PERFORMANCE EVALUATION OF EVOLUTIONARY HEURISTICS IN DYNAMIC ENVIRONMENTS

by

Demet Ayvaz

B.S., Computer Engineering, Marmara University, 2004

Submitted to Institute of Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering

Boğaziçi University

2006

PERFORMANCE EVALUATION OF EVOLUTIONARY HEURISTICS IN DYNAMIC ENVIRONMENTS

APPROVED BY:

Prof. Fikret Gürgen (Thesis Supervisor)	……...
Assoc. Prof. Haluk Topçuoğlu (Thesis Co-supervisor)	……..
Prof. Ahmet Ademoğlu	……..
Prof. Ethem Alpaydın	……..
Prof. Oğuz Tosun	……..

DATE OF APPROVAL: 09.06.2006

ACKNOWLEDGEMENTS

First of all, I would like to thank to Prof. Fikret Gürgen and Assoc. Prof. Haluk Topçuoğlu for their support, guidance and motivation throughout the thesis work and especially for giving time and helping me constantly. I also want to thank them for their contribution to my education during graduate and undergraduate years.

I thank Prof. Ethem Alpaydın, Prof. Ahmet Ademoğlu and Prof. Oğuz Tosun for participating in my thesis jury and giving me feedback.

I thank my friends Esma Kılıç and Reyhan Aydoğan for their support, motivation and the unique friendship that we share throughout these years.

Finally, I should express my gratefulness to my family, especially to my mother and father for their love, support, self-sacrifice and their endless thrust in me. Special thanks to my first teacher, my father, for teaching me everything I know. Without him I would never be myself and achieve my current position.

ABSTRACT

PERFORMANCE EVALUATION OF EVOLUTIONARY HEURISTICS IN DYNAMIC ENVIRONMENTS

In stationary optimization problems, it is assumed that no changes occur with respect to the problem solved during the course of computation. However many real-world optimization problems are non-stationary (dynamic) and subject to changes over time with respect to the objective function, the decision variables or the environmental parameters. For dynamic optimization problems the goal of an optimization algorithm is no longer to find a stationary solution, but to continuously track the changing or moving optimum in the problem space.

In this thesis, we present a complete and an extensive performance evaluation of leading evolutionary optimization techniques in dynamic environments. We have examined and implemented a set of 13 evolutionary optimization techniques on a common platform by using the moving peaks benchmark and by varying important problem parameters. Two new algorithms which are the hybridization of the leading techniques in the literature have been proposed in this thesis. Based on the experimental study, it was observed that the hybrid methods outperform the related work with respect to quality of solutions for various parameters of the given benchmark problems. Additionally, a new comparison metric which is based on signal similarity is proposed and used for performance evaluation of algorithms.

The comparison study is based on both artificial problems including moving peaks problems and some of the real-world problems such as scheduling. We have also implemented five evolutionary algorithms which have been designed to solve dynamic job shop scheduling problem. The algorithms are compared in both deterministic and stochastic scheduling environments. The results have shown that there is no algorithm that is best for all environmental conditions.

ÖZET

DİNAMİK ORTAMLARDA BULUŞSAL TEKNİKLER KULLANILARAK PERFORMANSIN DEĞERLENDİRİLMESİ

Statik optimizasyan problemlerinde optimizasyon süresince optimize edilen problemde herhangi bir değişiklik olmadığı varsayılır. Bununla birlikte gerçek hayattaki birçok optimizasyon problemi dinamiktir ve zaman içerisinde amaç fonksiyonu, karar değişkenleri ya da ortamsal parametrelerinde değişikliklere maruz kalır. Dinamik optimizasyon problemlerinde amaç statik bir optimal çözüm bulabilmekten ziyade, çözüm kümesi içerisinde sürekli bir şekilde yer değiştiren optimal değeri olabildiğince iyi takib edebilmektir.

Bu tezde önde gelen evrimsel dinamik optimizasyon tekniklerinin tam ve kapsamlı bir karşılaştırmasını sunuyoruz. 13 evrimsel optimizasyon algoritmasını ortak bir platform olan moving peaks problemi üzerinde önemli problem parametrelerini değiştirerek inceleyip karşılaştırdık. Bu algoritmalardan iki tanesi bu tez kapsamında geliştirilmiş, karma tekniklerdir. Yapılan karşılaştırma sonucunda, karma tekniklerin literatürdeki tekniklerden daha iyi sonuç verdiği gözlenmiştir. Bununla birlikte bu çalışmada sinyal benzerliğini temel alan yeni bir performans ölçütü geliştirilmiş ve algoritmaların karşılaştırılmasında kullanılmıştır.

Yapılan karşılaştırma çalısması moving peaks gibi sanal bir problemin yanı sıra scheduling gibi gerçek bir problemide içermektedir. Dinamik scheduling problemini çözmek üzere dizayn edilmiş 5 evrimsel algoritmayıda inceledik. Algoritmalar stokastik ve determinist scheduling ortamlarında karşılastırıldı. Sonuçlar her ortamda iyi çalısan tek bir algorimanın olmadığını göstermektedir.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iii
ABSTRACT iv
ÖZET v
LIST OF FIGURES ix
LIST OF TABLES xi
LIST OF SYMBOLS / ABBREVIATIONS xv
1. INTRODUCTION 1
2. EVOLUTIONARY ALGORITHMS 5
2.1. Genetic Algorithms 6
2.2. Major Elements of Genetic Algorithms 7
2.2.1. Representation of Individuals 7
2.2.2. Mutation Operator 7
2.2.2.1. Mutation operators for Binary Representation 8
2.2.2.2. Mutation Operators for Integer Representation. 8
2.2.2.3. Mutation Operators for Floating Point Representation 8
2.2.2.4. Mutation Operators for Permutation Representation. 9
2.2.3. Recombination Operator 9
2.2.3.1. Recombination Operators for Binary Representation. 9
2.2.3.2. Recombination Operators for Integer Representation. 10
2.2.3.3. Recombination Operators for Floating-Point Representation. 10
2.2.3.4. Recombination Operators for Permutation Representation. 11
2.2.4. Parent Selection 12
2.2.4.1. Tournament Selection.. 12
2.2.4.2. Proportional Selection 12
2.2.4.3. Rank-Based Selection 13
2.2.5. Population Model and Replacement Strategies 13
3. DYNAMIC ENVIRONMENTS 15
4. ALGORITHMS 21

4.1. SEA: Standard Evolutionary Algorithm 21
4.2. SEAm: Standard Evolutionary Algorithm with Memory 22
4.3. P3: Standard Evolutionary Algorithm with Three Sub-Populations 24
4.4. P3m: Standard Evolutionary Algorithm with Three Sub-Populations and Memory 24
4.5. RI: Random Immigrants 25
4.6. RIm: Random Immigrants with Memory 26
4.7. Memory/search 27
4.8. Memory/2search 28
4.9. SOS: Self Organizing Scouts 29
4.9.1. Creating a New Scout Population 29
4.9.2. Moving the Search Space of Scout Populations 31
4.9.3. Adjusting the Population Size 32
4.9.4. Computing the Next Generation 33
4.10. Multinational GA 34
4.11. Niching: Niching for Dynamic Landscapes 36
4.12. Proposed Hybrid Techniques 37
4.12.1. SOS+LS: Self Organizing Scouts + LS with Crossover Hill-Climbing .. 37
4.12.2. MN+RI : Multinational GA + Random Immigrants 39
5. EXPERIMENTAL STUDY 40
5.1. Moving Peaks Benchmark 40
5.2. Comparison Metrics 42
5.3. Parameter Settings 43
5.4. Experimental Results 45
5.4.1. The Influence of Severity Changes 45
5.4.2. The Influence of Change Frequencies 48
5.4.3. The Influence of Correlation 50
5.4.4. The Influence of Changing the Number of Peaks in the Search Space 52
5.5. Similarity Based Comparison 54
5.5.1. The Influence of Severity Changes 57
5.5.2. The Influence of Change Frequencies 60
5.5.3. The Influence of Correlation 63

5.5.4. The Influence of Changing the Number of Peaks in the Search Space 65
6. JOB SHOP SCHEDULING 68
6.1. The Static Model 68
6.2. The Dynamic Model 70
6.3. Algorithms 71
6.3.1. PSRS: Production Scheduling Rescheduling with Genetic Algorithms 71
6.3.2. FLEX : Anticipation and Flexibility in Dynamic Scheduling 75
6.3.3. SGA: A Genetic Algorithm Approach to Dynamic Scheduling Problem 77
6.3.4. OBGT: Order-Based Giffler and Thomson Genetic Algorithm 80
6.3.5. HCA: Heuristically-Guided GA 82
6.4. Dynamic Job Shop Scheduling Problem Generator 84
6.5. Performance Measures 85
6.6. Parameter Settings 87
6.7. Experimental Results 88
6.7.1. Stochastic Job Shop Scheduling Results 88
6.7.2. Deterministic Job Shop Scheduling Results 93
7. CONCLUSION 98
REFERENCES 100

LIST OF FIGURES

Figure 2.1. The general schema of an Evolutionary Algorithm as a flow-chart..... 5

Figure 2.2. Pseudo-code for typical evolutionary algorithm 6

Figure 3.1. Classification of evolutionary algorithms for dynamic optimization..... 16

Figure 4.1. Pseudo-code for typical evolutionary algorithm 22

Figure 4.2. Pseudo-code for typical evolutionary algorithm with memory..... 23

Figure 4.3. Illustration of standard evolutionary algorithm with memory 23

Figure 4.4. Pseudo-code for standard evolutionary algorithm with three islands 24

Figure 4.5. Pseudo-code for standard evolutionary algorithm with three islands and memory 25

Figure 4.6. Pseudo-code for random immigrants 26

Figure 4.7. Pseudo-code for random immigrants with memory..... 27

Figure 4.8. Illustration of memory based EA with two populations..... 28

Figure 4.9. Pseudo-code for memory/search algorithm..... 28

Figure 4.10. Illustration of memory based EA with three populations..... 29

Figure 4.11. The Self Organizing Scouts Algorithm in pseudo-code..... 30

Figure 4.12. Illustration of the creation of scout populations..... 30

Figure 4.13. Illustration of nation, government and policy 34

Figure 4.14. Illustration of hill-valley detection algorithm 34

Figure 4.15. The pseudo-code for hill-valley detection algorithm 35

Figure 4.16. Pseudo-code for multinational GA..... 36

Figure 4.17. Pseudo-code for niching algorithm 37

Figure 4.18. Pseudo-code for crossover hill-climbing..... 38

Figure 6.1. Example problem with 3 jobs and 3 machines..... 68

Figure 6.2. A feasible schedule for the example problem 69

Figure 6.3. The procedure for decoding a chromosome into a semi-active schedule.... 72

Figure 6.4. The procedure for decoding a chromosome into an active schedule 73

Figure 6.5. The procedure for decoding a chromosome into a non-delay schedule..... 74

Figure 6.6. The procedure for creating an hybrid scheduler..... 75

Figure 6.7. The Giffler and Thomson algorithm (G & T algorithm)..... 78

Figure 6.8. The modification made to G&T algorithm to solve deterministic JSSPs ... 78

Figure 6.9. The modification made to G&T algorithm to solve non-deterministic JSSPs..... 79

Figure 6.10. The modification made to G&T algorithm to adapt the population to the current problem..... 80

Figure 6.11. OBGT Algorithm 81

LIST OF TABLES

Table 5.1. Default settings for the moving peaks benchmark 41

Table 5.2. Default settings for SOS algorithm..... 44

Table 5.3. The offline error of algorithms for different shift lengths 46

Table 5.4. The average execution time and evaluation counts of algorithms for different shift lengths (s)..... 47

Table 5.5. The offline error of algorithms for different change frequencies 48

Table 5.6. The average execution time and evaluation counts of algorithms for different change frequencies (f)..... 49

Table 5.7. The offline error of algorithms for different correlations 51

Table 5.8. The average execution time and evaluation counts of algorithms for different correlations (λ) 51

Table 5.9. The offline error of algorithms for different number of peaks 52

Table 5.10. The average execution time and evaluation counts of algorithms for different number of peaks (n) 53

Table 5.11. Average Euclidean distance and cross correlation of algorithms for different shift lengths (s)..... 57

Table 5.12. Scaled Euclidean distance and cross correlation of algorithms for different shift lengths (s)..... 58

Table 5.13. Similarity based error values of algorithms for different shift lengths..... 59

Table 5.14. Average Euclidean distance and cross correlation of algorithms for different change frequencies (f)..... 60

Table 5.15. Scaled Euclidean distance and cross correlation of algorithms for different change frequencies (f)..... 61

Table 5.16. Similarity based error values of algorithms for different change frequencies..... 62

Table 5.17. Average Euclidean distance and cross correlation of algorithms for different correlations (λ)..... 63

Table 5.18. Scaled Euclidean distance and cross correlation of algorithms for different correlations (λ)..... 64

Table 5.19. Similarity based error values of algorithms for different correlations 64

Table 5.20. Average Euclidean distance and cross correlation of algorithms for different number of peaks (n)..... 66

Table 5.21. Average Euclidean distance and cross correlation of algorithms for different number of peaks (n)..... 66

Table 5.22. Similarity based error values of algorithms for different number of peaks.. 67

Table 6.1. Heuristics used for dynamic job shop problem..... 83

Table 6.2. The notation used in Table 6.1 83

Table 6.3. Weighted flow time values of different algorithms for various utilization rates 89

Table 6.4. Average execution time and evaluation counts for various utilization rates. 90

Table 6.5. Weighted tardiness values of different algorithms for various utilization rates 90

Table 6.6. Average execution time and evaluation counts for various utilization rates (U) 91

Table 6.7. Weighted lateness values of different algorithms for various utilization rates 91

Table 6.8. Average execution time and evaluation counts for various utilization rates(U) 91

Table 6.9. Weighted earliness + tardiness values of different algorithms for various utilization rates 92

Table 6.10. Average execution time and evaluation counts for various utilization rates 92

Table 6.11. Weighted flow time values of different algorithms for various utilization rates in deterministic environment 93

Table 6.12. Average execution time and evaluation counts for various utilization rates in deterministic environment 94

Table 6.13. Weighted tardiness values of different algorithms for various utilization rates in deterministic environment..... 95

Table 6.14. Average execution time and evaluation counts for various utilization rates in determinstihnc environment 95

Table 6.15. Weighted lateness values of different algorithms for various utilization rates in deterministic environment..... 96

Table 6.16. Average execution time and evaluation counts for..... 96

Table 6.17. Weighted earliness + tardiness values of different algorithms for various utilization rates in deterministic environment 97

Table 6.18. Average execution time and evaluation counts for various utilization rates in deterministic environment 97

LIST OF SYMBOLS / ABBREVIATIONS

A	Set of all beginning operations
a_m	Setup time of machine m
B	Set of all conflicting operations
$B(\vec{x})$	Base function for moving peaks benchmark
C_i	Completion time of job i
$cross_corr$	Cross correlation of signals
$\mathcal{D}$i	Relative dynamism of scout i
$\overline{ET}$	Weighted mean earliness + tardiness
e'_t	Best-so-far value up to generation t
$euclid_dist$	Euclidean distance between signals
$\overline{F}$	Weighted mean flow time
$\mathcal{F}$i	Relative fitness of scout i
f_k	Fitness of kth individual
$f_{min,new}$	Minimum relative fitness of a new scout population
$f_{min,old}$	Minimum relative fitness of an old scout population
$h_i(t)$	Height of scout i at time t
$\overline{L}$	Weighted mean lateness
L_i	Lower bound for dimension i
m	Number of machines
max_$clustersize$	Maximum radius for a scout population
n_{off}	Number of offsprings generated by crossover hill-climbing operator
n_t	Number of replacements made by crossover hill-climbing operator
o^*_{ik}	kth operation of job i
$o^*_{i,k+1}$	Ooperation after the kth operation of job i
P1	First parent (parent 1)

