method to create the initial population with *population size* and evaluate this population.

3. Derive extrema: Copy the *current population* as the *learning population* from which create the high fitness group (H-group) and low fitness group (L-group), according to fitness values and *threshold*. These two groups could have a joint set, or their union could be a subset of the whole population set or even equals to the whole population set. These two groups are stored for dwKNN algorithm.

4. Generate new generations: After reproducing the *current population*, apply the mutation, crossover operators to generate new individuals. Once a new offspring is generated (it is not evaluated and is not placed in the mating pool immediately), dwKNN is applied to calculate its estimated fitness value according to Equation 4.11, with regard to H-group and L-group (not the whole *learning population*). There will be two cases:

   i) if the estimated fitness is better than the survival fitness (the worst fitness in H-group), then it is evaluated and retained into the newly created population.

   ii) if the estimated fitness is worse than the survival fitness, then it is aborted.

   The generating procedure continues until this new population is filled with such newly generated individuals nearer to H-group. This finishes the generation of one generation.

5. Update H-group and L-group: When *learning gap* is reached, the *learning population* is replaced by the *current population*. The H-group and L-group are therefore recalculated according to the current *learning population* and the same *threshold*. The new H-group and L-group are stored for dwKNN.

6. Termination condition: The above steps 4 and 5 repeat until some termination conditions are satisfied:

   i) the optimal (if known) is reached; or

   ii) the maximum allowed generations number is reached; or

   iii) the best fitness value has not been improving for a certain number of generations.

The LEM(dwKNN) algorithm is modified based on the original LEM(KNN) algorithm, the pseudo-code for our specific instantiation of dwKNNGA is set out here as Algorithm 11.

In this section, we have developed the LEM(dwKNN) algorithm. The reason for developing LEM(dwKNN) is to overcome the potential drawbacks to LEM(KNN). As we mentioned before, the goal of LEM(KNN) is to achieve good optimization performance by reducing the unnecessary evaluations. To achieve this, LEM(KNN) uses KNN learning to predict the fitness of the new generated individual instead of evaluating them. Only those predicted as 'fit' will be evaluated later. However, the drawback of LEM(KNN) is that some unfit individuals are still able to survive due to the 'not perfect' KNN algorithm's prediction accuracy. To reduce the effect of this drawback, we first try to use the idea of 'verification' to verify each new individual which has survived the KNN classifier. However, this verification needs more evaluations to act, which is against our initial and main goal of developing LEM(KNN), which is to reduce the number of evaluations. As a better solution to this problem, we refine the initial LEM(KNN) algorithm as LEM(dwKNN), which maintains the KNN prediction capacity and also try to avoid the survivals of unfit individuals by calculating a predicted or estimated fitness for each individual, and the estimated fitness $\hat{f}$ is used to compare with the survival fitness. In this way, LEM(dwKNN) avoids the extra evaluations in KNNGA(V), and therefore in favor of the initial and main goal of LEM(KNN). Essentially, the estimated fitness value in LEM(dwKNN) has done two tasks together, that is, classifying the individual and verifying its quality. These two tasks are implemented in LEM(dwKNN) together, while in LEM(KNN)V, they are realized separately. We will prove the advantage of LEM(dwKNN) in the experiment part in the next section.

4.3.3 Experiments and Results

Test Functions

This section describes the experimental results derived from the comparison between GAs, LEM(KNN), LEM(dwKNN), and CMAES. First, we are always interested in the basic performance of LEM(KNN) vs GA, so that we could sample the degree to which the LEM framework could be successful when using the simplest possible learning scheme. Second, we are also interested in the performance of LEM(KNN) and LEM(dwKNN), therefore the distance-weights refinement techniques to overcome the drawbacks of LEM(KNN), if

Algorithm 11 pseudo-code for dwKNNGA

1: $population\_size = 100, i = 0$;
2: $generation\_number = 0, max\_generation\_number = 100$;
3: $k = 11, learning\_gap = 1, threshold = 0.3$;
4: Initialize a new population with *population_size*, evaluate it;
5: **repeat**
6: reproduce *current population*;
7: **if** (*generation_number*%*learning_gap* == 0) **then**
8: copy *current population* into *learning population*;
9: calculate the *H*-group and *L*-group according to *threshold*;
10: calculate the *survival_fitness*;
11: **end if**
12: **while** ($i <$ *population_size*) **do**
13: mutate a parent individual to generate a new child;
14: calculate the *estimated fitness* according to Equation 4.11 for this child;
15: **if** (the *estimated fitness* is better than *survival_fitness* in the *H*-group) **then**
16: evaluate and place it into the next generation;
17: *j*++;
18: **else**
19: child is aborted;
20: **end if**
21: apply crossover on two parent individuals in the *current population* to generate two new children;
22: for each of these two children, repeat steps 14-20;
23: **end while**
24: *generation_number*++;
25: **until** (*generation_number* == *max_generation_number*)

this refinement works, how well does it work? Finally, we also want to find out the performance of our LEM hybrid algorithms against state-of-art hybrid optimization algorithm like CMAES. We tested all of these algorithms using a collection of seven benchmark test functions widely used in the EC literature. Here, we name this set of functions as Test Suite 2 and the details of the test functions are described for completeness. Unless otherwise stated, in all cases, n (number of dimensions, genes) is 30.

1. The DeJong's function 3 is defined as:

$$f(x_1, \ldots, x_n) = \sum_{i=1}^{n} integer(x_i) \tag{4.13}$$

where $n = 30$ and $-5.12 \leq x_i \leq 5.12$. The global minimum of -150 is at the point $(x_1, \ldots, x_n) = (-5.12, \ldots, -5.12)$.

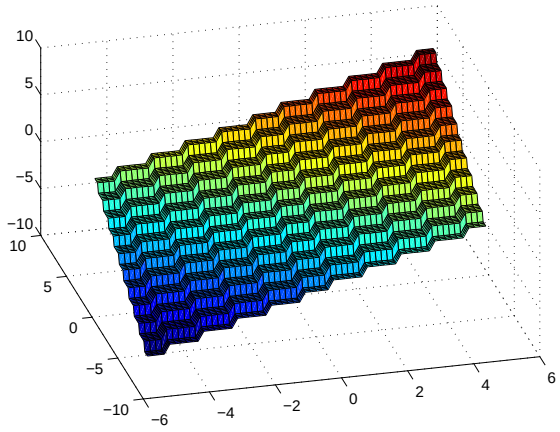


Figure 4.9: Landscape of the De Jong function 3 in 2 dimensions

2. The DeJong's function 4 is defined as:

$$f(x_1, \ldots, x_n) = \sum_{i=1}^{n} ix_i^4 + Gauss(0, 1) \tag{4.14}$$

where $n = 30$ and $-1.28 \leq x_i \leq 1.28$. The global minimum of zero is at the point $(x_1, \ldots, x_n) = (0, \ldots, 0)$.

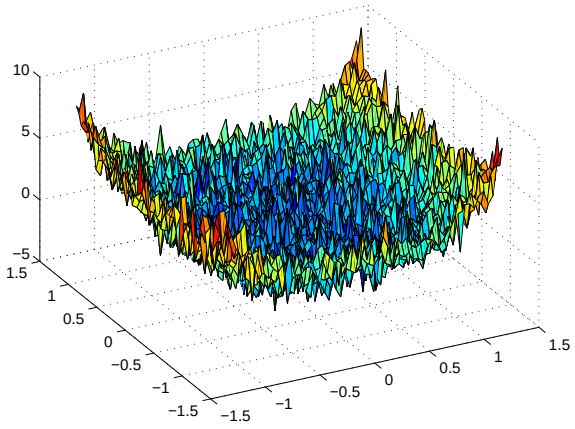


Figure 4.10: Landscape of the De Jong function 4 in 2 dimensions

3. The Rastrigin's function is defined as:

$$f(x_1, \ldots, x_n) = 10.0n + \sum_{i=1}^{n} (x_i^2 - 10.0\cos(2\pi x_i)) \tag{4.15}$$

where $n = 30$ and $-5.12 \leq x_i \leq 5.12$. The global minimum of zero is at the point $(x_1, \ldots, x_n) = (0, \ldots, 0)$.

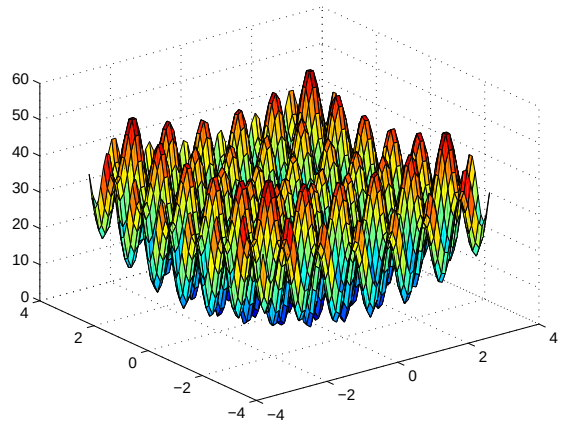


Figure 4.11: Landscape of the rastrigin function in 2 dimensions

4. The Griewank's function is defined as:

$$f(x_1, \ldots, x_n) = 1 + \sum_{i=1}^{n} \frac{x_i^2}{4000} - \prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}}\right) \tag{4.16}$$

where $n = 30$ and $-600 \leq x_i \leq 600$. The global minimum of zero is at the point $(x_1, \ldots, x_n) = (0, \ldots, 0)$.

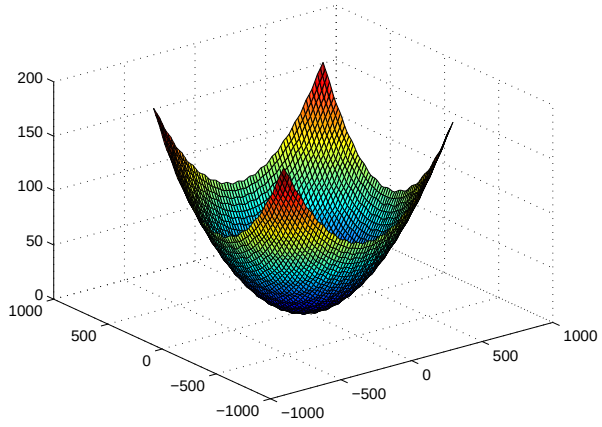


Figure 4.12: Landscape of the griewank function in 2 dimensions

5. The Rosenbrock's function is defined as:

$$f(x_1, \ldots, x_n) = \sum_{i=1}^{n-1} (100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2) \tag{4.17}$$

where $n = 30$ and $-2.048 \leq x_i \leq 2.048$. The global minimum of zero is at the point $(x_1, \ldots, x_n) = (1, \ldots, 1)$.

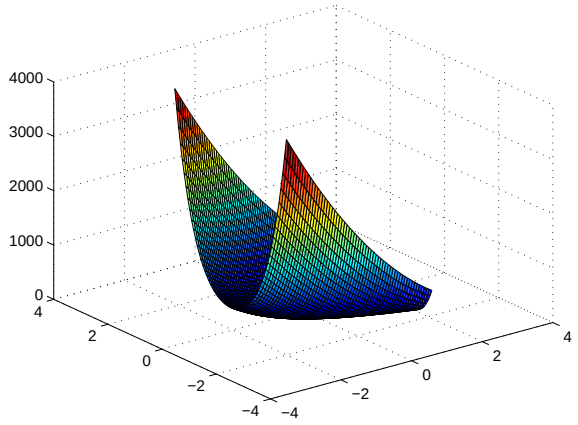


Figure 4.13: Landscape of the rosenbrock function in 2 dimensions

6. The Ackley's function is defined as:

$$f(x_1, \ldots, x_n) = 20 + e - 20exp\left(-0.2\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}\right) - exp\left(\frac{1}{n}\sum_{i=1}^{n}\cos(2\pi x_i)\right) \tag{4.18}$$

where $n = 30$ and $-30 \leq x_i \leq 30$. The global minimum of zero is at the point $(x_1, \ldots, x_n) = (0, \ldots, 0)$.

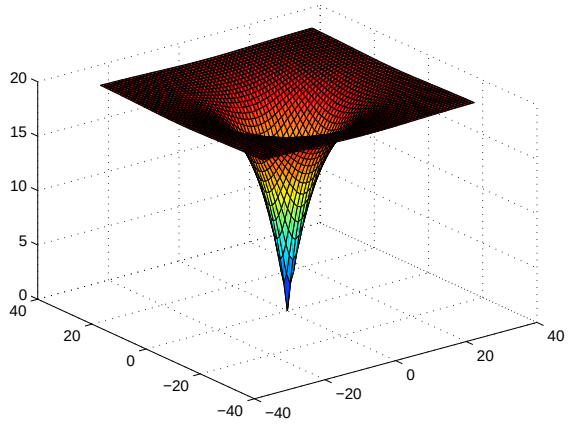


Figure 4.14: Landscape of the ackley function in 2 dimensions

7. The Schwefel's function is defined as:

$$f(x_1, \ldots, x_n) = 418.9829n + \sum_{i=1}^{n} x_i \sin\left(\sqrt{|x_i|}\right) \tag{4.19}$$

where $n = 30$ and $-500 \leq x_i \leq 500$. The global minimum of zero is at the point $(x_1, \ldots, x_n) = (420.9687, \ldots, 420.9687)$.

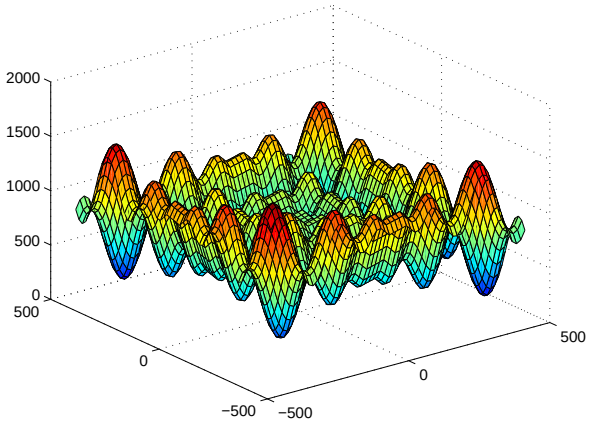


Figure 4.15: Landscape of the schwefel function in 2 dimensions

Parameters Settings

We have also improved our genetic algorithm implementations in this case over that used in Section 4.2, and made them more advanced for solving real-parameters optimization problems. For the new genetic algorithms developed here, we call them GA1 and GA2, we apply the steady-state model instead of the generational model, the steady-state model is

Table 4.3: Parameters settings for GA1 and GA2

Representation	Real numbers
Crossover	Blind crossover(BLX-0.5)
Crossover Probability	0.6
Mutation	Normal distribution mutation
Mutation step size	1 / 4 of the whole search range
Mutation Probability	GA1: 1.0 GA2: 1.0/*length of chromosome*
Parent selection	Binary Tournament Selection
Survival selection	Replace the worst of the population
Population size	100
Number of offspring	100
Initialization	Randomly generate for each run
Termination condition	After 10000 evaluations

Table 4.4: Parameters settings for LEM(KNN) and LEM(dwKNN)

Threshold	0.3
k value	11
Learning gap	1
Distance function	Euclidean distance
GA applied	GA2(share all GA2 settings if applied)

said to have better performance over the generational model; we also incorporate new real-parameters crossover operators developed in the GA literature into GA1 and GA2 here, the BLX crossover operator [ES93] introduced in Section 2.2.2 is applied with $\alpha = 0.5$, crossover probability of 0.6; we also apply the Gaussian perturbation mutation with a step-size of one quarter of a genes range, and apply to a new candidate with various probabilities, since in earlier experiments some advantage was sometimes shown for more frequent mutation, the mutation probability of 1.0 is used for GA1, and mutation probability of (1.0/*chromosome_length*) for GA2. Parameters for GAs are summarized in Table 4.3.

Both LEM(KNN) and LEM(dwKNN) used a threshold value of 0.3, $k = 11$, a learning gap of 1, with the underlying EA being GA2, as in Table 4.4.

The main parameters for CMAES(μ, λ) are μ, number of parent individuals, λ the num-

Table 4.5: Parameters settings for CMAES

Number of parents	50
Number of offspring	100
Mutation step size	1 / 4 of the whole search range
Initialization	Randomly generate for each run
Termination condition	After 10000 evaluations

Table 4.6: Means and standard deviation after 10 generations

Functions	GA(1.0)	GA(1.0/30.0)	LEM(KNN)	LEM(dwKNN)	CMAES
DeJong3	-78.3(4.06)	-90.26(4.74)	-98.07(4.88)	**-113.8**(4.80)	-94.55(3.88)
DeJong4	8.84(3.20)	3.47(2.02)	2.39(1.57)	**1.3**(1.23)	9.66(3.71)
Rastrigin	287.89(17.18)	232.08(20.3)	212.84(23.59)	**161.42**(20.05)	288.66(17.41)
Griewank	134.04(23.36)	76.26(13.62)	67.21(13.43)	**36.67**(10.88)	158.41(28.86)
Rosenbrock	1027.19(232.76)	583.57(148.74)	487.11(89.91)	**311.64**(84.76)	728.83(166.97)
Ackley	16.45(0.56)	14.50(0.71)	13.95(0.82)	**12.10**(34.72)	16.77(0.67)
Schwefel	9028.48(379.91)	7487.64(517.69)	6611.33(552.26)	**5312.65**(558.17)	9611.63(321.64)

ber of offspring, and the initial standard deviations σ. Here, we implement CMAES(50,100) and σ is set to one quarter of the range of each variable, as indicated in Table 4.5.

In all cases, the encoding was a vector of real-valued genes each encoding numbers within a specified interval, population size is 100, binary tournament selection and elitism (the next generation's population is always initialized with the best of the previous generation) are applied all the time. All experiments are repeated 100 times independently to provide sufficient evidence for claims of statistical significance.

Summary of Results

Table 4.6 to Table 4.9 summarises the results of 100 runs of each algorithm on each function, with means and standard deviations recorded at 10, 20, 50 and 100 generations (multiply by 100 for number of fitness evaluations). Meanwhile, Figure 4.16 to Figure 4.22 show the mean convergence curves for each algorithm on these test functions, respectively.

Table 4.7: Means and standard deviation after 20 generations

Functions	GA(1.0)	GA(1.0/30.0)	LEM(KNN)	LEM(dwKNN)	CMAES
DeJong3	-97.71(3.42)	-117.21(3.64)	-127.6(3.21)	**-141.36**(2.80)	-112.16(2.82)
DeJong4	1.35(0.99)	0.09(0.11)	**0.037**(0.051)	0.16(0.24)	0.070(0.12)
Rastrigin	248.74(17.74)	155.40(19.59)	134.73(16.62)	**87.74**(15.04)	228.39(14.78)
Griewank	55.2333(10.57)	21.65(4.74)	17.27(3.97)	**5.70**(2.27)	26.07(7.78)
Rosenbrock	436.4(89.7)	220.01(51.03)	185.23(41.61)	**133.99**(46.33)	152.95(37.93)
Ackley	13.17(0.76)	9.96(0.83)	9.05(0.82)	**6.33**(0.84)	10.36(1.06)
Schwefel	8706.26(381.46)	5686.49(533.59)	4641.64(536.89)	**3470.82**(461.21)	9564.46(291.09)

Table 4.8: Means and standard deviation after 50 generations

Functions	GA(1.0)	GA(1.0/30.0)	LEM(KNN)	LEM(dwKNN)	CMAES
DeJong3	-128.23(2.65)	-146.67(1.42)	-149.6(0.61)	**-149.93**(0.25)	-141.7(2.23)
DeJong4	8.63e-3(7.53e-3)	3.60e-3(4.86e-3)	2.52e-3(3.17e-3)	7.76e-3(1.93e-2)	**5.28e-4**(0.12)
Rastrigin	174.94(30.17)	84.36(14.51)	64.38(10.78)	**30.32**(7.05)	191.26(12.1)
Griewank	6.75(1.99)	2.04(0.4)	1.52(0.27)	**1.08**(0.081)	1.16(0.079)
Rosenbrock	116.74(27.65)	75.38(29.28)	69.29(30.96)	68.34(39.16)	**29.94**(0.73)
Ackley	6.601(0.71)	3.91(0.50)	3.02(0.50)	**2.22**(0.58)	2.32(0.409)
Schwefel	7380.06(1012.55)	3277.86(493.95)	2009.99(375.16)	**1685.07**(330.60)	9460.74(282.0)

Table 4.9: Means and standard deviation after 100 generations

Functions	GA(1.0)	GA(1.0/30.0)	LEM(KNN)	LEM(dwKNN)	CMAES
DeJong3	-145.60(2.12)	**-150**(0.0)	**-150**(0.0)	**-150**(0.0)	**-150**(0.0)
DeJong4	1.77e-3(1.73e-3)	8.8e-4(9.9e-4)	8.7e-4(0.9e-3)	1.53e-3(1.68e-3)	**1.52e-4**(1.6e-4)
Rastrigin	77.29(33.48)	44.75(8.997)	27.66(7.9)	**11.01**(2.79)	116.69(63.62)
Griewank	1.52(0.68)	0.95(0.11)	0.51(0.25)	0.68(0.23)	**0.029**(0.017)
Rosenbrock	55.17(20.87)	44.72(24.15)	45.68(27.14)	53.00(34.72)	**27.42**(0.53)
Ackley	2.59(0.83)	0.95(0.52)	0.65(0.57)	1.38(0.61)	**0.019**(0.0094)
Schwefel	5120.94(1078.51)	1540.0(289.45)	**1357.99**(336.98)	1446.53(301.62)	8948.33(805.8)

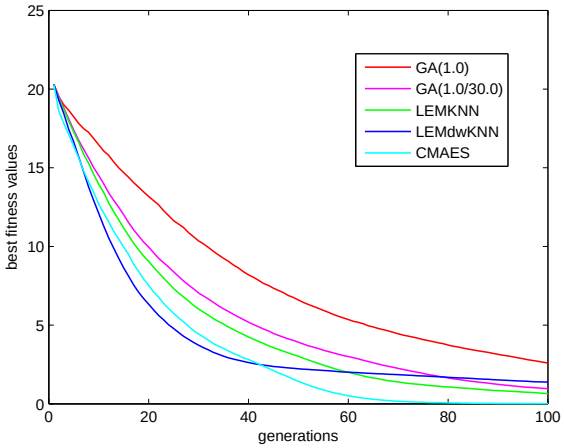


Figure 4.16: Results of running 5 algorithms on the DeJong3 problem

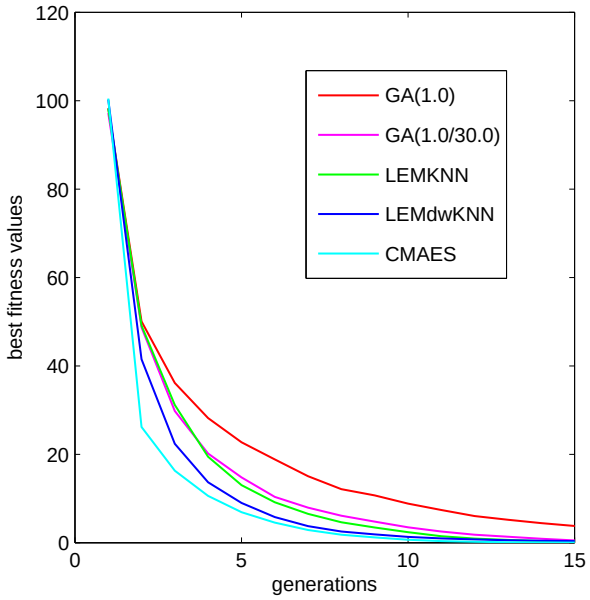


Figure 4.17: Results of running 5 algorithms on the DeJong4 problem

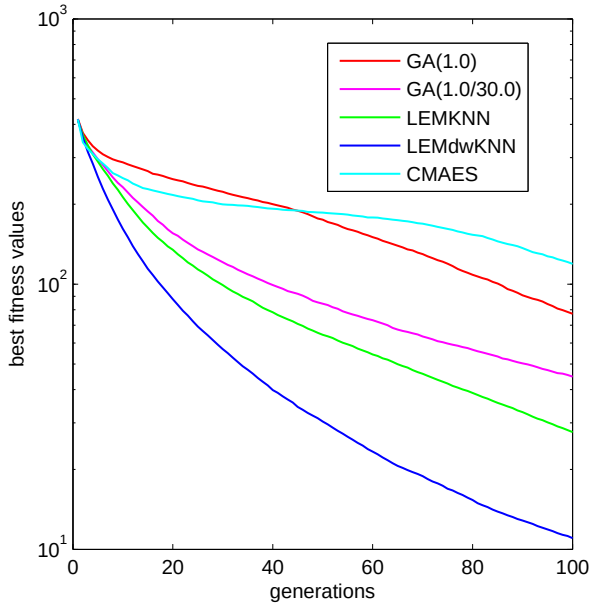


Figure 4.18: Results of running 5 algorithms on the Rastrigin problem

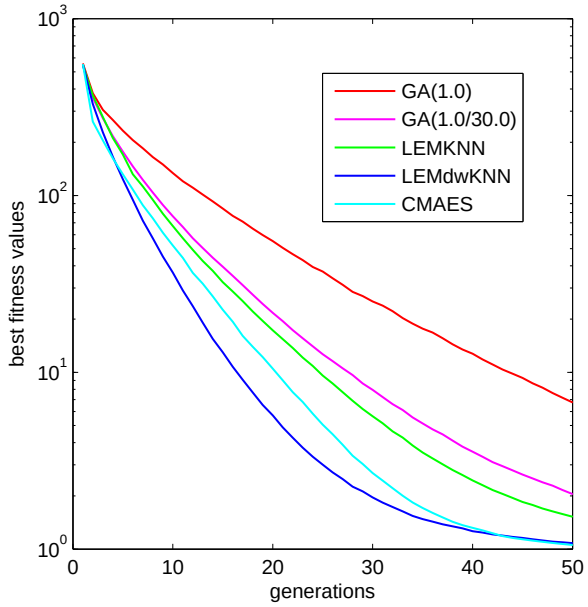


Figure 4.19: Results of running 5 algorithms on the Griewank problem

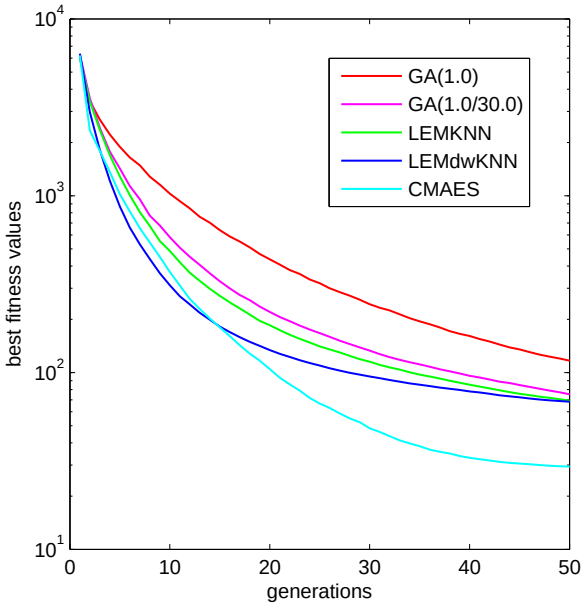


Figure 4.20: Results of running 5 algorithms on the Rosenbrock problem

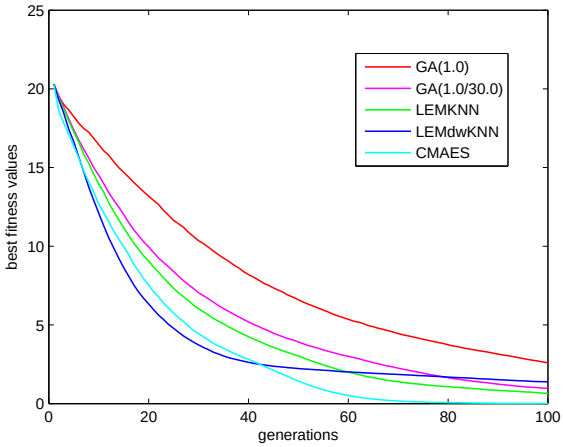


Figure 4.21: Results of running 5 algorithms on the Ackley problem

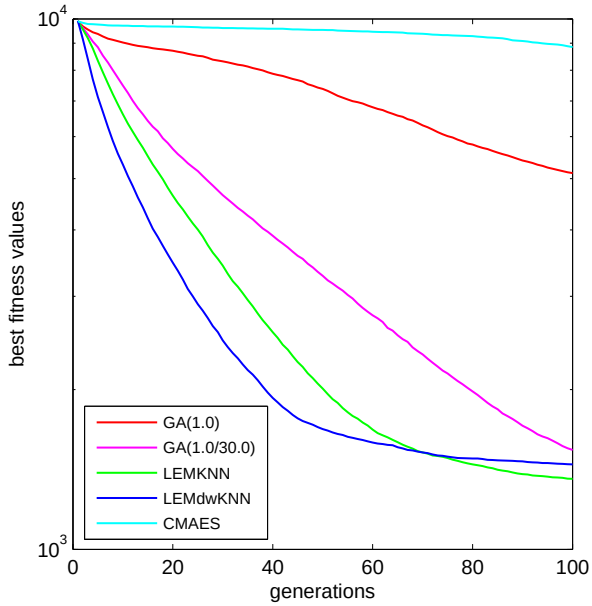


Figure 4.22: Results of running 5 algorithms on the Schwefel problem

First, for the GA1 and GA2 algorithms, the only difference between these two algorithms is the mutation probability. GA1 has a mutation probability of 1.0, while GA2 has a mutation probability of 1.0/(*length of chromosome*). As is clear from the tables, GA2 beats GA1 in all the functions for all generations. This has the clear implication that, when GA is used to solve real-parameter optimization problems, a lower mutation probability should be recommended. However, this recommendation is only derived from our test problems here, which may not represent all problem complexities and features.

Second, according to the algorithms performances, both LEM(KNN) and LEM(dwKNN) outperform the corresponding GA algorithms (GA2) in all problems for all generations. The performance improvements are more clear in the earlier generations, and are less clear in the later generations. This advantage in the early stages of optimization has shown the advantage and promise that LEM based hybrid algorithms can achieve important speedup over the standard evolutionary procedures in both the quality of solutions and speed of optimization by using the application of learning as a guide for the evolutionary search. This advantage of LEM(KNN) algorithms over the standard GA algorithm is crucial for expensive-evaluation problems.

Third, as shown in the results, the refined LEM(KNN) algorithm LEM(dwKNN) has more competitive performance than its original version. As we expected, LEM(dwKNN) is developed to overcome the potential drawbacks that LEM(KNN) may suffer, that is, the learning classifier KNN may misclassify some unfit individuals as survived individuals and

therefore lower the optimization performance. This speculation has been reflected in our experiment on test suite 2, as we can see, the LEM(dwKNN) algorithm significantly improves the performance of LEM(KNN) on six of these functions for all generations, this improvement over LEM(KNN) indicates that LEM(dwKNN) not only inherits the prediction capacity of LEM(KNN) by applying the distance-weight version of KNN algorithm but also improves the efficiency of LEM(KNN) by applying the estimated fitness values in order to verify the quality of the survived individuals without any extra evaluation expenses. The only exceptional result is De Jong's function 4 which contains a noisy component in the function definition, LEM(KNN) outperformance LEM(dwKNN) at the generation 20. This situation merits more investigation for our LEM(dwKNN) algorithm on problems with noisy input components.

Finally, LEM(dwKNN) performed significantly better than CMAES, and most of the other algorithms, on six of the seven test functions when measured at 1,000 evaluations (10th generation). The picture remains similar at 2,000 evaluations, and LEM(dwKNN) tends to maintain a strong advantage at 50 generations (5,000 evaluations) too, however it is overtaken between then and the 10,000 evaluations point, at which CMAES tends to be the dominant algorithm. The optimization performance derived by CMAES is quite reasonable. CMAES, at the beginning of optimization, tends to learn about the best mutation step sizes and will generally adapt to the best mutation step sizes, this procedure will take certain number of generations, that is why the performance of CMAES in the initial generations are not very promising. However, once the optimized mutation step sizes are found, CMAES rapidly converges to the optimum solutions either local or global, this is what makes the performances of CMAES so excellent, particularly in the quality of the solutions derived in the later generations.

Here we have found that LEM(dwKNN), which simply augments an EA with a pre-evaluation filter survival based on distance-weighted nearest neighbors, can drastically improve the performance of the underlying EA and LEM(KNN), and result in performance comparable or (usually) better than CMAES over limited number-of-evaluation regimes (up to around 5,000).

4.4 Concluding Discussion

We investigated a simple version of Michalski's LEM [Mic00] which used *k*-nearest-neighbors as the learning component, and had a straightforward interaction between the learning and

the GA, in which new individuals only entered the population if the majority of their *k* nearest neighbors in the current *learning population* were in the top good performance group. One contribution of this work is the LEM(KNN) algorithm, a simple instantiation of LEM with KNN, which very clearly trounces the corresponding GA in both speed and solution quality. The speed advantage is particularly impressive in general. Another contribution of this work is the fact that the LEM framework has been shown to work well in the context of using perhaps the simplest possible learning method. In other words, even the simplest learning approach hybridized with a normal GA in a very simple way can lead to considerable performance improvement over the GA alone. This is in contrast to published approaches which have either used AQ learners or C4.5. KNN is both simpler and more generic, suggesting that LEM(KNN) may be applied to large-scale optimization problems independently of the chromosome encoding required, needing only a suitable distance metric to be defined.