P2	Second parent (parent 2)
$\hat{P}$	Flexibility term
$\overline{P}$	Mean processing time
$p_i(t)$	Position of scout *i* at time *t*
$p_{job_predecessor}$	Processing time of the previous opration of the same job
$p_{mac_predecessor}$	Processing time of the previous opration on the same machime
p_{max}	Maximum number of individuals in a scout
p_{min}	Minimum number of individuals in a scout
p_{oper}	Processing time of operation *oper*
r_j	Release time of job *j*
s	Shift length
$\hat{T}$	Fitness term
$\overline{T}$	Weighted mean tardiness
$t_{job_predecessor}$	Starting time of the previous opration of the same job
$t_{mac_prodecessor}$	Starting time of the previous opration on the same machime
$t_{release}$	Arrival time of a new job to the shop floor
t_{oper}	Starting time of operation *oper*
U	Utilization rate
U_i	Upper bound for dimension *i*
$w_i(t)$	Width of scout *i* at time *t*
$weight_{euclid}$	Weight for Euclidean measure
$weight_{cross}$	Weight for cross correlation
α	Weighting factor
β	Length of time to be penalized
δ	Idle time allowance for hybrid scheduler
Λ	Mean inter-arrival time

λ	Correlation coefficient
σ_i	Standard deviation for scout *i*
τ	Time of last change

FLEX	Flexibility based algorithm
GT	Giffler and Thomson
G&T	Giffler & Thomson algorithm
HCA	Heuristically guided genetic algorithm
JSSP	Job Shop Scheduling Problem
Multi-National	Multinational genetic algorithm
MN+RI	Hybridization of multinational GA and Random Immigrants
Mem/search	Memory-search algorithm with one search and one memory population
Mem/2search	Memory-search algorithm with two search and one memory population
mindist	Minimum distance replacement strategy
Nichning	Niching for dynamic landscapes
ND	Non-delay
OBGT	Order-based Giffler & Thomson
P3	Evolutionary algorithm with 3 sub-populations
P3m	Evolutionary algorithm with 3 sub-populations and memory
PPX	Precedence Preservative Crossover
PSRS	Production scheduling re-scheduling
RI	Random Immigrants
RIm	Random Immigrants with Memory
SEA	Standard Evolutionary Algorithm
SEAm	Standard Evolutionary Algorithm with Memory
SGA	Simple genetic algorithm with THX crossover
SOS	Self Organizing Scouts
SOS+LS	Hybridization of Self Organizing Scouts and Local Search
THX	Time Horizon Exchange
WAMS	Worst Among Most Similar

1. INTRODUCTION

In stationary optimization problems, it is assumed that no changes occur with respect to the problem solved during the course of computation and the goal of an optimization algorithm is to find a stationary optimal solution. However many real-world optimization problems are non-stationary (dynamic) and subject to changes over time with respect to the objective function, the decision variables or the environmental parameters. Applications include scheduling, where new jobs have to be added all the time, or manufacturing where the quality of material is changing over time. For dynamic optimization problems the goal of an optimization algorithm is no longer to find a stationary optimal solution, but to continuously track the changing or moving optimum in the problem space.

The problems in dynamic environments are not the same and the characteristics of the changes in an environment may generate different dynamic environments. Some of these characteristics are the frequency of change, the severity of change, the predictability of change, and the cycle length/cycle accuracy [1]. The frequency of change is a property defining how often an environment changes, which is the time for an algorithm to come up with a solution. The severity of change is the property defining the strength of changes in an environment. The predictability of change is a measure of correlation between successive changes. If there is high correlation, it can be possible to predict next change given the past changes. Cycle accuracy/cycle length is the property defining the time for the optimum to return its previous location or a close location to old optimum.

The type of algorithm that will be applied to a problem depends on the characteristics of dynamic environment. Some algorithms may be more accurate than others while dealing with one of the above type of changes. These characteristics of the change may be varied on the same problem to examine the performance of different algorithms.

Genetic Algorithms have been widely used for solving optimization problems in stationary environments. In recent years, there has been a growing interest for investigating and improving the performance of these algorithms in dynamic environments where the fitness landscape changes. The main problem of evolutionary algorithms in dynamic environments is the loose of diversity. Losing diversity is required for convergence and hence and expected behavior in stationary environments. But in dynamic environments a diverse population is needed to adapt changing environmental conditions. A set of evolutionary optimization techniques in dynamic environments, which have different design philosophies and characteristics, have been proposed over the past few years. These techniques can be broadly classified into four categories [2, 3]:

- Approaches reacting after a change
- Approaches maintaining diversity throughout the run
- Memory base approaches
- Multi-population approaches

The algorithms in the first category take explicit actions to increase diversity after a change such as increasing mutation rate drastically or gradually. The algorithms in the second category do not make any action at the time of change. But they avoid convergence and aim to keep diversity high all the time. Memory-based approaches, as the name implies, uses some sort of memory to use old good solutions later. The last approach, multi-population algorithms, divides the population into smaller groups in order to find and track multiple optima in the search space.

Performance evaluation is important for understanding the effectiveness of different algorithms. In order to study the performance of various heuristics for dynamic optimization problems, researchers use some benchmark problems and have developed several dynamic problem generators. Different dynamic test problems are generated by changing the parameters of the stationary problems. The most prominent benchmark problem is the moving peaks problem [1, 4]. It consists of a number of peaks changing in height, width and location.

There are also other benchmarks which are commonly used in dynamic optimization such as dynamic bit-matching, time-varying knapsack problem, scheduling problems …etc.

One of the important research areas in dynamic environments is to measure performance. Performance measures used for standard evolutionary algorithms in static environments such as offline error, best-so-far curves and offline performance are not appropriate for measuring performance in dynamic environments. The most widely used one is the modified offline error described in this section. There are also some other measures for dynamic environments such as collective mean fitness, average of best-of-generation values…etc. A new performance measure is also proposed in this thesis which is based on signal similarity.

The aim of this thesis is to analyze the performance of various evolutionary heuristics in dynamic environments. The comparison study is based on both artificial problems including moving peaks problems and some of the real-world problems such as scheduling. We consider the various metrics to analyze and characterize the changes of a problem over time. Two new hybrid optimization techniques for dynamic environments have been designed by combining local search based techniques with EAs. In this thesis, computational effort of the heuristics will be considered in addition to various metrics for representing the solution quality. Computational effort is reported in terms of average execution time and average number of fitness evaluations. Solution quality is reported in terms of offline error and similarity based error which has been proposed in this thesis. Current performance measures for dynamic environments including the offline error only considers the solution quality and they are limited in measuring the ability to recover after a change. The new metric is proposed to take this ability into account while measuring performance.

The rest of this thesis is organized as follows. Section 2 gives a brief review of evolutionary algorithms. Section 3 presents general information about dynamic environments and summarizes the related work which has been done in the area of dynamic optimization. In section 4, the algorithms which are evaluated and compared in this work including the two heuristics which are the hybridization of leading techniques in the literature are explained in

detail. Section 5 describes the moving peaks benchmark and performance evaluation of the algorithms according to solution quality and computational effort. The results in this section are reported in terms of offline error and the new metric which is called similarity based error. In section 6, dynamic job shop scheduling problem is described. The algorithms which have been proposed to solve this problem are explained in detail and compared. Comparison is made according to four different objectives. Section 7, concludes the thesis.

2. EVOLUTIONARY ALGORITHMS

Evolutionary algorithms are heuristic search and optimization methods which get inspiration from natural evolution. Evolutionary algorithms are proven to be successful in solving optimization problems which could not be solved by conventional optimization techniques. They are applicable for a wide range of problems and pose no restriction on them. Evolutionary algorithms are blind algorithms and they do not use problem specific knowledge and heuristics but these can be easily integrated.

Evolutionary algorithms are not a single algorithm but a family of algorithms consisting of Genetic Algorithms, Evolutionary Programming, Evolution strategies and Genetic Programming. These variants have different historical backgrounds, representations, variation operators and selection schemes. Figure 2.1 shows the general schema for an evolutionary algorithm.

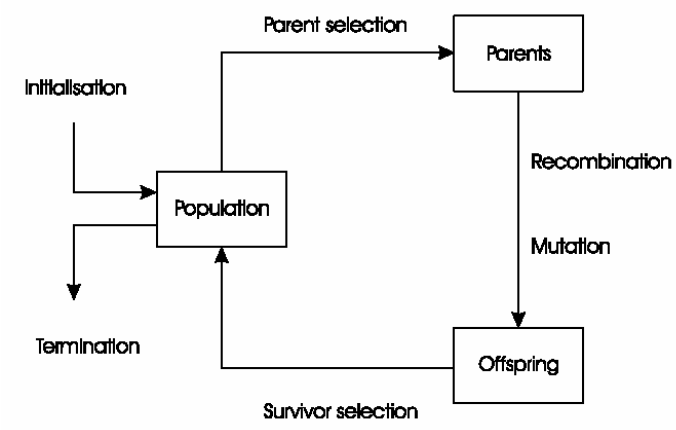


Figure 2.1. The general schema of an Evolutionary Algorithm as a flow-chart [5]

Although there are different variants of evolutionary algorithms, the basic idea for all of them is the same. An evolutionary algorithm starts with a set of candidate solutions which is the initial population, and then at each generation (iteration) these individuals are evaluated with a fitness function. Some individuals are selected according to their fitness and some

selection schema, to form the parent population. By mutating and recombining these parent solutions, new candidate solutions are created, and then new population is selected from these offspring candidate solutions and parent population by using a "survival selection" schema. This strategy resembles the "survival of the fittest" mechanism in natural evolution. The individuals which are better than others can survive and reproduce to generate new population. The basic components of an evolutionary algorithm are representation, parent selection, recombination, mutation and survival selection.

2.1. Genetic Algorithms

The most popular and widely used class of evolutionary algorithms is the Genetic Algorithms. GA is first developed by Holland in 1960's. There is not a single genetic algorithm that fits to any application. There is a variety of different evolutionary algorithm components such as different representations, selection operators …etc. By selecting operators and representations that are most suitable for an application, an effective genetic algorithm can be created for the particular problem at hand. Figure 2.2 shows the pseudo-code for a genetic algorithm.

```
BEGIN
  INITIALISE population with random candidate solutions;
  EVALUATE each candidate;
  REPEAT UNTIL ( TERMINATION CONDITION is satisfied ) DO
    1 SELECT parents;
    2 RECOMBINE pairs of parents;
    3 MUTATE the resulting offspring;
    4 EVALUATE new candidates;
    5 SELECT individuals for the next generation;
  OD
END
```

Figure 2.2. Pseudo-code for typical evolutionary algorithm [5]

2.2. Major Elements of Genetic Algorithms

The major components of an evolutionary algorithm are representation, mutation and recombination operators, selection and replacement strategy. This part describes these major components one by one.

2.2.1. Representation of Individuals

The first step of building a genetic algorithm is deciding how to represent a candidate solution or an individual. An individual consists of a chromosome (genom), a fitness value and some auxiliary variables such as sex and age. The chromosome consists of genes and is a solution to the problem .The encoding used to form the chromosome is the representation. The success of a genetic algorithm for solving an optimization problem heavily depends on the selected representation. The most widely used representations for genetic algorithms are as follows [5]:

- **Binary Representation:** Each chromosome consists of binary digits (a bit string)
- **Integer Representation:** Each chromosome consists of integer values.
- **Real-Valued Representation:** Each chromosome consists of real values.
- **Permutation Representation:** Each chromosome is a different sequence of a set of integer values.

Real-valued representation (floating point representation) is used for the moving peaks problem. Different permutation and direct representations are used for job shop scheduling problem.

2.2.2. Mutation Operator

Mutation is a genetic operator which creates an offspring by altering the genotype of a single parent. There is a large variety of mutation operators. Each type of representation has its

own specialized set of mutation operators. All of them are out of the scope of this work. Some of them will be described in this section.

2.2.2.1. Mutation operators for Binary Representation. The most common mutation operator used for binary representation is the "bit-flipping" operator. This operator considers each bit of an individual separately. If randomly generated value for each bit position is less than the mutation rate, the bit at that location is reversed. Other mutation operators also exist for binary representation but the most common one is the "bit-flipping".

2.2.2.2. Mutation Operators for Integer Representation. There are two commonly used mutation operators for integer representation. Both of these operators use a user-defined probability pm.

Random Resetting: This is an extended version of "bit-flipping". If randomly generated value for each bit position is less than the mutation rate, this operator changes that location of the individual with a randomly chosen integer value among the possible values.

Creep Mutation: This mutation operator is used for problems with ordinal attributes. This operator adds a small positive or negative integer value to each location with a probability of p. The random number generator used for this mutation rate should generate values symmetric about zero and is more likely to generate small values.

2.2.2.3. Mutation Operators for Floating Point Representation. Since allele values can take continuous values instead of discrete ones, above mentioned operators can not be used any more. Floating point operators change each allele value within its domain defined by the lower and upper bounds L_i and U_i respectively.

Uniform Mutation : This operator is similar to bit-flipping. Allele values are replaced by uniform randomly drawn values from $[L_i, U_i]$.

Non-uniform Mutation with a Fixed Distribution: This operator is the most common mutation operator for floating-point representation. It is similar to creep mutation. This operator adds a value which is randomly drawn from a Gaussian distribution with mean zero and user-defined standard deviation. The Gaussian distribution usually creates small values around the mean.

Non-uniform mutation with Gaussian distribution is used for this work.

2.2.2.4. Mutation Operators for Permutation Representation. This type of operators does not consider each gene value independently. Instead of single gene values, the whole chromosome is considered in mutation. Swap, insert, scramble and inversion mutations are the most widely used mutation operators.

Swap mutation: This operator selects two points randomly and swaps their gene values.
Inversion operator: This selects two points randomly and reverses the order of genes between those positions.

2.2.3. Recombination Operator

Recombination is the genetic operator which creates a new solution by combining two or more parent solutions. The term crossover is usually used interchangeably with the term recombination. It is seen as the source of diversity for genetic algorithms. The recombination operator that will be used depends on the type of representation.

2.2.3.1. Recombination Operators for Binary Representation. All of these operators start with two parents and create two offspring solutions.

One-Point Crossover: This operator selects a random position and exchanges the tails of both parents to create offspring solutions.

N-Point Crossover: This operator selects N random points and divides chromosomes into N-1 segments. Then it combines these segments randomly to create offspring solutions.

2.2.3.2. Recombination Operators for Integer Representation. In this representation each gene can take values from a set of integer values. Therefore it is possible to use the same operators used for binary representation.

2.2.3.3. Recombination Operators for Floating-Point Representation. Above mentioned operators can again be used for this type of representation. There are also more commonly used operators for this kind of representation. These operators depend on generating new allele values rather than exchanging old values.

Simple Recombination: This operator selects a random location, copies the parents to the children up to that point and for the rest of the children it takes the arithmetic average of parents.

Whole Arithmetic Recombination: This operator takes the weighted sum of the two parental alleles for each gene. This is the most widely used operator for floating point representation.

Simulated Binary Crossover: The child genes are computed as the linear combination of parent genes. The child genes d_i and e_i are calculated from the parent genes a_i and b_i with the following equations [1]:

$$d_i = 0.5[(1+\beta)a_i + (1-\beta)b_i] \tag{1.1}$$

$$e_i = 0.5[(1-\beta)a_i + (1+\beta)b_i] \tag{1.2}$$

β is a random variable with the following probability distribution.

$$p(\beta) = \begin{cases} 1.5\beta^2 & : \text{ if } \beta \le 1 \\ 1.5\beta^4 & : \text{ otherwise} \end{cases} \tag{1.3}$$

2.2.3.4. Recombination Operators for Permutation Representation. Recombination operators for permutation representation is a bit more complex than other operators since it is not possible to simply change some part of the parents and obtain a new legal permutation. For recombination operators, the information contained in a representation such as the order of elements or the linking between pairs of elements is used to create new offsprings. This kind of operators are designed to transmit as much as possible information from parents to children. There is a wide variety of this sort of operators. In this part we will only consider some the most widely used ones such as partially mapped crossover (PMX) and order based crossover.

Partially mapped crossover: This is the most widely used recombination operator for adjacency-type problems. This operator selects two points randomly. The elements between these points on the first parent are copied to same locations on the first child. Starting from the first point the elements on the same segment of second parent which are not copied to the first child are copied to the locations of the elements which stands on their location on the first child. These steps are described in [5] as follows:

1. Choose two crossover points at random, and copy the segment between them from the first parent (P1) into the first offspring.
2. Starting from the first crossover point look for elements in that segment of the second parent (P2) that have not been copied.
3. For each of these (say *i*), look in the offspring to see what element (say *j*) has been copied in its place from P1.
4. Place *i* into the position occupied *j* in P2, since we know that we will not be putting *j* there.
5. If the place occupied by *j* in P2 has already been filled in the offspring by an element k, put *i* in position occupied by *k* in P2.
6. Having dealt with the elements from the crossover segment, the rest of the offspring can be filled from P2, and the second child is created analogously with the parental roles reserved.

Order crossover: This operator is commonly used for order-based permutation problems. The operator selects two points randomly. The elements between these points on the first parent are copied to the same locations on the first child. The other locations on the first child are filled from the second parent. Starting from the second point on the second parent all unused elements are copied to the first child in the order they appear on the second parent. The other child obtained by reversing the parents.

2.2.4. Parent Selection

Selection is the process of selecting the parents of the next generation. This process enables better individuals to survive and removes the worse individuals by giving them smaller chances for reproduction. The individuals' survival rate is governed by the selection pressure. Too high selection pressure results in early convergence and sub-optimal solutions whereas a too low selection pressure leads to slow convergence.

2.2.4.1. Tournament Selection. For each position in the next generation, this operator makes a tournament to decide the parents. For each tournament k individuals are selected randomly from the current population. The best individual among these k individuals is selected as a parent for the next generation. The tournament size (k) is usually taken to be between 2 and 5. A value higher than five results strong selection pressure and leads to early convergence. Tournament selection is simple to implement, requires a small cost of computation, produces good results in short times. As a result it is the most commonly used selection operator nowadays.

2.2.4.2. Proportional Selection. This operator assigns a probability of survival to each individual. This probability is calculated by dividing the fitness of an individual to the sum of all fitnesses in the current population. According to this operator, the relative fitness of an individual defines the survival chance of an individual. The probabilities are calculated with the following equation.

$$p_i = \frac{fitness_i}{\sum_j fitness_j} \tag{1.4}$$

This selection strategy is also known as roulette wheel selection. Each individual is assigned to a slot according to its probability of survival. The drawback of this strategy is that a few good individuals can quickly take over the entire population since they dominate the large part of the roulette wheel.

2.2.4.3. Rank-Based Selection. Ranking selection is a variant of roulette-wheel selection. In order not to let a few good individuals take over all population, ranking of the individuals is used instead of their relative fitness. All individuals are sorted according to their fitness and each individual is assigned to a survival probability by using a ranking schema. For this work linear ranking is used for some of the algorithms. The ranking schema used for linear ranking is as follows:

$$p_i = \frac{b}{n} - (rank_i - 1)\frac{2(b-1)}{n(n-1)} \tag{1.5}$$

In this equation, n is the population size, $rank_i$ is the rank of the i^{th} member of the population ($rank_i$ is 1 for the best member of the population) and is a constant ($b \in [1.0,2.0]$).

2.2.5. Population Model and Replacement Strategies

There are two different GA models in the literature: the generational model and the steady-state model. The replacement strategy defines how the algorithm updates the population between successive iterations. Generational EAs replace the whole population while steady-state EAs replace only a fraction of it. Replacement is also known as survival selection.

In generational model, the entire population is replaced by the newly created offspring at every generation. In this model, a mating pool of population size is selected from the main population using one of the selection strategies. From these parents λ ($\lambda =$ population size) offsprings are generated with recombination and mutation. These offsprings replace the main population totally and become the main population for the next generation. By the way generational replacement works stochastically. There is a chance of replacing better individuals with worse ones. Even the best individual may not be preserved in the next generation. To prevent this, generational replacement is usually applied with elitism to preserve best k individuals between generations. In this work, generational replacement with elitism of 1 is used.

In steady-state model, only a fraction of population is replaced by the newly created offsprings. From μ (population size) parents only λ ($\lambda < \mu$) offsprings are generated and these offsprings replaces the worst individuals in the main population. The introduction of this model is Whitley's GENITOR algorithm [6]. Since than this model is widely used with $\lambda = 1$.

3. DYNAMIC ENVIRONMENTS

In stationary optimization problems, it is assumed that the fitness landscape does not change during optimization process. In such an environment the goal of an optimization algorithm is to locate a stationary optimum. For real world optimization problems, the environment is usually dynamic due to the changes in the fitness landscape or changes in the constraints of the problem such as design variables and environmental conditions. In such an environment the goal of an optimization algorithm is to locate a changing optimum continually. Such an algorithm should have an adaptive behavior.

Solving dynamic optimization problems is a challenging task and traditional evolutionary algorithms are not suitable for solving these problems. When genetic algorithms converge to an optimum, all population members become similar to each other and algorithm looses diversity. If the optimum changes due to environmental changes, the algorithm can not search effectively for the new optimum. As a result new approaches have been developed which are capable of adapting solutions to a changing environment. In order to use such an adaptive algorithm, the dynamic landscape should have some similarities before and after a change. Only this type of environments are considered in dynamic optimization otherwise adaptation is not possible.

The simplest approach for solving dynamic optimization problems is considering each change as the arrival of a new optimization problem. In this approach no information is transferred between the optimization problems before and after a change. This approach may be the best choice if the environment changes so severely that there is no relation between the old and new optimum. Most of the time this is not the case and the information should be transferred between the optimization problems. This can be done by transferring individuals. For some problems individuals should be adapted before being transferred. This is necessary for the type of changes that affect the individual's representation. The question that should be answered at this point is the amount of information that will be transferred. Transferring too

many individuals may result to an early convergence problem while transferring too few individuals may lead to loss of information.

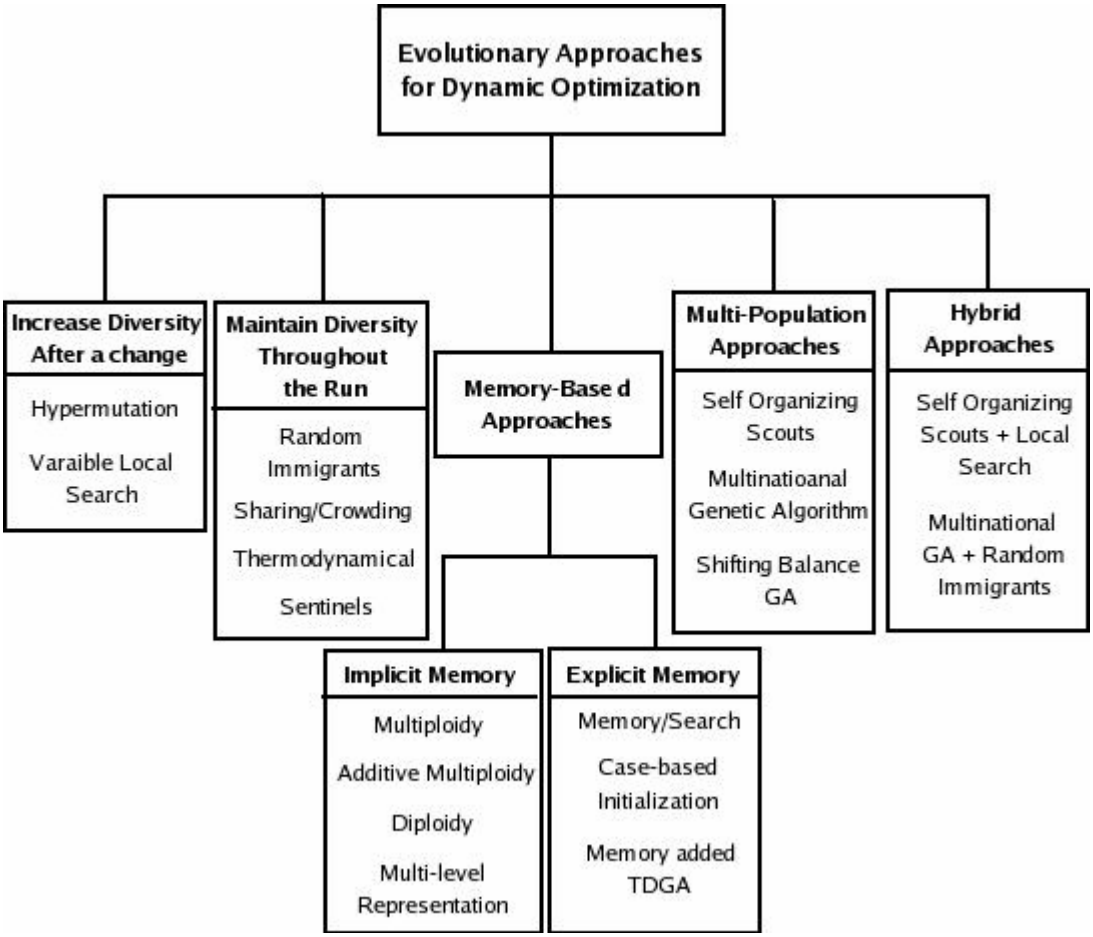


Figure 3.1. Classification of evolutionary algorithms for dynamic optimization

Several approaches have been developed to deal with optimization problems in dynamic environments and to adapt evolutionary algorithms to dynamic environments (see Figure 3.1). The four main approaches [2, 3] are increasing diversity after a change [7, 8, 9], maintaining diversity throughout the run [10, 11, 12], using implicit or explicit memory [13, 14, 15, 16] to reuse old solutions and multi-population approaches. As mentioned before, the main problem of evolutionary algorithms in dynamic environments is the loss of diversity which is necessary to adapt to a changing environment. Diversity is a key concept in dynamic optimization since

it defines the exploration capability of a search population. These four approaches take different actions to deal with diversity problem.

The first approach is increasing diversity after a change. The algorithms in this category take extra actions to increase diversity after a change. With other words, these algorithms react on changes. One of this type of algorithms is hypermutation [8]. The idea is to increase mutation rate drastically while keeping the main population after a change. Deciding the mutation rate is important for the performance of this algorithm. A small increase in the mutation rate may not create the needed diversity for finding new optimum. Whereas a great increase in the mutation rate may lead the same results as restart. Recent researches have shown that hypermutation rate is related to the frequency of changes. Higher frequency of changes requires higher hypermutation rates. Another algorithm in this category is Variable Local Search (VLS) [9]. This algorithm is a variant of hypermutation. VLS increases mutation rate gradually after detecting a change. First, small changes are made. If population fitness does not increase, higher changes are applied.

The second approach is to maintain diversity throughout the run, with other words avoiding convergence all the time. These type of algorithms aim to keep a diverse population all the time since a spread-out population may easily adapt to changing environmental conditions. The most known diversity maintaining algorithm is the Random Immigrants [10] schema. This algorithm replaces some individuals with random ones in every generation. Another technique which aims to control and maintain diversity all the time is Thermo-dynamical GA (TDGA) [11]. This algorithm controls a parameter called free energy to control diversity in the population. Another diversity maintaining technique has been proposed by Cedeno and Vemuri [12] which uses crowding selection and a replacement schema called "*Worst Among Most Similar (WAMS)*".

The third approach for solving dynamic optimization problems is to support evolutionary algorithm with some sort of memory. This memory support is used to store good individuals and reuse them later. The memory can be an implicit memory or an explicit one. This type of approach is especially useful for environments whose optimum oscillates between

some values which are known as periodic environments. Implicit memory means using redundant representations. In this approach the chromosome carries more information than phenotype of the individuals. The most known approach for redundant representations is diploidy. Another form of memory is the explicit memory which means using an extra memory to store some good individuals and reuse them later. The size of the memory, selection of individuals that will be stored, replacement strategy for the memory and for the population affect the performance of algorithms which use extra memory. One of these algorithms is proposed by Ramsey and Grefenstette [14]. This algorithm employs case base reasoning to select individuals which will be re-inserted to the main population by measuring the environmental conditions and indexing memory with these conditions. Branke proposed an algorithm called memory search [1, 13] which divides the population into two subpopulations, which are "*memory population*" and "*search population*".

The last approach developed to solve dynamic optimization problems is the multi-population approaches. The main idea of these algorithms is to divide population into smaller groups in order to find and track multiple optima in the search space. In a dynamic environment finding and controlling the global optimum is not sufficient since any local optimum may increase as a result of environmental changes and become the global optimum. This type of algorithms aims to follow all promising areas. One of these algorithms is the self organizing scouts (SOS) [17, 18] algorithm which is proposed by Branke. The main idea of this algorithm is to divide base population and the search space to find multiple optima in the search space. The algorithm divides the population into one base and many scout populations. The base population is responsible for exploring the search space and finding new peaks while the scout populations are responsible for exploiting small areas. Another multi-population approach is the shifting balance GA [19]. The idea of this algorithm is similar to SOS but for this algorithm, base population makes exploitation and the child populations are responsible for exploration. Another multi-population algorithm is multinational GA [20] which is proposed by Ursem. This algorithm divides the main population into nations by using a procedure called hill-valley detection. By this way, it aims to group the individuals according to peaks in the landscape and follow all promising areas.

On the other hand, not all dynamic optimization problems are the same. The characteristics of the changes in an environment may generate different dynamic environments. These characteristics are frequency of change, severity of change, predictability of change, and cycle length/cycle accuracy [1]. Frequency of change is a property defining how often an environment changes. This is the time for an algorithm to come up with a solution. Severity of change is the property defining the strength of changes in an environment. Predictability of change is a measure of correlation between successive changes. If there is high correlation then it can be possible to predict next change given the past changes. Cycle accuracy/cycle length is the property defining the time for the optimum to return its previous location or a close location to old optimum. Performance evaluation is important for understanding the effectiveness of different algorithms.

A set of evolutionary optimization techniques in dynamic environments, which have different design philosophies and characteristics, have been proposed over the past few years. The type of algorithm that will be applied to a problem depends on the characteristics of dynamic environment. Some algorithms may be more accurate than others while dealing with one of the above type of changes. These characteristics of the change may be varied on the same problem to examine the performance of different algorithms.

Additionally, researchers consider a set of syntactically-generated benchmarks [21, 22, 23] as well as benchmarks from real world problems [24, 25, 26, 27] to measure the performance of their methods. These problems are not the same and the characteristics of the changes in an environment may generate different dynamic environments. There are even more than 20 different dynamic optimization problems which have been used so far. Most of these problems are not much common in use. A good benchmark problem should have some properties [1]:

- It should be computationally efficient
- It should be simple to analyze
- It should be simple to implement

- It should be possible to vary environmental parameters
- It should allow conjectures to real world problems

Some of the problems which satisfy these properties and used commonly in dynamic optimization are dynamic bit-matching [28, 29], moving parabola [30, 31], time-varying knapsack problem [15, 32, 33], moving peaks [4, 22], scheduling problems [24, 25, 26, 27], oscillating peaks [1, 13]…etc.

One of the important research areas in dynamic environments is measuring performance. The most widely used performance measure is the offline error (modified version) which is based on the offline performance proposed by De Jong [34]. Offline error is the running average of the difference between the best individual encountered so far and the optimum at any time. Another measure used for measuring the performance of evolutionary algorithms in static environments is the best-so-far curves which are the plots of generation versus best fitness encountered so far. This measure is not appropriate for dynamic environments since old best values become meaningless after a change in the environment. In order to measure performance in dynamic environments, researches used new measures. The most widely used one is the modified offline error described in this section. There are also some other measures for dynamic environments [35]

- Average Euclidean distance to the optimum at each generation [36]
- Modified offline error [1]
- The difference between the best fitness obtained just before the change and the optimum value [37]
- Average of best-of-generation values for each generation [38]
- For a fixed number of generations (window) the difference between the best-of-generation and the worst fitness encountered, divided by the difference between the best and worst values encountered within the same [39]
- Collective mean fitness [35]

4. ALGORITHMS

Genetic Algorithms have widely been used for solving optimization problems in stationary environments. In recent years, there has been a growing interest for investigating and improving the performance of these algorithms in dynamic environments. In this work, we present a complete and an extensive performance evaluation of leading evolutionary optimization techniques in dynamic environments. We have examined and implemented a set of 11 evolutionary optimization techniques on a common platform and tested them using the same suite of benchmark with a wide range of parameters. Although there are comparison studies [7, 33, 40,] presented in the literature, they are limited in classes of algorithms considered and/or limited in diversity of metrics and parameters of the benchmark problem.
The algorithms in the set can be classified into three distinct categories [2, 3] according to their functionalities: i) approaches that maintain diversity throughout the run, ii) memory-based approaches, and iii) multi-population approaches.