The advantage of LEM(KNN) is also our main purpose to develop LEM(KNN), that is, it saves evaluations by using the KNN learning method as the survival section method to predict the 'good' or 'bad' for the new generated individuals rather than evaluating them exactly. However, the disadvantage of LEM(KNN) is that the prediction based on neighbors could be flawed and therefore bring unfit individuals into the next generations. This is not what we expected. To overcome this drawback, we have tried two approaches. One is the development of a 'verification' version of LEM(KNN), which results in the KNNGA(V) algorithm, and the other is the application of the distance-weight KNN algorithm, which results in the LEM(dwKNN) algorithm. The KNNGA(V) algorithm is not very successful in overcoming the disadvantage of LEM(KNN), because it causes more actual evaluations to verify the new generated individuals in order to exclude the unfit individuals, which inevitably violate the advantage and main goal of developing LEM(KNN) based methods. On the contrary, the LEM(dwKNN) seems very suited to overcoming the drawbacks of LEM(KNN) and therefore is able to perform better than the LEM(KNN) algorithm. It judges the quality of the new generated individuals by calculating an estimated fitness according to the *k* nearest neighbors and their distances weights to a new individual, and verify this individual using this estimated fitness against a predefined survival fitness. In this way, LEM(dwKNN) maintains the prediction capacity of LEM(KNN) while excluding the unfit individuals without any extra evaluations.

We note that it has been difficult to compare our KNNGA with the specific LEM method used in [Mic00], since not all parameters are provided in the LEM paper. However, while

the improvements in performance over the GA are similarly vast, it does seem that the LEM(AQ) implementation reported there provides superior results to KNNGA. Two clear explanations for this are available: the simplicity of KNN compared with the relative sophistication of AQ, and the differences in the way that learning influences the evolution in the two cases. We have deliberately opted for the simplest possible approaches in both cases here, and therefore can show that the bulk of the improvement afforded by the LEM framework is still present in these circumstances, suggesting that the specific choice of learning method and the design of the learning/evolution interaction provide opportunities for further improvement and refinement, rather than being crucial to being able to show superior performance at all.

Continued research on instantiations and variations of the LEM framework are clearly warranted. Lines of work that we expect to explore are: the relationship between the problem landscape and the choice of learning method; the interaction between the learning method and the learning gap, and the use of more than one learning method (with perhaps adaptive techniques to choose between them at different points). Further hybridization and comparisons with EDA style approaches, and EDA/search hybrids are also warranted. Importantly, however, LEM-based approaches would seem to have much to offer for the speedup of large scale optimization, and we recommend its application to real-world problems of that nature. A specific issue with some possible LEM variants, including the KNN case in many dimensions, is that the learning method itself may take up a considerable amount of time. This is why we recommend LEM-based research in particular for problems where this 'learning time' remains trivial in comparison to the other aspects of the search, either because a single fitness evaluation takes significant time, or because very many fitness evaluations are needed, or both. In conclusion, there remains a wealth of solutions to be found in the combination of optimization and learning, and we believe that helpful insight is easiest to grasp by exploring simple combinations, especially when such simple combinations perform so well.

Chapter 5

LEM Instantiated with Entropy-Based Discretization

5.1 Overview

In the previous chapter, we developed the LEM(KNN) and its variant algorithms. In LEM(KNN), the learning method KNN is used as the survival selection or filter to predict the newly generated individuals. The development of LEM(KNN) and its variant algorithms have investigated two important aspects of the LEM framework. One is the interaction between learning and evolution, the other is the flexibility of the LEM framework. In this chapter, we continue our investigation of the original LEM framework, this time, we are interested in the flexibility of the LEM framework. Namely, to explore how to replace AQ learning with another learning mechanism and explore the performance of the resulting LEM. As the main LEM instance algorithm, LEM(AQ) utilizes the AQ learning method to generate explicit hypotheses describing the search space, then the new individuals are instantiated according to these hypotheses or rules. This generating-and-instantiating method plays an important role in the LEM framework. Our investigation of the LEM framework begins from the flexibility and try to find new methods to fulfill this generating-and-instantiating method within the LEM framework, and we ask ourselves the following questions as our research motivations.

When we replace the AQ learning algorithm with the other well-known learning methods, will the resulting new LEM instance algorithms perform equally well with the LEM(AQ) algorithm? Alternatively, whether the equally significant performance improvements can still be obtained with a simpler learning method rather than the complex AQ learning algo-

rithm? If yes, the flexibility of the LEM framework is verified. If not, what will the performance of the new version of the LEM instance algorithm be? Based on these questions, our investigations of replacing the AQ algorithm begin with simpler algorithms, this is not only because of the fact that the implementation of learning methods could be expensive, (therefore the cheaper implementations are always preferred to more complex implementations), but also because beginning with simple ideas is a good research strategy. If the simpler methods result in bad algorithm performances due to their simplicities, then we can consider adding more complex elements to them or using new complex algorithms completely. Starting from simpler ideas is also useful in order to analyze and clarify the problems of the current 'simpler' version of our algorithms, and therefore can gain more understanding and inspiration to develop more advanced algorithms with improved performance. We have just experienced this procedure in our study of LEM(KNN) and its variant algorithms.

Before introducing the new instance LEM algorithm, we make an observation about the LEM(AQ) algorithm. Basically, there are two learning tasks in learning algorithms, they are classification and prediction tasks. However, in the LEM(AQ) method, in fact, only the classification task is used, the prediction task is never used. This is because the LEM algorithm, and also the evolutionary computation algorithms, are used to optimize the solutions and to find better solutions. This is quite different from the machine learning algorithms, where the task is to predict the classification of future unseen instances based on the currently available training instances. The prediction capacity or accuracy of the classification for future unseen instances are expected. Therefore, the observation is that the LEM framework based hybrid optimization algorithms only need a 'partial' learning method which only needs to learn from the current training instances, but does not need to make any prediction about any future unseen instances. Namely, what we need in this LEM framework is in fact only a 'classifier' rather than a 'predictor'. LEM distinguishes the current training data, indicating the high-performance districts from the low-performance districts, and the rules or hypothesis will be used to generate new promising individuals, rather than being used to make more accurate predictions for future unseen data.

Based on these motivations and observations, in this chapter we investigate a new LEM hybrid optimization algorithm which also uses a very simple learning strategy, but of a very different kind. This learning method is called Entropy-Based Discretization (ED) and we call the resulting algorithm, LEM Instantiated with ED algorithm (LEM(ED)) [SC09], where ED is used to guide the generating of new individuals, not to filter generated individuals. Entropy is an important concept and widely used technique in data mining and

machine learning, here, we use it as the discretization measurement method. ED simply finds a partition or 'cut point' for each given variable's range, these 'cut points' are expected to be the best points on each variable (attribute) at which to classify the training data. Through the development of LEM(ED), we want to find out whether the above development strategy is feasible, and also more importantly the performance of this resulting LEM(ED) algorithm. We test LEM(ED) on a suite of function optimization problems and compare its performance with other optimization algorithms.

In the remainder, we continue as follows. Section 5.2 introduces the Entropy-Based Discretization method. Section 5.3 provides complete detail of our LEM(ED) algorithm. Section 5.4 presents the experiments and results. Finally, we conclude and discuss in Section 5.5.

5.2 Entropy-Based Discretization

5.2.1 Discretization Techniques

Discretization is a process of quantizing continuous attributes. It is an important technique widely applied in data mining, machine learning and knowledge discovery. Discretized intervals of continuous numbers are able to represent, specify, and comprehend the knowledge domain more precisely than the continuous values. Also, the discretized features are easier to understand, use, and explain to users of any level. Discretization can also be used to reduce the complexity of the original continuous data set. In fact, many machine learning tasks and induction algorithms require discretization as a prior condition. For example, the rule-based learning algorithms require discretized input data, rules with discretized value are more compact and understandable with higher predictive accuracy.

There exist many discretization techniques in literature, they can be classified according to many standards. One important standard is the class information of the data need to be classified, if the data contain class information then the discretization method is called *supervised discretization*, otherwise, it is *unsupervised discretization*. The unsupervised methods simply divide the whole continuous number range into intervals with equal ranges, this may not achieve good results simply because the data is distributed in a very complex manner. The supervised methods utilize the class information to find the better or fitter intervals divided in the continuous range. Discretization techniques can also be grouped as top-down and bottom-up methods, also termed as *splitting* and *merging* methods. Top-down

(splitting) methods start with an empty list of cut-points and keep on adding new ones to the list by splitting intervals as the discretization progresses. Meanwhile, bottom-up (merging) methods start with the complete list of all the continuous values of the feature as cut-points and remove some of them by merging intervals as the discretization progresses.

In spite of these standards which characterize the discretization methods, a general discretization procedure is common to many concrete discretization methods. First, for each attribute of the data set, the continuous values are sorted in either descending or ascending order, this can make all the numerical values become candidates for 'cut-points' or 'merge-points' in a systematic way. After sorting, the next step is to find the best cut-point to split a range of continuous values or the best pair of adjacent intervals to merge. One typical evaluation function is then used to determine the correlation of a split or a merge with the class information of the data set. Examples of such evaluation methods include entropy based measurements and statistical measurements. When the evaluation method is applied, for the splitting method, the best cut-point is chosen, and it splits the range of continuous values into two partitions; for the merging method, all the adjacent intervals are evaluated to find the best pair of intervals to merge in each iteration. For both cases, discretization continues until a stopping criterion is satisfied.

5.2.2 Entropy-Based Discretization

Based on the above discussion, we are ready now to talk about Entropy-Based Discretization. Entropy is one of the most widely used discretization measurement in the literature, in Chapter 2, we give the definition of entropy in binary situation, here we define entropy in its more general form in information theory. Entropy for a variable X is defined as:

Definition

$$E(X) = -\sum_{x} p_x \log p_x \tag{5.1}$$

where x represents all the possible values of X and p_x is its estimated probability of occurring. It is the average amount of information for each value x. Information is high for less probable events and low otherwise, hence entropy E is highest when each value is equally probable, i.e., $p_{x_i} = p_{x_j}$ for all i, j; it is the lowest when there is one value with the appearing probability $p_x = 0$, and all the other values with probability 1.

So, from the definition of entropy, we can see that it is a method to measure the purity of a set of data. Alternatively, a low entropy value results in a more efficient classification

of two classes of instances. Finally, It belongs to the supervised discretization methods. There are two well-known discretization methods which apply entropy as the measurement method. First, in the ID3 [Qui86] and C4.5 [Qui93] decision tree construction algorithms, if the training data is represented as real numbers, the discretization method will apply entropy as a measure to discretize the real attributes for the training data, and the cut-point with minimum entropy for each attribute will be selected as the discretization point. That is, the point by which the real attribute is split into two intervals which distinguish the current training data most efficiently, is selected by the discretization method of ID3 or C4.5. In this way, the real-number data set containing continuous attributes is discretized to be able to avoid creating too many branches for one node. Indeed, ID3 considers each value (point) of each attribute as a potential cut-point, and calculates their entropy values, the point with the minimal entropy value is chosen as the cut-point to divide this feature into two intervals. Namely, it binaries a range at every splitting.

The second discretization method which applies entropy measurement is the D2 algorithm [Cat91]. Like ID3, it applies entropy to find a potential cut-point to split a range of continuous values into two intervals. Unlike ID3, which binaries a range of values while building up the decision tree, D2 is a static method that discretizes the whole continuous value range for all variables. Instead of finding only one cut-point for each variable, D2 recursively binaries ranges or subranges until a stopping criterion is met. The discretized data is then used for any learning algorithms, not only for ID3, therefore, D2 is a successor of ID3 discretization. However, the stop criterion for D2 could be difficult to indicate, for example, if the stop criterion contains that, the number of intervals is beyond a pre-fixed number, then the best or suitable pre-fixed number is difficult to define.

The ED discretization method in LEM(ED) hybrid algorithm is also known as Entropy-Based Supervised Binary Discretization. It is based on both the ideas of ID3 discretization and D2 algorithms, it discretizes all attributes with entropy measurement into two intervals all the time in a supervised way, which is similar to ID3 discretization; the discretization is conducted before any learning happens, in this sense, it is a static method and is similar to D2. Both ED and D2 are static, they discretize all the attributes before learning. ED is different to D2, ED binaries the interval, while D2 iteratively binaries the intervals and following subintervals. ED is also different from ID3 discretization, which acts dynamically during the construction of the decision tree. To put it more precisely, ED is a static version of ID3 discretization, and is a simplified or binary version of D2.

We have discussed three concrete discretization methods, they are all supervised and splitting discretization methods. There are still many other discretization methods available in the literature, such as merging based methods, the majority of which are still supervised, but based on merging two adjacent intervals rather than splitting. Also the AQ anchor adaptive discretization method [MC01], which is also a supervised and splitting method, but the measurement for splitting is based on accuracy rather than entropy.

5.3 LEM with Entropy-Based Discretization – LEM(ED)

After the discussion about the discretization methods, we will now describe the LEM(ED) algorithm [SC09], which follows the general LEM framework. Therefore the main development idea behind LEM(ED) is to use ED as the generating hypothesis method, and design a corresponding instantiating method for the ED learning output. That is, find a good concrete solution for the generating-and-instantiating method in the LEM framework. As LEM(ED) is a new LEM instance algorithm, many aspects and structures are inspired by the the LEM(AQ) algorithm, the description of LEM(ED) is straightforward.

5.3.1 The LEM(ED) Algorithm

As with LEM(AQ), LEM(ED) divides the *current population* into high-performance (H-group) and low-performance (L-group) groups according to their fitness values and a given *threshold*. This is then saved as the *learning population*. Individuals of the H-group and L-group in the *learning population* form the training examples used by the 'learning' algorithm ED. Again, there are many ways to generate the training data set as long as they can guarantee enough training examples are derived. After the generation of training data, ED is applied on each variable (dimension) of the data set to discretize each real number individual. ED simply finds the cut-points for every dimension using the entropy measurement according to the class information, the point with the lowest entropy is selected as the cut point for each variable, therefore the learned output is a set of interval pairs for all variables. For example, intervals $< min_i, cut\_point_i >$ and $< cut\_point_i, max_i >$ are the output for the ith variable, where cut_point_i is the splitting point with minimum entropy, and min_i and max_i are the smallest and biggest values on ith variable, respectively. The principle that entropy based discretization is employed as the learning component in the LEM framework is that, first, the class information is available for the individuals in the training data set, and then we want to find out in each variable, which discretized interval is contributing the most

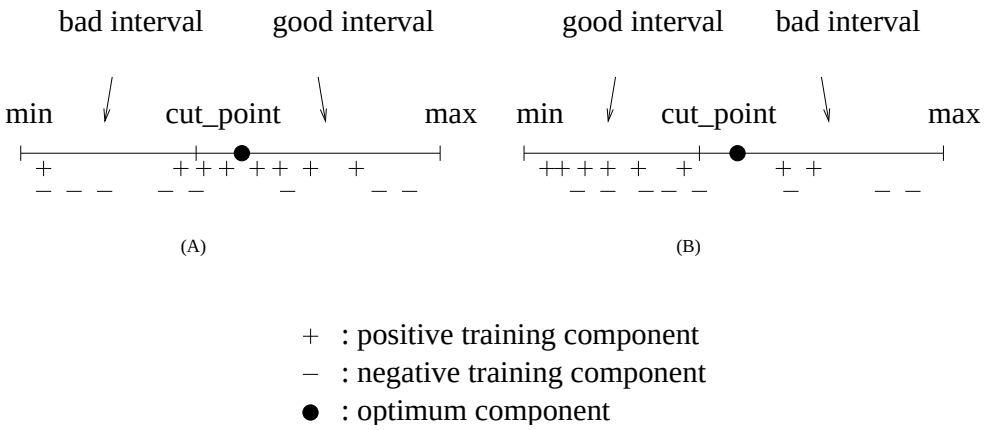


Figure 5.1: The correct and incorrect labellings for two intervals by LEM(ED).

to the current promising individuals in the training data. If these intervals can be indicated, they will be used to guide the generation of new individuals for following generations. To be able to realize this principle, when the intervals are discretized, we need to further label them to select the interval used to guide the generation of new individuals.

When all variables are discretized, each interval on each variable is explicitly labeled as a *good* or *bad* interval according to the class information. There are many methods to implement this labeling step, for example, we can simply count, if the majority of H-group individuals with *i*th variable values are lying in one interval, then label this interval as a *good* interval and another interval as a *bad* interval. As Figure 5.1(A) shows, after labeling, the pair of intervals $< min_i, cut\_point_i >$ and $< cut\_point_i, max_i >$ now indicates that individuals whose *i*th values are from different intervals very probably belong to different classes. The output intervals on each dimension in every generation are key concepts in LEM(ED), the output interval can be used in order to generate the new individuals for next generation or be used as guides for following the evolution search procedure. One important issue arises due to the quality of training data, the learning algorithms ED and the labeling procedure could produce intervals incorrectly. That is, the output *good* interval on *i*th dimension may also include many gene values from the L-group individuals, or simply miss out the optimum component for the *i*th dimension. This case can be caused by the poorly distributed training data, and also the strong interaction between dimensions in deciding the problem landscape can affect the accuracy of the output intervals. A simple bad distribution of the training data on *i*th dimension makes the *i*th output intervals pair wrong, as shown in Figure 5.1(B).

Labeling each interval according to category information is a simple but reasonable method to apply. First, according to the training data generated with the threshold, the

class information for each individual is available. Second, entropy can be used as the measurement of purity of the training data on each variable according to the class information, and the point which best distinguishes the classes can be found through this measurement. Finally, simply counting the current good individuals lying in each interval and selecting the dominating interval can reflect the current distribution of good individuals, although the distribution of bad individuals is ignored.

However, as we may notice the ED method has some similarities and differences with the well-known decision tree construction algorithm, where the relationships between intervals (domains) and class information are not explicitly indicated, they are implicitly involved in each path of the tree, or can be more clear from the translated rules. Here, we discuss the main differences between ED and the decision tree with a focus on the output forms. There are several important differences between the outputs of a decision tree and ED:

1. The output of ED is all variables; where for the decision tree, the output is a subset of all variables, some variables with high entropy are not used.

2. If translated into rules, the output of ED is a single rule which has the same number of conditions as it has variables; the output of a decision tree can be considered as a rule set.

3. Because the ED output is a single rule, there is exactly one interval (domain) on each variable for each category; for decision tree, there are possibly more than one intervals (domains) on each variable for each category.

When the intervals for all variables are labeled, LEM(ED) begins the instantiation procedure. The new genes of new individuals for the next generation are now generated according to the *good* intervals for variables. This can be achieved in a number of ways, for example, new gene values are generated only from the *good* interval in a random, or ordered way; or new gene values are generated from both the *good* and *bad* intervals, but with a high probability for the *good* interval and a low probability for the *bad* interval. The former can be considered as a greedy method, and the latter a non-greedy method. A typical instantiation procedure is illustrated in Figure 5.2.

When the new population is created, ED is applied again on the current population to generate new intervals updating the old intervals for every variable (dimension). Such a procedure is repeated until the learning mode termination condition is met. Such termination

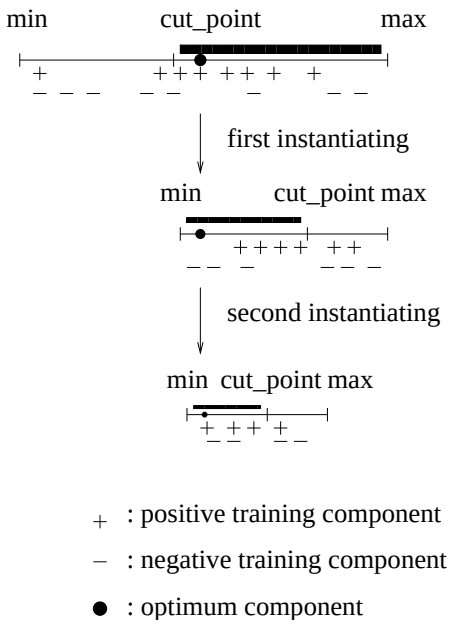


Figure 5.2: Instantiation procedure by LEM(ED).

conditions could include there is no fitness improvements for a certain number of generations; or a fixed number of generations is reached. When the learning mode is finished, there are a number of possible options to choose as the following algorithm components. For example, we can simply apply a normal genetic algorithm to finish the optimization procedure; or, we can restrict the evolutionary procedure within the learning intervals; or we begin the learning procedure again after a certain number of evolutionary procedures. We will develop these ideas into a series of LEM(ED) algorithms and will discuss these cases in the following sections. For now, we give the description of the general LEM(ED) algorithm and try to ensure a replicable explication with pseudo-code. 'Overview' pseudo-code for LEM(ED) algorithms is as follows:

1. Set parameters: Set values for *population size*, mutation probability, crossover probability, *learning threshold* and set elite-preserve operator option.

2. Generate initial population: Choose a method to create the initial population with *population size* and evaluate this population.

3. Begin learning mode :

   (a) Derive extrema: Copy the *current population* as the *learning population*, from which the high fitness group (H-group) and low fitness group (L-group) are created according to fitness values and *threshold*. These two groups could have a joint set, or their union could be a subset of the whole population set or even

equal to the whole population set. These two groups are stored as positive and negative training data for the learning algorithm.

(b) Apply ED on training data: For each variable (dimension), consider each value (point) as the potential cut-point and calculate the entropy values for all of these points and choose the point with the best entropy as the cut_point for this dimension. This point is the best point on this dimension in classifying the training data. The output of this step on each dimension is two intervals with the form $< min, cut\_point >$, and $< cut\_point, max >$.

(c) Label the learned intervals: For the output two intervals on each dimension, label them as *good* and *bad* intervals. This can be done by simply counting the gene values in each interval. If one interval has more values belonging to H-group individuals than the other interval does, then this interval is labeled as *good* interval, and the other is labeled as a *bad* interval. This procedure is repeated until all variables are labeled.

(d) Instantiate new individuals: After the discretization and labeling procedures, the new gene values of new individuals for next generation are generated from the good intervals in each dimension. There are many methods to do this, the simplest method is random generation.

(e) Update H-group and L-group: When the new population is generated, the H-group and L-group are regained by ED applied on the new population, and the *learning threshold* will be used again.

(f) Termination condition for learning: The above procedures will repeat to generate new individuals until there is no improvement which can be achieved for the best fitness value for a certain number of generations. When this condition is met, LEM(ED) switches to evolution mode.

4. Begin evolution mode: The evolutionary procedure can have a number of forms. It could be a normal genetic algorithm, or a genetic algorithm with specific search according to the output intervals information.

5. Termination condition for LEM(ED): LEM(ED) will stop, if any of the following termination conditions is satisfied:

   (a) the optimal (if known) is reached; or

(b) the maximum allowed number of generations is reached; or

(c) the best fitness value has not been improving for a certain number of generations.

The pseudo-code for our specific instantiation of LEM(ED) is set out here as Algorithm 12.

Algorithm 12 pseudo code for LEM(ED)

```
1: Set population_size, max_generation_number, threshold, generation_number;
2: Initialize a new population with population_size;
3: Evaluate current population;
4: repeat
5:    while (Termination condition for learning is not met) do
6:       Copy current population into learning population;
7:       Calculate the H-group and L-group according to fitness values and threshold;
8:       Apply ED on the training data to generate the learned intervals for each dimension;
9:       Instantiate the new current population for next generation;
10:      generation_number++;
11:   end while
12:   Select parents on current population;
13:   Crossover current population;
14:   Mutate current population; {LEM(ED)1}
15:   Mutate current population according to the output intervals; {LEM(ED)2}
16:   generation_number++;
17: until (generation_number==max_generation_number)
```

5.3.2 LEM(ED) Variant Algorithms

As noticed in Algorithm 12, there are two variant algorithms for a general LEM(ED) algorithm, we explain the motivations and differences behind them in this section.

LEM(ED)1 is our initial development of LEM(ED). It begins by applying ED as learning method on the training data, and the following generations are generated according to the output intervals, it stops learning when there are no further improvements for the best fitness for a certain number of generations. LEM(ED)1 then switches to a normal evolu-

tionary procedure to finish the optimization procedure, therefore, it does not include a loop of learning and evolution procedures. Due to the fact that LEM(ED)1 uses a normal evolution procedure to find the optimum, its performance in the ending part is very much like a normal evolutionary algorithm. We will show LEM(ED)1's performance in our experiment section.

LEM(ED)2 tries to achieve better performance in the ending part than LEM(ED)1. Thanks to the existence of the intervals as the output of learning, a natural idea is to utilize these intervals to guide or restrict the range of the following evolution search procedure. There is an important aspect about this algorithm, that is, if the learned intervals are not correct (the *good* interval is corresponding to the L-group individuals), then the following evolutionary procedure will be misled on those variables. We expect LEM(ED)2 to be able to converge more quickly to the optimum than a normal evolution procedure does in the ending part for some optimization problems.

5.4 Experiments and Results

This section describes the experiment and results of the LEM(ED) algorithms in comparison with the corresponding conventional GA and our first LEM hybrid algorithms, the LEM(KNN) and LEM(dwKNN) algorithms developed in Chapter 4. In addition to this comparison, we are also interested in comparing LEM(ED) with the state-of-art optimization algorithm, we choose the Covariance Matrix Adaptation Evolution Strategy (CMAES)[AH05b], [AH05a] again. The test function used here is the 'Test Suite 2' used in Chapter 4 as well, we refer to Section 4.3 for complete definitions.

5.4.1 Parameters Settings

We give the details of parameters settings for LEM(ED)1 and LEM(ED)2 algorithms as listed in Table 5.1. For algorithms, GA1, GA2, LEM(KNN), LEM(dwKNN) and CMAES, we refer to the settings in Chapter 4 Section 4.3. For all of our GAs and LEM hybrid algorithms, the encoding was a vector of real-valued genes each encoding numbers within a specified interval. We used binary tournament selection, elitism (the next generation's population is always initialized with the best of the previous generation). We apply BLX crossover with $\alpha = 0.5$, crossover probability 0.6, normal distribution mutation with mutation probability 1.0 or 1/(*size of chromosome*).

Table 5.1: Parameters settings for LEM(ED1) and LEM(ED2)

Threshold	0.3
Learning gap	1
Discretization method	Entropy based binary discretization
Instantiation method	Instantiate intervals with probabilities (80%, 20%)
GA applied	LEM(ED1) : GA2 LEM(ED2) : GA2 with very small mutation step size

Table 5.2: Means and standard deviation after 10 generations

Functions	GA(1.0)	GA(1.0/30.0)	LEM(KNN)	LEM(dwKNN)	LEM(ED)1	LEM(ED)2	CMAES
DeJong3	-78.3(4.06)	-90.26(4.74)	-98.07(4.88)	-113.8(4.80)	**-135.52**(4.48)	-132.83(5.11)	-94.55(3.88)
DeJong4	8.84(3.20)	3.47(2.02)	2.39(1.57)	1.3(1.23)	1.22(1.64)	**1.14**(1.54)	9.66(3.71)
Rastrigin	287.89(17.18)	232.08(20.3)	212.84(23.59)	**161.42**(20.05)	228.32(22.96)	230.01(21.80)	288.66(17.41)
Griewank	134.04(23.36)	76.26(13.62)	67.21(13.43)	**36.67**(10.88)	45.30(16.23)	45.93(17.52)	158.41(28.86)
Rosenbrock	1027.19(232.76)	583.57(148.74)	487.11(89.91)	**311.64**(84.76)	373.04(101.15)	372.30(99.90)	728.83(166.97)
Ackley	16.45(0.56)	14.50(0.71)	13.95(0.82)	**12.10**(34.72)	13.27(1.44)	12.93(1.49)	16.77(0.67)
Schwefel	9028.48(379.91)	7487.64(517.69)	6611.33(552.26)	**5312.65**(558.17)	6955.62(931.45)	6863.86(843.99)	9611.63(321.64)

Some important parameters for LEM(ED) are set as follows: the *learning threshold* is 0.3, the instantiation method is implemented according to probabilities; 80% of new individuals are generated from the *good* interval, 20% are from the *bad* interval. In LEM(ED)2's evolution mode, the mutation is implemented with a normal distribution with the mean equals to the gene value of the best individual so far and variance is a small value (1.0, here for all functions).

5.4.2 Summary of Results

Table 5.2 to Table 5.5 summarises the results of 100 runs of each algorithm on each function, with means and standard deviations recorded at 10, 20, 50 and 100 generations (multiply by 100 for number of fitness evaluations). Meanwhile, Figure 5.3 to Figure 5.9 show the mean convergence curves for each algorithm on these test functions, respectively.