4.1. SEA: Standard Evolutionary Algorithm

Evolutionary algorithms are heuristic search and optimization methods which get inspiration from natural evolution. An evolutionary algorithm starts with a set of candidate solutions which is the initial population, and then at each generation (iteration) these individuals are evaluated with a fitness function, some individuals are selected according to their fitness and some selection schema, to form the parent population. By mutating and recombining these parent solutions, new candidate solutions are created, and then new population is selected from these offspring candidate solutions and parent population by using a “survival selection” schema. The basic components of an evolutionary algorithm are representation, parent selection, recombination, mutation and survival selection. Figure 4.1 is the pseudo-code of standard evolutionary algorithm.

```
BEGIN
  INITIALISE population with random candidate solutions;
  EVALUATE each candidate;
  REPEAT UNTIL ( TERMINATION CONDITION is satisfied ) DO
    1 SELECT parents;
    2 RECOMBINE pairs of parents;
    3 MUTATE the resulting offspring;
    4 EVALUATE new candidates;
    5 SELECT individuals for the next generation;
  OD
END
```

Figure 4.1. Pseudo-code for typical evolutionary algorithm [5]

By standard evolutionary algorithm in this work, we mean a genetic algorithm which uses floating point representation, generational replacement with an elite of one, rank based selection, simulated binary crossover and gaussian mutation.

4.2. SEAm: Standard Evolutionary Algorithm with Memory

SEAm [1] is the standard evolutionary algorithm which uses an extra memory to store good individuals and reuse them later. The pseudo-code for this algorithm is shown in Figure 4.2

The algorithm starts as a standard evolutionary algorithm. At every x^{th} generation, the best of population is written into memory. If memory is full, a memory replacement strategy is used. For this work, mindist is used as a replacement strategy. According to mindist strategy, the most similar two old solutions are selected from the memory, and the worst one is replaced by the entering solution only if the new solution is better than the old one. Whenever the environment changes, the main population and memory are merged and the best n individuals are selected to form main population.

```
BEGIN
   INITIALIZE population with random candidate solutions;
   EVALUATE each candidate;
   REPEAT UNTIL (TERMINATION COND. is satisfied) DO
      1.  SELECT parents;
      2.  RECOMBINE pairs of parents;
      3.  MUTATE the resulting offspring;
      4.  EVALUATE new candidates;
      5.  IF(memory_update_generation)
            i.  Define the best member of generation
            ii. IF(memory_full)
                     Replace old_member with mindist
                ELSE
                     Put member to the memory
      6.  IF(Environment changes)
                Merge memory and the population;
      7.  SELECT individuals for the next generation;
    END REPEAT;
END;
```

Figure 4.2. Pseudo-code for typical evolutionary algorithm with memory

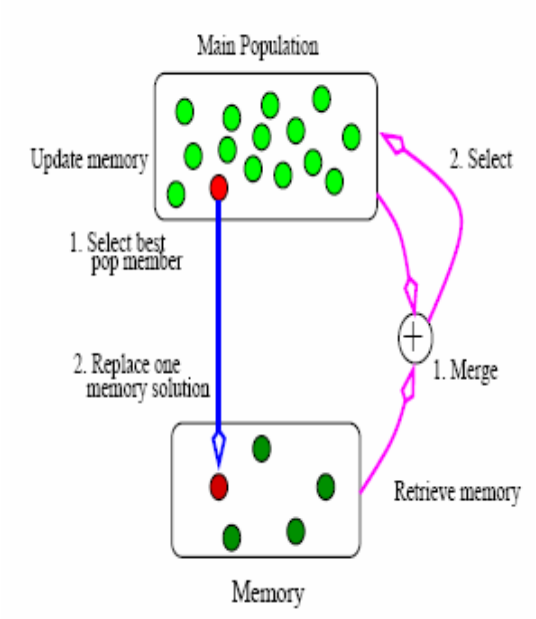


Figure 4.3. Illustration of standard evolutionary algorithm with memory

4.3. P3: Standard Evolutionary Algorithm with Three Sub-Populations

For this algorithm [1], the whole population is divided into three independent sub-populations of equal size. Each population explores the search space independently. Only the members of same island are allowed to recombine. Migration between islands is not allowed in order to keep diversity high. Figure 4.4 shows the pseudo-code for this algorithm.

```
BEGIN
   INITIALIXE the three sub-populations with random
   candidate solutions;
   EVALUATE each candidate for all populations;
   REPEAT UNTIL (TERMINATION COND. is satisfied) DO
         FOR each population DO
         1. SELECT parents;
         2. RECOMBINE pairs of parents;
         3. MUTATE the resulting offspring;
         4. EVALUATE new candidates;
         5. ELITISM of 1;
         6. SELECT individuals for the next generation;
         END FOR;
    END REPEAT;
END;
```

Figure 4.4. Pseudo-code for standard evolutionary algorithm with three islands

4.4. P3m: Standard Evolutionary Algorithm with Three Sub-Populations and Memory

For this algorithm [1], the whole population is divided into three independent sub-populations of equal size. Each population explores the search space independently. Only the members of same island are allowed to recombine. Migration between islands is not allowed in order to keep diversity high. The algorithm is also supported with an extra memory. At every x^{th} generation, the best of all three populations is written into memory. If memory is full, a memory replacement strategy is used. For this work, mindist is used as a replacement strategy. According to mindist strategy, the most similar two old solutions are selected from the memory, and the worst one is replaced by the entering solution only if the new solution is better than the old one. Whenever the environment changes, one of the populations and

memory are merged and the best n individuals are selected to form new population. Figure 4.5 shows the pseudo-code for this algorithm.

```
BEGIN
    INITIALIXE the three sub-populations with random
    candidate solutions;
    EVALUATE each candidate for all populations;
    REPEAT UNTIL (TERMINATION COND. is satisfied) DO
          FOR each population DO
          1.  SELECT parents;
          2.  RECOMBINE pairs of parents;
          3.  MUTATE the resulting offspring;
          4.  EVALUATE new candidates;
          5.  ELITISM of 1;
          6.  IF(memory_update_generation)
                i.  Define the best member of generation
                ii. IF(memory_full)
                        Replace old_member with mindist
                    ELSE
                        Put member to the memory
          7.  IF(Environment changes)
                    Merge memory and the population;
          8.  SELECT individuals for the next generation;
          END FOR;
     END REPEAT;
END;
```

Figure 4.5. Pseudo-code for standard evolutionary algorithm with three islands and memory

4.5. RI: Random Immigrants

Random immigrants method [7, 10] (proposed by Grefenstette in 1992) starts as a standard evolutionary algorithm. At each generation, random individuals are created and a fraction of the main population is replaced by these random individuals. Different replacement methodologies can be considered. Replacing the worst ones may be a good choice; and, inferior solutions may also be replaced. The algorithm considered in this work, replaces the worst individuals in the population; and by this way, it maintains diversity throughout the execution. At each generation 25 percent of the old population is replaced by the randomly generated individuals. Figure 4.6 shows the pseudo-code for this algorithm.

```
BEGIN
   INITIALIZE population with random candidate solutions;
   EVALUATE each candidate;
   REPEAT UNTIL (TERMINATION COND. is satisfied) DO
      1. SELECT parents;
      2. RECOMBINE pairs of parents;
      3. MUTATE the resulting offspring;
      4. EVALUATE new candidates;
      5. ELITISM of 1;
      6. GENERATE random individuals
      7. REPLACE the worst individuals with random ones
      8. SELECT individuals for the next generation;
    END REPEAT;
END;
```

Figure 4.6. Pseudo-code for random immigrants

4.6. RIm: Random Immigrants with Memory

The algorithm [1] starts as a standard evolutionary algorithm. At each generation percent of main population is replaced by randomly generated individuals. The algorithm is also supported with an extra memory. At every x^{th} generation, the best of population is written into memory. If memory is full, a memory replacement strategy is used. For this work, mindist is used as a replacement strategy. According to mindist strategy, the most similar two old solutions are selected from the memory, and the worst one is replaced by the entering solution only if the new solution is better than the old one. Whenever the environment changes, the main population and memory are merged and the best n individuals are selected to form main population. Figure 4.7 shows the pseudo-code for this algorithm.

```
BEGIN
   INITIALIZE population with random candidate solutions;
   EVALUATE each candidate;
   REPEAT UNTIL (TERMINATION COND. is satisfied) DO
     1. SELECT parents;
     2. RECOMBINE pairs of parents;
     3. MUTATE the resulting offspring;
     4. EVALUATE new candidates;
     5. ELITISM of 1;
     6. IF(memory_update_generation)
          i. Define the best member of generation
          ii. IF(memory_full)
                    Replace old_member with mindist
              ELSE
                    Put member to the memory
     7. IF(Environment changes)
           Merge memory and the population;
     8. GENERATE random individuals
     9. REPLACE the worst individuals with random ones
     10. SELECT individuals for the next generation;
     END REPEAT;
END;
```

Figure 4.7. Pseudo-code for random immigrants with memory

4.7. Memory/search

This algorithm [1, 13] divides the whole population into two sub-populations: “Memory Population” and “Search Population”. At every x^{th} generation, the best member of two populations (the whole population) is written into memory. If memory is full, a memory replacement strategy is used. For this work, mindist is used as a replacement strategy. Whenever the environment changes, the memory population and memory are merged and the best n individuals are selected to form memory population. The search population is reinitialized in order to increase diversity. Figure 4.8 is an illustration of memory/search algorithm. Figure 4.9 shows the pseudo-code for this algorithm.

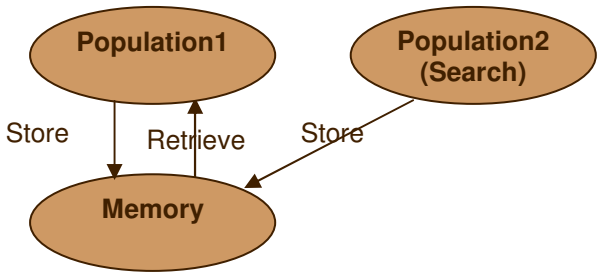


Figure 4.8. Illustration of memory based EA with two populations

```
BEGIN
    INITIALIXE the two sub-populations with random
    candidate solutions;
    EVALUATE each candidate for all populations;
    REPEAT UNTIL (TERMINATION COND. is satisfied) DO
          FOR each population DO
          1.  SELECT parents;
          2.  RECOMBINE pairs of parents;
          3.  MUTATE the resulting offspring;
          4.  EVALUATE new candidates;
          5.  ELITISM of 1;
          6.  SELECT individuals for the next generation;
          END FOR;
          7.  IF(memory_update_generation)
              i.  Define the best member of generation
              ii. IF(memory_full)
                      Replace old_member with mindist
                  ELSE
                      Put member to the memory
          8.  IF(Environment changes)
              i.  Merge memory and the memory population
              ii. Select best n individuals as the memory
                  population
              iii. Reinitialize search population
    END REPEAT;
END;
```

Figure 4.9. Pseudo-code for memory/search algorithm

4.8. Memory/2search

This algorithm is a variant of memory/search [1, 13]. The only difference between these two algorithms is the number of search populations. Memory/2search uses two search populations instead of one. Its working procedure is the same as memory/search. As in the

memory/search algorithm, whenever the environment changes, the memory population and memory are merged. Then the best n individuals are selected to form memory population. The two search populations are reinitialized. Figure 4.10 is an illustration of memory/search algorithm. The pseudo-code for this algorithm is the same as Figure 4.9.

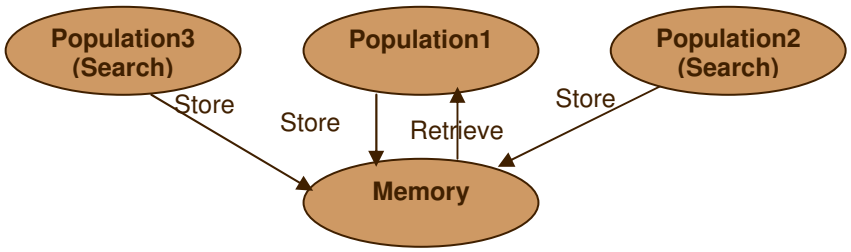


Figure 4.10. Illustration of memory based EA with three populations

4.9. SOS: Self Organizing Scouts

Self organizing scouts (SOS) [1, 17] is a multi-population optimization algorithm inspired from forking GA [41] method. The main idea of this algorithm is to divide base population and the search space whenever a peak has been found. In each forking generation the base population is analyzed to check whether the forking conditions are satisfied. If that is the case, a small fraction of base population (scout population) is split from the base population and assigned to watch a small region which is thought to be a peak. Whenever this peak moves, the scout watching this peak moves with it. The sizes of the scout and base population are not fixed. At each forking generation the total number of individuals is divided between base and scout populations according to their quality. Figure 4.11 shows the pseudo-code for this algorithm.

4.9.1. Creating a New Scout Population

Each scout population is defined by its center and range. Center is the location of the best member of a scout and range is the distance between the center and furthest member of

the scout. There are limits on the number of individuals in a scout and the ranges of scouts. The allowed minimum and maximum numbers of individuals in a scout are p_{min} and p_{max} respectively. The allowed minimum and maximum ranges for a scout are r_{min} and r_{max} respectively. Fitness of a scout population is the fitness of its best member. The fitness of a new scout population must be higher than some ratio of currently overall best individual. This ratio is $f_{min,new}$. The fitness of old scout populations must also be higher than some ratio of currently overall best individual. This ratio is $f_{min,old}$.

REPEAT

Compute next generation of base population and scout populations

Adjust search space of scout populations

IF (forking generation)

Create new scout population if suitable cluster is found

Adjust number of individuals in base and scout populations

UNTIL termination criterion

Figure 4.11. The Self Organizing Scouts Algorithm in pseudo-code [1]

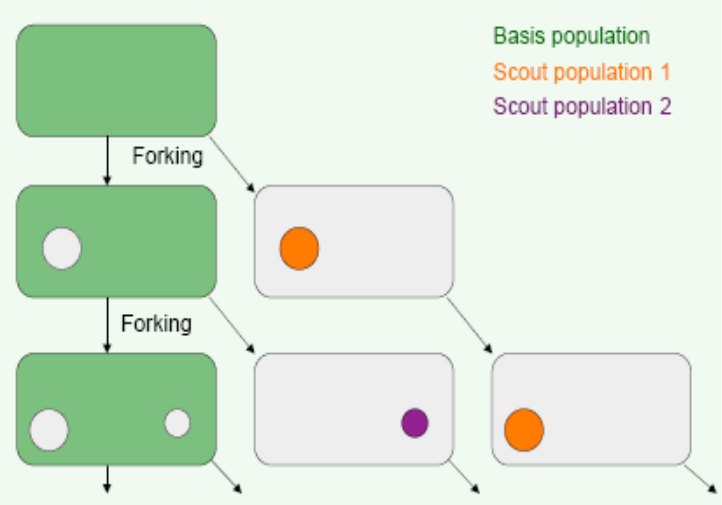


Figure 4.12. Illustration of the creation of scout populations [1]

Whenever the above constraints are satisfied by a group of individuals in the base population, these individuals are split from the base population to create a scout population.

The number of scout populations is also limited, but this limit is defined by the minimum number of individuals in a scout population. If this limit is exceeded, the newly created scout population replaces the old forking population with the lowest fitness.

4.9.2. Moving the Search Space of Scout Populations

During optimization each scout population evolves to find better solutions. If better solutions are found, the center of the scout population changes and the scout moves. Another cause of scout movement can be the movement of the peak which is being watched by the scout. Whenever the peak moves, the fitness of the members of the scout change and the center of the scout may change as well. During this movement some individuals may lie outside the scout's search space. These individuals are discarded as long as the number of individuals in a scout is above p_{min} otherwise these random individuals are replaced by random ones.

Overlapping of search spaces of scout populations is allowed as long as their centers do not fall into other scout's regions. Whenever this occurs, the scouts are merged. The radius of the new scout population is calculated as follows [1]:

$$radius = \min\left\{\sqrt[n]{radius_1^n + radius_2^n}, \max_clustersize\right\} \tag{4.1}$$

The center of new scout population is defined to be center of fitter scout. All members of merging scout populations are checked for the new center and new range. The ones whose distance to new center is less than *radius* is integrated into the new scout and others are discarded.

4.9.3. Adjusting the Population Size

Distributing individuals efficiently between base and scout populations is an important concept for SOS algorithm. The main idea is to place more individuals on more promising areas which have high quality and high dynamics. At each forking generation, scout populations which have fitness less than the minimum required fitness is deleted, and then a quality measure is calculated for all scout populations and the base population. The quality is the linear combination of a scout's dynamism and fitness measures. The quality is calculated as shown below [1]:

Numpops(t): total number of populations P_i existing at time t

$F_i(t)$: fitness of best individual of population i at time t

α : user defined parameter to weigh dynamism against fitness

$$fitpops(t) = \left|\{P_i \,|\, F_i(t) > F_i(t-1)\}\right|$$

$$\beta = \frac{fitpops(t)}{numpops(t)} \tag{4.2}$$

$$F_{\min}(t) = \min_{j=1\ldots numpops(t)} F_j(t) \tag{4.3}$$

$$D_i(t) = \max\left\{0, \frac{F_i(t) - F_i(t-1)}{F_i(t-1)}\right\} \tag{4.4}$$

The relative fitness $\mathcal{F}_i$ and relative dynamism $\mathcal{D}i$ of a population i is calculated with the following equations:

$$\mathcal{F}_i = \begin{cases} \dfrac{F_i(t) - F_{\min}(t)}{\sum_j (F_j(t) - F_{\min}(t))} & : \quad \sum_j (F_j(t) - F_{\min}(t)) > 0 \\ 1/n & : \quad otherwise \end{cases} \tag{4.5}$$

$$\mathcal{D}_i = \begin{cases} \dfrac{D_i(t)}{\sum_j D_j(t)} & : \ \sum_j D_j(t) > 0 \\ 1/n & : \ otherwise \end{cases} \tag{4.6}$$

Then the relative quality measure is calculated for each population with the following equation:

$$Q_i(t) = \alpha\beta D_i(t) + (1 - \alpha\beta) F_i(t) \tag{4.7}$$

Then the number of individuals that should be distributed to population i at time (t+1) is defined as:

$$S_i(t+1) = \frac{Q_i(t) \cdot (popsize) + S_i(t)}{2} \tag{4.8}$$

In this equation $S_i(t)$ is the number of individuals that scout i has at time t and $S_i(t+1)$ is the number of individuals that scout I will generate in the next generation. Popsize is the total number of individuals.

4.9.4. Computing the Next Generation

While computing the population of next generation, recombination is only allowed between the individuals of the same population. Each newly created individual is checked to see if it lies in the search space of its parent population. If it is not in the allowed search space, it is discarded and a new individual is created.

The mutation step length of a scout population is not equal to the mutation step length of the base population. It is calculated with the following equation:

$$\sigma_i = \sigma_0 \frac{3 \cdot r_i}{\text{maximal distance in one dimension}} \tag{4.9}$$

4.10. Multinational GA

This algorithm is proposed by Ursem [20]. The whole population is considered as the world consisting of nations. Each nation is a subpopulation and has a government, a policy and a population. Figure 4.13 shows an illustration of a nation. The algorithm starts with a single nation and all members of the world are assumed to be from this single nation. The government of a nation are the best g (g is usually taken to be 8) members of a nation. The policy is the average of these government members.

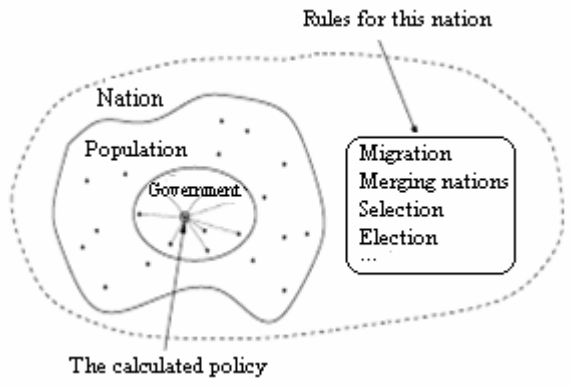


Figure 4.13. Illustration of nation, government and policy [20]

A method called hill-valley detection is used to cluster the whole population into nations. At each generation each individual is first compared to its nations' policy to detect if there is a valley between this member and its nation. If that is the case, this member migrates to another nation or it forms a totally new nation. To decide which nation to migrate hill-valley detection algorithm is used. Figure 4.14 shows an illustration of hill-valley detection algorithm.

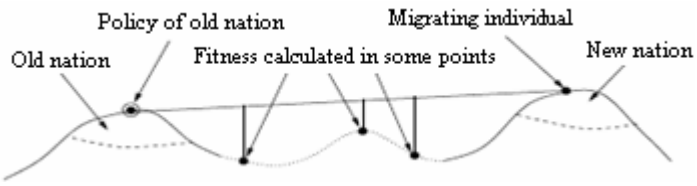


Figure 4.14. Illustration of hill-valley detection algorithm [20]

```
Hill-valley(point p, point q)
        Define n random sample points (s[]) between p and q
        FOR i=1 to n
                IF (fitness(s_i)>min(fitness(p),fitness(q))
                        return i;
        END FOR
RETURN 0;
```

Figure 4.15. The pseudo-code for hill-valley detection algorithm

Figure 4.15 shows the pseudo-code for hill-valley detection algorithm. If hill-valley detection returns a value other than zero, this means there is a valley between points p and q. If p is an individual and q is the policy of the nation which includes p, and there is a valley between them, then p should migrate to another nation. To decide which nation to migrate, p is compared to policies of other nations with hill-valley detection algorithm. If there does not exist a valley between one of them and p, p migrates to this nation. If none of the existing nations owns the individual p. This individual is assumed to lie on a totally new peak and it forms a new nation.

Another important part of clustering step is the merging of nations. In this step, hill-valley detection algorithm is applied to policies of nations. Assume p and q are the policies of two different nations. If no valley is detected between the two policies, the nations are combined to create a new nation.

After clustering step is complete for all individuals, new generation is created by applying selection, recombination and mutation to existing individuals. Recombination is done on global and national level. The amount of national and global recombination is defined with a weight. For national recombination only the members of the same scout are recombined, while in global recombination any member in the population can be recombined.

After recombination mutation is performed. Distance to policy based mutation is used for this purpose. The idea of this operator is to mutate individuals according to their distance to policy. The individuals which are far from the policy are mutated with a high mutation rate

and the individuals which are closer to policy are mutated with low mutation rates. The pseudo-code for multinational genetic algorithm is seen in Figure 4.16.

```
1. At time t=0 all individuals belong to same
   nation

2. REPEAT

      a) For each individual,
         Hill-valley is used to decide,
         if it should stay in its nation
         if it should migrate to another nation
         if it should form a new nation

      b) The nations approaching to same peak are merged

      c) Selection and recombination are made on national and
         globel level

      d) Distance to policy based mutation applied

3. UNTIL termination
```

Figure 4.16. Pseudo-code for multinational GA

4.11. Niching: Niching for Dynamic Landscapes

This technique [12] (proposed by Cedeno and Vemuri) uses crowding selection and a replacement schema called "*Worst Among Most Similar (WAMS)*". Crowding selection chooses the second parent according to the similarity to the first parent. First, an individual from the population is selected; then a small group of individuals are selected from the population randomly, where the size of the group is defined by "crowding selection group size". The individual which is the most similar to the first mate with respect to Euclidean distance is selected as the second mate. After recombination and mutation, the generated offspring replaces one of the old individuals according to WAMS replacement schema. This schema uses two parameters: the crowding factor and the crowding group size. Crowding factor defines the number of groups that will be created for selection and crowding group size is the size of these groups. These groups are selected at random (with replacement) from the

whole population. For each of these groups, the member which is most similar to the offspring is selected. And the one with the least fitness is replaced by the offspring.

```
BEGIN
   INITIALIZE population with random candidate solutions;
   EVALUATE each candidate;
   REPEAT UNTIL (TERMINATION COND. is satisfied) DO;
         FOR each individual from 1 to n
         1.  Use CROWDING SELECTION to find mate for
             individual
         2.  MATE nad MUTATE offspring
         3.  INSERT offspring into population using
             using WAMS replacement
    END REPEAT;
END;
```

Figure 4.17. Pseudo-code for niching algortithm

4.12. Proposed Hybrid Techniques

In this section, we present two hybrid techniques which are based on leading methods in the literature.

4.12.1. SOS+LS: Self Organizing Scouts + LS with Crossover Hill-Climbing

In this method, we propose a hybrid solution with the aim of improving the performance of SOS method [1, 17] with local search based on crossover hill-climbing [42]. The idea of SOS algorithm is to divide main population into smaller populations to explore several peaks in a multimodal landscape. Even though scout populations are responsible for exploiting smaller areas, the reduced size of a scout population, decreases ga's ability to explore and exploit these areas. That is because the scouts are isolated and recombination is only allowed for the members of same scout. Moreover the number of individuals that a scout will own in the next generation is determined by the linear sum of the fitness and dynamism of the scout. If a scout population does not explore its search space effectively, it can not have good relative fitness and dynamism values. This directly affects the division of population between scouts

for the next generation. As a result of this degradation, a dynamic and promising area may seem less promising and loose its members in succesive generations. To solve this problem, a way to increase the explorative power of a small population is needed. In this approach we support the scout population with a local search algorithm to increase its exploitation capability.

Cross-over hill-climbing(p_1, p_2, n_{off}, n_t)

1. $p'_1 = p_1$ $p'_2 = p_2$
2. REPEAT n_t times
   Generate n_{off} offspring o_1.....o_{noff}
   performing crossover on p'_1 and p'_2
   Evaluate o_1.....o_{noff}
   Find the offspring with the best fitness value o_{best}
   Replace the worst of p'_1 and p'_2 only if offspring is better
   RETURN p'_1 and p'_2

Figure 4.18. Pseudo-code for crossover hill-climbing [42]

For each scout population the mates for recombination are selected with rank based selection. Instead of using simulated binary crossover as in SOS method, this algorithm uses crossover hill-climbing operator [42] to generate offsprings. The cross-over hill-climbing operator (Figure 4.18) is a genetic operator which uses hill-climbing as a move accepting criterion and uses crossover as a move operator. For each pair crossover is applied for five times to create ten offsprings. The best of these offsprings replaces the worst parent. Then recombination is applied again to these new parents to create new offsprings. Again the worst parent is replaced and the resulting parents are returned as generated offsprings.

4.12.2. MN+RI : Multinational GA + Random Immigrants

Multinational GA is a multimodal optimization algorithm whose idea is to locate several peaks and valleys in the landscape. For this purpose it uses a procedure called hill-valley detection. This procedure takes two points and decides if these two are on the same peak or there is a valley in between them and they are on seperate peaks. As a result the performance of this algorithm strongly depends on the initial population. If the initial population only covers a small part of the whole search space, the algorithm only finds the peaks and valleys in this area and can not direct the search to the other areas of the search space. Even if the initial population covers the whole search space, it can converge around a few promising peaks and can not find new peaks if they appear as a result of dynamism.

We consider the combination of Multinational GA and random immigrants both of which provide promising results. By inserting random immigrants we aim to increase the area that this algorithm searches and keep the diversity high. By this way the performance of multinational GA is improved. At each generation, if the total number of nations is lower than a predefined number, a fraction of the population is replaced by randomly generated individuals. As can be seen from experimental results section, the combination of these algorithms works better than each of them.

5. EXPERIMENTAL STUDY

5.1. Moving Peaks Benchmark

Moving peaks benchmark is considered for performance evaluation of the algorithms, which is a multimodal landscape consisting of peaks with changing heights, widths and locations in an n dimensional search space. In this work, we used the problem generator suggested by Branke [1, 4] where the fitness landscape is defined as follows :

$$F(\vec{x},t) = \max(B(\vec{x}), \max_{i=1\ldots m} P(\vec{x}, h_i(t), w_i(t), p_i(t))) \tag{5.1}$$

In this equation, B(x) is the base function which is not dependent on time. The problem has m peaks and each of these peaks have its own time dependent height (h), width (w) and location (p) parameters.

There are several parameters that define the characteristics of this dynamic problem. The height severity, width severity and shift length are the parameters defining the severity of changes in height, width and location, respectively. Δe is the parameter defining the number of generations before the environment changes; and it is used for controlling frequency of changes. λ is the parameter defining the correlation between successive moves of a peak. For $\lambda = 0$, peaks move in random ways without any correlation. For $\lambda = 1$, peaks move in linear directions.

At every change of the environment, height, width and position of a single peak is calculated with the following equations [1]:

$$\sigma \in N(0,1)$$

$$h_i(t) = h_i(t-1) + height_severity \cdot \sigma \tag{5.2}$$

$$w_i(t) = w_i(t-1) + width_severity \cdot \sigma \tag{5.3}$$

$$\vec{p}_i(t) = \vec{p}_i(t-1) + \vec{v}_i(t) \tag{5.4}$$

$\vec{v}_i$ is the shift vector applied to peak i. It is the linear combination of the previous shift vector $\vec{v}_i(t-1)$ and a random vector $\vec{r}$. Both the random vector and the shift vector are normalized to s. The shift vector at time t is calculated with the following equation:

$$\vec{v}_i(t) = \frac{s}{\left|\vec{r} + \vec{v}_i(t-1)\right|}\left((1-\lambda)\vec{r} + \lambda\vec{v}_i(t-1)\right) \tag{5.5}$$

In this thesis, our selected problem consists of 10 peaks in a 5 dimensional search space. The lower bound and the upper bound for each dimension is 0 and 100 respectively. Changes are applied at every 50 generations. The successive movements are not correlated. λ is taken to be zero and the peaks make random movements. The shift length is 1, height severity is 7 and width severity is 1.0. These values are considered in all experiments, unless otherwise stated.

Table 5.1. Default settings for the moving peaks benchmark [1]

Parameter	Value
Number of Peaks p	10
Change Frequency	Every 50 generations
Height Severity	7.0
Width Severity	1.0
Peak Shape	Cone
Basis function	no
Correlation Lamda	0
Shift Length s	1
Number of Dimensions	5
Min and Max Allele value	[0,100]
Min and Max peak height	[30,70]

In dynamic environments changes can be deterministic or sthocastic. In the deterministic case changes are known in advance while in the sthocastic case changes are generated with random intervals. For the former type of changes there is not a need for a change detection mechanism but for sthocastically changing environments change detection is required. Genetic algorithms are very effective in detecting changes in the environment. since these algorithms iteratively improve a population of candidate solutions, a change is detected in the environment whenever the average fitness of the population reduces more than a predefined amount. Moving peaks problem uses a parameter to determine the frequency of change and it is a deterministic problem. As a result, all of the algorithms implemented in this work do not need to employ a change detection mechanism.

5.2. Comparison Metrics

Performance evaluation of algorithms is based on three main metrics: offline error, number of fitness function evaluations and running time. Offline error measure is based on offline performance proposed by De Jong [34]. Since moving peaks benchmark is considered in this study and its optimum value is known at any point in time, offline error is used instead of offline performance in our experiments.