Table 5.3: Means and standard deviation after 20 generations

Functions	GA(1.0)	GA(1.0/30.0)	LEM(KNN)	LEM(dwKNN)	LEM(ED)1	LEM(ED)2	CMAES
DeJong3	-97.71(3.42)	-117.21(3.64)	-127.6(3.21)	-141.36(2.80)	**-144.32**(2.74)	-140.99(3.68)	-112.16(2.82)
DeJong4	1.35(0.99)	0.09(0.11)	**0.037**(0.051)	0.16(0.24)	0.081(0.16)	0.096(0.24)	0.070(0.12)
Rastrigin	248.74(17.74)	155.40(19.59)	134.73(16.62)	**87.74**(15.04)	141.37(19.43)	130.41(21.71)	228.39(14.78)
Griewank	55.2333(10.57)	21.65(4.74)	17.27(3.97)	**5.70**(2.27)	14.40(4.74)	12.58(6.37)	26.07(7.78)
Rosenbrock	436.4(89.7)	220.01(51.03)	185.23(41.61)	**133.99**(46.33)	191.67(44.56)	178.08(46.77)	152.95(37.93)
Ackley	13.17(0.76)	9.96(0.83)	9.05(0.82)	**6.33**(0.84)	9.26(1.13)	8.83(1.49)	10.36(1.06)
Schwefel	8706.26(381.46)	5686.49(533.59)	4641.64(536.89)	**3470.82**(461.21)	5701.74(778.84)	5899.15(727.38)	9564.46(291.09)

Table 5.4: Means and standard deviation after 50 generations

Functions	GA(1.0)	GA(1.0/30.0)	LEM(KNN)	LEM(dwKNN)	LEM(ED)1	LEM(ED)2	CMAES
DeJong3	-128.23(2.65)	-146.67(1.42)	-149.6(0.61)	-149.93(0.25)	**-149.99**(0.1)	-149.12(1.28)	-141.7(2.23)
DeJong4	8.63e-3(7.53e-3)	3.60e-3(4.86e-3)	2.52e-3(3.17e-3)	7.76e-3(1.93e-2)	2.09e-3(2.22e-3)	3.09e-3(0.01	**5.28e-4**(0.12)
Rastrigin	174.94(30.17)	84.36(14.51)	64.38(10.78)	**30.32**(7.05)	72.27(14.42)	57.60(11.20)	191.26(12.1)
Griewank	6.75(1.99)	2.04(0.4)	1.52(0.27)	**1.08**(0.081)	1.77(0.33)	1.28(0.13)	1.16(0.079)
Rosenbrock	116.74(27.65)	75.38(29.28)	69.29(30.96)	68.34(39.16)	89.48(33.27)	69.02(33.27)	**29.94**(0.73)
Ackley	6.601(0.71)	3.91(0.50)	3.02(0.50)	**2.22**(0.58)	3.76(0.49)	2.83(0.50)	2.32(0.409)
Schwefel	7380.06(1012.55)	3277.86(493.95)	2009.99(375.16)	**1685.07**(330.60)	3506.38(641.16)	5050.98(673.27)	9460.74(282.0)

Table 5.5: Means and standard deviation after 100 generations

Functions	GA(1.0)	GA(1.0/30.0)	LEM(KNN)	LEM(dwKNN)	LEM(ED)1	LEM(ED)2	CMAES
DeJong3	-145.60(2.12)	**-150**(0.0)	**-150**(0.0)	**-150**(0.0)	**-150**(0.0)	-149.26(1.24)	**-150**(0.0)
DeJong4	1.77e-3(1.73e-3)	8.8e-4(9.9e-4)	8.7e-4(0.9e-3)	1.53e-3(1.68e-3)	5.83e-4(7.23e-4)	6.68e-4(7.72e-4)	**1.52e-4**(1.6e-4)
Rastrigin	77.29(33.48)	44.75(8.997)	27.66(7.9)	**11.01**(2.79)	38.67(10.09)	42.37(9.61)	116.69(63.62)
Griewank	1.52(0.68)	0.95(0.11)	0.51(0.25)	0.68(0.23)	0.92(0.14)	0.50(0.16)	**0.029**(0.017)
Rosenbrock	55.17(20.87)	44.72(24.15)	45.68(27.14)	53.00(34.72)	61.05(28.96)	44.71(27.80)	**27.42**(0.53)
Ackley	2.59(0.83)	0.95(0.52)	0.65(0.57)	1.38(0.61)	0.95(0.45)	0.17(0.10)	**0.019**(0.0094)
Schwefel	5120.94(1078.51)	1540.0(289.45)	**1357.99**(336.98)	1446.53(301.62)	1534.62(414.66)	4664.54(605.53)	8948.33(805.8)

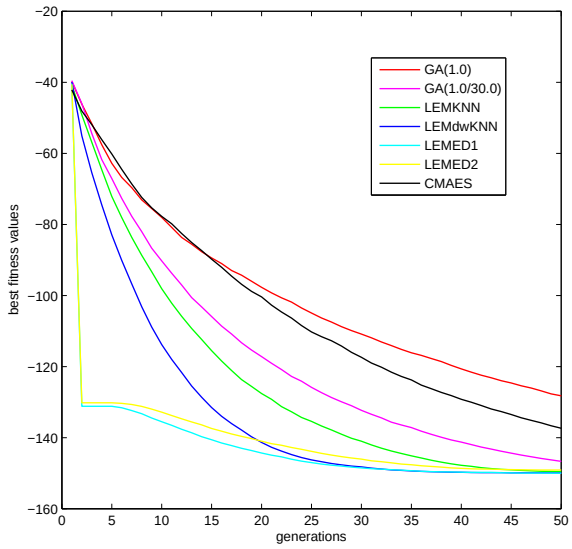


Figure 5.3: Results of running 7 algorithms on the DeJong3 problem

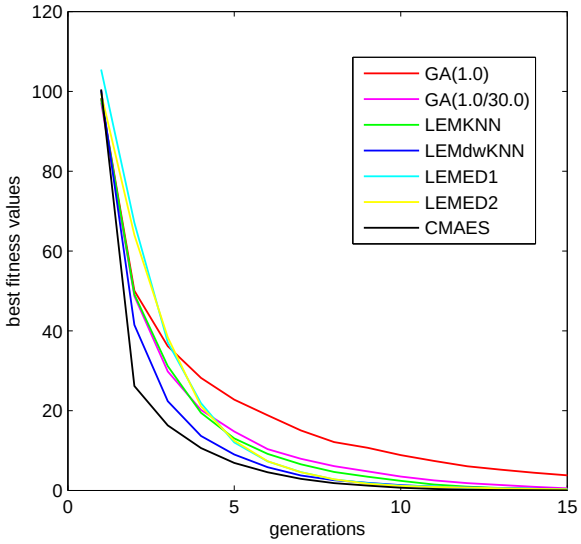


Figure 5.4: Results of running 7 algorithms on the DeJong4 problem

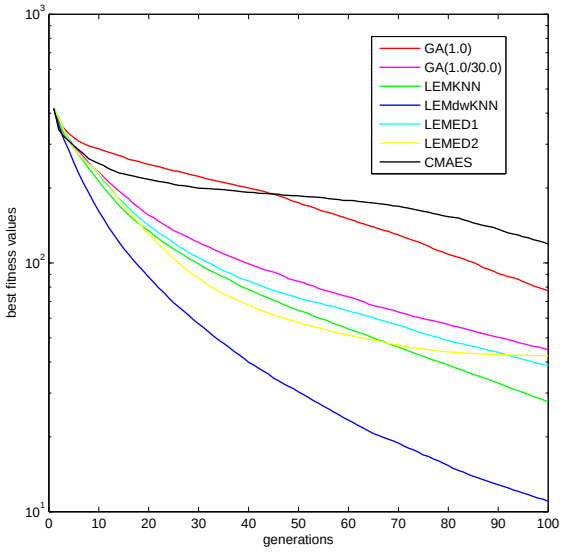


Figure 5.5: Results of running 7 algorithms on the Rastrigin problem

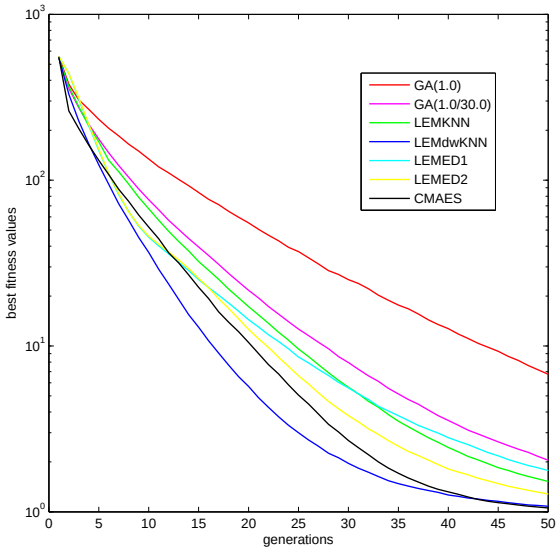


Figure 5.6: Results of running 7 algorithms on the Griewank problem

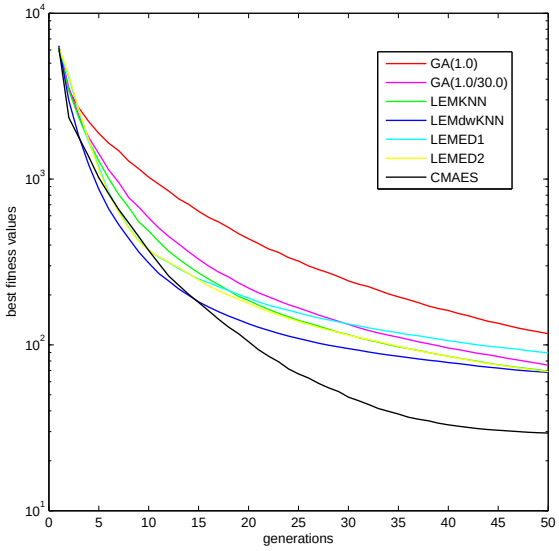


Figure 5.7: Results of running 7 algorithms on the Rosenbrock problem

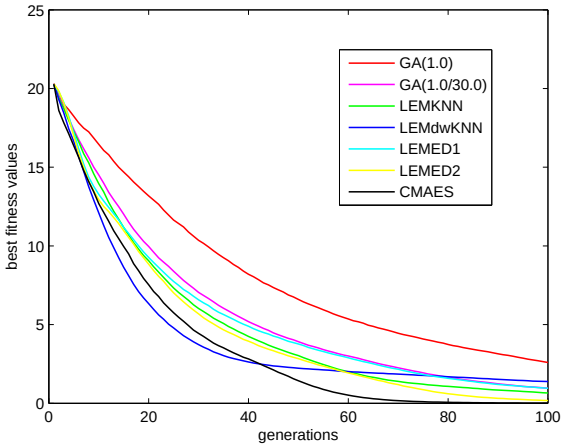


Figure 5.8: Results of running 7 algorithms on the Ackley problem

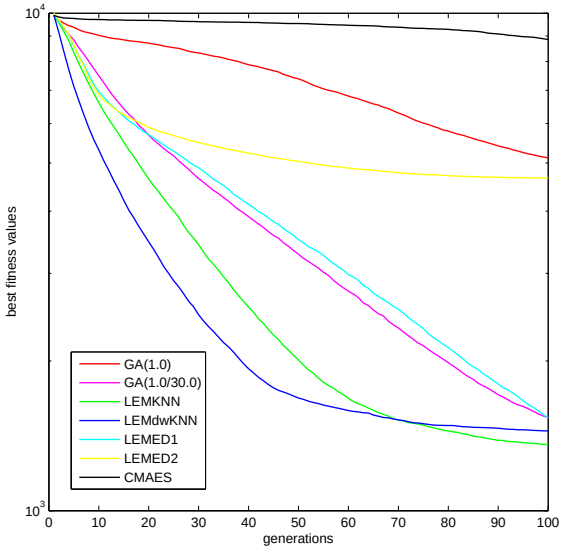


Figure 5.9: Results of running 7 algorithms on the Schwefel problem

In what comes below, statements of significance are based on randomization tests, and made only when confidence was above 99%. Inspection of standard deviations also clearly supports the statements made.

With only one exception (LEM(ED)2 on Schwefel at 50 and 100 generations) the LEM(ED) variants are always superior to the underlying GA1 and GA2. To some extent, the underlying EA can be seen as a 'straw man', and it is used here only as a baseline, with improvement to be expected. In the early generations (10 and 20 generations), all LEM(ED) versions have clearly beaten all GAs on all problems, except the Schwefel function. This shows the learning and instantiation operation in LEM(ED) clearly has its advantage over the normal evolutionary procedure. However, these outperformances are not as significant as we expected. In the later generations (50 and 100 generations), the range of problems in which LEM(ED) cannot beat GAs increased by one case of the Rosenbrock function (at generation 50).

LEM(ED) only beat LEM(KNN) variants on De Jong's functions and cannot outperform on any other functions at the earlier generation 10. From generation 20 to 50, LEM(ED) loses further superiority to the LEM(KNN) variant algorithms on almost all the problems. However, LEM(ED)2 seems to fight back and outperform LEM(dwKNN) at generation 100, the end stage of the optimization, on all problems except for DeJong 4, Rastrigin and Schwefel functions. We think this is due to the fact that LEM(ED)2 applies a very small mutation step size derived from the learnt intervals in the learning procedure. If the learnt output is correct, this mutation will help in the later stage of optimization.

With the exception of De Jong function 4 and Rosenbrock, one of the LEM(ED) variants is always superior to CMAES at the 1,000 and 2,000 evaluation points. At the 5,000 evaluations point, LEM(ED)'s advantage list over CMAES has been reduced by two more functions, Griewank and Ackley. And at 10,000 evaluations this list further reduces to Rastrigin and Schwefel, with ties for the De Jong 3 function. However, since LEM(ED)2 biases itself strongly to learned intervals, it is quite possible that some functions can lead to it being deceived and misled. It is interesting and promising that this usually does not seem to happen, however, LEM(ED)2's early fast progress on the Schwefel function clearly leads it in the wrong direction. This situation is the same for CMAES, which is always beaten by both the EA and LEM(ED)1 on Schwefel. In general, CMAES, which is itself a sophisticated hybrid of learning and evolution, overtakes LEM(ED) (and the vast majority of other algorithms) as we consume more function evaluations. Regarding the LEM(ED) design tested here, this is not surprising since our LEM(ED) variants use a single learning phase followed by an EA, while CMAES is continually learning and adapting. LEM(ED)'s performance in the 1,000-5,000 evaluations regime is nevertheless encouraging, and there may be considerable value in more sophisticated adaptive versions.

5.5 Concluding Discussion

We investigated a new LEM hybrid optimization algorithm LEM(ED), which incorporates a simple entropy-based discretization method as the learning component with a normal evolutionary procedure. The learning method ED applied here in LEM(ED) is a very simple mechanism compared with other well-known learning algorithms. ED simply takes the training data as input and uses an entropy measurement to find the best cut-points and therefore to identify the best interval to guide the generation of new individuals.

The LEM(ED) algorithm clearly outperforms its EA component alone both in the initial and later stages for all problems, however, in the later generations, the extent of advantages over EA begins to reduce. One of the LEM(ED) algorithms, LEM(ED)2 outperformans our first KNN based hybrid algorithm LEM(KNN) in general. However, neither of the LEM(ED) algorithms can beat the refined LEM(KNN) algorithm LEM(dwKNN) during the whole optimization procedure for almost all functions. Also, LEM(ED) generally outperforms CMAES during the initial several-thousand fitness evaluations. This adds to evidence that even straightforward learning mechanisms provide considerable benefit to an EA, especially for accelerating the search.

However, the outperformance LEM(ED) achieved over the normal GA is not as promising as the original LEM(AQ) algorithm, although their test functions are different from here. We think there are possibly the following reasons. First of all, the learning method applied in LEM(ED) is a very simple method, ED only considers binary discretization with entropy measurement. It does not distinguish the difference among variables, and therefore is not able to find out the relationship between dimensions, which could be very important to the success of the optimization procedure. Secondly, ED only binaries each variable range, which may not be the best way to fit the complex problem landscapes in a adaptive way. Ideally, discretization should be dividing the variable range into several subranges and change adaptively according to the optimization procedure. One such discretization method is the AQ learning method in LEM(AQ) algorithm. Finally, LEM(ED) does not contain repeated learning and evolution interactions, it has only one learning period, after that, the normal GA is applied to finish the rest of the optimization procedure. We think the absence of learning in the later stage also affects the optimization performance.

Lines of further work that seem warranted include testing on a more accepted set of optimization challenge functions, our choice of function suite follows those used in the original LEM publications. However, such suites are now superseded by those described in the CEC 2005 Challenge [SHL+05], which emphasizes non-separability and other measures that are likely to make functions difficult. Since most of the functions tested herein are, however, separable, the criticism can be made that the findings may well not generalize to nonseparable functions. However, following preliminary and ongoing work we can confirm that the LEM(ED) variants here show entirely similar relative (to basic GA and to CMAES) performance properties as found here, we deal with this later in Chapter 6.

Also there is a need to investigate repeated phases of ED-based learning (rather than a single phase at the beginning). Our investigations so far have focused on tightly coupled ED and instantiation, which (as we find in preliminary experiments) is best limited, rather than continued throughout the run, otherwise the learned intervals can be deceived and results suffer. However we are yet to investigate (which would be highly suited to the LEM framework) the interleaving of further ED/instantiation phases with phases of several generations of evolution. Meanwhile, the information inherent to the learned intervals could be used more creatively in later phases, in various ways. Also, a more sophisticated termination criterion for the ED phase would be beneficial, since we note that the better sets of intervals are often those learned a handful of generations before the cessation of fitness improvement.

More generally, more study seems warranted in the area of learning/evolution combinations (both in terms of LEM-framework instantiations, and also in terms of organizing the knowledge in this important area that is currently widespread in the literature).

Chapter 6

LEM Instantiated with Decision Tree Learning

6.1 Overview

In this chapter, we investigate the LEM(ID3) algorithm [SC10] which is a hybrid of evolutionary search with ID3 decision tree learning algorithm. The reasons for which we choose the ID3 decision tree learning algorithm as the learning component in our investigation of LEM framework are based on the following considerations.

First of all, the development of LEM(ID3) offers us an important chance to explore, for the first time, the effect of replacing AQ learning in LEM framework with a different but equally sophisticated learning algorithm, therefore to examine the flexibility of the LEM framework. ID3 and AQ learning algorithms have huge similarities in common, among which the most important are that both of them are supervised learning methods and the output of hypothesis descriptions are all rule based. The decision tree constructed by the ID3 learning algorithm can be transformed into set of rules. Therefore, many important properties of the LEM(AQ) algorithm can be studied and understood through the development of rule-based LEM(ID3) algorithm. For instance, the instantiation operation of the generation of new individuals for next generations according to learned rules. Through the development of LEM(ID3), we can understand how to design and apply this important operation in more detail, and how it influences the effectiveness of the resulting hybrid algorithms.

Second, another main reason for developing LEM(ID3) and applying the ID3 learning algorithm is due to the fact that, in the field of solving complex optimization problems,

many problem features make these problems very difficult and challenging to solve, therefore these difficulties and complexities require better learning algorithms to capture the relationships among the problem variables and characteristics involved in the problems. The decision tree learning algorithm ID3 is one of the good and practical learning methods widely applied in the machine learning community and the learned rules are able to navigate interactions between any number of parameters for many practical application problems. ID3 is expected to be able to gain better results within the LEM(ID3) hybrid algorithm and also to be able to tackle complex applications in practice.

Finally, another reason to develop LEM(ID3) is from the point of view or practical considerations. At the time that this PhD study is being conducted, the LEM(AQ) learning algorithm and the details of its implementation are not openly available. Implementation of our own LEM instance algorithm is necessary for experiments and solving our own problems at hand. For all the above reasons, LEM(ID3) is an important development step for our LEM investigation, and deserves to be a baseline algorithm for further development and improvement, and also should be applied to solve more practical and challenging application problems.

As with the previous developments of LEM instance algorithms LEM(KNN) and LEM(ED), LEM(ID3) involves interleaved periods of learning and evolution, adopting the decision tree construction algorithm ID3 as the learning component, and a steady-state EA as the evolution component. In the learning periods, based on chromosome data and evaluated fitnesses, ID3 is used to repeatedly find and infer rules that attempt to identify and predict whether a chromosome is 'good' or 'not good' based on the values of one or more other genes. The rules are then used to guide the generating of new individuals. When the learning component is finished, LEM(ID3) is switched to the evolution component, after that the learning starts again.

Without any preliminary parameters tuning, we evaluate LEM(ID3) on the 'test suite 2' used in previous chapters and also on the test suite of 25 functions designed for the CEC 2005 special session on real-parameter function optimization. We describe the results, and in particular compare with the three most successful algorithms from the CEC 2005 competition, the K-PCX algorithm [STD05], and two versions of Auger and Hansen's CMAES algorithms [AH05b, AH05a]. We find that LEM(ID3)'s performance is competitive with these algorithms, increasingly so as the problem dimensionality increases. In the case of 50-Dimensions, LEM(ID3) clearly records better overall performance on this function suite than the three comparative algorithms.

In the following sections, we provide complete details on the LEM(ID3) algorithm in Section 6.2, and describe the details of our experiments and analyze the results in Section 6.3, we conclude in Section 6.4.

6.2 LEM with Decision Tree Learning – LEM(ID3)

We have already discussed the ID3 decision tree learning algorithm [Qui86] and experienced the LEM(AQ), LEM(KNN) and LEM(ED) algorithms. Apart from the ID3 learning algorithm, LEM(ID3) shares many other aspects with the LEM(AQ) algorithm. In this way, we will have our own LEM(AQ) algorithm straightforwardly. LEM(ID3) contains two main components: learning and evolution. As with other LEM algorithms, in the learning component, LEM(ID3) divides the current population into high-performance (H-group) and low-performance (L-group) groups according to their fitness values and a given threshold. ID3 then uses the H-group and L-group as the training data to construct the decision tree, which is then transformed into a set of rules. These sets of rules are the hypotheses that differentiate between the two groups. New individuals are generated by instantiating these hypotheses. The learning mode continues until there is no better individual generated for a certain number of generations, or the diversity of the population is too small. The evolution mode begins when the learning mode is finished, in the evolution mode, a standard evolutionary algorithm is applied. The main purpose of evolution is that it offers the opportunity to escape from local optima and also preserves diversity for the current population, which is crucial to the success in the subsequent learning phase. Evolution continues for a certain number of generations, before the learning phase begins again. The overall pseudo-code of LEM(ID3) is set out here as Algorithm 13, with some components elaborated further with more details later in the paper.

6.2.1 Learning Mode

In the learning mode, basically, there are three main steps. They are Creating the Training Data; Learning and Generating Hypotheses; Instantiating Hypotheses and Generating New Individuals. We discuss all of these three steps in more detail.

Algorithm 13 pseudo code for LEM(ID3)

```
1: Generate initial population and evaluate each chromosome;
2: repeat
3:    while (Termination condition for learning is not satisfied) do
4:       Form the H-group and L-group from the current population;
5:       Learn a decision tree using the H-group and L-group, and transform it into a set
         of rules;
6:       Generate some new individuals for the next generation by instantiating new chro-
         mosomes guided by the learned rules; or
7:       Generate some new individuals for the next generation by evolution (mutation and
         crossover ) or at random;
8:    end while
9:    while (Termination condition for evolution is not satisfied) do
10:      Operate a standard evolutionary algorithm;
11:   end while
12:   Adjust discretization;
13: until (Termination condition for LEM(ID3) is satisfied)
```

Creating Training Data

In the learning mode, the first important step is to create high quality training data. High quality training data is crucial to the success of the learning algorithm. In LEM(ID3), the training data is generated from the current evolving population, therefore the quality of training data depends on the current population and its distribution. We use 'population-based selection' ([Mic00]) to generate the training data, in which we specify that a given percentage of the population will be in the H-group and a given percentage will be in the L-group. We use 30% in both cases – i.e., after sorting the individuals by fitness values, the top 30% are placed into the H-group and the lowest 30% are put in the L-group. Due to the individuals we consider are all real-parameter, these selected individuals in both groups are then discretized into discrete training instances. There are some practical implementation issues in the generating and discretization procedure, first, when the optimization procedure continues to progress the whole population will intend to converge to a few promising solutions, which will cause the population to lose its diversity, this is common for evolutionary algorithms. However, this situation of similarity in the later stage of evolution affects the creation of enough training data for the learning algorithm ID3, therefore affecting the qual-

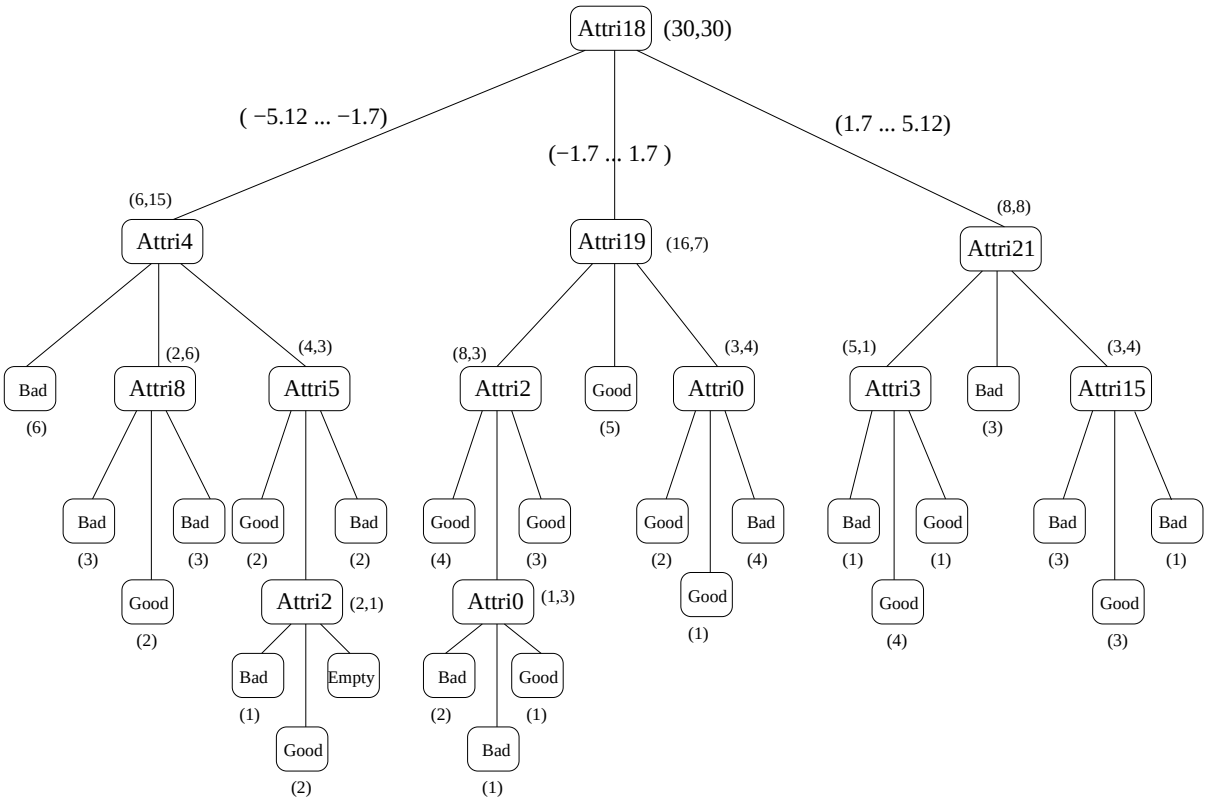


Figure 6.1: A decision tree learned by LEM(ID3) for Rastrigin function at generation 1

ity of training data. During our implementation, we found in the later evolution stage, if no good mechanism is employed to release this problem, then very little training data will be generated, and when ID3 is applied to this small-sized training data, the resulting decision tree and corresponding rules are less meaningful. We will come back to this issue and talk about our solution later on.

Learning Hypotheses

When a good set of training data has been generated from the current population of individuals, LEM(ID3) uses ID3 learning algorithms to construct a decision tree. The construction procedure is straightforward, as discussed in Section 2.3.1. The resulting tree is then transformed into a set of rules, which can then be seen as hypotheses discriminating H-group and L-group individuals of the current population. We call this stage the Learning Hypotheses, which contain a number of important issues which we will discuss next in more detail. For now, we give an example decision tree constructed by a real LEM(ID3) run on the Rastrigin's function as defined in 'Test Suite 2' in Figure 6.1, and the ruleset produced from this decision tree in Table 6.1:

Where for both decision tree and rule, $attri_i$ are decision tree *attribute* terms correspond-

Table 6.1: The ruleset transformed from the DT for positive data in Figure 6.1

1	$attri_{18} = (-5.12 \cdots -1.7) \wedge attri_4 = (-1.7 \ldots 1.7) \wedge attri_8 = (-1.7 \ldots 1.7) \Longrightarrow G$
2	$attri_{18} = (-5.12 \cdots -1.7) \wedge attri_4 = (1.7 \ldots 5.12) \wedge attri_5 = (-5.12 \cdots -1.7) \Longrightarrow G$
3	$attri_{18} = (-5.12 \cdots -1.7) \wedge attri_4 = (1.7 \ldots 5.12) \wedge attri_5 = (-1.7 \ldots 1.7) \wedge attri_2 = (-1.7 \ldots 1.7) \Longrightarrow G$
4	$attri_{18} = (-1.7 \ldots 1.7) \wedge attri_{19} = (-5.12 \cdots -1.7) \wedge attri_2 = (-5.12 \cdots -1.7) \Longrightarrow G$
5	$attri_{18} = (-1.7 \ldots 1.7) \wedge attri_{19} = (-5.12 \cdots -1.7) \wedge attri_2 = (-1.7 \ldots 1.7) \wedge attri_0 = (1.7 \ldots 5.12) \Longrightarrow G$
6	$attri_{18} = (-1.7 \ldots 1.7) \wedge attri_{19} = (-5.12 \cdots -1.7) \wedge attri_2 = (1.7 \ldots 5.12) \Longrightarrow G$
7	$attri_{18} = (-1.7 \ldots 1.7) \wedge attri_{19} = (-1.7 \ldots 1.7) \Longrightarrow G$
8	$attri_{18} = (-1.7 \ldots 1.7) \wedge attri_{19} = (1.7 \ldots 5.12) \wedge attri_0 = (-5.12 \cdots -1.7) \Longrightarrow G$
9	$attri_{18} = (-1.7 \ldots 1.7) \wedge attri_{19} = (1.7 \ldots 5.12) \wedge attri_0 = (-1.7 \ldots 1.7) \Longrightarrow G$
10	$attri_{18} = (1.7 \ldots 5.12) \wedge attri_{21} = (-5.12 \cdots -1.7) \wedge attri_3 = (-1.7 \ldots 1.7) \Longrightarrow G$
11	$attri_{18} = (1.7 \ldots 5.12) \wedge attri_{21} = (-5.12 \cdots -1.7) \wedge attri_3 = (1.7 \ldots 5.12) \Longrightarrow G$
12	$attri_{18} = (1.7 \ldots 5.12) \wedge attri_{21} = (1.7 \ldots 5.12) \wedge attri_{15} = (-1.7 \ldots 1.7) \Longrightarrow G$

ing to each dimension or gene in individuals, the real number ranges (...) are generated through discretizing the current learning population (training data) individuals and can be seen as *domain* for each *attribute*. G indicates the classification information in the training data set and represents High-group of individuals in the training data. For the decision tree being constructed here, each path rooted from attribute $attri_{18}$ to leaf nodes *Good* or *Bad* can be transformed into a rule. For example, the rule $attri_{18} = (-5.12 \cdots -1.7) \wedge attri_4 = (-1.7 \ldots 1.7) \wedge attri_8 = (-1.7 \ldots 1.7) \Longrightarrow G$ is a path in the decision tree constructed by ID3, and any training instances satisfying this path are classified into class G, representing some individuals in H-group. Also, in this decision tree, there is some useful extra information which can be derived, such as the coverage value for each rule (the number of instances satisfying the rule). And the average fitness values for each rule (the average fitness value for the instances satisfying this rule), which are not shown in our illustration.

After the decision tree is constructed and its ruleset transformed, we still face some important issues, two of these are highlighted here due to their importance in the success of our LEM(ID3) implementation. First, as seen in Figure 6.1, there are many attributes (totaling 30 attributes for Rastrigin's function) which do not appear in the constructed tree. Therefore, when the instantiation hypothesis operation is implemented, we will face the problem of choosing new values for these attributes, for which it is difficult to find good methods. Second, the ruleset transformed from the constructed decision tree consists of a huge number of rules with different coverage values, as seen in Table 6.1. The *rules 5, 9, 11* have only coverage value 1 and *rules 1, 2, 3, 8* have only coverage value 2, while *rule 7* has

a coverage value 5. Rules with different coverage values should be treated differently, small coverage values mean the corresponding rules could be meaningless or representing noisy data. Therefore, we have to make a decision on the choosing of these rules. We discuss the first issue here which is more relavent to the Learning Hypothesis stage, and will discuss the second issue in the Instantiating Hypotheses stage.

Forest Model

We face an important issue after the construction of the decision tree. That is, many attributes do not appear in the tree and ruleset. This is due to the feature of ID3 learning algorithm, ID3 always tries to classify the training data as efficiently as possible, it starts from the root attribute with best information gain, and excludes those training data and the current selected attribute, and repeats to find another attribute with the remaining training data and attributes. This procedure repeats until all training data are excluded. However, such a procedure is very efficient in the sense that not too many attributes will be involved in this procedure. Namely, very few attributes are used to classify the training data and to construct the decision tree. This naturally raises the problem for instantiation algorithms, that is, what should be done to those attributes which do not appear in the rules? In LEM(AQ), the authors have mentioned possible solutions for this question, for example, if one attribute does not appear in any rule, then the corresponding parent gene values are inherited into the new individuals. In the development of LEM(ID3), we reconsider this problem and attempt to solve it by indicating one disadvantage of the ID3 algorithm. Namely, a single run of the ID3 algorithm is not able to learn or mine all useful patterns existing in the given training data.

To analyze this, we reconsider a constructed decision tree by ID3 based on the training data from the current population. One path, from $attri_i$ (the root) down to the leaf node G, within the decision tree says that if an instance satisfies the attributes and their domain values, then this individual will result in the class 'G'. Essentially, each path and its rule is a pattern representing some instances with common features which distinguish them from the others in the training data. And if the pattern can represent the instances well, how many such patterns exist? With this question in mind, we summarize three important statements:

1. One decision tree (or its ruleset) represents a pattern involved in the training data;

2. One tree is 'searched' by the ID3 algorithm in a greedy fashion according to the information gain criterion, and ID3 is a local search algorithm;

3. According to the information gain criterion, one constructed tree is the best pattern for the training data at hand.

Evidently, these three statements can be extended or be more precisely stated, if we consider the decision tree construction procedure in a more global view:

1. One decision tree represents one of the patterns (not only one) involved in the training data;

2. Other patterns can be 'searched' by global algorithms according to other criteria;

3. According to the information gain criterion, one constructed tree by ID3 may or may not be the best pattern for the whole data space (including training data and future unseen data).

Considering ID3 decision tree construction procedure in a global view and distinguishing the concepts of 'local ' and 'global' are our goals and the reasons to build up the *forest model*. Namely, we want to capture all the other patterns or trees 'hidden' in the training data. Constructing by ID3 can only offer one pattern which is the most efficient according to the information gain criterion. Of course, this tree being the most efficient is very useful. However, the other trees or patterns may also be very helpful and could reflect more relationships between attributes for the training data.

In order to build the forest model, a number of steps need to be followed. Firstly, the ID3 algorithm is used to construct the first tree in the normal way. After the first tree is constructed, a number of other trees will be constructed in sequence, the construction method is still ID3, but this time with a fixed or pre-selected root attribute indicated. Namely, we pre-selected for each following tree a fixed root attribute which must be different to the first and previous tree's root attributes. For the following trees, ID3 is applied as the construction algorithm only with the exception of the root's attributes. Once an attribute is selected as the root attribute, it is not available for the following selection. Namely, all attributes including the first root attribute selected by ID3 can only be used once during the construction of the forest model. When there is no attribute left or an indicated number of attributes are selected, the construction procedure is finished, and the resulting forest model is derived.

In this way, apart from the normally built first decision tree, we will also have a number of extra trees which are built with pre-selected root attributes. These attributes do not have the best information gains compared with the root attribute selected by ID3 in the normal way, however, more patterns with also useful information for the training data

could be represented by trees rooted at these attributes. Finally, we do not design any new measurement criterion, like information gain, the only thing we do is to change the ID3 algorithm slightly and apply it several times, (with each time a different attribute for one tree), to construct a 'forest' model. A brief illustration of the construction procedure for a forest model is given in Figure 6.2. Given a population of individuals, each of which consists of 10 genes (the length of chromosome). When this population is used as the *learning population* and discretized. We construct a forest model from the training data by applying the ID3 algorithm repeatedly, the forest model is defined as a set of trees, $\{(T_r, T_i, \ldots, T_j), r, i, j \in (1 \ldots 10), r \neq i \neq j\}$, where r, i, j indicate root attributes for these trees.

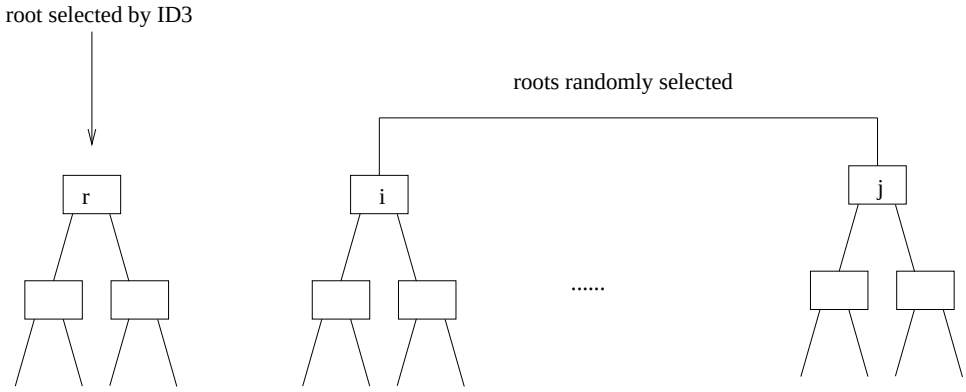


Figure 6.2: An illustrative example for the forest model

The forest model can find more patterns existing in the training data. Not only is the most efficient tree useful, the slightly less efficient trees are believed to be still very useful and helpful in indicating good patterns. This is more convincing if we consider the ID3 construction algorithm in a more global view, and consider each tree as one of the many possible classification problem solutions. And also, more importantly, for our problem posed before, more decision trees being constructed will make more attributes appearing in the learnt patterns. During the instantiating procedure for new individuals, there will be fewer situations where for a given attribute, we do not know how to assign a new value.