Offline error is the running average of the difference between the best individual encountered so far and the optimum at any time. The equation for offline error is as follows [1]:

$$E(T) = \frac{1}{T}\sum_{t=1}^{T}\left(opt_t - e_t'\right) \tag{5.6}$$

$$e_t' = \max(e_\tau, e_{\tau+1}, ..., e_t) \tag{5.7}$$

In this equation τ is the time of last change. While calculating offline error, only the evaluations made after the last change are considered.

Our second metric is the running time, which is the average time that passes for a single execution of each algorithm. The third metric is the number of fitness function evaluations. This metric is also calculated as the average of twenty executions. Both of these metrics are used to measure the computational effort required for different algorithms.

Some of the algorithms included in this work require different amount of computational effort. In order to take these computational cost differences into account, we report running times of algorithms. Number of fitness function evaluations is also reported as a measure of computational effort. But using only this measure can be misleading, if an evolutionary algorithm uses hidden labour [5]. Two algorithms using equal amount of fitness evaluations can have much different computational costs as a result of hidden labour (hill valley detection procedure in multinational ga, forking ga techniques in SOS, WAMS schema in niching...etc).

The disadvantage of using running time is that the time required to execute an algorithm can change on different machines; but for this work all the tests are made on the same machine and the ranking of these algorithms does not change on different machines and the number of fitness function evaluations does not change with the computational environment.

5.3. Parameter Settings

In this subsection, we present the values of algorithm-specific and experimental parameters used in our performance study. In our experiments, a population of 100 individuals is considered for each algorithm. The results in the tables in this study are the averages of 20 runs, where each run is performed with a different random seed to create different instances of the same problem. Real-valued representation and generational replacement with elitism of 1 is used. Unless otherwise stated, two-point simulated binary cross-over and gaussian random mutation is applied. The crossover probability is taken 0.6 and the mutation rate is set to 0.2.

For memory supported algorithms, the memory size has been set to 10. Random immigrants (RI) and random immigrants with memory (RIm) algorithms replace 25 percent of

the individuals at each generation. These parameter settings are the ones given in [1]. The values of parameters given in the SOS algorithm are taken from their paper [17], which are summarized in Table 5.2. The same settings are also considered in our modified version of the SOS algorithm.

Multinational GA uses a parameter defining the ratio of local and global crossover. For the experiments in this part, this ratio is set to 0.7. This algorithm uses tournament selection with a tournament size of 5. The same parametric settings are also considered in our modified version of the multinational GA called MN+RI.

Table 5.2. Default settings for SOS algorithm [1]

Parameter	**Value**
min. radius of scout population search space r_{min}	6
max. radius of scout population search space r_{max}	12.0
relative weight of dynamism α	0.5
min. number of individuals in a scout p_{min}	4
min number of individuals in base population	8
total number of individuals	100
number of base populations	1
min. relative fitness for a new scout	0.6
min relative fitness for an existing scout	0
factor for shrinking scout population search space	0.99

The extra parameters used by niching algorithm are crowding selection size, crowding factor and crowding group size. In the experiments, the crowding selection group size is set to 10 for selection. Crowding group size and crowding factor parameters are set to 10 and 3, respectively. These parameters are not the best settings for this algorithm. But increasing the values of parameters increases the computation time severely. Therefore, these parameters are selected to optimize the trade off between performance and computation time.

Crossover hill-climbing operator also uses new parameters such as n_{off} and n_t. n_{off} is the number of offsprings generated for each pair of parents and n_t is the number of repetitions

used in the operator for each pair of parents. The values of n_{off} and n_t are 10 and 2 respectively.

5.4. Experimental Results

Selected parameters of the moving peaks benchmark are varied to provide different test cases or experiments, which are given in this section. It should be noted that results in the tables are grouped according to the classification of the algorithms. In addition to the algorithms mentioned in this thesis, tables also include results of the SEA algorithm, which is the simple evolutionary algorithm.

5.4.1. The Influence of Severity Changes

Shift length is the norm of the vectors that will be added to peak location. If the shift length is taken to be 1, this means that each peak moves by one unit at each change. This experiment presents the effect of "severity of change" on the performance of different algorithms. The shift lengths considered in our test cases are 1, 2 and 3. A change with a shift length of 3 is much more severe than a change with shift length 1.

Table 5.3 shows the offline errors of algorithms for different shift lengths. As shift length increases, the performance of all algorithms degrades and offline errors increase. That is a normal consequence of increasing the dynamism of the environment. As shift length increases, the degree of similarity between the environments before and after a change decreases and the exploitable information level decreases.

All approaches which use a simple memory performs worse than their standard approaches without memory and the difference increases as shift length increases. That is because while peaks move, the memory becomes more useless. Since these approaches reserve some part of the population to memory and can not use it effectively, they have worse performance than others. Mem/Search outperforms all other algorithms except the

multinational ones and the Niching algorithm. But as the environment changes, memory/search is affected more than the others since memory becomes less useful. When Memory/2Search is considered, the results show that one search population is enough; therefore, the extra search population does not make any improvement. Niching algorithm performs better than all memory based approaches since it uses crowding selection and WAMS replacement to maintain high diversity. It is better than Memory/Search while exploring search space and not affected severely by the increase of shift length.

Table 5.3. The offline error of algorithms for different shift lengths

Algorithm	**Shift Length**		
	1.0	**2.0**	**3.0**
SEA	17,008	18,154	18,319
RI	11,070	11,890	12,058
P3	11,945	12,744	14,056
Niching	6,334	7,179	7,308
SEAm	17,236	18,200	18,714
RIm	13,193	13,626	14,008
P3m	12,398	13,196	14,911
Mem/Search	7,383	8,785	9,789
Mem/2Search	7,492	9,075	10,950
SOS	4,340	5,042	5,908
Multi-National	5,973	5,894	5,949
SOS+LS	**3,413**	**4,091**	**4,488**
MN+RI	4,337	5,052	5,242

Multi-population approaches are much more efficient than others and they are less affected from the change in severity since they are capable of detecting and tracking multiple optima. The least affected algorithm is the multinational GA. That is the result of using hill-valley detection algorithm. This clustering algorithm is able to predict locations of peaks and valleys even if they move severely. The drawback of this algorithm is that, its performance strongly depends on the initial solutions. The hybrid algorithm of multinational GA and random immigrants perform better than multinational GA since random immigrants makes this algorithm explore more unexplored regions and increase diversity. But it is more affected by the change in severity.

SOS+LS gives the best results since it increases the search capacity of scout populations with a local search technique.

Table 5.4. The average execution time and evaluation counts of algorithms for different shift lengths (s)

Algorithms	Execution Time			Number of Evaluations		
	s =1.0	s = 2.0	s = 3.0	s = 1.0	s = 2.0	s = 3.0
SEA	1,850	2,000	1,950	102,00	102,00	102,00
RI	2,200	2,350	2,200	127,00	127,00	127,00
P3	1,750	1,800	1,700	104,00	104,00	104,00
Niching	11,05	11,30	10,70	202,00	202,00	202,00
SEAm	1,650	1,700	1,650	91,00	91,00	91,000
RIm	2,000	2,150	2,050	116,00	116,00	116,00
P3m	1,550	1,600	1,550	92,00	92,000	92,000
Mem/Search	1,600	1,650	1,600	94,00	94,000	94,000
Mem/2Search	1,550	1,600	1,550	92,000	92,000	92,000
SOS	2,450	2,600	2,500	114,85	114,85	114,45
Multi-national	8,100	8,600	8,050	617,30	617,70	616,00
SOS+LS	8,150	8,200	7,800	573,45	568,60	561,65
MN+RI	8,500	8,750	8,250	626,70	626,20	625,95

Table 5.4 shows the average running time and fitness function evaluation numbers of different approaches. Memory supported approaches work faster than the versions without memory. That is because memory is separated from the main population. The size of the main population decreases as a result running time decreases. Multi-population algorithms divide the population into smaller groups but they make more computation to decide dividing or merging population. Niching algorithm has highest time overhead.

Average number of fitness function evaluations also reflects the results reported as running time except for the Niching algorithm. Even though Niching makes less number of evaluations than SOS+LS and MN+RI, it has the highest time overhead since it uses crowding selection and WAMS replacement. As it is observed in the table all algorithms except the multi-population algorithms makes fixed number of fitness evaluations at each run and this numbers does not change with shift length. The changes in multi-population approaches are

due to size constraints of the sub-populations. There are limits on the minimum number of members that a sub-population must have for these algorithms. Whenever a sub-population has less number of members than the lower bound, some individuals which fit this sub-population are created randomly and added to this sub-population.

5.4.2. The Influence of Change Frequencies

The second experiment aims to see the effect of frequency of change for different algorithms. For this purpose, algorithms are run in environments which change at every 10, 25 or 50 generations. As the frequency of changes increases, each algorithm has a shorter time to come up with a solution and offline error increases for all algorithms (Table 5.5).

Table 5.5. The offline error of algorithms for different change frequencies

Algorithm	Change Frequency		
	10	**25**	**50**
SEA	19,565	18,148	17,008
RI	15,568	13,027	11,070
P3	14,840	12,256	11,945
Niching	8,551	7,417	6,334
SEAm	18,646	18,175	17,236
RIm	15,043	13,716	13,193
P3m	14,430	12,838	12,398
Mem/Search	14,991	9,561	7,383
Mem/2Search	16,027	10,776	7,492
SOS	6,117	5,137	4,340
Multi-National	7,868	5,664	5,973
SOS+LS	**4,897**	**3,983**	**3,413**
MN+RI	5,329	4,628	4,337

The most heavily affected algorithms are the memory based ones. That is because the memorized individuals will not be good solutions after a short time and the memory becomes less useful as the environment changes more quickly.

Diversity maintaining algorithms are less affected from the quick changes since they have high diversity and may explore for the new optimum more effectively. However, they also degrade in performance since they have less time to find an optimum solution before it changes.

The least affected algorithms are the multi-population approaches. They are also degraded but not as much as other algorithms. Multi-population approaches divide the population to track the changes in all promising regions. Even if the environment changes more quickly, those algorithms still can detect and track changes in those promising areas. However they have also less time to climb the hills. As a result they can not find as good solutions as they find when changes are less frequent. As a consequence, their performance also degrades with the increase in frequency of change.

Table 5.6. The average execution time and evaluation counts of algorithms for different change frequencies (f)

Algorithm	Execution Time			Number of Evaluations		
	f = 10	f = 25	f = 50	f = 10	f = 25	f = 50
SEA	2,00	1,850	1,850	110,00	104,0	102,00
RI	2,400	2,200	2,200	135,00	129,0	127,00
P3	1,850	1,700	1,750	112,00	106,0	104,00
Niching	11,00	10,75	11,05	210,00	204,0	202,00
SEAm	1,800	1,650	1,650	99,000	93,00	91,00
RIm	2,150	2,000	2,000	124,00	118,0	116,00
P3m	1,650	1,550	1,550	99,00	93,00	92,00
Mem/Search	1,800	1,600	1,600	102,00	96,00	94,00
Mem/2Search	1,650	1,550	1,550	99,00	93,00	92,000
SOS	2,600	2,400	2,450	121,05	116,0	114,85
Multi-national	8,350	7,900	8,100	626,70	622,7	617,30
SOS+LS	9,000	8,250	8,150	629,25	606,7	573,45
MN+RI	8,650	8,200	8,500	637,00	629,9	626,70

The best performing algorithm is the SOS+LS algorithm because of employing a local search algorithm. The hybrid algorithm of multinational GA and random immigrants (MN+RI) performs better than the multinational GA; and the MN+RI algorithm is the least

affected algorithm with respect to frequency of change, which can be due to insertion of immigrants. It takes an explicit action to increase diversity during optimization. These immigrants increase the adaptability of the algorithm as the environment changes.

The execution times of algorithms given in Table 5.6 show that the computational effort slightly increases as the changes in the environment become more frequent. That increase is due to the fitness update operations performed at each change. That effect is clearly seen from both measures. As the change interval decreases, algorithms deal with more changes and at each change they update the fitness by re-evaluating the main population. This extra cost is seen in both running time and number of fitness evaluation results for all of the algorithms.

5.4.3. The Influence of Correlation

This experiment shows the effect of changing correlation between successive changes for different algorithms. For the above cases we always set λ to 0 and deal with peaks moving in random direction. In such a situation, peaks move around their initial locations. If λ is set to 1, each peak moves along a line. Different test cases are generated by changing the value of λ to see the effect of change predictability for different algorithms. The different λ values used in these test cases are 0, 0.5 and 1.

Table 5.7 shows offline errors of different approaches for different λ values. The algorithms that maintain diversity throughout the run and multi-population approaches does not feel the effect of changes in correlation much. Only small changes occur when compared to memory based approaches, which is due to the memorization process. If peaks move around their initial conditions, the memorized solutions become more useful. Otherwise peaks move along and the old solutions in memory become useless and degrade the performance of algorithms.

Table 5.7. The offline error of algorithms for different correlations

Algorithm	Correlation		
	0	**0.5**	**1**
SEA	17,008	17,240	17,412
RI	11,070	11,507	11,595
P3	11,945	12,825	13,291
Niching	6,334	6,662	6,812
SEAm	17,236	18,157	18,588
RIm	13,193	13,663	13,963
P3m	12,398	13,423	13,557
Mem/Search	7,383	8,590	9,888
Mem/2Search	7,492	8,970	10,137
SOS	4,340	4,540	4,563
Multi-National	5,973	5,772	5,661
SOS+LS	**3,413**	**3,625**	**3,535**
MN+RI	4,337	4,280	4,377

Table 5.8. The average execution time and evaluation counts of algorithms for different correlations (λ)

Algorithm	Execution Time			Number of Evaluations		
	λ = 0	**λ = 0.5**	**λ = 1**	**λ = 0**	**λ = 0.5**	**λ = 1**
SEA	1,850	1,800	1,900	102,00	102,00	102,00
RI	2,200	2,150	2,300	127,00	127,00	127,00
P3	1,750	1,700	1,800	104,00	104,00	104,00
Niching	11,05	10,65	10,850	202,00	202,00	202,00
SEAm	1,650	1,650	1,700	91,00	91,00	91,00
RIm	2,000	1,950	2,050	116,00	116,00	116,00
P3m	1,550	1,500	1,550	92,00	92,00	92,00
Mem/Search	1,600	1,600	1,650	94,00	94,00	94,00
Mem/2Search	1,550	1,500	1,600	92,000	92,00	92,00
SOS	2,450	2,400	2,500	114,85	114,75	115,25
Multi-national	8,100	7,850	8,250	617,30	618,40	617,75
SOS+LS	8,150	7,750	8,000	573,45	563,95	585,60
MN+RI	8,500	8,200	8,600	626,70	626,80	626,35

All multi-population approaches are nearly not affected and all diversity maintaining algorithms are slightly affected. The SOS+LS algorithm outperforms other methods considered in the experiments.

Table 5.8 shows the average running time of different algorithms for different λ values. The results are similar for all λ values, since algorithms do the same amount of computation as before even if the environment changes linearly or randomly. The results are obvious from both measures.

5.4.4. The Influence of Changing the Number of Peaks in the Search Space

Changing the number of peaks in the landscape or changing the dimensionality of search space generates test cases with varying complexities. In this work, the dimensionality of the search space is changed to create test cases by considering 10, 50 and 200 peaks.

Table 5.9. The offline error of algorithms for different number of peaks

Algorithm	**Number of Peaks**		
	10	**50**	**200**
SEA	17,008	19,112	16,702
RI	11,070	9,802	7,592
P3	11,945	13,244	9,009
Niching	6,334	7,381	5,791
SEAm	17,236	17,968	14,409
RIm	13,193	10,923	8,930
P3m	12,398	13,025	9,821
Mem/Search	7,383	6,147	4,797
Mem/2Search	7,492	5,736	4,376
SOS	4,340	3,217	2,494
Multi-national	5,973	3,558	2,667
SOS+LS	**3,413**	**2,626**	**1,883**
MN+RI	4,337	3,242	2,656

When number of peaks is increased from 10 to 50 the complexity of the environment increases. The expected result is degradation in the performance of all algorithms. Table 5.9

shows the offline error of algorithms for different peak numbers. When all algorithms other than mem/search, mem/2search and the multi-population approaches are considered, performance degrades when number of peaks is set to 50. That is because the environment becomes more complex; and any of these peaks may be increased in height and it may become the new optimum. Therefore, the location of the optimum may change a great deal. These algorithms have difficulty in jumping from one peak to another.