Instantiating Hypotheses

The last step in the learning mode is to instantiate the learned hypotheses and generate new individuals for next generations. In LEM(AQ) and LEM(ID3) algorithms, during the learning mode, the new individuals are generated by instantiating the learned hypotheses rather than by genetic operators. Therefore, the instantiation procedure is very important

in the success of LEM algorithms. In the original LEM(AQ) algorithm, there are basically three different instantiation algorithms, they are developed according to different considerations. In our LEM(ID3) algorithm, the instantiation procedure is different from those in LEM(AQ), this is because despite many common features, ID3 and AQ are different learning methods; the former employs the divide-and-conquer strategy to construct decision tree and rules, while the latter employs the separate-and-conquer strategy, as we discussed in chapter 2. These strategies are quite different, therefore the constructed rules will have different representation forms as well for the same given data set. It is these differences that result in independent development of new instantiation procedures for our LEM(ID3) algorithm. However, before we introduce our instantiation algorithm, we discuss another important issue related to instantiating hypotheses first, that is the rule selection issue, which we did not solve in the previous section.

Rule Selection

As we have discussed before, there are many practical problems within the Learning and Instantiating procedures for ID3 generated rules. First, there are not enough attributes appearing in a single tree or ruleset, we solve this problem with the suggested 'forest' model, where extra trees are constructed with more attributes being involved in representing the learnt patterns. Namely, we have solved the 'quantity' issue, and it is time to solve the 'quality' issue for the rules in each ruleset. We expect the rules in a ruleset should catch enough useful 'patterns' residing in the training data, and these patterns should correctly reflect the relationships between the genes and their corresponding classification. That is, the rules should be useful and correct. By useful, we mean the rules should be representative enough to describe a pattern; and by correct, we mean these patterns should be important enough or be able to reflect the global properties of the current population space. Rules that satisfy these requirements should contain enough attributes, rather than only one or two attributes, and also should cover enough training instances. Therefore, we define two criteria used to decide the quality of rules generated by ID3 algorithm. First, if a rule contains enough attributes in its condition part, then we call this rule an *informative rule*, in contrast, if a rule contains little attributes in its condition part, then we call this rule an *uninformative rule*. Second, if a rule covers enough training instances, then we call this rule a *significant rule*, in contrast, if a rule contains little training instances, then we call this rule an *insignificant rule*. Note, we define these criteria in a relative way, which means we use these criteria to qualify the rules always in comparison with other rules. Table 6.2

Table 6.2: Meaning of a preferred rule

	Informative	Uninformative
Significant	preferred	not preferred
insignificant	not preferred	not preferred

shows what we mean by a *preferred rule*.

Based on this definition of *preferred rule*, one extra step we need to take during instantiation hypothesis is to select qualified rules from the set of all rules. The selection criterion is the *preferred rule* which is defined as being *Informative* and *Significant*. There are many measurement methods which exist in the machine learning community for these concepts or similar concepts. Here, we are only concerned with the implementation issue for our LEM(ID3) algorithm. In LEM(ID3) construction, *Informative* is quantified by the length of the rule, and *Significant* is quantified by the coverage value for the rule. We think such an extra step of 'rule-selecting' or contrarily 'rule-excluding' is necessary for improving our LEM(ID3) algorithm for the following reasons: an uninformative rule should be excluded from the current ruleset, because it contains too few attributes and therefore cannot catch any useful pattern from the training data even if it is significant (covering enough training instances); an insignificant rule should also be excluded from the current ruleset, because it covers too few training instances which are possible noisy data in the training data. Either of these rules and rules which have both properties should be excluded from the final ruleset, avoiding the possibility of misleading our instantiation procedure.

To this point, we are now ready to talk about the instantiating algorithm. In our LEM(ID3) algorithm, we designed the instantiation operator with the consideration of three important aspects. First, for the attributes which do not appear in the learned rule set, some particular methods need to be implemented to assign values for these attributes for each individual under generation. Second, for each of rule, the number of individuals which needs to be generated from this rule also needs to be calculated in a proper way, or according to a reasonable standard. Finally, the way in which the new individuals are generated also needs to be considered carefully, for example, whether to get a value randomly or modify an existing value can affect the optimization performance for LEM algorithms. Based on all of these considerations, we designed the instantiation algorithm as Algorithm 14. The core idea is based on the coverage value of each rule. Namely, rules with high coverage are used more frequently in generating new individuals.

Algorithm 14 pseudo code for instantiation

```
1:  Declare rule coverage variables c_r for each rule r;
2:  Declare number of training examples t;
3:  for all (rules in the ruleset) do
4:      Calculate the coverage of the rule c_r;
5:      Initialize 'generation distribution' variables p_ij to 0.0;
6:      for (each Attribute_i that appears in the rule) do
7:          for (each Interval_j of the Attribute_i) do
8:              p_ij += c_r/t;
9:          end for
10:     end for
11: end for
12: while (the new individuals are still needed) do
13:     for (each Attribute_i in the individual) do
14:         Select interval_j for Attribute_i with probability p_ij/T, where T sums the p_ij values
            for Attribute_i, and randomly creates a new value within interval_j.
15:     end for
16: end while
```

In the instantiation Algorithm 14, first, for each attribute and domain pair of the discretized problem space, a probability value is calculated according to the learned ruleset. For each rule in the current ruleset, if this attribute-domain pair appears in one rule, then the corresponding rule coverage value is recorded and accumulated as p_{ij}. The more times this attribute-domain pair appears, the bigger the probability value is. When all the rules in the ruleset are examined, the resulting probability values are then used to assign values for the new individuals being generated. Second, when the new individuals are generated, for each gene (attribute) of one individual, the value is generated from a particular interval (domain) of that gene, the frequency of a value created from a domain depends on the probability p_{ij} calculated previously. When all the new individuals are generated, we finish one instantiation procedure.

In fact, the instantiation procedure could have many variants implemented based on many other criteria, such criteria may include accuracy of the rule on training data evaluated by cross-validation method, but this method may need more training data. We can investigate this idea in our further research.

6.2.2 Evolution Mode

In LEM(ID3), when the Learning Mode can not find any better individuals by Learning Hypothesis and Instantiation Hypothesis. LEM(ID3) will switch to the Evolution Mode, where the traditional evolutionary computation operations are applied. Here, we emphasize one important issue in many population based optimization algorithms. That is, the diversity of the population. This problem turns out to be more severe for LEM methods, due to the application of learning techniques. Loss of the diversity for the current population does not only affect the evolutionary search procedure, but more importantly affects the learning procedure. This is because it can result in a lack of enough training data for the supervised learning algorithms. Generally, when the population is in the early stages of optimization, the individuals in the population tend to be very different to each other, therefore, the population has a good diversity. However, as the optimization progresses into the later generations, the diversity is lost in general, this is because the selection operations are making more copies of the promising individuals, therefore the search is focusing on some particular regions, containing either the local or global optimum. This is a common situation in most EAs, but in the case of LEM, we note that it causes particular problems for the learning process, and does not make the generation of training data an easy task.

Furthermore, some other factors will make the situation even worse. These factors include: first, the individuals in the population are very similar (this is the case commonly for the end phrase); second, the threshold is not adaptive, (say not changed from 30% to 10%); third, for real-parameter optimization problems, discretization needs to be applied on the training data and will cause the worse phenomena that, contradictory is possible occurring, that is after discretization, two or more individuals having the same chromosome values are put into different classification groups. Contradictory will cause more difficulty for the ID3 algorithm to generate a useful and correct decision tree. For all of these factors in the design of the LEM(ID3) algorithm, we need to consider diversity-preserve mechanisms in the evolution mode regularly before the learning mode starts again. To conquer this problem, for the moment, LEM(ID3) employs the simplest possible diversity preservation method: when diversity is too low, we perturb the population with a very high mutation rate. Through our observation in our experiments, we find this method works very well in preserving our population diversity for our problem, although it has the disadvantage that the evolution mode will contribute to less good optimization performance.

6.2.3 Switch Conditions

There is also another very important issue in the design of any LEM based hybrid optimization algorithms, that is the Switch Conditions. These conditions define the boundary between the learning and evolution procedure, and will decide when the learning procedure should stop and the evolution procedure begins, or vice versa. Therefore, the good design on these switch conditions is crucial for the success of the hybrid optimization algorithm. In fact, these conditions are very difficult to define precisely in practice. In LEM(ID3), we have attempted to develop switch conditions in a number of ways. Here, we give two methods applied in our implementations.

The first switch condition is coined as the *progress rate*, which is based on the algorithm's optimization performance during the past generations until now. Progress rate is the percentage of generations which achieves optimization performance improvement to its immediate previous generation over a fixed period of generations. If within a generation with the expense of certain evaluations (say, 100 evaluations), the best fitness of this population of this generation is improved with regard to its previous generation, then we call such a generation an *improved generation*, otherwise *unimproved generation*. For a given period of generations, the progress rate for this period is simply the number of improved

generations divided by the number of the generations for this period. Say we monitor 10 generations as a learning period, and find there are a total of 6 generations where the best fitnesses are improved from their previous generations, then the progress rate for this learning period is 0.6. Based on this progress rate, we also need to define a *progress rate threshold* which will decide the logic value of the switch conditions. Namely, if the current progress rate is above the progress rate threshold, then the current mode should continue otherwise a switch action should happen and another mode begins.

The second switch condition is related to the diversity of the population, and is defined as the minimum-allowed training data set size. As we mentioned before that, the quality of the training data set is crucial to our LEM(ID3) algorithm, because it is related to and used directly by the ID3 decision tree construction algorithm to generate the learnt hypotheses and the transformed ruleset. If the training data is noisy or does not have enough training instances, then the resulting decision tree is either meaningless or useless in representing the pattern of the training data. The later situation can be more easily dealt with or avoided if we assume the training data is noisy-free, then we require the size of the generated training data set after discretization cannot be lower than a minimum threshold. This forms the second switch condition, namely, if the size of the training data set is smaller than a given threshold (the minimum allowed training data set size), then LEM(ID3) is switched to evolution mode immediately without the learning mode to be conducted. We will come back to the two switch conditions later in the parameters settings part in the experiment section.

6.2.4 Discretization

Before any application of ID3 algorithm, the population needs to be (for learning use only) discretized. Instead of regarding genes as real-valued variables, each gene must range over a small set of intervals that partition each range. There are many discretization methods available as we discussed in Chapter 5, however, in our current development of LEM(ID3) algorithm, we use a very simple fixed interval discretization, but we adapt the number of intervals when the fitness seems to have stagnated. This is done simply by multiplying the number of intervals by an integer factor. Figure 6.3 illustrates this by showing the difference in the search space before and after such an adjustment in the discretization with factor 2.

The simplification of the discretization method in our current development of our LEM(ID3) algorithm is because we are paying more attention to the design of the LEM(ID3) algorithm

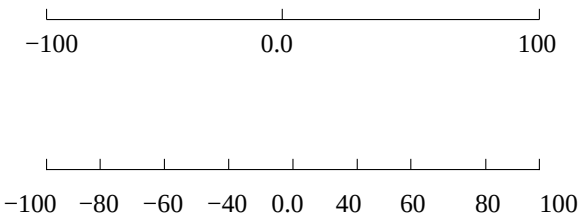


Figure 6.3: Before and after adjusting discretization representation

structure, rather than on the specific component technique, like discretization, however, in future development, we will incorporate more complex discretization techniques into our LEM(ID3) algorithm.

6.2.5 Instantiation, Evolution and Randomization

Until now, we have finished the introduction of our original LEM(ID3) algorithm and analyze many important aspects about this hybrid algorithm. However, these are not all features involved in the development of the LEM(AQ) algorithm, there are some other advanced considerations which are used to tackle the complexity of problems met in practice. These ideas are applied to our LEM algorithm and a new version of the LEM(ID3) algorithm, LEM(ID3) algorithm extended with Instantiation, Evolution and Randomization (LEM(ID3)IER) is developed.

Although Learning and Instantiation play key roles in the learning phase, they are not the only possible operations applied in the generation of new individuals. Part of the population can still be generated by the standard evolutionary operators, or even by random generation. This requirement for more various individuals generating methods is due to the fact that practical problems features could be very complex, and also there is a need to maintain diversity during the learning phase (not only in the evolution mode), which is essential in order to generate an informative tree. Namely, in the learning phase, a new individual could be generated either by the instantiation method described above, or by a standard evolution procedure, or at random. The randomization could be implemented by generating a random value from the whole search space.

In order to realize these ideas, we need to apply a parameter setting to decide the probabilities (percentages) for each operations. Ideally, these percentages would adapt as optimization progresses, however, for simplicity we use fixed (unoptimized) values in the current work. Therefore, the resulting algorithm LEM(ID3)IER modifies the original LEM(ID3) algorithm by allowing some percentage of the new individuals to be generated

by either evolution (crossover and mutation) or randomly through the whole search space. We will test the performance of both LEM(ID3) and LEM(ID3)IER in our following experiment part.

6.3 Experiments and Results

In this section, we begin to explore the performance of our LEM(ID3) algorithms on a number of real-parameter functions optimization problems. Although still being well-defined functions, the test functions now are more complex and equipped with more complex features. We compare the LEM(ID3) algorithms with other contemporary advanced hybrid optimization algorithms, not only the standard CMAES algorithm this time. To illustrate, we group our experiments into two parts. We will see all of the details about these two parts of experiments in the following sections.

6.3.1 Experiment Study 1

In this experiment study, we test the LEM(ID3) algorithm following the series of experiments we did before for our LEM algorithms. That is, we compare LEM(ID3) with the standard GAs, our developed LEM variant algorithms, and the standard CMAES algorithm.

Test Functions

The first test functions set used here is the old 'Test Suite 2' used in previous Chapters 4, 5, we refer to these chapters for the definitions of these functions.

Parameters Settings

Due to the various performance derived from our previous experiments, in this experiment we only test some of our previous testing algorithms, they are GA2, LEM(dwKNN) and CMAES. We also refer to the previous chapters for the parameters settings for these algorithms and only give the parameters settings for the LEM(ID3) algorithm. LEM(ID3) is implemented with the following settings. For the learning phase, we set the *threshold* as 0.3, initial discretization divides each gene's range into an *interval number* of 3 intervals. When we adjust discretization, the *interval number* is multiplied by the integer discretization factor 2. In the evolution phase, we use a steady-state strategy with binary tournament selection, a normal distribution (0,σ) mutation operator with mutation probability *1.0/size*

Table 6.3: Parameters settings for LEM(ID3)

Threshold	0.3
Switch conditions	Progress Rate: 0.7 Minimal allowed training data set size: 10
Learning gap	1
Discretization method	Fixed discretization with initial interval_number 3, and integer discretization factor 2.
Instantiation method	Instantiate intervals with probabilities (80%, 20%)
GA applied	GA2

Table 6.4: Means and standard deviations after 10 generations

Functions	GA(1.0/30.0)	LEM(dwKNN)	LEM(ID3)	CMAES
DeJong3	-90.26(4.74)	-113.8(4.80)	**-126.78**(6.88371)	-94.55(3.88)
DeJong4	3.47(2.02)	**1.3**(1.23)	2.98115(1.55961)	9.66(3.71)
Rastrigin	232.08(20.3)	161.42(20.05)	**75.1045**(16.4482)	288.66(17.41)
Griewank	76.26(13.62)	36.67(10.88)	**14.8608**(8.00999)	158.41(28.86)
Rosenbrock	583.57(148.74)	311.64(84.76)	**221.745**(62.123)	728.83(166.97)
Ackley	14.50(0.71)	12.10(34.72)	**8.81757**(1.66572)	16.77(0.67)
Schwefel	7487.64(517.69)	5312.65(558.17)	**3365.08**(533.216)	9611.63(321.64)

of chromosomes applied, where the mutation step size σ is a value always bigger than the current interval size in the discretized search space. We summarize the complete parameters setting for the LEM(ID3) algorithm in Table 6.3. The *population size* is 100 for all problems.

Summary of Results

The experimental results are are summarized in the same way as before and are listed from Table 6.4 to Table 6.7. Meanwhile, Figure 6.4 to Figure 6.10 show the mean convergence curves for each algorithm on these test functions, respectively.

Table 6.5: Means and standard deviations after 20 generations

Functions	GA(1.0/30.0)	LEM(dwKNN)	LEM(ID3)	CMAES
DeJong3	-117.21(3.64)	**-141.36**(2.80)	-139.06(4.32708)	-112.16(2.82)
DeJong4	0.09(0.11)	0.16(0.24)	0.918659(0.828816)	**0.070**(0.12)
Rastrigin	155.40(19.59)	87.74(15.04)	**48.2334**(10.342)	228.39(14.78)
Griewank	21.65(4.74)	**5.70**(2.27)	5.93735(2.54205)	26.07(7.78)
Rosenbrock	220.01(51.03)	**133.99**(46.33)	180.653(42.7118)	152.95(37.93)
Ackley	9.96(0.83)	**6.33**(0.84)	6.62236(1.2462)	10.36(1.06)
Schwefel	5686.49(533.59)	3470.82(461.21)	**2835.22**(504.211)	9564.46(291.09)

Table 6.6: Means and standard deviations after 50 generations

Functions	GA(1.0/30.0)	LEM(dwKNN)	LEM(ID3)	CMAES
DeJong3	-146.67(1.42)	-149.92(0.25)	**-149.93**(0.320826)	-141.7(2.23)
DeJong4	3.60e-3(4.86e-3)	7.76e-3(1.93e-2)	3.24e-2(5.64e-2)	**5.28e-4**(0.12)
Rastrigin	84.36(14.51)	**30.32**(7.05)	34.786(8.62315)	191.26(12.1)
Griewank	2.04(0.4)	1.08(0.081)	**1.02904**(0.0534997)	1.16(0.079)
Rosenbrock	75.38(29.28)	68.34(39.16)	115.768(37.5217)	**29.94**(0.73)
Ackley	3.91(0.50)	**2.22**(0.58)	2.56555(0.464773)	2.32(0.409)
Schwefel	3277.86(493.95)	**1685.07**(330.60)	1765.76(443.485)	9460.74(282.0)

Table 6.7: Means and standard deviations after 100 generations

Functions	GA(1.0/30.0)	LEM(dwKNN)	LEM(ID3)	CMAES
DeJong3	**-150**(0.0)	**-150**(0.0)	**-150**(0.0)	**-150**(0.0)
DeJong4	8.8e-4(9.9e-4)	1.53e-3(1.68e-3)	2.69-3(3.45-3)	**1.52e-4**(1.6e-4)
Rastrigin	44.75(8.997)	**11.01**(2.79)	22.1808(6.17295)	116.69(63.62)
Griewank	0.95(0.11)	0.68(0.23)	**0.020**(0.013)	0.029(0.017)
Rosenbrock	44.72(24.15)	53.00(34.72)	83.5877(35.9244)	**27.42**(0.53)
Ackley	0.95(0.52)	1.38(0.61)	0.119332(0.0599154)	**0.019**(0.0094)
Schwefel	1540.0(289.45)	1446.53(301.62)	**1188.65**(331.967)	8948.33(805.8)

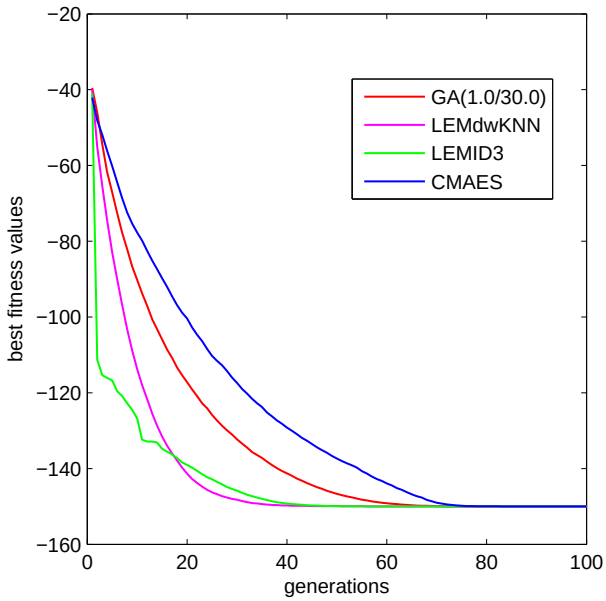


Figure 6.4: Results of running 4 algorithms on the DeJong3 problem

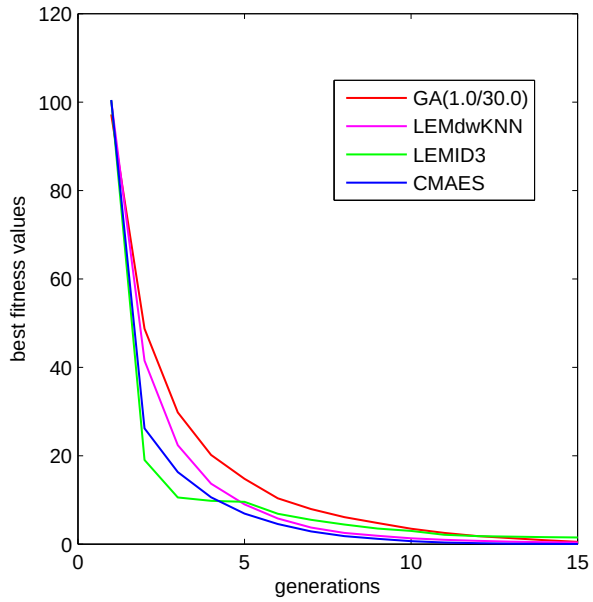


Figure 6.5: Results of running 4 algorithms on the DeJong4 problem

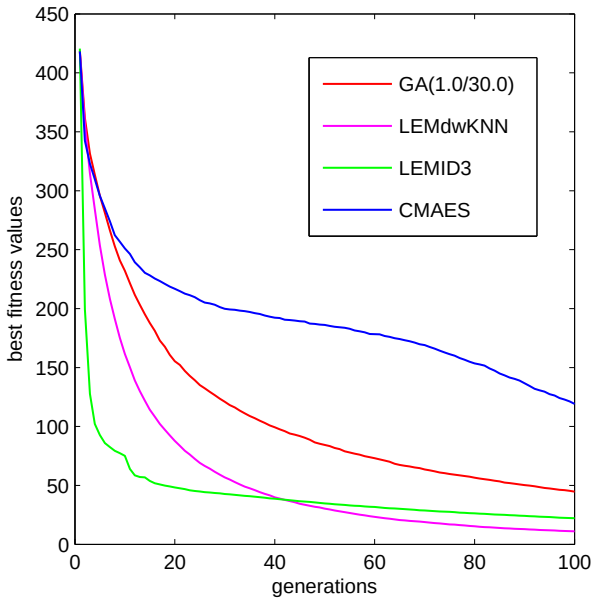


Figure 6.6: Results of running 4 algorithms on the Rastrigin problem

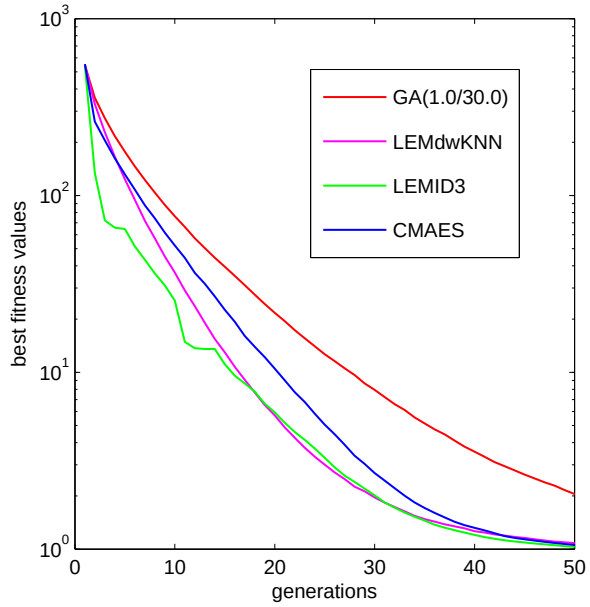


Figure 6.7: Results of running 4 algorithms on the Griewank problem

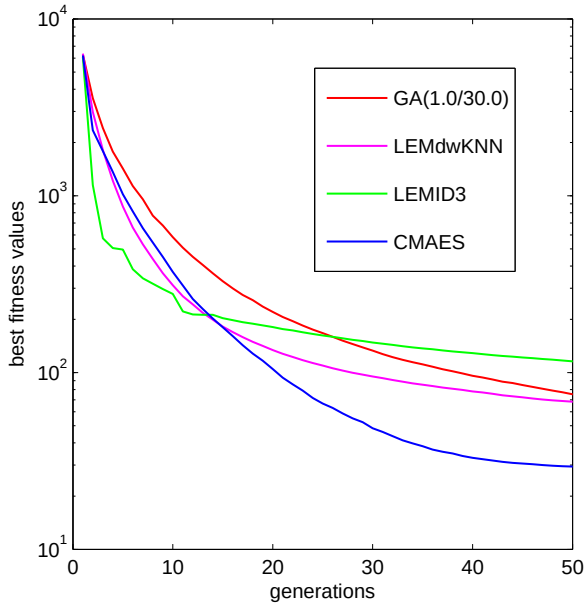


Figure 6.8: Results of running 4 algorithms on the Rosenbrock problem

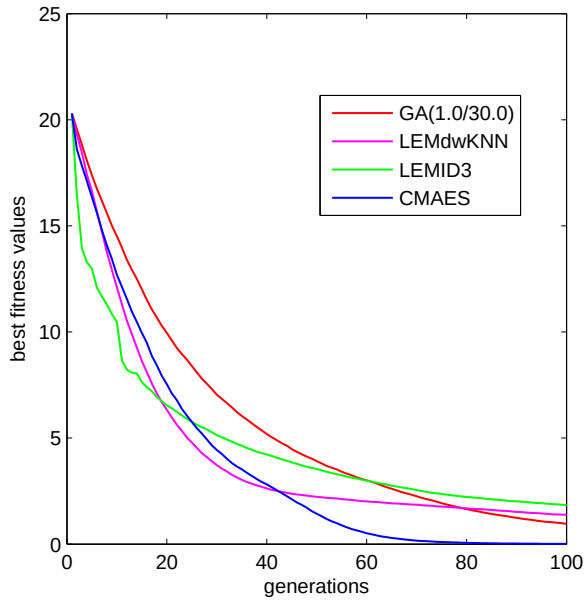


Figure 6.9: Results of running 4 algorithms on the Ackley problem

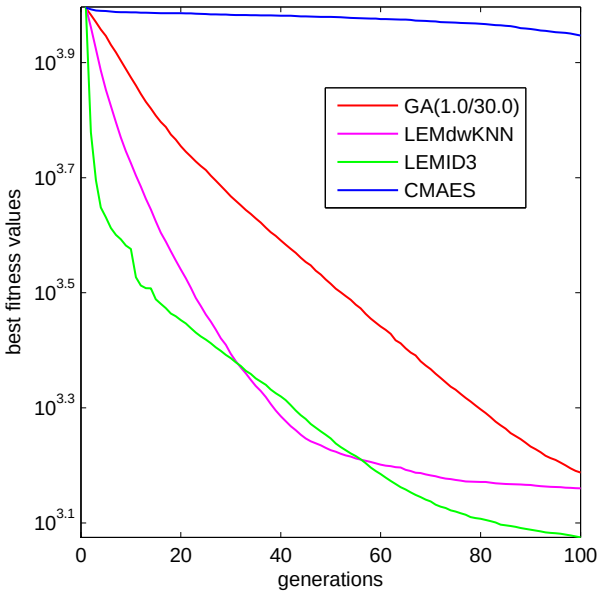


Figure 6.10: Results of running 4 algorithms on the Schwefel problem

From these experiment results in this study, we can see that LEM(ID3) outperforms the normal genetic algorithm GA2 (which is tuned based on the experience of GA1) in almost all generations for all problems, except for some cases for function DeJong4 and Rosenbrock at the final (100th) generations. LEM(ID3)'s advantage over GA2 is particularly evident and significant in the earlier generations of optimization, as seen in generations 10 and 20. LEM(ID3) also outperforms the other two hybrid algorithms, LEM(dwKNN) and CMAES, especially in the earlier 50 generations, except for Ackley and Rosenbrock functions. To highlight one of the most important advantages of the LEM(ID3) algorithm, we summarize that LEM(ID3) speeds up the optimization procedure strongly in the early generations by applying the ID3 learning and the instantiation algorithm, this advantage is crucial for many practical optimization problems where evaluations are expensive, and relative good quality feasible solutions are expected to be derived quickly.

6.3.2 Experiment Study 2

In this experiment study, we test the extended version of LEM(ID3) algorithm, the LEM(ID3)IER algorithm, on a more complex set of real-parameters function optimization problems, that is the test suite of 25 test problems for the CEC 2005 Special Session on Real-Parameter Optimization [SHL+05]. First, those real-parameter optimization problems are themselves more challenging in the character of their problems. Second, these optimization problems are tested by a number of advanced algorithms widely used in the evolutionary search op-

timization field, so it is handy for our LEM(ID3)IER algorithm to be compared with those contemporary algorithms. Therefore, we compare LEM(ID3)IER with these learning and evolution hybrid algorithms, particularly two advanced variant CMAES algorithms and a carefully-designed genetic algorithm with advanced genetic operators, instead of the standard genetic algorithm.

Test Functions

For the 2005 special sessions on real-parameter optimization, we have the following notes about the problem definitions which will reflect the complexity of these problems. Most of these problems are multimodal functions, only a few of them are unimodal; they are discontinuous problems and embedded with noise; they are shifted and rotated versions of the well-known optimization function, that is, the optimum is shifted and rotated randomly to different values, causing difficulties for many specific algorithms; they are expanded and hybrid composited functions which makes the function landscape even more complex to solve and extremely irregular; all of these functions are scalable to huge dimensions of search space. Finally, most of the algorithms compared there failed in finding the global optimum. For more details of these functions we refer to [SHL+05]. The number of solved problems for all algorithms in the competition are summarized as below:

1. Note that problems 1 to 6 are unimodal functions, and problems 7 to 25 are multimodal.

2. Also, the set of thirteen 10D problems {8, 13, 14, 16–25} were never 'solved' by any algorithm in the CEC 2005 competition, where 'solved' indicates reaching a certain level of accuracy specified in [SHL+05], which in turn was a function of the problem and its dimensionality. On 30D problems, problem 15 is another unsolved problem. On 50D problems, more problems are turned into the unsolved set.

3. Even for the 30D problems, the performances of the algorithms presented in the CEC 2005 session are not good enough.

Parameters Settings

The parameters settings for the LEM(ID3)IER algorithm is the same as the original LEM(ID3) algorithm's setting in Table 6.3, apart from two differences. First, there is an extra percentages set to indicate the number of new individuals being generated via learning, evolution,

and random selection, respectively. This set is given as (70%, 20%, and 10%). Second, there is no crossover operator in the current LEM(ID3)IER implementation, due to the consideration of verifying the capacity of diversity-preserving by mutation.

The algorithms used for comparison are the three algorithms with the best performance in the CEC 2005 competition. According to various quality criteria, these were IPOP-CMAES [AH05b], a restart version of CMAES with population resizing, also the dominant algorithm on this problem set so far; LR-CMAES [AH05a], an alternative local version of CMAES; and K-PCX [STD05], a carefully designed evolutionary algorithm with a specialized crossover operator (PCX). We have already introduced the general CMAES algorithm in Chapter 3, here we simply introduce the two variant CMAES algorithms and the K-PCX algorithm.

The IPOP-CMAES was developed based on the investigation of the impact of the population size of CMAES on multi-modal functions. Those investigations show that increasing the population size improves the performance on multi-modal functions, therefore in IPOP-CMAES, the restarting strategy with successively increasing population size is applied to solve the CEC optimization problems with very good performance. The IPOP-CMAES algorithm is also called (μ_W, λ)-CMAES, the parameters for the normal distribution are adapted based on the covariance matrix $\mathbf{C}$ for the next generation in the same way as in the CMAES algorithm. For the restart strategy, the (μ_W, λ)-CMAES is stopped, whenever one stopping criterion described below is met, and a restart is launched with the population size increased by a factor of 2. Therefore, in (μ_W, λ)-CMAES, the default values are used except for the population size, starting from the default value but then repeatedly increased. For completeness, we list the restart criteria:

- Stop if the overall change in the objective function value is below $Tolfun$ for $10 + \lfloor 30n/\lambda \rfloor$ generations.
- Stop if the standard deviation of the normal distribution is smaller than $TolX$ in all coordinates, and if the evolution path is smaller than $TolX$ in all components.
- Stop if the condition number of the covariance matrix exceeds 10^{14}.

LR-CMAES is developed to explore the performance of a restart local search strategy. To do this, the CMAES algorithm discussed before is cooperated with small initial step sizes, an initial step-size which is a hundred times smaller than is recommended as default. The default population size is applied. As a result, the algorithm can be then regarded as an

advanced local search, because the complete covariance matrix of the search distribution is efficiently adapted to the local topography of the objective function, and the step-size adaptation can result in comparatively large steps even when the initial step-size is chosen to be small.

K-PCX is a steady-state, population-based search algorithm for real parameter optimization. The main character of this algorithm is that it designed the main search components independently by defining four plans, the Selection Plan (SP), Generation Plan (GP), Replacement Plan (RP), Update Plan (UP). As the names of these plans suggest, they appear in different stages of the evolution search procedure. In each plan, the important aspects about solving multi-modal functions are considered, such as diversity preservation multi-modal parameters. K-PCX starts with an initial population generated randomly with the size N. Then, it uses the Selection Plan to choose μ parents from the initial population. In this selection scheme, first, sort the entire population in ascending order based on the function values. It then divides the population into k equal segments, where k is a user-defined parameter within the range 1 to N, indicating the extent of modality of the problem. For uni-modal problems, a small value and for multi-modal problems a large value of k is suggested. The best solution of each segment is picked and stored in $\mathcal{B}$. Then it randomly picks one solution from the set of best solutions with $\mathcal{B}$ as the first parent, this solution is also called the *index* solution. Thereafter, the other (μ - 1) parents are picked randomly from the population. In the Generation Plan, it creates λ offspring solutions from the chosen μ parent solutions by using the parent centric recombination (PCX) [DJA02] operator with modification for the purpose of recombination and producing λ offspring solutions. After describing the generation plan, next the Replacement Plan is to choose r solutions from the population. In the present scheme, the solutions are chosen randomly from the entire population, then a pool of size (r + λ) is formed, consisting of r solutions chosen from the population by the replacement plan and λ newly created offspring solutions by the generation plan. The current population is then updated using the Update Plan, in which r solutions chosen in the replacement plan are replaced by the best r solutions of the pool. This operation ensures an elite-preservation strategy.

For this K-PCX algorithm, the iteration continues until a prescribed number of function evaluations is achieved or a pre-defined termination criterion is met. If the diversity in the population is lost, cataclysmic mutation is used, and the best individual obtained so far is chosen as the index parent. Normally, the polynomial mutation with a mutation probability $p_m = 1/n$ is applied, where n is the number of real variables. K-PCX is an algorithm

designed specifically for complex function optimization, the parameter like k need to be indicated carefully and is problem-dependent. The bad choice for this parameter can make the algorithm perform poorly.

Summary of Results

We tested LEM(ID3)IER on all of the 25 problems in the CEC 2005 competition, for each of the 10-dimensional, 30-D, and 50-D cases (hence 75 problems altogether). Following the CEC2005 rules [SHL+05], 25 trials were run for each problem, and a variety of result indicators were recorded. Tables 6.8, 6.9 and 6.10 respectively show results for the 10D, 30D and 50D problems. In each case, we see the mean of 25 trials, reported for each of IPOP-CMAES, LR-CMAES, K-PCX, and LEM(ID3)IER. For the comparative algorithms, we take the mean results directly from the cited publications. Note that in the case of K-PCX, results for 50D problems were not reported.

If we observe the results summarized in Table 6.8 and compare the means, we see that IPOP-CMAES, LR-CMAES, K-PCX and LEM(ID3)IER respectively 'win' 13, 5, 4 and 6 of the contests on 10-dimensional functions. This includes some, but quite few, cases in which more than one of the algorithms shares the best mean for that problem. Table 6.9 shows the corresponding results for the 30D functions, and we now see that the numbers of 'wins' are 6, 4, 6 and 9 respectively for IPOP-CMAES, LR-CMAES, K-PCX and LEM(ID3)IER. As we scale from 10D to 30D, the relative performance of LEM(ID3)IER clearly seems to improve. Finally, although results for K-PCX on the 50D problems are not available, we note that the numbers of wins for IPOP-CMAES, LR-CMAES and LEM(ID3)IER on 50D problems are respectively 7, 7 and 11. A basic statistical analysis of these findings can be carried out using multinomial distributions. For example, if we assume that each algorithm has an equal chance of achieving a 'win' in the 30D case, then we find that the chance of a single algorithm achieving 9 or more wins has a probability of 0.15. In the case of 10D, simplifying the situation by ignoring problems 8 and 24, we find, analogously, that achieving 11 or more wins by chance from 23 four-way contests is 0.015. Finally, referring to the 50D case, the probability of achieving 11 or more wins in such a three way contest, assuming equal algorithm performance, is 0.18. The superiority of IPOP-CMAES in the 10D cases therefore seems significant, although LEM(ID3)IER achieves multiple wins on the 30D and 50D cases, the degrees of significance are less marked. However, the improvement in the relative performance of LEM(ID3)IER as we scale up is significant, and it seems clear that LEM(ID3)IER has

Table 6.8: Means for two CMAES, KPCX, LEM(ID3)IER, 10D, CEC05, 100K Evas.

Problems	IPOP-CMAES	LR-CMAES	K-PCX	LEM(ID3)IER
1	5.20e-9	5.14e-9	8.71e-9	**9.5497e-14**
2	4.70e-9	5.31e-9	9.40e-9	**1.18234e-13**
3	5.60e-9	**4.94e-9**	3.02e+4	4.75951e+4
4	**5.02e-9**	1.79e+6	7.94e-7	1.53131e-8
5	6.58e-9	**6.57e-9**	4.85e+1	1.08404e+2
6	**4.87e-9**	5.41e-9	2.07e+1	5.32101e+1
7	**3.31e-9**	4.91e-9	6.40e-2	7.82496e-2
8	**2.0e+1**	**2.00e+1**	**2.00e+1**	2.015e+1
9	2.39e-1	4.49e+1	1.19e-1	**3.52629e-7**
10	**7.96e-2**	4.08e+1	2.39e-1	4.73601e+0
11	9.34e-1	3.65e+0	9.11e+0	**2.97556e-3**
12	**2.93e+1**	2.09e+2	2.44e+4	3.30583e+1
13	6.96e-1	4.94e-1	6.53e-1	**3.00063e-1**
14	3.01e+0	4.01e+0	**2.35e+0**	2.52033e+0
15	2.28e+2	**2.11e+2**	5.10e+2	4.10562e+2
16	**9.13e+1**	1.05e+2	9.59e+1	9.91853e+1
17	1.23e+2	5.49e+2	**9.73e+1**	9.93189e+1
18	**3.32e+2**	4.97e+2	7.52e+2	5.40254e+2
19	**3.26e+2**	5.16e+2	7.51e+2	5.20259e+2
20	**3.00e+2**	4.42e+2	8.13e+2	6.40241e+2
21	5.00e+2	**4.04e+2**	1.05e+3	4.84205e+2
22	7.29e+2	7.40e+2	**6.59e+2**	7.43115e+2
23	**5.59e+2**	7.91e+2	1.06e+3	7.30581e+2
24	**2.00e+2**	8.65e+2	4.06e+2	**2.00064e+2**
25	**3.74e+2**	4.42e+2	4.06e+2	3.91368e+2

Table 6.9: Means for two CMAES, KPCX, LEM(ID3)IER, 30D, CEC05, 300K Evas.

Problems	IPOP-CMAES	LR-CMAES	K-PCX	LEM(ID3)IER
1	5.42e-9	5.28e-9	8.95e-9	**3.47882e-13**
2	6.22e-9	6.93e-9	1.44e-2	**1.65321e-10**
3	5.55e-9	**5.18e-9**	5.07e+5	2.72353e+5
4	1.11e+4	9.26e+7	**1.11e+3**	3.85297e+3
5	8.62e-9	**8.30e-9**	2.04e+3	3.13183e+3
6	**5.90e-9**	6.31e-9	9.89e+2	1.50812e+2
7	**5.31e-9**	6.48e-9	3.63e-2	2.95802e-2
8	2.01e+1	**2.00e+1**	2.00e+1	2.01516e+1
9	9.38e-1	2.91e+2	2.79e-1	**7.8419e-7**
10	1.65e+0	5.63e+2	**5.17e-1**	3.69056e+1
11	5.48e+0	1.52e+1	2.95e+1	**8.40942e-3**
12	4.43e+4	1.32e+4	1.04e+6	**4.91148e+3**
13	2.49e+0	2.32e+0	1.19e+1	**1.0437e+0**
14	1.29e+1	1.40e+1	1.38e+1	**1.20617e+1**
15	**2.08e+2**	2.16e+2	8.76e+2	3.6229e+2
16	**3.50e+1**	5.84e+1	7.15e+1	3.36537e+2
17	2.91e+2	1.07e+3	**1.56e+2**	3.10781e+2
18	9.04e+2	8.90e+2	**8.30e+2**	9.11234e+2
19	9.04e+2	9.03e+2	**8.31e+2**	9.10634e+2
20	9.04e+2	8.89e+2	**8.31e+2**	9.11151e+2
21	5.00e+2	**4.85e+2**	8.59e+2	5.00162e+2
22	**8.03e+2**	8.71e+2	1.56e+3	9.14701e+2
23	**5.34e+2**	5.35e+2	8.66e+2	5.41424e+2
24	9.10e+2	1.41e+3	2.13e+2	**2.00283e+2**
25	2.11e+2	6.91e+2	2.13e+2	**2.00294e+2**

Table 6.10: Means for two CMAES, KPCX, LEM(ID3)IER, 50D, CEC05, 500K Evas.

Problems	IPOP-CMAES	LR-CMAES	K-PCX	LEM(ID3)IER
1	5.87e-9	6.20e-9	-	**5.34328e-13**
2	7.86e-9	7.96e-9	-	**1.31335e-9**
3	6.14e-9	**6.04e-9**	-	2.11398e+5
4	4.68e+5	4.46e+8	-	**1.91824e+4**
5	**2.85e+0**	3.27e+0	-	9.82375e+3
6	7.13e-9	**7.12e-9**	-	1.12253e+2
7	**7.22e-9**	7.49e-9	-	1.48301e-2
8	2.01e+1	**2.00e+1**	-	2.01318e+1
9	1.39e+0	5.67e+2	-	**1.2652e-6**
10	**1.72e+0**	1.48e+3	-	1.17804e+2
11	1.17e+1	3.41e+1	-	**1.4332e-2**
12	2.27e+5	8.93e+4	-	**4.85485e+4**
13	4.59e+0	4.70e+0	-	**1.92194e+0**
14	2.29e+1	2.39e+1	-	**2.1577e+1**
15	**2.04e+2**	2.50e+2	-	4.04874e+2
16	**3.09e+1**	7.09e+1	-	9.8556e+1
17	2.34e+2	1.05e+3	-	**1.23137e+2**
18	9.13e+2	**9.06e+2**	-	9.38592e+2
19	9.12e+2	**9.11e+2**	-	9.40204e+2
20	9.12e+2	**9.01e+2**	-	9.40267e+2
21	1.00e+3	**5.00e+2**	-	6.07759e+2
22	**8.05e+2**	9.10e+2	-	1.00155e+3
23	1.01e+3	6.37e+2	-	**5.955e+2**
24	9.55e+2	8.43e+2	-	**2.00544e+2**
25	**2.15e+2**	4.77e+2	-	2.18497e+2

Table 6.11: Summary of solved problems by CEC05 session algorithms on 30D

	Number of solved problems	
	single-modal	multi-modal
IPOP-CMAES	6	5
K-PCX	3	4
LR-CMAES	5	1
LEM(ID3)IER	2	4
Other algorithms	≤ 4	≤ 2

promising properties with regard to scalability. Finally, with respect to the definition of 'successful run', which means that the algorithm achieves the fixed accuracy level (mean error values, $\leq 1e-6$ for unimodal; $\leq 1e-2$ for multimodal) within the maximum allowed evaluation number for the particular dimension, we summarize for the case of 30D the number of problems solved by the three algorithms being compared and LEM(ID3)IER in Table 6.11.

Meanwhile, in Appendix B we show Tables with the full set of result indicators (as specified in [SHL+05]) for LEM(ID3)IER on the 10D, 30D and 50D versions of the problems, to support comparative experiments of other researchers.

6.4 Concluding Discussion

Continuing to explore the LEM framework, we have described and evaluated our new LEM hybrid algorithms that combine evolutionary search with ID3 decision tree learning. In earlier work [SC08] [SC09], we found that hybridizations of quite simple learning strategies with evolutionary search were able to improve optimization performance considerably upon the unchanged EA, in particular, similar or better solution quality was achieved with significant savings in fitness evaluations. In this chapter, we examined a less simple, but still quite straightforward LEM variant algorithm in which decision tree learning, with instantiation generation and adaptive discretization, was interleaved with evolutionary search, and tested this approach on a number of test functions, especially the CEC 2005 real parameter optimization function suite. When compared with our well-tuned GA, KNN-based LEM hybrid algorithm LEM(dwKNN), and three of the best-performing function optimization algorithms previously published, we found that LEM(ID3) as the first version of our development has clear and significant advantages over the standard genetic

algorithm, this strongly proves the initial goal of our development of more LEM instance algorithms and verifies the claim again made by the original LEM authors, that LEM and its instance algorithms, like LEM(AQ), can speed up the traditional evolutionary optimization procedure to gain relative high-quality solutions. And this advantage also maintains over one of our LEM(KNN) algorithms developed earlier in this thesis, especially in the early stages of optimization, this feature of LEM(ID3) gives us more confidence in the development and application of Learning-and-Instantiating based LEM algorithms for practical complex optimization problems where the evaluations could be expensive. In another of our experiments on the CEC 2005 real parameter function optimization suite, as an extended version of the LEM(ID3) algorithm, LEM(ID3)IER is clearly competitive in performance with three dominating hybrid optimization algorithms in that competition, which are in fact particularly well-designed and well-tuned from their standard versions. Finally, one feature of LEM(ID3)IER which is worth mentioning is that its relative performance improves as problem dimensionality increases, with tentative evidence to suggest that it may be a recommended choice in general for high-D problems. With the performance tested here, we recommend LEM(ID3) algorithm as a baseline algorithm that should be further investigated and studied, more importantly, should be applied to solve more challenging problems in practice. Research-strength LEM(ID3)IER code is freely available at http://www.macs.hw.ac.uk/~gls3/LEMID3/LEMID3.zip.

Chapter 7

Cancer Chemotherapy Treatments Optimized by LEMs

7.1 Overview

In previous chapters, we have investigated new instance algorithms under the general LEM framework with the development of LEM(KNN), LEM(ED), LEM(ID3) and their variant algorithms. However, the test problems applied on these algorithms are all typical real-parameters function optimization problems. Although, they are particularly designed for the purpose of testing the performance of new optimization algorithms, they do not represent the practical optimization problems directly. Namely, we still did not apply our LEM instance algorithms to solve any practical hard optimization problems, and such an application is important and worthy. Practical problems may include more complex features which may reflect more real requirements to the problem solving algorithms than the well-defined testing functions. Practical problems may include more constraints which are the real considerations from the reality view. To satisfy these constraints during the optimization procedure, the search algorithms need to tackle more difficulties, and any violation of these constraints will make the derived solutions not feasible. Solving problems with constraint satisfaction is a more demanding task than the function optimization problems. Finally, our purpose of investigating LEM based hybrid optimization algorithms is to speed up the traditional evolutionary optimization procedure through the application of learning, and such a purpose is more strongly to be achieved due to the reality that many practical problems contain many practical aspects which make the evaluation of the solutions for the problems very expensive in real implementations or operations. Therefore, our developed LEM algo-

rithms are more suitable to solve these evaluation-expensive problems. As we have stated in Chapter 1, this property of speedup is very useful in solving many practical optimization problems, where the time complexity for working out the fitness of a single solution is quite poor. In this chapter, we investigate such a practical and complex problem which explores all of the aspects considered above, especially the property of time-consuming evaluations, with our LEM algorithms, LEM(dwKNN) and LEM(ID3). The problem is the Cancer Chemotherapy Treatments problem, solving this optimization problem provides a good test for our LEM algorithms in solving practical optimization problems.

In the following sections, we introduce the medical aspects of cancer treatment in Section 7.2, and give the mathematical formulation for this problem in Section 7.3, we solve this problem with our LEM instance algorithms in Section 7.4, and concludes in Section 7.5

7.2 Introduction

Amongst the modalities of cancer treatment, chemotherapy is often considered as inherently the most complex [Whe88]. As a consequence of this, it is extremely difficult to find effective chemotherapy treatments without a systematic approach. In order to realize such an approach, we need to take into account the medical aspects of cancer treatment.

Drugs used in cancer chemotherapy all have narrow therapeutic indices. This means that the dose levels at which these drugs significantly affect a tumour are close to those levels at which unacceptable toxic side-effects occur. Therefore, more effective treatments result from balancing the beneficial and adverse effects of a combination of different drugs, administered at various dosages over a treatment period [PM01].The beneficial effects of cancer chemotherapy correspond to treatment objectives which oncologists want to achieve by means of administering anti-cancer drugs. A cancer chemotherapy treatment may be either curative or palliative. Curative treatments attempt to eradicate the tumour; palliative treatments, on the other hand, are applied only when a tumour is deemed to be incurable, with the objective of maintaining a reasonable quality of life for as long as possible.

The adverse effects of cancer chemotherapy stem from the systemic nature of this treatment: drugs are delivered via the bloodstream and therefore affect all body tissues. Since most anti-cancer drugs are highly toxic, they inevitably cause damage to sensitive tissues elsewhere in the body. In order to limit this damage, toxicity constraints need to be placed on the amount of drug applied at any time interval, on the cumulative drug dosage over

the treatment period, and on the damage caused to various sensitive tissues [Whe88]. In addition to toxicity constraints, the tumour size (i.e. the number of cancerous cells) must be maintained below a lethal level during the whole treatment period for obvious reasons. The goal of cancer chemotherapy therefore is to achieve the beneficial effects of treatment objectives without violating any of the above mentioned constraints.

7.3 Mathematical Problem Formulation

In order to solve the optimization problem of cancer chemotherapy, we need to find a set of treatment schedules, which satisfies toxicity and tumour size constraints while also yielding acceptable values of treatment objectives. This set will allow the oncologist to make a decision on which treatment schedule to use, given his/her preferences or certain priorities. In the remainder of this section we will define the decision vectors and the search space for the cancer chemotherapy optimization problem, specify the constraints, and particularize the optimization objectives.

Anti-cancer drugs are usually delivered according to a discrete dosage program in which there are s doses given at times $t_1, t_2, \ldots t_s$ [MT94]. In the case of multi-drug chemotherapy, each dose is a cocktail of d drugs characterized by the concentration levels C_{ij},$i \in 1, \ldots, s$, $j \in 1, \ldots, d$ of anti-cancer drugs in the bloodplasma. Optimization of chemotherapeutic treatment is achieved by modification of these variables. Therefore, the solution space Ω of the chemotherapy optimization problem is the set of control vectors $\mathbf{c} = (C_{ij})$ representing the drug concentration profiles. However, not all of these profiles will be feasible, as chemotherapy treatment must be constrained in a number of ways. Although the constraint sets of chemotherapeutic treatment vary from drug to drug as well as with cancer types, they have the following general forms:

1. Maximum instantaneous dose C_{max} for each drug acting as a single agent:

$$g_1(c) = \{C_{max\,j} - Cij \geq 0 : \forall i \in 1 \ldots s, \forall j \in 1 \ldots d\} \tag{7.1}$$

2. Maximum cumulative C_{cum} dose for drug acting as a single agent:

$$g_2(c) = \{C_{cum\,j} - \sum_{i=1}^{s} Cij \geq 0 : \forall j \in 1 \ldots d\} \tag{7.2}$$

3. Maximum permissible size of the tumour:

$$g_3(c) = \{N_{max} - Nt_i \geq 0 : \forall i \in 1 \ldots s\} \tag{7.3}$$

4. Restriction on the toxic side-effects of multi-drug chemotherapy:

$$g_4(c) = \{C_{s-effk} - \sum_{j=1}^{d} \eta_{kj} Cij \geq 0 : \forall i \in 1 \ldots s, \forall k \in 1 \ldots m\} \tag{7.4}$$

The factors η_{kj} in the last constraint represent the risk of damaging the k^{th} organ or tissue (such as heart, bone marrow, lung etc.) by administering the j^{th} drug. Estimates of these factors for the drugs most commonly used in treatment of breast cancer, as well as the values of maximum instantaneous and cumulative doses, can be found in [DJR95].

Regarding the objectives of cancer chemotherapy, we focus our study on the primary objective of cancer treatment - tumour eradication. We define eradication to mean a reduction of the tumour from an initial size of around 10^9 cells (minimum detectable tumour size) to below 10^3 cells. In order to simulate the response of a tumour to chemotherapy, a number of mathematical models can be used [MT94]. The most popular is the Gompertz growth model with a linear cell-loss effect [Whe88]:

$$\frac{dN}{dt} = N(t) \cdot [\lambda \ln(\frac{\Theta}{N(t)}) - \sum_{j=1}^{d} K_j \sum_{i=1}^{s} C_{ij}\{H(t-t_i) - H(t-t_{i+1})\}] \tag{7.5}$$

where $N(t)$ represents the number of tumour cells at time t; λ ,Θ are the parameters of tumour growth, $H(t)$ is the Heaviside step function; k_j are the quantities representing the efficacy of anti-cancer drugs, and C_{ij} denote the concentration levels of these drugs. One advantage of the Gompertz model from the computational optimization point of view is that the equation (5) yields an analytical solution after the substitution $u(t) = \ln(\frac{\Theta}{N(t)})$ [MT94]. Since $u(t)$ increases when $N(t)$ decreases, the primary optimization objective of tumour eradication can be formulated as follows [Pet99]:

$$F(c) = \sum_{i=1}^{s} N(t_i) \tag{7.6}$$

subject to the Equation 7.5 and Constraints 7.1-7.4.

7.4 Solving using LEM Hybrid Algorithms

After having formatted the problem, we present the methods used to solve this problem. The methods used are evolutionary and hybrid search algorithms. For the evolutionary search methods, we use the standard genetic algorithm to optimize the treatment plans. For the hybrid algorithms, we apply the LEM hybrid algorithms developed in this thesis. This

problem has also been solved in [PSM06], and we include that work with EDA variant algorithm in this thesis as a comparison.

7.4.1 Problem Representation and Evaluation

After the mathematical formulation of the problem, we need to define the problem's representation space, before any evolutionary search based methods can be applied to solve this problem. Originally, the cancer chemotherapy optimization problem was solved using the binary representation of solutions. However, for the following two reasons, we apply integer representation for this problem in this thesis. First, it has been reported that integer encoding of GA solutions can improve the algorithm's performance [PBM05] by the original authors. Second, in order to make a fair comparison with the LEM hybrid algorithms, which are all developed for solving real parameters optimization problems.

For the convenience of illustration, we still begin by introducing the binary string representation of the problem. The multi-drug chemotherapy schedules problem is represented by decision vectors $c = (C_{ij}), i \in 1, \ldots s,\ j \in 1, \ldots d$, which are encoded as binary strings known as *chromosomes*. The representation space $\mathbf{I}$ (a discretized version of Ω) can then be expressed as a Cartesian product:

$$\mathbf{I} = A_1^1 \times A_1^2 \times \ldots \times A_1^d \times A_2^1 \times A_2^2 \times \ldots \times A_2^d \times \ldots \times A_s^1 \times A_s^2 \times \ldots \times A_s^d$$

of allele sets A_i^j. Each allele set uses a 4-bit representation scheme:

$$A_i^j = \{x_1 x_2 x_3 x_4 : x_k \in \{0, 1\} \forall k \in 1, \ldots 4\}$$

so that each concentration level C_{ij} takes an integer value in the range of 0 to 15 concentration units [PM01]. In general, with s treatment intervals and up to 2^p concentration levels for d drugs, there are up to 2^{spd} individual elements. Henceforth we assume that $s = 10$ and that the number of available drugs is restricted to ten [Pet99]. These drugs are delivered sequentially - one after another - to form a multi-drug dose, which is administered periodically over the treatment period that consists of up to s cycles. The values $s = 10$ and $d = 10$ result in the individual (search) space of power $|I| = 2^{400}$ individuals, referred to as chromosomes.

$$x = \{x_1 x_2 x_3 \ldots x_{4sd} : x_k \in \{0, 1\} \forall k \in 1, \ldots 4sd\}$$

and the mapping function $m : \mathbf{I} \rightarrow \mathbf{C}$ between the individual $\mathbf{I}$ and the decision vector $\mathbf{C}$ spaces can be defined as:

$$C_{ij} = \Delta C_j \sum_{k=1}^{4} 2^{4-k} x_{4d(i-1)+4(j-1)+k}, \forall i \in 1 \ldots s, j \in 1 \ldots d \tag{7.7}$$

where ΔC_j represents the concentration unit for drug j. This function symbolizes the decoding algorithm to derive a decision vector from a chromosome x. Applying the evaluation function F to $\mathbf{c}$ yields the value of the fitness function for both algorithms.

$$F(c) = \sum_{p=1}^{n} \sum_{j=1}^{d} k_j \sum_{i=1}^{p} C_{ij} e^{\lambda(t_{i-1}-t_p)} - \sum_{s=1}^{4} P_s d_s \tag{7.8}$$

where d_s are the distance measures specifying how seriously Constraints 7.1 - 7.4 are violated, and P_s are the corresponding penalty coefficients. If all constraints are satisfied (i.e. a treatment regime is feasible), then the second term in Equation 7.8 will be zero, significantly increasing the value of the fitness function.

7.4.2 Problem Solving and Results

We have given the definition of the representation space, and also the fitness evaluation function. As we have seen, the representation space of this problem can be transformed to real variable space, and there are constraints conditions on this real space. Therefore, the solutions are divided into two sets, one is the feasible solutions and the other is the infeasible solutions. When our evolutionary and hybrid optimization algorithms are applid for this constraint satisfaction problem, we have to consider and deal with the newly-generated solutions carefully, because these solutions could be within two differenct sets, feasible and infeasible. Our consideration is based on two ideas, the first, the feasible solutions are always prefered to the infeasible ones according to the problem task requirement; the second, the infeasible solutions can also contain good combinations of genes which can result in feasible solutions immediately in a few following generations, therefore they need to be explored as well. Based on these two ideas, our evolutionary and hybrid optimization algorithms have all been modified on the survival selection operations as follows:

- The evolution mode operate on the whole search space rather than is limited into the feasible regions only. When a new solution is generated, it is then tested by the Constraints 7.1 - 7.4 for its feasibility.

- During the learning mode (the instantiation operator), only unfeasible solutions can be replaced (by either feasible solutions or infeasible solutions), feasible solutions are never replaced by any solutions. During the evolution mode, survival selection is still based on fitness.

The C++ implementation codes for the evaluation function for this cancer chemotherapy optimization problem is given in Appendix C based on the above descriptions and formulae. Now, we can apply the LEM hybrid algorithms to solve this problem and make a comparison of the performance for this problem with other algorithms which have been applied to solve this problem. The algorithms involved are a standard genetic algorithm, a variant EDA algorithm called PBIL, the LEM(dwKNN) algorithm and the LEM(ID3) algorithm developed in Chapters 4 and 6, and also the CMAES algorithm. First, we give the complete description of the PBIL algorithm for the sake of completeness.

Population Based Incremental Learning

PBIL [Bal94, BD97] is a simple EDA variant algorithm, it is a non-dependence EDA, that is PBIL does not consider or model the dependence relationship between the variables. For the EDA algorithms and its classification, we refer to Chapter 3.

PBIL starts by initializing a probability vector $p = \{p_1, p_2, \ldots, p_n\}$ where each $p_i = 0.5$. Each p_i represents the probability of 1 being presented in i^{th} position of a chromosome. p is then sampled M times to create a population P of chromosomes. N chromosomes are then selected from P according to the quality or fitness value of that chromosome. As with GA, a number of selection mechanisms can be applied for this purpose. In the original investigation of PBIL for the cancer chemotherapy problem, the authors use truncation selection [LL02] which is to select the best N solutions from P. After selection, the marginal probability ρ_i for each i^{th} allele position is calculated from the selected N solutions. (ρ_i can be simply calculated by dividing the frequency of 1 in i^{th} position of allele in selected set by N). ρ_i is then used to update the probability vector p. This updated probability vector replaces the initial probability vector. This process continues until termination criteria are satisfied.

Parameters Settings for All Algorithms

The parameters settings for all the algorithms on the chemotherapy problem are listed here. All of the parameters are following the normal settings used in our previous chapters with-

Algorithm 15 pseudo code for PBIL

1: Initialize a probability vector $p = \{p_1, p_2, \ldots, p_n\}$, where each $p_i = 0.5$;
2: Sample p to generate an initial population P of size M;
3: Select the N fittest solutions from P where $N \leq M$;
4: For each allele x_i, calculate the marginal probability ρ_i from selected N solutions;
5: Update p using following updating rule:
6: **for** $i = 1$ to n **do**
7: $\quad p_i = p_i \times (1 - \lambda) + \rho_i \times \lambda$
8: **end for**
9: where, $0 \leq \lambda \leq 1$. λ is known as *learning rate parameter* chosen by the user.
10: Go to Step 2 until the termination criterion is satisfied.

out any particular tunings for this problem and previous runs. Therefore, we will not give all the details about the parameters and only give some general settings here. We refer to the corresponding chapters for complete details of the parameters, which are used here consistently.

- GAs: population size 100, crossover probability 0.6, mutation probability 1.0 for GA1 and 1/*length of chromosome* for GA2.
- PBIL: population size 100, selection size 20, learning rate $\lambda = 0.3$
- LEM(dwKNN): population size 100, learning threshold 0.3, initial discretization interval 3, multiplication factor is 2.
- LEM(ID3): population size 100, learning threshold 0.3, initial discretization interval 3, multiplication factor is 2.
- CMAES: (μ, λ) is set as (50,100) respectively, the initial mutation step size σ is set as one quarter of the whole search range.

Summary of Results

The performance of these algorithms is measured according to two main standards. The first standard is efficiency, which means the number of the fitness evaluations taken by the algorithms to find the first feasible solution. According to this standard, Table 7.1 shows the mean evaluation numbers expended by all these algorithms to find the first feasible solution, respectively. The second standard is the quality of the found solutions, which is quantified

by the best fitness values obtained by the algorithms. According to this standard, Table 7.2 gives the best fitness values of the found feasible solutions for all these algorithms at the maximum allowed evaluation number 200,000 after 1000 runs.

Table 7.1: Evaluation numbers for the first feasible solution: mean(sd)

Problems	GA1	GA2	PBIL	LEM(dwKNN)	LEM(ID3)	CMAES
Chemotherapy	12794(406.697)	5821(39.3107)	4871(620)	2930.4(52.3069)	3789(8.96655)	3815(11.5007)

Table 7.2: Best fitness values after 200k evaluation: mean(sd)

Problems	GA1	GA2	PBIL	LEM(dwKNN)	LEM(ID3)	CMAES
Chemotherapy	0.5605(0.0029)	0.572769(0.00270306)	0.428(0.112)	0.602562(0.00200748)	0.6055(0.0020)	0.6060(0.0020)

We can see from the results that LEM(dwKNN) and LEM(ID3) have very good performance both in the evaluation numbers and the quality of the solution found, compared with the standard well-tuned genetic algorithm GA2, PBIL and CMAES. It is interesting that the best algorithm with regard to the two standards for this problem is the LEM(dwKNN) algorithm, where only half of the evaluation number is needed to find the first feasible solution compared with the GA2 algorithm. Also, the speedup derived by LEM(ID3) algorithm over GA2 is also clear and significant. Finally, both of our LEM instance hybrid algorithms LEM(dwKNN) and LEM(ID3) algorithms are competitive to the hybrid optimization algorithms PBIL and CMAES. Such speedup derived by our LEM algorithms in optimization performance or savings of evaluations are certainly plausible for solving evaluation-expensive practical problems.

7.5 Concluding Discussion

We have investigated the application of our LEM instance algorithms to solve a practical complex optimization problem, the cancer chemotherapy treatments optimization problem. The main features of this problem are that it is complex on its own - when represented in integer encoding, its chromosome length is 100 with each gene value range from $(0 \ldots 16)$. This problem contains constraint satisfactions conditions, which put more limitations on

the search space. More importantly, this is an evaluation-expensive problem, due to the practical aspects of the problem.

Two advanced LEM hybrid algorithms developed in this thesis are applied on solving this cancer chemotherapy, and very good performances are derived. First, our LEM hybrid algorithms, based on KNN and ID3 decision tree learning, are able to solve practical complex optimization problems, this shows they are general problem solvers. The performances have beaten the corresponding GA significantly and are competitive with other advanced hybrid optimization algorithms, PBIL and CMAES, both in the evaluation number and quality of solutions. Based on these excellent performances of our LEM instance algorithms, we reclaim the fact that the original LEM authors had claimed before, that is, LEM based instance algorithms have significant advantages over the traditional evolutionary search algorithms, and this advantage can be proved not only on well-defined real-parameters function optimization problems but also remains for practical optimization problems with expensive evaluations and complex problem features.

Chapter 8

Conclusion

8.1 Summary

In this thesis, we have finished a series of important pieces of work on designing new hybrid algorithms and testing their performances. We summarize this work in this chapter.

Our first choice for a learning algorithm to be applied in the LEM framework is the *k*-nearest-neighbors learning. The reasons for this are that, first, KNN is a relatively simple algorithm to implement compared with the bulk of other learning paradigms in the machine learning community. Second, despite being simple, KNN is an excellent learning algorithm both in theoretical study and practical applications, particularly, it has a global view on the problem solutions space, this capacity is not universal for many other more complex learning algorithms. Simple does not mean incapable. Finally, based on these two reasons, KNN is our best choice to investigate LEM based hybrid optimization algorithms, and the effect of the development can be visible more quickly, due to KNN's excellency in learning capacity and efficiency in implementation.