On the other hand performance of mem/search and mem/2search and multi-population approaches increases at the 50 peak case. That is because the peaks are close to each other and these algorithms easily jump to new optimums. Another reason is that the smaller peaks become hidden under the higher peaks and the average fitness of the landscape increases. This effect is more apparent at 200 peaks case and the performance of all algorithms becomes better. As in the previous experiments, our SOS+LS extension outperforms the other methods for all tested dimensionalities of the search space.

Table 5.10. The average execution time and evaluation counts of algorithms for different number of peaks (n)

Algorithm	**Execution Time**			**Number of Evaluations**		
	n = 10	**n = 50**	**n = 200**	**n = 10**	**n = 50**	**n = 200**
SEA	1,850	5,200	18,350	102,00	102,00	102,00
RI	2,200	6,400	22,600	127,00	127,00	127,00
P3	1,750	5,100	18,400	104,00	104,00	104,00
Niching	11,05	17,30	42,000	202,00	202,00	202.00
SEAm	1,650	4,650	16,549	91,00	91,00	91,00
RIm	2,000	5,850	20,850	116,00	116,00	116,00
P3m	1,550	4,550	16,400	92,00	92,00	92,00
Mem/Search	1,600	4,650	16,600	94,00	94,00	94,00
Mem/2Search	1,550	4,550	16,250	92,000	92,00	92,00
SOS	2,450	5,950	20,000	114,85	108,90	108,00
Multi-national	8,100	29,25	113,00	617,30	648,65	675,00
SOS+LS	8,150	41,65	173,00	573,45	909,90	1032,0
MN+RI	8,500	29,30	115,55	626,70	650,60	676,95

Table 5.10 shows the average execution times of algorithms for different number of peaks. As the number of peaks increases, the execution time of algorithms also increases. This

increase is due to the increase of the time for a single evaluation. This effect is most apparent for multinational GA, MN+RI and SOS+LS, since these algorithms include more number of evaluations than the other algorithms. For SOS, SOS+LS algorithms the number of fitness function evaluations required changes with the number of peaks in the environment. Since there are more peaks in the environment, these algorithms create more scouts and at each creation they reinitialize and re-evaluate the base population. For SOS+LS the difference is more serious, this is a consequence of using crossover hill-climbing operator. As the number of scout populations increases, this operator is applied more often and increases the number of evaluations for each run.

5.5. Similarity Based Comparison

One of the important search areas in dynamic environments is measuring performance. The most widely used performance measure is the offline error which is proposed by De Jong [34]. Offline error is the running average of the difference between the best individual encountered so far and the optimum at any time. The quality of solution or the accuracy of the algorithm is the primary concern for this metric. But in dynamic environments, recovering after a change is also important for an algorithm. In order to take this ability into account, another performance measure is proposed in this work which is based on signal similarity. This measure is a weighted sum of two similarity measures: Euclidean distance and Cross correlation.

The true optimal values at each generation forms a signal and best values reached by the algorithm at each generation forms another signal. At each change the best-so-far value is reset since old best values become meaningless after a change. For a perfect algorithm which has an error value of zero, the two signals will be the same and dissimilarity of signals (error) will be zero. In order to measure the dissimilarity between these two signals, we use two signal similarity measures: Euclidean distance and cross correlation.

Euclidean distance between two signals (x and y) is calculated with the following equation:

$$Euclid_dist = \sqrt[2]{\sum_{i=1}^{\max gens} (x_i - y_i)^2} \tag{5.8}$$

Cross correlation is also a measure for defining how two signals are similar and correlated. This measure is calculated for two signals x and y with the following equation:

$$Cross_corr = \frac{\sum_i \left((x_i - m_x) \cdot (y_i - m_y)\right)}{\sqrt{\sum_i (x_i - m_x)^2} \cdot \sqrt{\sum_i (y_i - m_y)^2}} \tag{5.9}$$

In this equation, m_x is the estimated mean of signal x and m_y is the estimated mean of signal y. The means are estimated with the following equations:

$$m_x = \frac{1}{N} \sum_i x_i \tag{5.10}$$

$$m_y = \frac{1}{N} \sum_i y_i \tag{5.11}$$

Cross correlation returns values between -1 and 1. The closer the cross correlation to 1, the more similar the signals are. Since our purpose is to measure dissimilarity we use 1 minus cross correlation which changes between 0 and 2 and normalize it to interval [0,1].

Euclidean distance and cross correlation measure the similarity between the real optimal values and the found optimal values by the algorithm. A small Euclidean distance shows that the algorithm is able to find close results to the true optimal solution but it does not matter on which generation this solution is obtained. On the other hand a high correlation value indicates that the shape of the signal formed by true optimal values and the signal formed by the algorithm bests are similar to each other even if they are apart in distance. By the way

Euclidean distance shows the effectiveness of the algorithm and cross correlation shows the ability to recover after a change.

For each algorithm an average cross-correlation and Euclidean distance measure is computed as the average of twenty executions. The equation for the new measure is as follows:

$$Dissimilarity_i = weight_{euclid} * scaled_euclid_i + weight_{cross} * scaled_cross_i \quad (5.12)$$

$weight_{euclid}$ and $weight_{cross}$ are the coefficients of Euclidean distance and the cross correlation respectively. These coefficients are used to define effectiveness of these measures in the total. The sum of the coefficients is 1. $scaled_euclid_i$ and $scaled_cross_i$ are the scaled values of Euclidean distance and cross correlation. After scaling, both of these measures are mapped onto interval [0, 1].

The scaled values are calculated with the following equations:

$$scaled_euclid_i = \frac{euclid_i - \min_{j=1..numof\exp s}\left(euclid_j\right)}{\max_{j=1..numof\exp s}\left(euclid_j\right) - \min_{j=1..numof\exp s}\left(euclid_j\right)} \quad (5.13)$$

$$scaled_cross_i = \frac{cross_i - \min_{j=1..numof\exp s}\left(cross_j\right)}{\max_{j=1..numof\exp s}\left(cross_j\right) - \min_{j=1..numof\exp s}\left(cross_j\right)} \quad (5.14)$$

This measure represents the performance of each algorithm as a value between 0 and 1. The algorithm which has the least value is the best one.

5.5.1. The Influence of Severity Changes

In this part we perform the same experiment reported in section 4.4.1 which presents the effect of "severity of change" on the performance of different algorithms; but we report the results with the new similarity based metric.

Table 5.11 shows the Euclidean distance and cross correlation values for different algorithms and Table 5.12 shows the scaled values. As shift length increases, performance of all algorithms degrades. This result is obvious from the increase in the Euclidean distances and the decrease in the cross correlations. This degradation is a normal consequence of the increased dynamism in the environment. As shift length increases, the degree of exploitable information before and after a change decreases and the algorithms have more difficulty in adapting to the new environment.

Table 5.11. Average Euclidean distance and cross correlation of algorithms for different shift lengths (s)

Algorithm	Euclidean Distance			Cross Correlation		
	1.0	2.0	3.0	1.0	2.0	3.0
SEA	1436,336	1510,254	1522,215	0,171	0,186	0,154
RI	1010,012	1064,978	1084,539	0,306	0,281	0,283
P3	1048,156	1096,103	1197,976	0,280	0,262	0,251
Niching	645,145	700,972	717,831	0,406	0,374	0,388
SEAm	1452,939	1506,051	1549,402	0,151	0,156	0,119
RIm	1131,983	1155,945	1187,755	0,294	0,316	0,312
P3m	1089,185	1142,326	1270,390	0,259	0,240	0,198
Mem/Search	696,971	815,070	900,855	0,450	0,398	0,358
Mem/2Search	713,178	840,556	992,715	0,432	0,377	0,326
SOS	476,319	524,059	598,625	0,602	0,585	0,528
Multi-national	625,596	613,592	627,370	0,455	0,455	0,487
SOS+LS	411,195	475,261	512,517	0,623	0,582	0,535
MN+RI	496,487	548,655	563,926	0,537	0,507	0,494

Table 5.12. Scaled Euclidean distance and cross correlation of algorithms for different shift lengths (s)

Algorithm	Scaled Euclidean Distance			Scaled Cross Correlation		
	1.0	2.0	3.0	1.0	2.0	3.0
SEA	0,901	0,966	0,976	0,896	0,867	0,930
RI	0,526	0,574	0,592	0,630	0,679	0,674
P3	0,560	0,602	0,691	0,681	0,716	0,738
Niching	0,206	0,255	0,269	0,431	0,494	0,466
SEAm	0,915	0,962	1,000	0,935	0,926	1,000
RIm	0,633	0,654	0,682	0,653	0,610	0,617
P3m	0,596	0,642	0,755	0,722	0,760	0,843
Mem/Search	0,251	0,355	0,430	0,343	0,445	0,525
Mem/2Search	0,265	0,377	0,511	0,378	0,489	0,589
SOS	0,057	0,099	0,165	0,042	0,074	0,188
Multi-national	0,188	0,178	0,190	0,332	0,333	0,269
SOS+LS	0,000	0,056	0,089	0,000	0,082	0,174
MN+RI	0,075	0,121	0,134	0,171	0,230	0,256

Table 5.13 shows the results obtained by the new metric. While commenting on these results, we will interpret them in two ways: rows and columns. The same column shows the performance of different algorithms under the same dynamic environmental conditions. On the same column the algorithm which has the lowest value is the best performer while the algorithm with the highest value is the worst performer. The same row shows the performance of the same algorithm under different environmental conditions. By inspecting rows we see how the algorithms are affected by the change. If this value increases, this means the algorithm performs worse. If this value decreases, this means the algorithm performs better than the previous case. The amount of increase or decrease in the same row defines how much the algorithm is affected in comparison to other algorithms. The algorithm with the highest decrease or the lowest increase is the algorithm which is least affected by the change.

Using these guidelines we can comment that all approaches which use a simple memory performs worse than their standard approaches without memory and the difference increases as shift length increases. That is because while peaks move severely, the memory becomes more useless. The idea of memory based approaches is to store old good solutions and use them later. Since severe changes occur in the environment, old good solutions may not be as

good for the new environment or they may also be bad solutions for the new environment. Since these approaches reserve some part of the population to memory and can not use it effectively, they have worse performance than others. Mem/Search outperforms all other algorithms except the multinational ones and the Niching algorithm. But as the environment changes more severely, memory/search is affected more than the others since memory becomes less useful. When Memory/2Search is considered, the results show that one search population is enough; therefore, the extra search population does not make any improvement. Niching algorithm performs better than all memory based approaches since it uses crowding selection and WAMS replacement to maintain high diversity. It is better than Memory/Search while exploring search space and not affected severely by the increase of shift length.

Table 5.13. Similarity based error values of algorithms for different shift lengths

Algorithm	**Shift Length [0.7, 0.3]**		
	1.0	**2.0**	**3.0**
SEA	0,899	0,936	0,962
RI	0,557	0,606	0,616
P3	0,596	0,636	0,705
Niching	0,273	0,326	0,328
SEAm	0,921	0,951	1,000
RIm	0,639	0,641	0,663
P3m	0,633	0,678	0,781
Mem/Search	0,279	0,382	0,459
Mem/2Search	0,299	0,411	0,534
SOS	0,053	0,092	0,172
Multi-National	0,232	0,224	0,214
SOS+LS	**0,000**	**0,064**	**0,115**
MN+RI	0,104	0,154	0,171

When we consider the results for multi-population approaches in table 5.13, it is seen that similarity based error values for multi-population approaches increase slightly or decrease. We can conclude that multi-population approaches are much more efficient than others and they are less affected from the change in severity since they are capable of detecting and tracking multiple optima. The least affected algorithm is the multinational GA. That is the result of using hill-valley detection algorithm. This clustering algorithm is able to

predict locations of peaks and valleys even if they move severely. The drawback of this algorithm is that, its performance strongly depends on the initial solutions. The hybrid algorithm of multinational GA and random immigrants perform better than multinational GA since random immigrants makes this algorithm explore more unexplored regions and increase diversity. But it is more affected by the changes in severity. The error value for multinational GA for the last case (shift length = 3) falls below the initial case (shift length = 1). This result indicates that it is the least affected algorithm. SOS+LS gives the best results since it increases the search capacity of scout populations with a local search technique.

5.5.2. The Influence of Change Frequencies

This part represents the results of the experiment described in section 4.4.2 which examines the effect of "change frequency" on the performance of different algorithms. The results are reported in terms of the new metric. Table 5.14 shows the Euclidean distance and cross correlation values. Table 5.16 shows the performances in terms of similarity based metric.

Table 5.14. Average Euclidean distance and cross correlation of algorithms for different change frequencies (f)

Algorithm	Euclidean Distance			Cross Correlation		
	10	25	50	10	25	50
SEA	1603,798	1508,005	1436,336	0,169	0,166	0,171
RI	1331,154	1155,092	1010,012	0,224	0,244	0,306
P3	1247,305	1070,954	1048,156	0,248	0,265	0,280
Niching	775,440	714,572	645,145	0,378	0,350	0,406
SEAm	1542,133	1512,155	1452,939	0,204	0,164	0,151
RIm	1268,630	1171,390	1131,983	0,268	0,289	0,294
P3m	1226,097	1118,968	1089,185	0,259	0,253	0,259
Mem/Search	1283,589	870,765	696,971	0,261	0,335	0,450
Mem/2Search	1357,832	966,762	713,178	0,241	0,307	0,432
SOS	591,573	532,598	476,319	0,523	0,549	0,602
Multi-national	754,805	586,216	625,596	0,412	0,475	0,455
SOS+LS	501,857	452,657	411,195	0,562	0,578	0,623
MN+RI	527,069	500,154	496,487	0,544	0,518	0,537

As change frequency increases, environment becomes more dynamic and algorithms deal with more changes. With other words frequency of change defines the time that an algorithm has before coming up with a solution. As this time decreases, performance of all the algorithms degrades. But some are affected more than the others. According to these results the most heavily affected algorithms are the memory-based ones. That is due to memorization. As environment changes faster and faster, the old good solutions become obsolete and useless.

Diversity maintaining algorithms are less affected from the quick changes since they have high diversity and may explore for the new optimum more effectively. However, they also degrade in performance since they have less time to find an optimum solution before it changes.

Table 5.15. Scaled Euclidean distance and cross correlation of algorithms for different change frequencies (f)

Algorithm	Scaled Euclidean Distance			Scaled Cross Correlation		
	10	25	50	10	25	50
SEA	1,000	0,920	0,860	0,962	0,969	0,958
RI	0,771	0,624	0,502	0,846	0,804	0,673
P3	0,701	0,553	0,534	0,794	0,759	0,728
Niching	0,305	0,254	0,196	0,520	0,579	0,461
SEAm	0,948	0,923	0,874	0,888	0,973	1,000
RIm	0,719	0,637	0,604	0,752	0,709	0,698
P3m	0,683	0,593	0,568	0,772	0,784	0,772
Mem/Search	0,732	0,385	0,240	0,767	0,610	0,367
Mem/2Search	0,794	0,466	0,253	0,811	0,670	0,404
SOS	0,151	0,102	0,055	0,213	0,157	0,045
Multi-national	0,288	0,147	0,180	0,447	0,314	0,355
SOS+LS	0,076	0,035	0,000	0,129	0,094	0,000
MN+RI	0,097	0,075	0,072	0,167	0,221	0,182

Multi-population algorithms are the least affected ones. These algorithms are able to divide the population and explore several promising areas in parallel. Even if the environment is more dynamic, these algorithms can still track changes effectively and adapt to the environment easily. Their performance is also degraded but not as much as memory-based and

diversity maintaining approaches. This degradation is due to having less time to come up with a solution.