LEM(KNN) and its variant algorithms, as the immediate results of development, significantly outperform the corresponding GAs in both speed and solution quality on a number of testing problems presented in this thesis, with the speed advantage being particularly impressive in general. Apart from the improved performances, we indicated two important aspects about the LEM framework, first, this framework is flexible, any PAC-learning algorithms can be applied in this framework to incorporate with the standard evolutionary search for solving optimization problems. This flexibility has been proved by the LEM(KNN) algorithms, where KNN replaces the AQ learning algorithm in the original LEM(AQ) algorithm. Furthermore, this flexibility is not limited by replacing one learning algorithm with another, it can also be reflected in the way by which learning and evolution interact.

LEM(KNN) once again shows this feature, the new individuals generated by evolution can only enter the population if the KNN learning judges them as 'good' individuals, otherwise they will be discarded.

These two contributions, in particular the latter, have opened the door for more possibilities of how the LEM based hybrid algorithms and even the learning and evolution hybrid algorithms should be developed. Learning can be used more flexibly to incorporate with evolution in more ways to achieve more varied performances.

The advantage of LEM(KNN) is that it can speed up the optimization procedure and save evaluations by using KNN learning method as the survival selection method to predict the 'good' or 'bad' for the new generated individuals rather than exactly evaluating them. However, the disadvantage of LEM(KNN) is that the prediction based on neighbors could make mistakes and therefore bring unfit individuals into the next generations. To overcome these drawbacks, we have tried two methods. One is the development of a 'verification' version of LEM(KNN), which results in the KNNGA(V) algorithm, and the other is the application of a distance-weight KNN algorithm, which results in the LEM(dwKNN) algorithm. The KNNGA(V) algorithm is not very successful in overcoming the disadvantage of LEM(KNN), because it uses more actual evaluations to verify the new generated individuals in order to exclude the unfit individuals, which inevitably violates the advantage and main goal of developing LEM(KNN) based methods. On the contrary, the LEM(dwKNN) algorithm seems very suitable to overcoming the drawbacks of LEM(KNN) and is therefore able to perform better than the LEM(KNN) algorithm. It judges the quality of the newly generated individuals through calculating an estimated fitness according to the k nearest neighbors, and verifies this individual using this estimated fitness against a predefined survival fitness. In this way, LEM(dwKNN) maintains the prediction capacity of LEM(KNN) while excluding the unfit individuals without any extra evaluations.

After the development of LEM(KNN) and its variant algorithms, we move our research focus to the concrete LEM(AQ) algorithm rather than the LEM framework. This change of the research focus is due, firstly to wanting to develop a complex and rule-based learning and evolution hybrid algorithm, which is equal to the LEM(AQ) algorithm both in the optimization performance and the functionalities of the algorithm. LEM(AQ), as the main instance algorithm of the LEM framework, has shown advantages in the optimization performance for a number of complex problems and also explored many advanced techniques into its algorithm design. To further evaluate this LEM algorithm and also compare it with our LEM(KNN) algorithm has become an urgent question to answer and is of the

utmost interest to us. Second, in the LEM framework, the core component which is believed to be the driving force of the promising performance of the LEM(AQ) algorithm, is the Learning-and-Instantiating method which was not verified by our development of the LEM(KNN) algorithms. This rule-based method remains interesting to us and merits more research efforts. Finally, through the development of our 'own' LEM(AQ) algorithm, the problems we want to solve at hand can be attempted, especially some practical problems with both complex problem features and expensive-evaluations costs.

However, the development procedure is not simple, the first step towards these goals is the LEM(ED) algorithm, which incorporates a simple Entropy-Based Discretization method as the learning component with a normal evolutionary procedure. The learning method ED applied here in LEM(ED) is a very simple mechanism compared with other well-known learning algorithms. ED simply takes the training data as input and use entropy measurement to find the best cut-points and therefore to identify the best interval to guide the generation of new individuals. Despite being able to outperform the standard GA algorithm in general, however, this advantage is limited and fades in the later stages of optimizations.

Although not very promising in the performance of LEM(ED), it is still a good attempt which may result in successful development of our LEM rule-based algorithm with excellent performances. The worthwhile experience derived from LEM(ED) is that, first of all, the learning method applied in LEM should be complex enough to distinguish the differences between variables, and therefore is able to find out the relationships between dimensions, which could be very important in the success of the optimization procedure. Second, the discretization method should fit the complex problem landscapes in an adaptive way. Ideal discretization should divide the variable range into several subranges and change adaptively according to the optimization procedure. Finally, the expected LEM algorithm should contain repeated learning and evolution interactions, containing many learning periods mediating the normal GA to finish the whole optimization procedure.

Based on the experience derived from the development of LEM(ED) and also on its development structure, continuing exploration on the LEM framework has resulted in the LEM(ID3) algorithm, which combines an evolutionary search with the ID3 decision tree learning. As with the LEM(KNN) algorithms, LEM(ID3) and its variant algorithms have achieved significant optimization performance on a number of real-parameters testing problems, including not only well-defined function problems, but also complex practical problems, over the standard GA algorithms, hybrid algorithms CMAES and variant of EDA algorithms. The successful development of LEM(ID3) verifies the importance of the Learning-

and-Instantiating method within the LEM framework, and also points out that this method can be efficient in achieving promising performances only if it is applied with many other techniques, such as the good design of the instantiation algorithm, forest model, rule selection method, and discretization methods. Without the good designs for these techniques, the successful development of LEM(ID3) cannot be expected. Finally, the LEM(ID3) algorithm should be used as a baseline algorithm that will be further investigated and studied, more importantly, should be applied to solve more challenging problems in practice.

The practical application oriented problem we solve with our LEM instance algorithms is the cancer chemotherapy treatments optimization problem. The main features of this problem are that, it is complex in its own, when represented in integer encoding, its chromosome length is 100 with each gene value ranges from $(0 \ldots 16)$. This problem contains constraint satisfactions conditions, which put more limitations on the search space. More important, this is an evaluation-expensive problem, due to the practical aspects related to the cancer chemotherapy problem. All of these features can pose challenges for our LEM hybrid optimization algorithms, and question them as general problem solvers. However, with the successful application of our LEM algorithms on this problem, especially, two LEM hybrid algorithms developed in this thesis, very good performance results are derived. These performances have beaten the corresponding GA significantly and are competitive with other hybrid optimization algorithms, PBIL and CMAES, both in the evaluation number and quality of solutions.

Based on these excellent performances of our LEM instance algorithms, we reclaim what the original LEM authors had claimed before, that is that LEM based hybrid optimization algorithms developed in this thesis have significant advantages over the traditional evolutionary search algorithms, and these advantages remain to variants of CMAES and EDA algorithms on both well-defined real-parameters functions and practical complex optimization problems with expensive evaluations features.

8.2 Contributions

As stated in Chapter 1 Section 1.1.5, the contributions of this thesis are restated as follows:

Contribution 1 A simple genetic algorithm combined with k-nearest-neighbor learning algorithm, called LEM(KNN), is developed. KNN in this LEM instance algorithm is used as a 'filter' deciding the survival of the newly generated individuals. Also, a variation of the LEM(KNN) algorithm, called LEM(dwKNN), is developed. LEM(dwKNN)

extends LEM(KNN) with the consideration of distance contributions. The performances of these algorithms are compared with the standard genetic algorithms, showing that significant improvements can be achieved by hybridizing even these very simple learning algorithms with the normal evolution algorithms.

Contribution 2 Simple genetic algorithm combined with Entropy-Based Discretization (ED), ID3 decision tree learning algorithm, and their variant algorithm are developed, respectively. These algorithms are all designed under the general LEM framework and are based on the Learning-and-Generating Hypotheses method, showing the flexibility of this framework. With the development of these LEM instance algorithms, we have also investigated different techniques and methods which are important components of the hybrid algorithms and affect the functions and performances of the hybrid algorithms.

Contribution 3 The resulting algorithms LEM(KNN), LEM(ID3) and their variant algorithms are compared with other hybrid algorithms, such as CMAES and EDA, on a number of test problems, including the CEC 2005 real-parameter functions optimization and the cancer chemotherapy optimization problem. Performance on these problems have shown that these LEM instance algorithms are promising and compete well against state of the art hybrid algorithms.

Contribution 1 was explored in Chapter 4, where LEM(KNN) and LEM(dwKNN) were described on pages 85 and 101, and the experiments were carried out that compared them with the standard genetic algorithm and the CMAES algorithm. These experiments show that LEM(KNN) and LEM(dwKNN) have significant advantages over the traditional evolutionary procedures and are competitive with the adaptive mutation step sizes strategy CMAES.

Contribution 2 was explored in Chapter 5 and Chapter 6, where LEM(ED) and LEM(ID3) were described on pages 127 and 145, and the experiments were done that compared them with GA, LEM(KNN), LEM(dwKNN) and CMAES. These experiments show that both LEM(ED) and LEM(ID3) can beat standard GAs in the earlier stages of optimization, and LEM(ID3) is superior to all the other algorithms on real-parameters function test set 'Test Suite 2'.

Contribution 3 was explored in Chapter 4, Chapter 6, and Chapter 7, respectively. Where the experiments were described on pages 159, 178, and the experiments were carried out that compared LEM(ID3)IER with other advanced GA and CMAES variant al-

gorithms. These experiments show that LEM(ID3)IER is competitive to these variant algorithms on real-parameters optimization functions in CEC 2005 competition, and also shows that LEM(dwKNN) and LEM(ID3) can solve the cancer chemotherapy treatments optimization problem with promising performances over traditional GA, CMAES and PBIL algorithms both in speed and quality of the solutions derived.

8.3 Future Work

Based on what we have investigated in this thesis, we will have the following work on which to carry out more research on the development of LEM based hybrid optimization algorithms.

1. More investigations of different learning algorithms in the LEM framework are needed. This will provide us with more data and experience that will help to guide a general theory on how best to construct a LEM instantiation for a given problem. We have applied several learning methods in our LEM instance algorithms, however, they are neither sufficiently representative nor advanced in the machine learning community. We need to explore more learning techniques in our future research.

2. More experiments and explorations on the interaction between learning and evolution phases need to be investigated. These interaction principles are central to the development of any hybrid optimization algorithms. However, the inspiration cannot be immediate and needs long-term research into both the learning and evolution sides with their own novel features in order to be understood in more depth.

3. More attempts on adaptive and multi-learner versions of LEM, where different learning phases may have different learning algorithms, should be investigated. For example, in the early and later stages of evolutionary search, the learning algorithms could change or switch from one to another according to the nature and difficulties of the optimization task at the various stages.

Appendix A

Brief Introduction on Probability

To represent and reason with uncertain knowledge, a formal language needs to be developed to deal with two issues: the degree of belief for assertion of a probability and dependence of the degree of belief which includes evidence and experience. Probability theory typically uses a language that is slightly more expressive than the proposition logic, and less expressive than first order logic. The basic element of this language is called a *random variable*, which is a map from an event in the real world to a value. Each random variable has a domain of values that it can take on, the value can be of many types, such as binary, discrete, and continuous. Once a value of the domain is assigned to the random variable, it means one event or some events have happened. For example, *Headache* is a random variable, its meaning is that someone has a headache, we could assign it two binary values $\langle true, false \rangle$, (Headache = true) means a headache happens to that person, or vice versa. *Weather* is another random variable with the domain $\langle sunny, rainy, cloudy, and snowy \rangle$ and *Length* is a random variable with real number domain $\langle 0.0, \ldots, 1.0 \rangle$. An event is exact assignment of the random variable, for example, (Headache = true), (weather = cloudy) are all events.

Prior or *unconditional probability* of an event is the frequency of the event which happened in a number of experiments. For example, for a year's observation, the probability of raining in a district can be represented as simply P(Weather = raining) = 0.3. Probability cannot be negative, and for all domains the sum of the probabilities for each value is 1.0. That is, P(Weather = sunny) = 0.5, P(Weather = raining) = 0.3, P(Weather = cloudy) = 0.05, and P(Weather = snow) = 0.15. Also, $P(Weather) = \{0.5, 0.3, 0.05, 0.15\}$ defines a probability distribution for the random variable *Weather*. Meanwhile, P(Weather, Headache) denotes the probabilities of all combinations of the values of the set of random variables, Weather and Headache, which is a 4×2 table of probabilities. This is called the *joint prob-*

ability distribution of Weather and headache. Generally, the joint probability distribution of some random variables $X_1, X_2, \ldots, X_n$ are indicated as, $P(X_1, X_2, \ldots, X_n)$.

For continuous random variables, it is not possible to write out the entire distribution as a table, because there are infinitely many values. Also, for a continuous random variable, the probability for one value happens to be a particular value x_0 is always 0.0. For continuous random variables, we are always concerned that the probability lies in a certain interval, although this interval can be very small. So, we have:

$$P(a < X \leq b) = \int_a^b f(x)dx \tag{A.1}$$

where X is the continuous random variable, $f(x)$ is called the *probability density function* for X.

If we want to talk about the probability given some evidence, that is, the probability when some evidence or events have happened, then the unconditional probability is not applicable anymore. We use the *conditional probability* indicated as:

$$P(a|b) = \frac{P(a \wedge b)}{P(a)} \tag{A.2}$$

where $P(a|b)$ means the probability of a, given that evidence b. For example, $P(Headache = true|Weather = raining)$ indicates the probability of a person having a headache, when the weather is rainy. Conditional probability is defined by unconditional probabilities, and can also be written as:

$$P(a \wedge b) = P(b|a)P(a) \tag{A.3}$$

or

$$P(a \wedge b) = P(a|b)P(b) \tag{A.4}$$

These two formulas are also called *rule product*. The more general joint probability distribution form for this rule product now becomes:

$$P(X, Y) = P(X|Y)P(Y) \tag{A.5}$$

What will happen when one random variable X does not influence the other random variable Y? That is the two random variables are independent to each other, then we have random variables' *independence property*:

$$P(X|Y) = P(X) \tag{A.6}$$

$$P(Y|X) = P(Y) \tag{A.7}$$

$$P(X, Y) = P(X)P(Y) \tag{A.8}$$

So far, the syntax of probability propositions for the prior and conditional probability statements are defined. However, the semantics for probability inference statements remains. We begin with the basic axioms that serve to define the probability scale and its endpoints:

1. All probabilities are between 0.0 and 1.0, for any proposition a,

$$0.0 \leq P(a) \leq 1.0 \tag{A.9}$$

2. Necessarily true propositions have probability 1.0, and necessarily false propositions have probability 0.0.

$$P(true) = 1.0 \tag{A.10}$$

$$P(false) = 0.0 \tag{A.11}$$

3. The probability of a disjunction is given by

$$P(a \vee b) = P(a) + P(b) - P(a \wedge b) \tag{A.12}$$

   This rule states that the cases where a holds, together with the cases where b holds, certainly cover all the cases where $a \vee b$ holds; but summing the two sets of cases counts their intersection twice, so we need to subtract $P(a \wedge b)$.

These three axioms are often called *Kolmogorov's axioms*, which showed how to build up the rest of probability theory from this simple foundation.

Before we reach the Bayesian Network inference, we first introduce a probabilistic inference method based on the full joint distribution. Assume we have the full joint distribution of a number of random variables $X_1 X_2 \ldots X_n$. Then, we can calculate any proposition from this variables set by adding any atomic events in which the proposition is true, and

add up their probabilities. There are two very useful rules called the *marginalization* rule and the *conditioning* rule:

$$P(X) = \sum_{y} P(X, y) \text{ (marginal probability)} \tag{A.13}$$

$$P(X) = \sum_{y} P(X|y)P(y) \text{ (conditioning probability)} \tag{A.14}$$

The marginal probability of random variable X is the procedure of summing out all the other variables from any joint distribution containing Y. The conditioning rule can be seen as a variant of the marginalization rule, and it involves conditional probabilities instead of joint probabilities using product rule. In most cases, we will be interested in computing conditional probabilities of some variables, given evidence about others. Conditional probabilities can be found by unconditional probabilities and then evaluating the expression from the full joint distribution. Then the general inference procedure can be formulated as:

Let X be the query variable, let E be the set of evidence variables, let e be the observed values for them, and let Y be the remaining unobserved variables. The query $P(X|e)$ can be evaluated as the queries of probability:

$$P(X|e) = \alpha P(X, e) = \alpha \sum_{y} P(X, e, y) \tag{A.15}$$

This formula gives the general form of inference for answering probabilistic queries for discrete variables, given the full joint distribution. In principle, the full joint distribution is capable of answering any query, however, it is not efficient, for a domain described by n boolean variables, it requires an input table of size $O(2^n)$. In a realistic problem, there might be hundreds or thousands of random variables to consider. It quickly becomes completely impractical to define the vast numbers of probabilities required. For this reason, the full joint distribution is not a practical tool for building reasoning systems. Instead, it should be viewed as the theoretical foundation on which more effective approaches may be built. The bayesian network is one of such more efficient techniques for inference.

Appendix B

LEM(ID3)IER Performance on CEC2005 Test Functions

Table B.1: Error values at FEs = 1e3, 1e4, 1e5 for problems 1-9(10D)

FE	Prob	1	2	3	4	5	6	7	8	9
1e3	1st(Best)	1.76627e+3	2.39601e+3	1.38448e+7	3.23016e+3	5.85876e+3	2.2349e+7	7.20519e+1	2.04625e+1	4.74269e+1
	7st	2.0475e+3	4.72171e+3	3.57553e+7	5.11679e+3	7.8962e+3	1.5933e+8	1.60004e+2	2.05996e+1	6.04818e+1
	13st(Median)	2.43771e+3	5.81598e+3	4.56641e+7	7.05193e+3	8.71907e+3	2.49027e+8	2.24438e+2	2.07639e+1	6.5823e+1
	19st	3.0907e+3	6.62843e+3	5.39912e+7	8.84371e+3	9.42126e+3	3.24026e+8	2.86345e+2	2.08247e+1	7.34762e+1
	25st(Worst)	4.16712e+3	1.0272e+4	6.82112e+7	1.20544e+4	1.0052e+4	5.01842e+8	7.28072e+2	2.09604e+1	8.31533e+1
	Mean	2.60096e+3	5.84884e+3	4.41267e+7	7.18504e+3	8.52183e+3	2.44197e+8	2.44954e+2	2.07373e+1	6.65439e+1
	Std	6.86132e+2	1.70132e+3	1.53174e+7	2.5182e+3	1.11491e+3	1.26305e+8	1.30463e+2	1.34418e-1	8.70724e+0
1e4	1st(Best)	5.13795e+0	1.93259e+1	1.54573e+5	4.07703e+1	5.26389e+2	7.1199e+3	4.41663e-1	2.02456e+1	8.06948e+0
	7st	1.00512e+1	5.59889e+1	4.06354e+5	7.04e+1	7.9056e+2	3.03004e+4	7.43838e-1	2.04018e+1	1.03437e+1
	13st(Median)	1.25371e+1	6.62806e+1	5.95762e+5	8.63e+1	9.09958e+2	4.55171e+4	8.05048e-1	2.05315e+1	1.1874e+1
	19st	1.4646e+1	9.17244e+1	8.51530e+5	1.09733e+2	9.89365e+2	7.59082e+4	8.8556e-1	2.05635e+1	1.37784e+1
	25st(Worst)	4.26447e+1	1.26841e+2	1.7486e+6	1.47519e+2	1.12607e+3	1.51208e+5	1.24752e+0	2.06824e+1	1.69805e+1
	Mean	1.38463e+1	7.11185e+1	7.10946e+5	8.81647e+1	8.85757e+2	5.5315e+4	7.94232e-1	2.04899e+1	1.21247e+1
	Std	7.3711e+0	2.59667e+1	4.15420e+5	2.99544e+1	1.58884e+2	3.61606e+4	1.71467e-1	1.08675e-1	2.44804e+0
1e5	1st(Best)	5.68434e-14	5.68434e-14	1.34808e+4	2.28096e-9	3.45007e+0	8.17715e-3	1.969e-6	2.00251e+1	1.03587e-7
	7st	5.68434e-14	5.68434e-14	2.77637e+4	8.3237e-9	2.73404e+1	1.37221e-2	4.68252e-2	2.00542e+1	1.79642e-7
	13st(Median)	1.13687e-13	1.13687e-13	3.68286e+4	1.3652e-8	1.11132e+2	7.15347e-1	7.87038e-2	2.01197e+1	3.01609e-7
	19st	1.13687e-13	1.7053e-13	6.96483e+4	1.88572e-8	1.55475e+2	8.48882e+1	1.08172e-1	2.02052e+1	3.95465e-7
	25st(Worst)	1.7053e-13	2.27374e-13	1.11079e+5	4.08672e-8	2.61796e+2	2.44872e+2	1.74597e-1	2.0408e+1	1.20204e-6
	Mean	9.5497e-14	1.18234e-13	4.75951e+4	1.53131e-8	1.08404e+2	5.32101e+1	7.82496e-2	2.015e+1	3.52629e-7
	Std	3.84524e-14	5.31296e-14	2.76208e+4	9.87347e-9	7.58561e+1	7.6474e+1	4.00416e-2	1.13729e-1	2.69572e-7

Table B.2: Error values at FEs = 1e3, 1e4, 1e5 for problems 10-17(10D)

FE	Prob	10	11	12	13	14	15	16	17
1e3	1^{st}(Best)	6.03503e+1	9.24089e+0	1.73259e+4	6.72208e+0	3.9688e+0	6.00501e+2	2.24909e+2	2.90355e+2
	7^{st}	7.93597e+1	1.04947e+1	2.85826e+4	8.73893e+0	4.21752e+0	6.71116e+2	2.87828e+2	3.47873e+2
	13^{st}(Median)	8.57247e+1	1.08896e+1	3.4430e+4	9.26566e+0	4.29112e+0	6.88087e+2	3.16287e+2	3.7172e+2
	19^{st}	9.0203e+1	1.11844e+1	4.16826e+4	1.05651e+1	4.44604e+0	6.98248e+2	3.34822e+2	3.81825e+2
	25^{st}(Worst)	1.01031e+2	1.23842e+1	6.47692e+4	1.36315e+1	4.49775e+0	7.33734e+2	3.65627e+2	4.01018e+2
	Mean	8.32572e+1	1.08257e+1	3.59067e+4	9.60275e+0	4.30392e+0	6.86153e+2	3.1068e+2	3.63629e+2
	Std	9.59376e+0	8.22642-1	1.18429e+4	1.68765e+0	1.41298e-1	2.78322e+1	3.17017e+1	2.79814e+1
1e4	1^{st}(Best)	1.71368e+1	1.99912e-1	3.21224e+2	4.38839e-1	2.62867e+0	4.54592e+2	1.30139e+2	1.47817e+2
	7^{st}	2.3299e+1	2.78322e-1	4.98497e+2	8.45902e-1	3.11283e+0	5.18905e+2	1.52923e+2	1.67544e+2
	13^{st}(Median)	3.01671e+1	3.27077e-1	6.70841e+2	9.2815e-1	3.27107e+0	5.28368e+2	1.65368e+2	1.84851e+2
	19^{st}	3.21286e+1	3.61697e-1	1.04811e+3	1.2037e+0	3.49629e+0	5.52708e+2	1.70085e+2	1.97498e+2
	25^{st}(Worst)	3.76535e+1	4.42254e-1	1.88115e+3	1.74414e+0	3.72812e+0	5.67095e+2	1.80192e+2	2.12473e+2
	Mean	2.8402e+1	3.25925e-1	8.30122e+2	1.02803e+0	3.26201e+0	5.3038e+2	1.61226e+2	1.82878e+2
	Std	5.54662e+0	6.04629e-2	4.31466e+2	3.24871e-1	2.82177e-1	2.59343e+1	1.28016e+1	1.7353e+1
1e5	1^{st}(Best)	1.98992e+0	2.18436e-3	1.43965e-4	1.66616e-1	1.50405e+0	3.28553e+2	6.21503e+1	2.06999e-1
	7^{st}	2.98488e+0	2.52192e-3	2.77244e-4	2.46125e-1	2.20657e+0	4.0068e+2	9.72366e+1	9.83996e+1
	13^{st}(Median)	4.9748e+0	2.93663e-3	9.43502e-2	2.96311e-1	2.53041e+0	4.10353e+2	1.02549e+2	1.04368e+2
	19^{st}	5.96975e+0	3.32401e-3	1.00034e+1	3.39652e-1	2.91899e+0	4.17695e+2	1.06642e+2	1.09433e+2
	25^{st}(Worst)	7.95967e+0	4.78371e-3	7.12254e+2	4.91012e-1	3.49936e+0	4.4737e+2	1.14427e+2	1.17578e+2
	Mean	4.73601e+0	2.97556e-3	3.30583e+1	3.00063e-1	2.52033e+0	4.10562e+2	9.91853e+1	9.93189e+1
	Std	1.71825e+0	5.99007e-4	1.38825e+2	8.12673e-2	4.68469e-1	2.03516e+1	1.30559e+1	2.33564e+1

Table B.3: Error values at FEs = 1e3, 1e4, 1e5 for problems 18-25(10D)

FE	Prob	18	19	20	21	22	23	24	25
1e3	1^{st}(Best)	1.04381e+3	9.83915e+2	9.65748e+2	1.26662e+3	9.71363e+2	1.03734e+3	8.9836e+2	1.5985e+3
	7^{st}	1.0939e+3	1.09785e+3	1.09047e+3	1.31223e+3	1.02432e+2	1.29795e+3	1.17436e+3	1.66137e+3
	13^{st}(Median)	1.10849e+3	1.11893e+3	1.1134e+3	1.3191e+3	1.04612e+2	1.31966e+3	1.20427e+3	1.69936e+3
	19^{st}	1.13111e+3	1.13196e+3	1.12525e+3	1.33983e+3	1.06773e+2	1.33677e+3	1.25144e+3	1.73652e+3
	25^{st}(Worst)	1.15192e+3	1.16046e+3	1.16109e+3	1.35745e+3	1.12643e+2	1.34714e+3	1.29982e+3	1.81718e+3
	Mean	1.10981e+3	1.10992e+3	1.10248e+3	1.319e+3	1.04432e+3	1.29423e+3	1.18882e+3	1.70262e+3
	Std	2.53905e+1	3.89784e+1	4.09853e+1	2.41123e+1	3.54599e+1	7.27649e+1	9.60612e+1	5.51437e+1
1e4	1^{st}(Best)	4.04967e+2	3.82633e+2	3.99736e+2	3.55975e+2	4.80714e+2	5.59505e+2	2.07512e+2	2.00489e+2
	7^{st}	4.64447e+2	4.40013e+2	4.7628e+2	5.26191e+2	7.81811e+2	5.62769e+2	2.29216e+2	4.10639e+2
	13^{st}(Median)	4.99639e+2	4.76394e+2	8.07968e+2	5.33244e+2	7.87769e+2	7.37565e+2	2.40137e+2	4.11148e+2
	19^{st}	8.0845e+2	8.07145e+2	8.11529e+2	5.65451e+2	7.90547e+2	7.68824e+2	2.49739e+2	4.11611e+2
	25^{st}(Worst)	8.15605e+2	8.13793e+2	8.17476e+2	1.06731e+3	8.53264e+2	1.0888e+3	2.70228e+2	4.1342e+2
	Mean	6.24096e+2	6.02761e+2	6.95493e+2	6.00961e+2	7.79636e+2	7.53007e+2	2.39711e+2	3.94488e+2
	Std	1.79093e+2	1.85373e+2	1.6831e+2	1.88466e+2	6.34221e+1	1.76176e+2	1.5304e+1	5.72005e+1
1e5	1^{st}(Best)	3.00244e+2	3.00127e+2	3.0018e+2	3.0019e+2	3.00411e+2	5.59468e+2	2.00035e+2	2.00e+2
	7^{st}	3.00295e+2	3.00283e+2	3.0036e+2	3.00427e+2	7.52446e+2	5.59469e+2	2.00055e+2	4.07417e+2
	13^{st}(Median)	3.00387e+2	3.00408e+2	8.00185e+2	5.00053e+2	7.5516e+2	7.21227e+2	2.00067e+2	4.07713e+2
	19^{st}	8.00211e+2	8.00187e+2	8.0025e+2	5.00064e+2	7.60027e+2	7.21234e+2	2.00072e+2	4.08418e+2
	25^{st}(Worst)	8.00314e+2	8.00288e+2	8.00346e+2	8.00572e+2	8.14518e+2	1.0888e+3	2.00084e+2	4.09425e+2
	Mean	5.40254e+2	5.20259e+2	6.40241e+2	4.84205e+2	7.43115e+2	7.30581e+2	2.00064e+2	3.91368e+2
	Std	2.4976e+2	2.48148e+2	2.33218e+2	1.6417e+2	9.2104e+1	1.65589e+2	1.28788e-2	5.6435e+1

Table B.4: Error values at FEs = 1e3, 1e4, 1e5, 3e5 for problems 1-9(30D)

FE	Prob	1	2	3	4	5	6	7	8	9
1e3	1^{st}(Best)	1.84816e+4	5.80713e+4	4.28226e+8	5.0508e+4	1.82683e+4	2.75641e+9	2.90741e+3	2.10927e+1	2.70007e+2
	7^{st}	2.25798e+4	7.29666e+4	5.86475e+8	9.13281e+4	2.0474e+4	5.1011e+9	4.35936e+3	2.11867e+1	2.85565e+2
	13^{st}(Median)	2.64077e+4	8.42053e+4	7.15101e+8	1.01778e+5	2.17553e+4	6.044e+9	4.86979e+3	2.12448e+1	2.925 e+2
	19^{st}	2.70371e+4	9.20998e+4	7.89873e+8	1.12835e+5	2.25345e+4	6.89223e+9	5.62334e+3	2.12742e+1	3.00243e+2
	25^{st}(Worst)	3.03603e+4	1.16946e+5	9.85705e+8	1.23943e+5	2.50816e+4	9.12403e+9	7.92250e+3	2.13298e+1	3.13635e+2
	Mean	2.52901e+4	8.36879e+4	7.03711e+8	1.00589e+5	2.13896e+4	6.05625e+9	5.11976e+3	2.12319e+1	2.9203e+2
	Std	3.16511e+3	1.40996e+4	1.49345e+8	1.54325e+4	1.80975e+3	1.46572e+9	1.1333e+3	6.07645e-2	1.21531e+1
1e4	1^{st}(Best)	1.20915e+2	5.21172e+3	1.30194e+7	5.83001e+3	4.72914e+3	1.49874e+7	1.86878e+1	2.09447e+1	5.77231e+1
	7^{st}	3.16374e+2	7.21918e+3	2.69825e+7	1.04962e+4	5.10548e+3	2.55644e+7	3.04849e+1	2.10612e+1	8.0596e+1
	13^{st}(Median)	4.36052e+2	8.88937e+3	3.39962e+7	1.34794e+4	5.43215e+3	3.24627e+7	4.19273e+1	2.10919e+1	8.96532e+1
	19^{st}	5.08726e+2	1.00818e+3	3.80224e+7	1.92009e+4	5.98524e+3	3.9388e+7	4.96740e+1	2.11286e+1	9.72164e+1
	25^{st}(Worst)	6.55245e+2	1.78049e+3	5.22573e+7	2.64618e+4	6.72032e+3	5.16719e+7	7.07495e+1	2.11979e+1	1.56192e+2
	Mean	4.08674e+2	8.90177e+3	3.32012e+7	1.49361e+4	5.58086e+3	3.20665e+7	4.20861e+1	2.10885e+1	9.20516e+1
	Std	1.27406e+2	2.52734e+3	9.00112e+6	5.62299e+3	5.56515e+2	9.61939e+6	1.35805e+1	5.68839e-2	2.02191e+1
1e5	1^{st}(Best)	2.27374e-13	2.09752e-10	1.72283e+5	6.38446e+2	2.32731e+3	1.28609e+1	8.14726e-6	2.01207e+1	5.35201e-6
	7^{st}	2.82217e-13	3.00207e-9	4.18630e+5	3.0072e+3	2.83805e+3	1.74484e+1	9.87778e-3	2.01977e+1	8.25543e-6
	13^{st}(Median)	3.41061e-13	1.24215e-8	8.53689e+5	5.06429e+3	2.98984e+3	1.50646e+2	2.45918e-2	2.02462e+1	1.02973e-5
	19^{st}	3.94904e-13	3.78838e-8	1.10303e+6	7.99879e+3	3.39687e+3	3.11994e+2	3.69159e-2	2.02938e+1	1.26634e-5
	25^{st}(Worst)	5.11591e-13	6.53588e-6	1.86449e+6	1.49205e+4	4.75612e+3	2.3377e+3	9.77006e-2	2.03878e+1	1.90699e-5
	Mean	3.47882e-13	2.93777e-7	8.26271e+5	5.8051e+3	3.13192e+3	2.90743e+2	2.95932e-2	2.02484e+1	1.10303e-5
	Std	8.24836e-14	1.27535e-6	4.53980e+5	3.7501e+3	5.2927e+2	4.89983e+2	2.54645e-2	6.922e-2	3.82911e-6
3e5	1^{st}(Best)	2.27374e-13	1.11982e-10	4.55579e+4	4.67497e+2	2.32729e+3	9.0764e-3	6.36728e-7	2.00574e+1	2.21295e-7
	7^{st}	2.82217e-13	1.44553e-10	1.56862e+5	1.89276e+3	2.83794e+3	8.20841e-2	9.85899e-3	2.01232e+1	6.09847e-7
	13^{st}(Median)	3.41061e-13	1.65699e-10	2.55728e+5	2.77784e+3	2.98982e+3	4.28868e+0	2.45743e-2	2.01563e+1	8.0644e-7
	19^{st}	3.94904e-13	1.82808e-10	3.72265e+5	5.28579e+3	3.39684e+3	1.73462e+2	3.69067e-2	2.01797e+1	9.5403e-7
	25^{st}(Worst)	5.11591e-13	2.58524e-10	5.81497e+5	1.1503e+4	4.75597e+3	9.07077e+2	9.76969e-2	2.02474e+1	1.32837e-6
	Mean	3.47882e-13	1.65321e-10	2.72353e+5	3.85297e+3	3.13183e+3	1.50812e+2	2.95802e-2	2.01516e+1	7.8419e-7
	Std	8.24836e-14	3.07934e-11	1.49584e+5	2.72695e+3	5.29252e+2	2.27077e+2	2.54659e-2	5.29173e-2	2.65556e-7

Table B.5: Error values at FEs = 1e3, 1e4, 1e5, 3e5 for problems 10-17(30D)

FE	Prob	10	11	12	13	14	15	16	17
1e3	1^{st}(Best)	3.41255e+2	3.82201e+1	9.96896e+5	6.95205e+1	1.36733e+1	9.40723e+2	5.71773e+2	5.96475e+2
	7^{st}	3.90662e+2	4.20253e+1	1.20119e+6	1.18234e+2	1.39339e+1	9.75799e+2	7.539e+2	7.3549e+2
	13^{st}(Median)	3.99057e+2	4.34894e+1	1.33726e+6	1.52431e+2	1.41018e+1	9.99283e+2	7.61828e+2	7.80179e+2
	19^{st}	4.13888e+2	4.43235e+1	1.39075e+6	1.83935e+2	1.41912e+1	1.0145e+3	7.85456e+2	8.61838e+2
	25^{st}(Worst)	4.4102e+2	4.63654e+1	1.57195e+6	2.51668e+2	1.43086e+1	1.06072e+3	8.45321e+2	9.67846e+2
	Mean	3.98611e+2	4.31598e+1	1.29445e+6	1.52042e+2	1.40726e+1	9.99565e+2	7.5464e+2	7.83124e+2
	Std	2.40552e+1	1.78288e+0	1.50328e+5	4.58589e+1	1.59441e-1	2.80225e+	5.40453e+1	1.00916e+2
1e4	1^{st}(Best)	1.88981e+2	1.7328e+0	7.41974e+4	5.88643e+0	1.2691e+1	4.88315e+2	2.11566e+2	2.65653e+2
	7^{st}	2.06258e+2	1.99844e+0	1.03273e+5	1.05982e+1	1.30513e+1	5.22805e+2	3.25737e+2	3.36452e+2
	13^{st}(Median)	2.1438e+2	2.18223e+0	1.24851e+5	1.64844e+1	1.31767e+1	5.48447e+2	5.53186e+2	4.68171e+2
	19^{st}	2.21266e+2	2.56902e+0	1.34856e+5	1.90537e+1	1.33215e+1	5.55105e+2	5.58166e+2	4.76113e+2
	25^{st}(Worst)	2.35254e+2	4.0414e+0	1.61766e+5	2.19337e+1	1.35207e+1	5.97066e+2	5.64091e+2	6.0056e+2
	Mean	2.12129e+2	2.44088e+0	1.19431e+5	1.51152e+1	1.31806e+1	5.40233e+2	4.44799e+2	4.25361e+2
	Std	1.1162e+1	6.63626e-1	2.4534e+4	4.81024e+0	1.83088e-1	2.86614e+1	1.30854e+2	1.04967e+2
1e5	1^{st}(Best)	1.39294e+1	1.41317e-2	1.14927e+1	7.92733e-1	1.11401e+1	3.2788e+2	5.4761e+1	6.38138e+1
	7^{st}	3.28336e+1	1.71758e-2	1.39058e+3	9.39681e-1	1.18155e+1	3.43616e+2	1.40888e+2	1.58613e+2
	13^{st}(Median)	3.71272e+1	1.82181e-2	2.87416e+3	1.16307e+0	1.22074e+1	3.51428e+2	5.01258e+2	4.34726e+2
	19^{st}	4.37781e+1	2.01791e-2	6.89031e+3	1.34709e+0	1.23845e+1	3.68657e+2	5.01848e+2	4.43602e+2
	25^{st}(Worst)	5.57176e+1	2.16151e-2	2.30486e+4	1.61207e+0	1.2699e+1	5.04367e+2	5.0349e+2	5.51299e+2
	Mean	3.69056e+1	1.84897e-2	5.093e+3	1.16044e+0	1.2063e+1	3.65085e+2	3.3684e+2	3.1954e+2
	Std	1.05106e+1	2.04441e-3	5.84165e+3	2.27263e-1	4.33187e-1	3.63679e+1	1.87927e+2	1.76305e+2
3e5	1^{st}(Best)	1.39294e+1	6.86314e-3	1.41528e-1	6.42879e-1	1.11378e+1	3.27641e+2	5.46082e+1	6.30197e+1
	7^{st}	3.28336e+1	7.77225e-3	1.37885e+3	8.83175e-1	1.18153e+1	3.39228e+2	1.40675e+2	1.57352e+2
	13^{st}(Median)	3.71271e+1	8.51561e-3	2.3158e+3	9.4174e-1	1.22042e+1	3.47151e+2	5.00979e+2	4.08135e+2
	19^{st}	4.37781e+1	8.85325e-3	6.36668e+3	1.17963e+0	1.23828e+1	3.6498e+2	5.01265e+2	4.31788e+2
	25^{st}(Worst)	5.57176e+1	9.96577e-3	2.30486e+4	1.48736e+0	1.26974e+1	5.03997e+2	5.02861e+2	5.39386e+2
	Mean	3.69056e+1	8.40942e-3	4.91148e+3	1.0437e+0	1.20617e+1	3.6229e+2	3.36537e+2	3.10781e+2
	Std	1.05106e+1	8.35477e-4	5.80581e+3	2.13731e-0	4.33793e-1	3.76284e+1	1.87872e+2	1.70242e+2

Table B.6: Error values at FEs = 1e3, 1e4, 1e5, 3e5 for problems 18-25(30D)

FE	Prob	18	19	20	21	22	23	24	25
1e3	1^{st}(Best)	1.09079e+3	1.07891e+3	1.08112e+3	1.18057e+3	1.1585e+3	1.21983e+3	1.29292e+3	1.2962e+3
	7^{st}	1.1147e+3	1.11035e+3	1.11859e+3	1.27844e+3	1.32045e+3	1.27208e+3	1.31483e+3	1.32914e+3
	13^{st}(Median)	1.13889e+3	1.12744e+3	1.13398e+3	1.28565e+3	1.3812e+3	1.28845e+3	1.33795e+3	1.34256e+3
	19^{st}	1.15725e+3	1.14656e+3	1.15064e+3	1.29709e+3	1.40928e+3	1.29647e+3	1.35684e+3	1.34969e+3
	25^{st}(Worst)	1.19718e+3	1.19202e+3	1.18751e+3	1.30451e+3	1.53828e+3	1.31618e+3	1.37563e+3	1.38775e+3
	Mean	1.1387e+3	1.12807e+3	1.1357e+3	1.28202e+3	1.3728e+3	1.281e+3	1.33572e+3	1.34032e+3
	Std	2.88961e+1	2.9308e+1	2.48189e+1	2.49337e+1	7.63534e+1	2.42088e+1	2.44945e+1	2.04276e+1
1e4	1^{st}(Best)	9.25083e+2	9.27342e+2	9.25014e+2	6.37034e+2	9.40359e+2	6.70569e+2	4.33243e+2	4.21729e+2
	7^{st}	9.29202e+2	9.28322e+2	9.28197e+2	6.65115e+2	9.52231e+2	6.87358e+2	4.63463e+2	5.02874e+2
	13^{st}(Median)	9.30119e+2	9.30361e+2	9.30676e+2	6.81871e+2	9.63811e+2	6.99054e+2	4.92972e+2	5.28702e+2
	19^{st}	9.34063e+2	9.31857e+2	9.35422e+2	6.99827e+2	9.7007e+2	7.18622e+2	5.20792e+2	5.75859e+2
	25^{st}(Worst)	9.45033e+2	9.42689e+2	9.38501e+2	7.65766e+2	9.9015e+2	7.42968e+2	5.89875e+2	6.82416e+2
	Mean	9.32184e+2	9.30698e+2	9.3145e+2	6.83808e+2	9.6255e+2	7.03473e+2	5.01424e+2	5.37929e+2
	Std	5.34395e+0	3.11483e+0	4.00502e+0	3.1823e+1	1.31899e+1	2.12136e+1	4.36122e+1	6.00467e+1
1e5	1^{st}(Best)	9.07615e+2	9.0739e+2	9.08242e+2	5.00159e+2	8.92648e+2	5.34165e+2	2.00236e+2	2.00255e+2
	7^{st}	9.10146e+2	9.09473e+2	9.09738e+2	5.00182e+2	9.0693e+2	5.34607e+2	2.00319e+2	2.00331e+2
	13^{st}(Median)	9.11624e+2	9.11141e+2	9.11128e+2	5.00201e+2	9.17885e+2	5.41573e+2	2.00345e+2	2.00348e+2
	19^{st}	9.1281e+2	9.12696e+2	9.13106e+2	5.00211e+2	9.27433e+2	5.45034e+2	2.00381e+2	2.00373e+2
	25^{st}(Worst)	9.16808e+2	9.14885e+2	9.1493e+2	5.00247e+2	9.40901e+2	5.67242e+2	2.0043e+2	2.00396e+2
	Mean	9.11523e+2	9.10891e+2	9.11336e+2	5.00198e+2	9.17556e+2	5.41424e+2	2.00346e+2	200.345e+2
	Std	2.08454e+0	1.95765e+0	1.97788e+0	2.15796e-2	1.22352e+1	7.64926e+0	5.00088e-2	3.51386e-2
3e5	1^{st}(Best)	9.07241e+2	9.06839e+2	9.07918e+2	5.00126e+2	8.89319e+2	5.34164e+2	2.00208e+2	2.00184e+2
	7^{st}	9.09834e+2	9.09196e+2	9.09694e+2	5.00152e+2	9.06081e+2	5.34607e+2	2.00266e+2	2.00274e+2
	13^{st}(Median)	9.11151e+2	9.10634e+2	9.10936e+2	5.00168e+2	9.12543e+2	5.41573e+2	2.00285e+2	2.00304e+2
	19^{st}	9.12553e+2	9.12673e+2	9.12857e+2	5.00172e+2	9.23771e+2	5.45034e+2	2.00306e+2	2.00312e+2
	25^{st}(Worst)	9.16264e+2	9.14885e+2	9.14851e+2	5.00195e+2	9.32868e+2	5.67242e+2	2.00329e+2	2.00362e+2
	Mean	9.11234e+2	9.10634e+2	9.11151e+2	5.00162e+2	9.14701e+2	5.41424e+2	2.00283e+2	2.00294e+2
	Std	2.08711e+0	2.04968e+0	1.9567e+0	1.76338e-2	1.22709e+1	7.6494e+0	2.96582e-2	3.496e-2

Table B.7: Error values at FEs = 1e3, 1e4, 1e5, 3e5, 5e5 for problems 1-9(50D)

FE	Prob	1	2	3	4	5	6	7	8	9
1e3	1^{st}(Best)	5.51142e+4	1.51879e+5	2.10005e+9	1.67442e+5	3.69882e+4	3.49642e+10	1.09999e+4	2.12583e+1	5.85913e+2
	7^{st}	7.12981e+4	1.83003e+5	2.72427e+9	2.32920e+5	3.87181e+4	4.97113e+10	1.15068e+4	2.13225e+1	6.24486e+2
	$13^{s}t$(Median)	7.66952e+4	2.04321e+5	3.21028e+9	2.54673e+5	4.02342e+4	5.75621e+10	1.25735e+4	2.13514e+1	6.5709e+2
	19^{st}	8.08763e+4	2.27296e+5	3.31233e+9	2.74659e+5	4.27958e+4	6.48353e+10	1.28912e+4	2.13788e+1	6.74407e+2
	25^{st}(Worst)	9.14238e+4	2.69784e+5	3.97722e+9	3.38369e+5	4.51263e+4	7.42571e+10	1.35734e+4	2.14255e+1	7.23742e+2
	Mean	7.58587e+4	2.05962e+5	3.02158e+9	2.56436e+5	4.05242e+4	5.69746e+10	1.22939e+4	2.13513e+1	6.49598e+2
	Std	8.70748e+3	3.06062e+4	4.44698e+8	4.08042e+4	2.50896e+3	1.11868e+10	7.6882e+2	4.06245e-2	3.8782e+1
1e4	1^{st}(Best)	9.98461e+2	2.73658e+4	1.12281e+8	3.37432e+4	1.20098e+4	5.57072e+8	5.05316e+2	2.11475e+1	3.65384e+2
	7^{st}	1.54895e+3	4.51881e+4	1.58228e+8	5.90627e+4	1.46458e+4	9.9109e+8	6.78866e+2	2.12413e+1	3.96714e+2
	13^{st}(Median)	1.72223e+3	4.95447e+4	1.7982e+8	7.04506e+4	1.57016e+4	1.17615e+9	8.24632e+2	2.12665e+1	4.13965e+2
	19^{st}	1.88534e+3	5.50288e+4	2.18898e+8	7.91098e+4	1.64814e+4	1.40983e+9	1.00922e+3	2.12941e+1	4.34711e+2
	25^{st}(Worst)	2.66011e+3	7.18056e+4	2.44658e+8	1.13046e+5	1.78503e+4	1.76553e+9	1.44983e+3	2.13482e+1	4.57782e+2
	Mean	1.7396e+3	4.99849e+4	1.83442e+8	6.90953e+4	1.5422e+4	1.18959e+9	8.69208e+2	2.12664e+1	4.13594e+2
	Std	3.68971e+2	9.19741e+3	3.63204e+7	1.57892e+4	1.52509e+3	3.05817e+8	2.5899e+2	4.24611e-2	2.52316e+1
1e5	1^{st}(Best)	3.41061e-13	1.87307e+2	1.00166e+6	2.07346e+4	7.23403e+3	2.47339e+1	5.55714e-8	2.01349e+1	2.42035e-5
	7^{st}	4.54747e-13	9.05602e+2	1.38678e+6	3.48667e+4	9.08045e+3	4.00485e+1	2.08563e-7	2.01938e+1	5.82229e-5
	13^{st}(Median)	5.11591e-13	1.35331e+3	1.92911e+6	3.91607e+4	9.69999e+3	4.76419e+1	9.85736e-3	2.02497e+1	7.45986e-5
	19^{st}	6.25278e-13	1.78202e+3	2.40636e+6	4.49693e+4	1.09219e+4	2.48135e+2	2.94594e-2	2.02981e+1	9.15903e-5
	25^{st}(Worst)	7.38964e-13	3.24683e+3	3.29172e+6	5.67418e+4	1.22664e+4	9.36661e+3	5.63526e-2	2.03497e+1	1.57744e-4
	Mean	5.34328e-13	1.42029e+3	1.89689e+6	3.89832e+4	9.82417e+3	1.15715e+3	1.48301e-2	2.02492e+1	7.64826e-5
	Std	9.91099e-14	7.23942e+2	5.98377e+5	8.6373e+3	1.26849e+3	2.67887e+3	1.68684e-2	6.05549e-2	2.76112e-5
3e5	1^{st}(Best)	3.41061e-13	7.23333e-10	2.09420e+5	1.34239e+4	7.23402e+3	4.80782e+0	8.36877e-10	2.00779e+1	2.01438e-6
	7^{st}	4.54747e-13	1.15472e-9	2.62135e+5	2.0845e+4	9.07959e+3	1.7834e+1	3.21262e-9	2.01243e+1	3.18409e-6
	13^{st}(Median)	5.11591e-13	1.37845e-9	3.88935e+5	2.40103e+4	9.69999e+3	2.06e+1	9.85729e-3	2.017e+1	3.48123e-6
	19^{st}	6.25278e-13	1.74771e-9	4.41208e+5	2.95921e+4	1.09219e+4	1.46261e+2	2.94591e-2	2.01979e+1	4.45399e-6
	25^{st}(Worst)	7.38964e-13	2.53847e-8	6.40857e+5	4.35825e+4	1.22664e+4	1.0344e+3	5.63524e-2	2.02488e+1	8.04654e-6
	Mean	5.34328e-13	2.38457e-9	3.79589e+5	2.56436e+4	9.82398e+3	1.30426e+2	1.48301e-2	2.01648e+1	3.79203e-6
	Std	9.91099e-14	4.70832e-9	1.19670e+5	6.75653e+3	1.26849e+3	2.62602e+2	1.68684e-2	4.32217e-2	1.30416e-6
5e5	1^{st}(Best)	3.41061e-13	7.23219e-10	1.03835e+5	7.76829e+3	7.23402e+3	2.15868e-1	2.15636e-10	2.00631e+1	4.52469e-7
	7^{st}	4.54747e-13	1.15028e-9	1.52246e+5	1.50123e+4	9.07959e+3	3.937e+0	1.01704e-9	2.01019e+1	1.07226e-6
	13^{st}(Median)	5.11591e-13	1.28637e-9	2.00057e+5	1.83313e+4	9.69999e+3	6.08975e+0	9.85728e-3	2.01386e+1	1.30084e-6
	19^{st}	6.25278e-13	1.41921e-9	2.67678e+5	2.30653e+4	1.09219e+4	1.36537e+2	2.94591e-2	2.01598e+1	1.40499e-6
	25^{st}(Worst)	7.38964e-13	2.03244e-9	3.38318e+5	3.63462e+4	1.22664e+4	9.65119e+2	5.63524e-2	2.02026e+1	2.00881e-6
	Mean	5.34328e-13	1.31335e-9	2.11398e+5	1.91824e+4	9.82375e+3	1.12253e+2	1.48301e-2	2.01318e+1	1.2652e-6
	Std	9.91099e-14	2.78563e-10	6.76307e+4	5.71984e+3	1.26857e+3	2.45579e+2	1.68684e-2	3.53769e-2	3.48828e-7

Table B.8: Error values at FEs = 1e3, 1e4, 1e5, 3e5, 5e5 for problems 10-17(50D)

FE	Prob	10	11	12	13	14	15	16	17
1e3	1^{st}(Best)	9.55087e+2	7.42498e+1	4.41679e+6	4.40894e+2	2.31195e+1	9.77567e+2	6.57047e+2	8.19088e+2
	7^{st}	1.04246e+3	7.57477e+1	5.64745e+6	8.16527e+2	2.38016e+1	1.04476e+3	7.46701e+2	9.07803e+2
	13^{st}(Median)	1.09997e+3	7.75215e+1	6.55796e+6	1.05628e+3	2.40312e+1	1.10186e+3	7.86151e+2	9.24502e+2
	19^{st}	1.12314e+3	7.85936e+1	6.79599e+6	1.09935e+3	2.41171e+1	1.13654e+3	8.31648e+2	9.80848e+2
	25^{st}(Worst)	1.18023e+3	8.35322e+1	7.36667e+6	1.53577e+3	2.42551e+1	1.16253e+3	9.21854e+2	1.11515e+3
	Mean	1.08081e+3	7.7621e+1	6.26256e+6	9.83086e+2	2.39534e+1	1.09006e+3	7.93349e+2	9.33038e+2
	Std	5.67469e+1	2.35302e+0	7.59139e+5	2.53989e+2	2.5106e-1	5.25278e+1	6.38147e+1	7.23728e+1
1e4	1^{st}(Best)	4.52249e+2	1.06331e+1	6.94560e+4	4.16305e+1	2.2704e+1	5.00881e+2	3.06611e+2	3.6226e+2
	7^{st}	4.77605e+2	1.23251e+1	9.91115e+5	4.45412e+1	2.29962e+1	6.09731e+2	3.21565e+2	3.78356e+2
	13^{st}(Median)	4.84931e+2	1.37686e+1	1.07617e+6	4.59726e+1	2.30894e+1	6.45887e+2	3.40662e+2	3.86797e+2
	19^{st}	5.06135e+2	1.59975e+1	1.13921e+6	4.85981e+1	2.326e+1	6.6742e+2	3.49884e+2	3.99851e+2
	25^{st}(Worst)	5.4627e+2	2.05923e+1	1.33412e+6	5.09829e+1	2.34369e+1	7.06334e+2	3.74912e+2	4.31716e+2
	Mean	4.91592e+2	1.4347e+1	1.05603e+6	4.61968e+1	2.30998e+1	6.24054e+2	3.38922e+2	3.89905e+2
	Std	2.5674e+1	2.48334e+0	1.48582e+5	2.64645e+0	2.01406e-1	6.05553e+1	1.86864e+1	1.72941e+1
1e5	1^{st}(Best)	8.75562e+1	3.81584e-2	1.44276e+4	1.6658e+0	2.05739e+1	3.46052e+2	7.74716e+1	9.57355e+1
	7^{st}	1.07455e+2	4.27134e-2	3.12412e+4	1.98964e+0	2.11749e+1	3.91017e+2	9.39309e+1	1.16354e+2
	13^{st}(Median)	1.15415e+2	4.50361e-2	4.34969e+4	2.38846e+0	2.16154e+1	4.023e+2	9.78339e+1	1.2483e+2
	19^{st}	1.30339e+2	4.71833e-2	6.12136e+4	2.70468e+0	2.18551e+1	4.23389e+2	1.0489e+2	1.32262e+2
	25^{st}(Worst)	1.68148e+2	5.29221e-2	1.29204e+5	3.16681e+0	2.24938e+1	5.05451e+2	1.19914e+2	1.58008e+2
	Mean	1.17804e+2	4.47662e-2	5.40739e+4	2.35217e+0	2.15798e+1	4.10944e+2	9.92568e+1	1.2542e+2
	Std	1.99834e+1	3.51986e-3	3.31723e+4	4.05752e-1	4.65305e-1	3.6672e+1	1.11199e+1	1.51036e+1
3e5	1^{st}(Best)	8.75562e+1	1.69114e-2	1.34048e+4	1.27279e+0	2.05719e+1	3.42822e+2	7.64194e+1	9.43639e+1
	7^{st}	1.07455e+2	1.8854e-2	2.8812e+4	1.75218e+0	2.11712e+1	3.81269e+2	9.33667e+1	1.15444e+2
	13^{st}(Median)	1.15415e+2	1.99417e-2	4.30683e+4	2.01806e+0	2.16104e+1	4.01718e+2	9.78339e+1	1.23132e+2
	19^{st}	1.30339e+2	2.1826e-2	6.12136e+4	2.20224e+0	2.18543e+1	4.21828e+2	1.0408e+2	1.30267e+2
	25^{st}(Worst)	1.68148e+2	2.35981e-2	1.20129e+5	2.53098e+0	2.2493e+1	5.03771e+2	1.19914e+2	1.5679e+2
	Mean	1.17804e+2	2.01512e-2	5.11606e+4	1.96793e+0	2.15774e+1	4.06304e+2	9.87115e+1	1.23639e+2
	Std	1.99834e+1	1.91237e-3	3.16112e+4	3.53218e-1	4.65591e-1	3.86759e+1	1.12072e+1	1.47723e+1
5e5	1^{st}(Best)	8.75562e+1	1.12915e-2	1.2641e+4	1.18787e+0	2.05718e+1	3.4065e+2	7.62284e+1	9.36489e+1
	7^{st}	1.07455e+2	1.31512e-2	2.6452e+4	1.72212e+0	2.11707e+1	3.81153e+2	9.32198e+1	1.15403e+2
	13^{st}(Median)	1.15415e+2	1.46848e-2	4.13691e+4	1.93704e+0	2.16094e+1	4.01405e+2	9.78048e+1	1.23132e+2
	19^{st}	1.30339e+2	1.50056e-2	6.12136e+4	2.16827e+0	2.18542e+1	4.20961e+2	1.0408e+2	1.2919e+2
	25^{st}(Worst)	1.68148e+2	1.8692e-2	1.14555e+5	2.51907e+0	2.24929e+1	5.03659e+2	1.19836e+2	1.55669e+2
	Mean	1.17804e+2	1.4332e-2	4.85485e+4	1.92194e+0	2.1577e+1	4.04874e+2	9.8556e+1	1.23137e+2
	Std	1.99834e+1	1.55761e-3	2.93073e+4	3.48701e-1	4.65643e-1	3.8859e+1	1.12841e+1	1.47763e+1

Table B.9: Error values at FEs = 1e3, 1e4, 1e5, 3e5, 5e5 for problems 18-25(50D)

FE	Prob	18	19	20	21	22	23	24	25
1e3	1^{st}(Best)	1.32012e+3	1.28895e+3	1.27409e+3	1.35554e+3	1.41029e+3	1.39336e+3	1.48128e+3	1.81075e+3
	7^{st}	1.34924e+3	1.32921e+3	1.32205e+3	1.41301e+3	1.47406e+3	1.4202e+3	1.50679e+3	1.90004e+3
	13^{st}(Median)	1.3719e+3	1.34251e+3	1.35202e+3	1.43062e+3	1.53781e+3	1.45175e+3	1.545e+3	1.91905e+3
	19^{st}	1.37878e+3	1.35255e+3	1.38102e+3	1.44662e+3	1.53781e+3	1.45673e+3	1.56103e+3	1.9374e+3
	25^{st}(Worst)	1.41859e+3	1.42743e+3	1.40995e+3	1.45713e+3	1.62434e+3	1.48599e+3	1.57763e+3	1.96508e+3
	Mean	1.36427e+3	1.34641e+3	1.3522e+3	1.42729e+3	1.52263e+3	1.44181e+3	1.53718e+3	1.91294e+3
	Std	2.43472e+1	3.02611e+1	3.7055e+1	2.35834e+1	5.63886e+1	2.31314e+1	2.87624e+1	3.52847e+1
1e4	1^{st}(Best)	9.97832e+2	9.89552e+2	9.7263e+2	9.87067e+2	1.02721e+3	9.93378e+2	1.03987e+3	4.61938e+2
	7^{st}	1.0087e+3	1.00917e+3	1.01018e+3	1.03841e+3	1.0634e+3	1.04926e+3	1.10792e+3	5.49586e+2
	13^{st}(Median)	1.01283e+3	1.01412e+3	1.01506e+3	1.05378e+3	1.07579e+3	1.07128e+3	1.15488e+3	6.78404e+2
	19^{st}	1.01973e+3	1.02425e+3	1.02636e+3	1.10462e+3	1.08577e+3	1.10827e+3	1.1638e+3	8.15937e+2
	25^{st}(Worst)	1.04242e+3	1.04429e+3	1.05164e+3	1.14175e+3	1.10565e+3	1.13687e+3	1.23733e+3	1.22294e+3
	Mean	1.01567e+3	1.01688e+3	1.01607e+3	1.06888e+3	1.07398e+3	1.07397e+3	1.14328e+3	7.34723e+2
	Std	9.9784e+0	1.23553e+1	1.60617e+1	4.51729e+1	1.80038e+1	3.93552e+1	4.91449e+1	2.12427e+2
1e5	1^{st}(Best)	9.28534e+2	9.33703e+2	9.29386e+2	5.00296e+2	9.68637e+2	5.49105e+2	2.00486e+2	2.20288e+2
	7^{st}	9.33376e+2	9.36249e+2	9.34088e+2	5.00353e+2	9.96375e+2	5.65554e+2	2.00661e+2	2.22291e+2
	13^{st}(Median)	9.36262e+2	9.39147e+2	9.38245e+2	5.00367e+2	1.00482e+3	5.80241e+2	2.00705e+2	2.23946e+2
	19^{st}	9.40687e+2	9.42993e+2	9.44787e+2	5.00414e+2	1.0204e+3	5.85571e+2	2.00749e+2	2.31217e+2
	25^{st}(Worst)	9.64881e+2	9.65428e+2	9.68877e+2	1.04828e+3	1.04072e+3	1.06117e+3	2.00837e+2	2.63998e+2
	Mean	9.39207e+2	9.40836e+2	9.40921e+2	6.08119e+2	1.00673e+3	5.9556e+2	2.00707e+2	2.29575e+2
	Std	9.51087e+0	6.63746e+0	9.82973e+0	2.15538e+2	1.73734e+1	9.60541e+1	7.71071e-2	1.17577e+2
3e5	1^{st}(Best)	9.2779e+2	9.33436e+2	9.283e+2	5.00274e+2	9.66662e+2	5.49103e+2	2.00409e+2	2.16496e+2
	7^{st}	9.32988e+2	9.358e+2	9.33721e+2	5.00296e+2	9.93031e+2	5.65553e+2	2.00547e+2	2.18164e+2
	13^{st}(Median)	9.36227e+2	9.38894e+2	9.38036e+2	5.00309e+2	1.00007e+3	5.80239e+2	2.00602e+2	2.18902e+2
	19^{st}	9.40216e+2	9.42198e+2	9.43444e+2	5.00325e+2	1.01668e+3	5.85571e+2	2.00618e+2	2.19667e+2
	25^{st}(Worst)	9.63709e+2	9.64643e+2	9.68343e+2	1.04755e+3	1.04072e+3	1.06116e+3	2.00654e+2	2.25634e+2
	Mean	9.38732e+2	9.40333e+2	9.40451e+2	6.07948e+2	1.00278e+3	5.95501e+2	2.00576e+2	2.19157e+2
	Std	9.34306e+0	6.55205e+0	9.84473e+0	2.15314e+2	1.7955e+1	9.60678e+1	6.23622e-2	1.9333e+0
5e5	1^{st}(Best)	9.27555e+2	9.33436e+2	9.283e+2	5.00231e+2	9.66662e+2	5.49103e+2	2.00409e+2	2.16285e+2
	7^{st}	9.32923e+2	9.358e+2	9.33485e+2	5.00276e+2	9.89824e+2	5.65553e+2	2.00507e+2	2.17589e+2
	13^{st}(Median)	9.36003e+2	9.38848e+2	9.37865e+2	5.00297e+2	9.98235e+2	5.80239e+2	2.00558e+2	2.184e+2
	19^{st}	9.39908e+2	9.42198e+2	9.43444e+2	5.0031e+2	1.0162e+3	5.85568e+2	2.00595e+2	2.18941e+2
	25^{st}(Worst)	9.63641e+2	9.64643e+2	9.67628e+2	1.04649e+3	1.04072e+3	1.06115e+3	2.00628e+2	2.21402e+2
	Mean	9.38592e+2	9.40204e+2	9.40267e+2	6.07759e+2	1.00155e+3	5.955e+2	2.00544e+2	2.18497e+2
	Std	9.27171e+0	6.57274e+0	9.78984e+0	2.14974e+2	1.88383e+1	9.6066e+1	6.13193e-2	1.30321e+0

Table B.10: Number of FES to achieve the accuracy level for problems 1 - 25(D = 10)

Prob	1^{st} (Best)	7^{th}	13^{th} (Median)	19^{th}	25^{th} (Worst)	Mean	Std	Succ. rate	succ. Perf.
1	22000	23700	24600	25200	26900	24416	1236.67	100%	24416
2	32600	34000	34200	35300	35800	34528	860.474	100%	34528
3	-	-	-	-	-	-	-	0%	-
4	38200	40100	41100	41600	43400	40900	1439.17	100%	40900
5	-	-	-	-	-	-	-	0%	-
6	98600	-	-	-	-	98600	0.0	4.0%	2.465e+6
7	52100	-	-	-	-	52100	0.0	4.0%	1.3025e+5
8	-	-	-	-	-	-	-	0%	-
9	20500	22000	22400	23200	26400	22620	1376.66	100%	22620
10	-	-	-	-	-	-	-	0%	-
11	28400	32500	36000	38400	42300	35556	4000.71	100%	35556
12	30800	53200	-	-	-	54109.1	15494.1	44%	122975
13-25		-	-	-	-	-	-	0%	-

Table B.11: Number of FES to achieve the accuracy level for problems 1 - 25(D = 30)

Prob	1^{st} (Best)	7^{th}	13^{th} (Median)	19^{th}	25^{th} (Worst)	Mean	Std	Succ. rate	succ. Perf.
1	30400	31700	32200	32800	33800	32180	813.88	100%	32180
2	77800	88200	92500	97800	102000	92420	6279.62	100%	92420
3-5	-	-	-	-	-	-	-	-	-
6	297100	-	-	-	-	297100	0.0	4%	7427500
7	29800	59300	-	-	-	51627.3	12492.5	44%	117335
8	-	-	-	-	-	-	-	0%	-
9	30900	32500	33100	35100	45500	34636	3610.53	100%	34636
10	-	-	-	-	-	-	-	0%	-
11	161600	210300	222200	242200	293200	228752	33891.1	100%	228752
12-25	-	-	-	-	-	-	-	0%	-

Table B.12: Number of FES to achieve the accuracy level for problems 1 - 25(D = 50)

Prob	1^{st} (Best)	7^{th}	13^{th} (Median)	19^{th}	25^{th} (Worst)	Mean	Std	Succ. rate	succ. Perf.
1	37400	37900	38600	39300	40100	38540	782.304	100%	38540
2	224200	230600	243900	251800	275600	242192	12952.8	100%	242192
3-6	-	-	-	-	-	-	-	0%	-
7	48700	54200	61400	-	-	-	-	60%	95211.1
8	-	-	-	-	-	-	-	0%	-
9	43400	48000	51800	54100	71900	51632	6007.51	100%	51632
10-25	-	-	-	-	-	-	-	0%	-

Appendix C

Chemotherapy Problem in C++ Source Codes

```
#include "StdAfx.h"
#include <stdlib.h>
#include <math.h>
#include "chemotherapy.h"

/*****************************************************************************
This is the main method of the class. It takes the chromosome as a parameter
and calculates both fitnesses. This method does not return a fitness value, it
just calculates and stores values for each fitness objective. Use getFitnessObjOne()
and getFitnessObjTwo() to extract fitness values rep tumour size and PST
(patient survival time).
*****************************************************************************/

//////////////////////////////Constant declarations////////////////////////////////
int NUM_DRUGS = 10;                   // number of drugs in treatment
int NUM_DOSES = 10;                   // number of doses in treatment
int NUM_ORGANS = 5;                   // body organs affected by drugs
int DRUG_UNIT = 5;                   // multiplier to get gene value of dose into mg
long double LAMBDA = 4;                 // Gompertz constant (tumour growth) 1.9 for
                                        // Nmax (10^12) at end of 7 doses
long double P1 = 5;                 // penalty multiplier for exceeding max
                                        // instantaneous dose of a drug
long double P2 = 5;                 // ditto for exceeding max cumulative
                                        // dose for a drug
long double P3 = 500;                 // ditto for exceeding max tumour size YMIN
long double P4 = 5;                 // ditto for exceeding side effect on each
                                        // organ at each time step
long double Y0 = 4.605;              // Initial tumour size y = ln(theta/N)
                                        // corresponding to $N = 10^10$
long double YMIN = 4.605;               // Y value corresp to max allowed tumour
                                        // size (no of cells = $10^10$)
long double BETA = 0.5;                 // some ratio penalizing the current tumour size
long double KAPPA_SCALE_FACTOR = 0.001; // Scaling Factor For Dose Efficacy values
```

```
int MAX_INST_DOSE[] = {75, 75, 100, 2000, 3000, 120, 10000, 15, 100, 2};
int MAX_CUM_SIDE_EFF[] = {90, 90, 90, 90, 90};
long double POTENCY_FACTOR[] = {1, 1, 0.75, 0.0375, 0.025, 0.625, 0.0075, 5, 0.75, 200};
long double MAX_CUM_DOSE[] = {550, 700, 1000, 10000, 30000, 600, 100000, 40, 1000, 30};
long double KAPPA[] = {5.605, 4.484, 7.29, 3.9235, 2.242, 4.335, 1.6815, 2.242, 1.121, 2.242};
int RISK_FACTOR[5][10] = {
  //table of 'stars' for side effect on organs
    {3, 3, 3, 2, 0, 1, 1, 2, 0, 0},// Bone marrow
    {0, 0, 0, 0, 0, 3, 1, 0, 0, 0},// Kidney
    {0, 0, 2, 0, 0, 3, 0, 0, 0, 2},// Peripheral nerves
    {0, 0, 0, 0, 0, 0, 1, 1, 0, 0},// Liver
    {2, 1, 1, 0, 0, 0, 0, 0, 0, 0} // Heart
};

//////////////////////////////////variables/////////////////////////////////////////
long double **realDoseMatrix;
int **baseDoseMatrix;
long double cellKillAtTimeP = 0.0; // cell kill term
long double totalPenaltyForInstantaneousDoses = 0.0;
long double totalPenaltyForCumulativeDoses = 0.0;
long double totalPenaltyForSideEffect = 0.0;
long double totalPenaltyForTumourSize = 0.0;
long double fitness = 0.0;
long double yti = 0.0;               // current tumour size(N) expressed as y = ln(theta/N),
                                        // where theta = 10^12 cells.
long double diff = 0.0;
int violate = 0;                     // no of tumour size violations
int pCount = 0;                     // no of penalties for side effect, instantaneous and
                                        // cummulative dose exceeded.
int patientSurvivalTime = -1;           // set to negative until it is set by calculateTumourSize

void allocate_memory()
{
 //allocate memory for realDoseMatrix.
 realDoseMatrix = (long double **)malloc(NUM_DOSES*sizeof(long double));

 for (int i = 0; i < NUM_DOSES; i++)
   {
      realDoseMatrix[i] = (long double *)malloc(NUM_DRUGS*sizeof(long double));
 }

 //allocate memory for baseDoseMatrix.
 baseDoseMatrix = (int **)malloc(NUM_DOSES*sizeof(int));

 for (int i = 0; i < NUM_DOSES; i++)
   {
      baseDoseMatrix[i] = (int *)malloc(NUM_DRUGS*sizeof(int));
 }
}
```

```
void free_memory()
{
 //free realDoseMatrix.
 for (int i = 0; i < NUM_DOSES; i++)
   {
      free(realDoseMatrix[i]);
 }

 free(realDoseMatrix);

 //free baseDoseMatrix.
 for (int i = 0; i < NUM_DOSES; i++)
   {
      free(baseDoseMatrix[i]);
 }

 free(baseDoseMatrix);
}

void calculateFitness(int *chromosome)
{
 //Populate 2D DoseMatrices
   int chromosomeIndex = 0;
 int temp1;
 long double temp2;

 for(int i = 0; i < NUM_DOSES; i++)
   {
  for(int j = 0; j < NUM_DRUGS; j++)
      {
     //Populate DoseMatrices. realDoseMatrix contains absolute value of drug dose.
     //baseDoseMatrix is gene integer value.
         temp1 = chromosome[chromosomeIndex];
   baseDoseMatrix[i][j] = temp1;

   temp2 = (long double)((long double)chromosome[chromosomeIndex]*DRUG_UNIT)/POTENCY_FACTOR[j];
   realDoseMatrix[i][j] = temp2;

         chromosomeIndex++;
      }
 }

    // For time step (which corresponds to a dose), calculate total tumour
    // size and side effects. Add a penalty if the tumour goes above a
    // given threshold. Also add penalty for side effects.  p -> time.

  //Calculate Instantaneous dose penalty.
    calculateInstantaneousDosePenalty();

  //Calculate cumulative dose penalty.
    calculateCumulativeDosePenalty();
```

```
  for (int p = 0; p < NUM_DOSES; p++)  // for timesteps 1 to NUM_DOSES
    {
    //Calculate tumour size at p.
        calculateTumourSize(p);

        //Calculate side effect penalty at p.
        calculateSideEffectPenalty(p);
  }

  //Work out fitness objective 1 Final tumour size including penalties.
    calculateFitnessObjOne();

    //Work out fitness objective 2 Palliative care, Patient Survival Time.
    //calculateFitnessObjTwo();
}

void calculateInstantaneousDosePenalty()
{
  for (int j = 0; j < NUM_DRUGS; j++) // for each Drug
    {
        //check each dose for each time i for that drug
        for (int i = 0; i < NUM_DOSES; i++)
        {
      // Find the difference between the maximum allowed amount and the current amount
            diff = realDoseMatrix[i][j] - MAX_INST_DOSE[j];

            if (diff <= 0.0)  // less than limit, OK
            {
                diff = 0.0;
            }
            else      // else get square of difference and apply penalty
            {
                diff = diff * diff;
               pCount = pCount + 1; // increment counter for penalties applied
            }

            totalPenaltyForInstantaneousDoses += diff * P1;
     }
   }
}

void calculateCumulativeDosePenalty()
{
  long double sumD; // local sum of doses in real values

  for (int j = 0; j < NUM_DRUGS; j++) // for each Drug
    {
    sumD = 0.0; //initialize

        // sum the doses for each time i for that drug
```

```
        for (int i = 0; i < NUM_DOSES; i++)
        {
     sumD += realDoseMatrix[i][j];
    }

        diff = sumD - MAX_CUM_DOSE[j];   // Find the difference between the maximum
                                          // allowed amount and the current amount
        if (diff <= 0.0)  // less than limit, OK
        {
           diff = 0.0;
      }
        else              // else get square of difference and apply penalty
        {
            diff = diff * diff;
            pCount = pCount + 1; // increment counter for penalties applied
     }

        totalPenaltyForCumulativeDoses += diff * P2;
    }
}

//This method takes a time step value as a parameter and calculates the tumour size at
//time p. It then applies a penalty if the tumour has grown above a certain size.
void calculateTumourSize(int p)
{
  long double sumDoses = 0.0;
    long double cellKillByJthDrugForPthTime = 0.0;
    long double totalPenaltyForTumourSizeForPthTime = 0.0;

    cellKillAtTimeP = 0.0;

  //For each drug, loop from time 0 to current time (p) and sum up the total amount of drug
  //administered.  Then use this amount to calculate cell kill. At the end of the loop, you've
  //reduced the total tumour size by cumulative dose of each drug up to time p.

    for (int j = 0; j < NUM_DRUGS; j++)
    {
    //Outer for loop goes through each drug. sumDoses represent the cumulative dose for
    //each single drug. therefore, it needs to be reset for each new drug.
    sumDoses = 0.0;

    //Loop from time 0 up to current time (p)
        for (int i = 0; i < p; i++)
        {
      //i represents time step, which corresponds to a dose given at that time step tim1
      //corresponds to previous time step.
            int tim1 = i;
            //sum the multiplication of all dose till time p for jth drug with
            //exp(LAMBDA*(previous time and current time))
            sumDoses += (realDoseMatrix[i][j] * exp(LAMBDA*(tim1 - p)));
    }
```

```
        //multiply calculated sum of dose for jth drug with the efficacy coefficient
        //KAPPA for that drug.
        cellKillByJthDrugForPthTime = KAPPA_SCALE_FACTOR * KAPPA[j] * sumDoses;
        cellKillAtTimeP += cellKillByJthDrugForPthTime;
  }

  //calculate the tumour size and penalize if infeasible.
    yti = (Y0*exp(-LAMBDA*p)) + (((exp(LAMBDA)-1)*cellKillAtTimeP)/LAMBDA);

  //check if current tumour size is less than allowed tumour size or not
    diff = YMIN - yti;

  if (diff <= 0.0)  // i.e. yti > YMIN so tumour has not exceeded threshold size.
    {
   diff = 0.0;
    }
    else
    {
       diff = diff * diff; //i.e. yti < YMIN so tumour HAS exceeded threshold size.

    if (p >= 1)               //do not count if first timestep (initial tumour)
        {
      violate++;                  //count no. of times threshold violated
        }

        if (patientSurvivalTime < 0) // if PST hasn't been set yet
        {
     patientSurvivalTime = p;   // the first timestep at which tumour > max
        }
  }

  int ti = p;
    totalPenaltyForTumourSizeForPthTime = exp(-BETA * ti) * diff;
    totalPenaltyForTumourSize += totalPenaltyForTumourSizeForPthTime;
}

//calculate the penalty for effect on other organs @param p = timestep
void calculateSideEffectPenalty(int p)
{
  long double penaltyForSideEffectForLthOrganPthDose = 0.0;
    long double totalPenaltyForSideEffectforPthDose = 0.0;
    long double DrugSideEff;

  //calculate the side effect penalty
    totalPenaltyForSideEffectforPthDose = 0.0;

    //for each organ calculate the toxicity penalty
    for (int l = 0; l < NUM_ORGANS; l++)
    {
    DrugSideEff = 0.0;
```

```
    //for each Drug calculate toxicity level it makes to particular organ
        for (int j = 0; j < NUM_DRUGS; j++)
        {
            DrugSideEff += RISK_FACTOR[l][j]*baseDoseMatrix[p][j]; // use integer originals
    }

        //now compare it with allowed toxicity for that organ
        diff = DrugSideEff - MAX_CUM_SIDE_EFF[l];

    if (diff <= 0.0)  //less than limit OK
        {
      diff = 0.0;
        }
    else              //if not then penalize
        {
            diff = diff * diff;
            pCount = pCount + 1; // increment counter for penalties applied
        }

    penaltyForSideEffectForLthOrganPthDose = diff * P4;
        totalPenaltyForSideEffectforPthDose += penaltyForSideEffectForLthOrganPthDose;
     }

   totalPenaltyForSideEffect += totalPenaltyForSideEffectforPthDose;
}

void calculateFitnessObjOne()
{
  long double expr1, expr2, expr3, expr4, expr5;

    expr1 = cellKillAtTimeP;           // cell kill

    expr2 = totalPenaltyForCumulativeDoses;    // Cumulative Doses

    expr3 = P3 * totalPenaltyForTumourSize;   // Tumour size

    expr4 = totalPenaltyForSideEffect;      // side effect

    expr5 = totalPenaltyForInstantaneousDoses; // Instantaneous Dose

    fitness = (expr1 - expr3) / (violate + 1);

    fitness = fitness - expr2 - expr4 - expr5;
}

long double getFitnessObjOne()
{
  return fitness;
}
```

```
long double getFitnessObjTwo()
{
    return patientSurvivalTime;
}

long double getFinalTumourSize()
{
  return yti;
}

int getViolationCount()
{
  return violate;
}

int getPenaltyCount()
{
  return pCount;
}

int getPST()
{
  return patientSurvivalTime;
}

void clearStoredValues()
{
    //Clear DoseMatrices
    for(int i = 0; i < NUM_DOSES; i++)
    {
    for(int j = 0; j < NUM_DRUGS; j++)
        {
      realDoseMatrix[i][j] = 0.0;
            baseDoseMatrix[i][j] = 0;
    }
  }

  cellKillAtTimeP = 0.0;
    totalPenaltyForTumourSize = 0.0;
    totalPenaltyForSideEffect = 0.0;
    totalPenaltyForCumulativeDoses = 0.0;
    totalPenaltyForInstantaneousDoses = 0.0;
    fitness = 0.0;
    yti = 0.0;
    diff = 0.0;
    violate = 0;
    pCount = 0;
    patientSurvivalTime = -1;
}
```

Bibliography

[AH05a] A. Auger and N. Hansen. Performance evaluation of an advanced local search evolutionary algorithm. In David Corne, Zbigniew Michalewicz, Bob McKay, Gusz Eiben, David Fogel, Carlos Fonseca, Garrison Greenwood, Gunther Raidl, Kay Chen Tan, and Ali Zalzala, editors, *Proceedings of the 2005 IEEE Congress on Evolutionary Computation*, volume 2, pages 1777–1784, Edinburgh, Scotland, UK, 2-5 September 2005. IEEE Press. 133, 144, 167

[AH05b] A. Auger and N. Hansen. A restart CMA evolution strategy with increasing population size. In David Corne, Zbigniew Michalewicz, Bob McKay, Gusz Eiben, David Fogel, Carlos Fonseca, Garrison Greenwood, Gunther Raidl, Kay Chen Tan, and Ali Zalzala, editors, *Proceedings of the 2005 IEEE Congress on Evolutionary Computation*, volume 2, pages 1769–1776, Edinburgh, Scotland, UK, 2-5 September 2005. IEEE Press. 133, 144, 167

[AK89] Emile Aarts and Jan Korst. *Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing*. Wiley Interscience Series in Discrete Mathematics and Optimization. John Wiley & Sons, Chichester, 1989. 27, 30, 31

[Bäc95] Thomas Bäck. Generalized convergence models for tournament- and (μ, λ)-selection. In Larry J. Eshelman, editor, *ICGA*, pages 2–8. Morgan Kaufmann, 1995. 34, 96

[Bäc96] Thomas Bäck. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*. Oxford University Press, New York, 1996. 32, 41

[Bak87] James E. Baker. Reducing bias and inefficiency in the selection algorithm. In John J. Grefenstette, editor, *Genetic Algorithms and their Applications*

(ICGA'87), pages 14–21, Hillsdale, New Jersey, 1987. Lawrence Erlbaum Associates. 34

[Bal94] Shummet Baluja. Population-based incremental learning: A method for integrating genetic search based function optimization and competitive learning. Technical Report CS-94-163, Carnegie Mellon University, School of Computer Science, June 1994. 181

[BD97] Shumeet Baluja and Scott Davies. Using optimal dependency-trees for combinatorial optimization: Learning the structure of the search space. In *Proc. 14th International Conference on Machine Learning*, pages 30–38. Morgan Kaufmann, 1997. 181

[Ben75] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, September 1975. 58

[BMLL02] E. Bengoextea, T. Miquelez, P. Larranga, and J.A. Lozano. Experimental results in function optimization with edas in continuous domain. 2002. 81

[BS02] Hans-Georg Beyer and Hans-Paul Schwefel. Evolution strategies - A comprehensive introduction. *Natural Computing*, 1(1):3–52, 2002. 39

[BT96] Tobias Blickle and Lothar Thiele. A comparison of selection schemes used in evolutionary algorithms. *Evolutionary Computation*, 4(4):361–394, 1996. 34, 96

[Cat91] J. Catlett. On changing continuous attributes into ordered discrete attributes. In Y. Kodratoff, editor, *Proceedings of the European Working Session on Learning : Machine Learning (EWSL-91)*, volume 482 of *LNAI*, pages 164–178, Porto, Portugal, March 1991. Springer Verlag. 126

[CH67] T. M. Cover and P. E. Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, IT-13(1):21–7, January 1967. 55

[CV95] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20:273, 1995. 17

[DBT00] Marco Dorigo, Eric Bonabeau, and Guy Theraulaz. Ant algorithms and stigmergy. *Future Generation Comp. Syst*, 16(8):851–871, 2000. 43

[Deb01] Kalyanmoy Deb. *Multi-Objective Optimization using Evolutionary Algorithms*. John Wiley & Sons, Chichester, UK, 2001. 12, 38

[Den] P. J. Denning. Bayesian learning. *American Scientist*, 77:216–218. 17

[DJ75] Kenneth Alan De Jong. *An analysis of the behavior of a class of genetic adaptive systems.* PhD thesis, Ann Arbor, MI, USA, 1975. 32, 94

[DJA02] Kalyanmoy Deb, Dhiraj Joshi, and Ashish Anand. Real-coded evolutionary algorithms with parent-centric recombination. In David B. Fogel, Mohamed A. El-Sharkawi, Xin Yao, Garry Greenwood, Hitoshi Iba, Paul Marrow, and Mark Shackleton, editors, *Proceedings of the 2002 Congress on Evolutionary Computation CEC2002*, pages 61–66. IEEE Press, 2002. 168

[DJR95] D. Dearnaly, I. Judson, and T. Root, editors. *Handbook of adult cancer chemotherapy schedules*. The Medicine Group (Education) Ltd., Oxfordshire, 1995. 178

[Edg86] Eugene S. Edgington. *Randomization tests*, volume 77 of *Statistics: textbooks and monographs*. Marcel Dekker, New York, second edition, 1986. 96

[ES93] Larry J. Eshelman and J. David Schaffer. Real–coded genetic algorithms and interval-schemata. In L. Darrell Whitley, editor, *Foundations of Genetic Algorithms 2*, pages 187–202, San Mateo, 1993. Morgan Kaufmann. 37, 112

[Esh91] Larry J. Eshelman. The CHC adaptive search algorithm: How to have safe search when engaging in nontraditional genetic recombination. In Gregory J. E. Rawlins, editor, *Foundations of Genetic Algorithms*, pages 265–283, San Mateo, 1991. Morgan Kaufmann. 35

[GD91] David E. Goldberg and Kalyanmoy Deb. A comparative analysis of selection schemes used in genetic algorithms. In Gregory J. E. Rawlins, editor, *Foundations of Genetic Algorithms*, pages 69–93, San Mateo, 1991. Morgan Kaufmann. 34, 96

[Glo96] Fred Glover. Tabu search and adaptive memory programing advances, applications and challenges. In *Interfaces in Computer Science and Operations Research*, pages 1–75. Kluwer, 1996. 31

[Gol89] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Reading, Mass., 1989. 32

[Hay99] S. Haykin. *Neural Networks: A Comprehensive Introduction*. Prentice Hall, 1999. 17

[HMMP08] Pierre Hansen, Nenad Mladenovic, and José A. Moreno-Pérez. Variable neighbourhood search: methods and applications. *4OR*, 6(4):319–360, 2008. 30

[HO96] Nikolaus Hansen and Andreas Ostermeier. Adapting arbitrary normal mutation distributions in evolution strategies: the covariance matrix adaptation. In *Proc. of the 1996 IEEE Int. Conf. on Evolutionary Computation*, pages 312–317, Piscataway, NJ, 1996. IEEE Service Center. 68

[HO97] Nikolaus Hansen and Andreas Ostermeier. Convergence properties of evolution strategies with the derandomized covariance matrix adaptation: The $(\mu/\mu_I, \lambda)$-CMA-ES, October 06 1997. 68, 69, 71

[HO01] Nikolaus Hansen and Andreas Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary Computation*, 9(2):159–195, 2001. 71

[Hol75] John H. Holland. *Adaption in Natural and Artificial Systems*. The University of Michigan Press, Ann Arbor, 1975. 32, 34, 35

[JCSW05] Laetitia Jourdan, David Corne, Dragan Savic, and Godfrey Walters. Preliminary Investigation of the 'Learnable Evolution Model' for Faster/Better Multiobjective Water Systems Design. In Carlos A. Coello Coello, Arturo Hernández Aguirre, and Eckart Zitzler, editors, *Evolutionary Multi-Criterion Optimization. Third International Conference, EMO 2005*, pages 841–855, Guanajuato, México, March 2005. Springer. Lecture Notes in Computer Science Vol. 3410. 81, 84, 101

[JKES95] Wnek Janusz, Kaufman Kenneth, Bloedorn Eric, and Michalski Ryszard S. Inductive learning system aq15c: The method and user's guide. MLI 95-4, Mar-1995. 55, 81

[KES01] James Kennedy, Russell C. Eberhart, and Yuhi Shi. *Swarm Intelligence*. Evolutionary Computation Series. Morgan Kaufman, San Francisco, 2001. 44

[KGV83] S. Kirkpatrick, C. D. Gelatt_jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–679, May 1983. 30

[KM99] Kenneth A. Kaufman and Ryszard S. Michalski. Learning from inconsistent and noisy data: The AQ18 approach. In Zbigniew W. Ras and Andrzej Skowron, editors, *ISMIS*, volume 1609 of *Lecture Notes in Computer Science*, pages 411–419. Springer, 1999. 55

[Koh95] Ron Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *IJCAI*, pages 1137–1145, 1995. 16

[Koz92] J. R. Koza. *Genetic Programming*. MIT Press, Cambridge, MA, 1992. 13

[Koz94] J. R. Koza. *Genetic Programming II*. MIT Press, Cambridge, MA, 1994. 13

[LELP99] P. Larrañaga, R. Etxeberria, J. A. Lozano, and J. M. Peña. Optimization by learning and simulation of bayesian and gaussian networks, 1999. 71

[LELP00] Pedro Larrañaga, Ramon Etxeberria, José A. Lozano, and José M. Peña. Combinatonal optimization by learning and simulation of bayesian networks. In Craig Boutilier and Moisés Goldszmidt, editors, *Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence (UAI-00)*, pages 343–352, SF, CA, June 30– July 3 2000. Morgan Kaufmann Publishers. 76

[LL02] P. Larranaga and J. A. Lozano. *Estimation of Distribution Algorithms:A New Tool for Evolutionary Computation*. Kluwer Academic, 2002. 81, 181

[LLB02] P. Larranaga, J.A. Lozano, and E. Bengoextea. Estimation of distribution algorithms based on multivariate normal and gaussian networks. Technical report, Dept Computer Science and Artificial Intelligence, University of the Basque Country, Spain, 2002. 81

[LnLB01] Pedro Larrañaga, Jose A. Lozano, and Endika Bengoetxea. Estimation of distribution algorithms based on multivariate normal distributions and Gaussian networks. Technical Report KZZA-IK-1-01, Dept. of Computer Science and Artificial Intelligence, University of Basque Country, 2001. 76

[MC01] Ryzsard S. Michalski and Guido Cervone. Adaptive anchoring discretization for learnable evolution model. In *George Mason University*, pages 01–3, 2001. 127

[MHF93] Melanie Mitchell, John H. Holland, and Stephanie Forrest. When will a genetic algorithm outperform hill climbing. In Jack D. Cowan, Gerald Tesauro, and Joshua Alspector, editors, *NIPS*, pages 51–58. Morgan Kaufmann, 1993. 28

[Mic69] R. S. Michalski. On the quasi-minimal solution of the general covering problem. In *Proc. Fifth Int. Symposium on Information Processing, FCIP 69*, volume A3, Bled, Yugoslavia, 1969. 51, 53

[Mic96] Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag, 1996. Contains introductory chapter on LCS. 32

[Mic00] Ryszard S. Michalski. Learnable evolution model: Evolutionary processes guided by machine learning. *Machine Learning*, 38(1-2):9–40, 2000. 53, 54, 77, 94, 100, 119, 120, 146

[Mit97] Tom Mitchell. *Machine Learning*. McGraw-Hill, 1997. 16

[Mos89] P. Moscato. On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms. Technical Report Caltech Concurrent Computation Program, Report. 826, California Institute of Technology, Pasadena, California, USA, 1989. 21

[MP96] Heinz Mühlenbein and Gerhard Paass. From recombination of genes to the estimation of distributions I. binary parameters. In Hans-Michael Voigt, Werner Ebeling, Ingo Rechenberger, and Hans-Paul Schwefel, editors, *PPSN*, volume 1141 of *Lecture Notes in Computer Science*, pages 178–187. Springer, 1996. 71

[MT94] R. Martin and K. Teo. Optimal control of drug administration in cancer chemotherapy. *World Scientific*, 1994. 177, 178

[Müh97] Heinz Mühlenbein. The equation for response to selection and its use for prediction. *Evolutionary Computation*, 5(3):303–346, 1997. 75

[MZ00] Ryszard S. Michalski and Qi Zhang. Initial experiments with the LEM1 learnable evolution model: An application to function optimization and evolvable hardware. Technical report, 2000. 94, 100

[PBM05] Andrei Petrovski, Alexander E. I. Brownlee, and John A. W. McCall. Statistical optimisation and tuning of GA factors. In *IEEE Congress on Evolutionary Computation*, pages 758–764. IEEE, 2005. 179

[Pea01] K. Pearson. On lines and planes of closest fit to systems of points in space. *The London, Edinburgh and Dublin Philosophical Magazine and Journal of Science*, 2:559–572, 1901. 58

[Pet99] A. Petrovski. *An Application of Genetic Algorithms to Chemotherapy Treatment*. PhD thesis, Aberdeen, UK, 1999. 178, 179

[PGL99] Martin Pelikan, David E. Goldberg, and O Lobo. A survey of optimization by building and using probabilistic models, October 08 1999. 71

[PH90] G. Pagallo and D. Haussler. Boolean feature discovery in empirical learning. *Machine Learning*, 5(1):71–99, 1990. 51

[PM01] Andrei Petrovski and John A. W. McCall. Multi-objective optimisation of cancer chemotherapy using evolutionary algorithms. In Eckart Zitzler, Kalyanmoy Deb, Lothar Thiele, Carlos A. Coello Coello, and David Corne, editors, *EMO*, volume 1993 of *Lecture Notes in Computer Science*, pages 531–545. Springer, 2001. 176, 179

[PRL+04] J.M. Pena, V. Robles, P. Larranaga, V. Herves, F. Rosales, and M.S. Perez. Ga-eda: Hybrid evolutionary algorithm using genetic and estimation of distribution algorithms. 2004. 81

[PSM06] Andrei Petrovski, Siddhartha Shakya, and John A. W. McCall. Optimising cancer chemotherapy using an estimation of distribution algorithm and genetic algorithms. In Mike Cattolico, editor, *GECCO*, pages 413–418. ACM, 2006. 179

[Qui86] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986. 47, 50, 51, 126, 145

[Qui93] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, CA, 1993. 47, 50, 81, 126

[Rec73] I. Rechenberg. *Evolutionsstrategie: optimierung technischer systeme nach prinzipien der biologischen evolution*. Frommann-Holzboog, 1973. 39, 40

[RIG+00] Michael L. Raymer, William F. Punch III, Erik D. Goodman, Leslie A. Kuhn, and Anil K. Jain. Dimensionality reduction using genetic algorithms. *IEEE Trans. Evolutionary Computation*, 4(2):164–171, 2000. 58

[SC08] Guleng Sheri and David W. Corne. The simplest evolution/learning hybrid: LEM with KNN. In Jun Wang, editor, *2008 IEEE World Congress on Computational Intelligence*, pages 3244–3251, Hong Kong, 1-6 June 2008. IEEE Computational Intelligence Society, IEEE Press. 84, 93, 173

[SC09] Guleng Sheri and David W. Corne. Evolutionary optimization guided by entropy-based discretization. In Mario Giacobini, Anthony Brabazon, Stefano Cagnoni, Gianni A. Di Caro, Anikó Ekárt, Anna Esparcia-Alcázar, Muddassar Farooq, Andreas Fink, Penousal Machado, Jon McCormack, Michael O'Neill, Ferrante Neri, Mike Preuss, Franz Rothlauf, Ernesto Tarantino, and Shengxiang Yang, editors, *EvoWorkshops*, volume 5484 of *Lecture Notes in Computer Science*, pages 695–704. Springer, 2009. 123, 127, 173

[SC10] Guleng Sheri and David Corne. Learning-assisted evolutionary search for scalable function optimization: LEM(ID3). In *IEEE Congress on Evolutionary Computation*, pages 1–8. IEEE, 2010. 143

[Sch88] Hans-Paul Schwefel. Collective intelligence in evolving systems. In W. Wolff, C. J. Soeder, and F. Drepper, editors, *Ecodynamics, Contributions to Theoretical Ecology*, pages 95–100. Springer, Berlin, 1988. 40

[Sch95] Hans Paul Schwefel. *Evolution and Optimum Seeking*. Sixth-Generation Computer Technology Series. John Wiley & Sons, Inc., New York, 1995. 39

[SG91] P. Spirtes and C. Glymour. An algorithm for fast recovery of sparse causal graphs. *Social Science Computer Review*, 9(1):62–73, 1991. 74

[Sha01] C. E. Shannon. A mathematical theory of communication. *SIGMOBILE Mob. Comput. Commun. Rev.*, 5(1):3–55, 2001. 48

[SHL+05] 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. Technical report, 2005. 141, 165, 166, 169, 173

[SP95] Rainer Storn and Kenneth Price. Differential evolution - a simple and efficient adaptive scheme for global optimization over continuous spaces, 1995. 45

[STD05] A. Sinha, S. Tiwari, and K. Deb. A population-based, steady-state procedure for real-parameter optimization. In David Corne, Zbigniew Michalewicz, Bob McKay, Gusz Eiben, David Fogel, Carlos Fonseca, Garrison Greenwood, Gunther Raidl, Kay Chen Tan, and Ali Zalzala, editors, *Proceedings of the 2005 IEEE Congress on Evolutionary Computation*, volume 1, pages 514–521, Edinburgh, Scotland, UK, 2-5 September 2005. IEEE Press. 144, 167

[Sys89] G. Syswerda. Uniform crossover in genetic algorithms. In J. D. Schaffer, editor, *Proceeding of the Third International Conference on Genetic Algorithms*, pages 2–9. Morgan Kaufmann, 1989. 96

[Sys91] Gilbert Syswerda. A study of reproduction in generational and steady state genetic algorithms. In Gregory J. E. Rawlins, editor, *Foundations of Genetic Algorithms*, pages 94–101, San Mateo, 1991. Morgan Kaufmann. 35

[TFM99a] Bäck Thomas, David B. Fogel, and Zbigniew Michalewicz, editors. *Advanced Algorithms and Operators*. IOP Publishing Ltd., Bristol, UK, UK, 1999. 32

[TFM99b] Bäck Thomas, David B. Fogel, and Zbigniew Michalewicz, editors. *Basic Algorithms and Operators*. IOP Publishing Ltd., Bristol, UK, UK, 1999. 32

[Whe88] T. Wheldon. *Mathematical models in cancer research*. Adam Hilger, Bristol, Philadelphia, 1988. 176, 177, 178

[WK88] D. Whitley and J. Kauth. GENITOR: A different genetic algorithm. In *Proceedings of the Rocky Mountain Conference on Artificial Intelligence*, pages 118–130. Denver, CO, 1988. 35

[WM97] David H. Wolpert and William G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, April 1997. 21

[WM05] Janusz Wojtusiak and Ryszard S. Michalski. The LEM3 system for non-darwinian evolutionary computation and its application to complex function optimization. Technical report, George Mason University, 2005. 81

[WM06] Janusz Wojtusiak and Ryszard S. Michalski. The LEM3 implementation of learnable evolution model and its testing on complex function optimization problems. In Mike Cattolico, editor, *GECCO*, pages 1281–1288. ACM, 2006. 81

[YI96] Mutsunori Yagiura and Toshihide Ibaraki. Metaheuristics as robust and simple optimization tools. In *In Proc. IEEE International Conf. Evolutionary Computation*, pages 541–546, 1996. 29

[YS94] L. Yao and William A. Sethares. Nonlinear parameter estimation via the genetic algorithm. *IEEE Transactions on Signal Processing*, 42(4):927–935, 1994. 94, 95, 98

[ZSTF03] Qingfu Zhang, Jianyong Sun, Edward Tsang, and John Ford. Hybrid estimation of distribution algorithm for global optimisation. *Engineering Computations*, 21:2003, 2003. 81

[ZSTF06] Qingfu Zhang, Jianyong Sun, Edward Tsang, and John Ford. Estimation of distribution algorithm with 2-opt local search for the quadratic assignment problem. In *Towards a New Evolutionary Computation. Advances in Estimation of Distribution Algorithm*, pages 281–292. Springer-Verlag, 2006. 81