Table 5.16. Similarity based error values of algorithms for different change frequencies

Algorithm	Change Frequency		
	10	**25**	**50**
SEA	0,989	0,934	0,889
RI	0,794	0,678	0,553
P3	0,729	0,615	0,592
Niching	0,370	0,352	0,276
SEAm	0,930	0,938	0,911
RIm	0,729	0,659	0,632
P3m	0,710	0,651	0,629
Mem/Search	0,742	0,453	0,278
Mem/2Search	0,799	0,527	0,298
SOS	0,170	0,118	0,052
Multi-National	0,336	0,197	0,232
SOS+LS	**0,092**	**0,053**	**0,000**
MN+RI	0,118	0,119	0,105

The least affected algorithm is the hybrid algorithm of multinational GA and random immigrants. As change frequency decreases from 50 to 10, dissimilarity of this algorithm increases only from 0,105 to 0.118 which means that this algorithm is more successful in adapting to more frequent changes. This may be due to insertion of immigrants. It takes an explicit action to increase diversity during optimization. These immigrants increase the adaptability of the algorithm as the environment changes.

The best performing algorithm is the SOS+LS algorithm because of employing a local search algorithm. The hybrid algorithm of multinational GA and random immigrants (MN+RI) performs better than the multinational GA.

5.5.3. The Influence of Correlation

This experiment shows the effect of changing correlation between successive changes for different algorithms such as in section 4.4.3. For the above cases we always set λ to 0 and deal with peaks moving in random direction. In such a situation, peaks move around their initial locations. If λ is set to 1, each peak moves along a line. Different test cases are generated by changing the value of λ to see the effect of change predictability for different algorithms. The different λ values used in these test cases are 0, 0.5 and 1 as in section 4.4.3.

Table 5.17. Average Euclidean distance and cross correlation of algorithms for different correlations (λ)

Algorithm	Euclidean Distance			Cross Correlation		
	0	0.5	1	0	0.5	1
SEA	1436,336	1445,057	1458,838	0,171	0,159	0,137
RI	1010,012	1045,286	1046,842	0,306	0,282	0,284
P3	1048,156	1112,860	1144,954	0,280	0,253	0,229
Niching	645,145	672,881	671,890	0,406	0,377	0,399
SEAm	1452,939	1501,990	1528,307	0,151	0,152	0,156
RIm	1131,983	1165,742	1184,887	0,294	0,312	0,318
P3m	1089,185	1165,676	1164,571	0,259	0,248	0,248
Mem/Search	696,971	815,939	926,622	0,450	0,376	0,338
Mem/2Search	713,178	847,275	942,117	0,432	0,391	0,325
SOS	476,319	493,094	493,750	0,602	0,585	0,600
Multi-national	625,596	618,693	609,869	0,455	0,451	0,484
SOS+LS	411,195	435,948	431,928	0,623	0,605	0,604
MN+RI	496,487	490,794	501,081	0,537	0,555	0,531

Table 5.17 shows Euclidean distance and cross correlation values and table 5.19 shows the similarity based error values of different approaches for different λ values. The algorithms that maintain diversity throughout the run and multi-population approaches does not feel the effect of changes in correlation much. Memory-based approaches are affected more when compared to others. That is a result of memorization process. For small λ values, peaks move around their initial conditions and the memorized solutions become more useful.

For larger λ values, peaks move along and the old solutions in memory become useless and degrade the performance of algorithms.

Table 5.18. Scaled Euclidean distance and cross correlation of algorithms for different correlations (λ)

Algorithm	Scaled Euclidean Distance			Scaled Cross Correlation		
	0	0.5	1	0	0.5	1
SEA	0,918	0,925	0,938	0,929	0,954	1,000
RI	0,536	0,568	0,569	0,653	0,702	0,698
P3	0,570	0,628	0,657	0,706	0,762	0,810
Niching	0,209	0,234	0,233	0,447	0,506	0,461
SEAm	0,933	0,976	1,000	0,970	0,969	0,960
RIm	0,645	0,675	0,693	0,677	0,639	0,628
P3m	0,607	0,675	0,674	0,748	0,772	0,772
Mem/Search	0,256	0,362	0,461	0,356	0,507	0,587
Mem/2Search	0,270	0,390	0,475	0,392	0,478	0,613
SOS	0,058	0,073	0,074	0,044	0,079	0,047
Multi-national	0,192	0,186	0,178	0,345	0,354	0,285
SOS+LS	0,000	0,022	0,019	0,000	0,037	0,038
MN+RI	0,076	0,071	0,080	0,177	0,140	0,189

Table 5.19. Similarity based error values of algorithms for different correlations

Algorithm	Correlation		
	0	0.5	1
SEA	0,921	0,934	0,956
RI	0,571	0,608	0,608
P3	0,611	0,668	0,703
Niching	0,281	0,316	0,302
SEAm	0,944	0,974	0,988
RIm	0,655	0,665	0,673
P3m	0,649	0,704	0,704
Mem/Search	0,286	0,406	0,499
Mem/2Search	0,307	0,417	0,517
SOS	0,054	0,075	0,066
Multi-National	0,238	0,236	0,210
SOS+LS	**0,000**	**0,027**	**0,025**
MN+RI	0,107	0,092	0,113

The results in Table 5.19 reveals that the dissimilarity values for multi-population approaches decreases or slightly increases and, for all diversity maintaining algorithms these values slightly increases which means that all multi-population approaches are nearly not affected and all diversity maintaining algorithms are slightly affected.

The SOS+LS algorithm outperforms other methods considered in the experiments. The hybrid algorithm of multinational GA and random immigrants (MN+RI) performs better than multinational GA.

5.5.4. The Influence of Changing the Number of Peaks in the Search Space

In this part we repeat the experiment in section 4.4.5 which aims to see the effect of changing the complexity of the landscape on the performance of different algorithms. In this work, the complexity of the search space is changed to create test cases by considering 10, 50 and 200 peaks. As the number of peaks in the landscape increases, the complexity of the environment increases as well. In this part the results are reported in terms of similarity based metric.

Table 5.20 shows the Euclidean distance and cross correlation values and Table 5.21 shows the errors in terms of the new metric for different algorithms. When number of peaks increases from 10 to 50, the environment becomes more complex. The expected result of this change is degradation in the performance of all algorithms. This is true for all diversity maintaining algorithms and memory based algorithms except mem/search and mem/2search. That is because the environment becomes more complex; and any of these peaks may be increased in height and it may become the new optimum. Therefore, the location of the optimum may change a great deal. These algorithms have difficulty in jumping from one peak to another.

Table 5.20. Average Euclidean distance and cross correlation of algorithms for different number of peaks (n)

Algorithm	Euclidean Distance			Cross Correlation		
	10	50	200	10	50	200
SEA	1436,336	1574,019	1367,724	0,171	0,034	0,078
RI	1010,012	840,976	647,241	0,306	0,229	0,298
P3	1048,156	1108,809	779,275	0,280	0,153	0,231
Niching	645,145	670,048	525,357	0,406	0,270	0,340
SEAm	1452,939	1472,150	1215,314	0,151	0,086	0,102
RIm	1131,983	893,711	728,065	0,294	0,261	0,303
P3m	1089,185	1093,276	831,437	0,259	0,157	0,218
Mem/Search	696,971	542,025	417,954	0,450	0,411	0,479
Mem/2Search	713,178	505,260	378,622	0,432	0,441	0,536
SOS	476,319	317,552	234,181	0,602	0,671	0,746
Multi-national	625,596	344,237	238,764	0,455	0,545	0,702
SOS+LS	411,195	277,799	192,558	0,623	0,645	0,762
MN+RI	496,487	312,254	239,250	0,537	0,585	0,699

Table 5.21. Average Euclidean distance and cross correlation of algorithms for different number of peaks (n)

Algorithm	Scaled Euclidean Distance			Scaled Cross Correlation		
	10	50	200	10	50	200
SEA	0,900	1,000	0,851	0,811	1,000	0,939
RI	0,592	0,469	0,329	0,627	0,731	0,638
P3	0,619	0,663	0,425	0,662	0,836	0,730
Niching	0,328	0,346	0,241	0,489	0,675	0,580
SEAm	0,912	0,926	0,740	0,838	0,928	0,906
RIm	0,680	0,508	0,388	0,643	0,688	0,631
P3m	0,649	0,652	0,462	0,690	0,831	0,747
Mem/Search	0,365	0,253	0,163	0,428	0,482	0,388
Mem/2Search	0,377	0,226	0,135	0,452	0,441	0,310
SOS	0,205	0,090	0,030	0,220	0,124	0,022
Multi-national	0,313	0,110	0,033	0,421	0,297	0,082
SOS+LS	0,158	0,062	0,000	0,191	0,160	0,000
MN+RI	0,220	0,087	0,034	0,309	0,243	0,087

The two memory based algorithms (mem/search and mem/2search) and all multi-population algorithms perform better when number of peaks in the landscape increases from 10 to 50. That is because the peaks are close to each other and these algorithms easily jump to new optimums. Another reason is the increase in the average fitness of the landscape. As number of peaks increases small peaks are covered by the higher peaks and the average fitness of the landscape increases. This affect is more apparent in the 200 peaks case; all algorithms perform better, even if they try to solve a problem which is more complex.

Table 5.22. Similarity based error values of algorithms for different number of peaks

Algorithm	**Number of Peaks [0.7, 0.3]**		
	10	**50**	**200**
SEA	0,874	1,000	0,877
RI	0,602	0,548	0,422
P3	0,632	0,715	0,516
Niching	0,376	0,444	0,342
SEAm	0,890	0,927	0,790
RIm	0,669	0,562	0,461
P3m	0,661	0,706	0,548
Mem/Search	0,384	0,322	0,231
Mem/2Search	0,400	0,291	0,187
SOS	0,210	0,101	0,028
Multi-National	0,346	0,166	0,048
SOS+LS	**0,168**	**0,091**	**0,000**
MN+RI	0,247	0,134	0,050

6. JOB SHOP SCHEDULING

Job shop scheduling problem is the most often used real world scheduling problem. In this work, job shop scheduling is selected as the second dynamic environment to evaluate performance of different algorithms. In this part both static and dynamic models of scheduling is being described.

6.1. The Static Model

Job shop scheduling is a multistage production process. The environment includes *m* machines ($M_1, M_2,...M_m$) which are dedicated to process *n* jobs ($J_1, J_2,...,J_n$).

In this model each job consists of a set of operations. There is a predefined sequence of these operations. Each operation is processed by only one machine in the predefined order. As a result, a job can at most consist of *m* operations. It also does not have to pass all machines and can not pass from the same machine twice. Figure 6.1 shows an example job shop scheduling problem consisting of three jobs and three machines.

Processing time operations				Machine sequence operations			
Job	1	2	3	Job	1	2	3
j1	3	3	2	j1	m1	m2	m3
j2	1	5	3	j2	m1	m3	m2
j3	3	2	3	j3	m2	m1	m3

Figure 6.1. Example problem with 3 jobs and 3 machines

The machines are not identical and can not replace each other. There is no preemption and only one operation of a job can be processed by one machine. The system is assumed to be ideal which means there is no machine breakdowns. Passing times of jobs between machines and machine specific set-up times are neglected.

A schedule is simply an assignment of operations to machines and a table of starting times of operations. Completion time of an operation is the sum of starting time and processing time of the operation. The earliest possible starting time of an operation depends on the completion times of the predecesor operation of the same machine and the same job.

$$starting_time = \max(t_{job_predecessor} + p_{job_predecessor}, t_{mac_predecessor} + p_{mac_predecessor}) \quad (6.1)$$

In this equation $t_{job_predecessor}$ and $t_{mac_predecessor}$ are the starting times of the previous oprations on the same job and on the same machine respectively. $p_{job_predecessor}$ and $p_{mac_predecessor}$ are the processing times of the previous operations on the same job and on the same machine respectively. If the operation is the same operation of a job then the first term becomes zero. Or if the operation is the first operation on a machine then the second term becomes zero. A feasible schedule for the example problem in Figure 6.1 is shown in Figure 6.2.

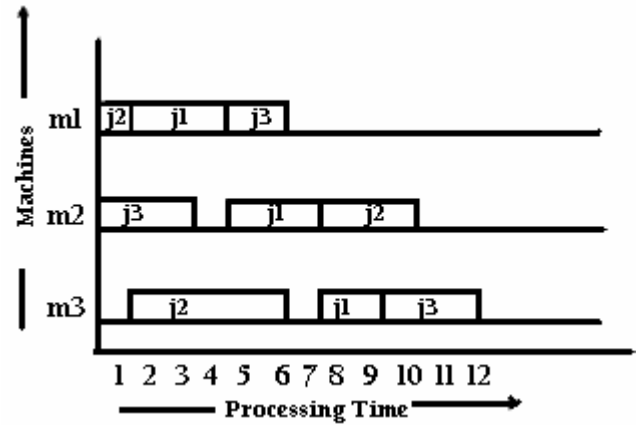


Figure 6.2. A feasible schedule for the example problem

The completion time of a job is the completion time of its last operation. The completion time of a schedule is the completion time of the job which has the longest comletion time. With other words, it is the maximum of all completion times in the schedule.

6.2. The Dynamic Model

A dynamic job shop scheduling problem can be deterministic or sthocastic (non-deterministic). Both of these models are extensions of the static job shop model. In deterministic model, each job has a release time. Release time is the time that a job enters to the shop floor. In this model only the earliest starting time of the first operation of a job changes. It is the maximum of release time of the job and the completion time of the previous operation on the same machine.

$$start_time_{i,1} = \max\left(release_time_i, t_{mac_predecessor} + p_{mac_predecessor}\right) \tag{6.2}$$

Whenever the release times of jobs are taken into account, the model is assumed to be dynamic. All release times are known in advance as a result this is a dynamic but deterministic scheduling problem.

In non-deterministic model, all release times of jobs are not known in advance. Jobs come to the shop floor sthocastically. This is a more realistic case. For this type of environments, the problem is decomposed into several smaller deterministic problems. Each arrival of a new job is considered as a new deterministic problem.

Whenever a new job enters to the shop floor, the release time of this job is asumed to be the starting time of the new problem. The best schedule up to this point is executed and the operations which are executed are removed from the system. If all operations of a job are executed at this release time, this job is totally removed from the system. New release times are calculated for the remaining jobs. If a job has an operation which is being processed at the time of new arrival, this operation is executed and removed from the system and the completion time of this operation is set as the new release time of this job. Otherwise, the release time of the new job becomes the release time of this existing job.

$$release_i = \max_{operations_job_i} \left(t_{oper} + p_{oper} \middle| t_{oper} < t_{release}, \quad t_{release}\right) \tag{6.3}$$

Machine setup times are also considered in this new system. Some machines can be busy at the time of new job arrival. These machines becomes available at he completion times of these jobs. If a machine is idle at the time of new job arrival, the setup time is set to the job arrival time. If a job arrives at time $t_{release}$, machine setup times are calculated with the following equation:

$$setup_i = \max_{operations_machine_i} \left(t_{oper} + p_{oper} \middle| t_{oper} < t_{release}, \quad t_{release} \right) \tag{6.4}$$

Machine setup times are taken into account while defining the earliest starting time for the first operation procesed on this machine. The starting time is the maximum of the completion time of predecessor operation of the same job and the setup time of the machine. If the operation is the first operation of a job and the first operation on a machine then the starting time is the maximum of the job release time and the machine setup time.

6.3. Algorithms

The majority of research in evolutionary computation for job shop scheduling proplem has concentrated around the static case in which all of the jobs are known in advance and starts at time zero. In dynamic jop shop scheduling case, jobs can enter to the system after the beginning of the optimization process. There are only a few evolutionary approaches dealing with dynamic job shop scheduling. In this work, an extensive comparison of these algorithms is performed on a common benchmark problem by varying system workload. In this part, the implemented algorithms are described.

6.3.1. PSRS: Production Scheduling Rescheduling with Genetic Algorithms

For this algorithm [26] a permutation based representation is used in order to serve as a sequence for the decoding procedure. The length of each cromosome is equal to the total number of operations in the system. Each operation is represented with the *id* of the job which it belongs to. As a result each job appears in the chromosome as much as the number of

operations it has. The first occurance represents the first operation, the second one the second operation and so for. For a problem including 3 jobs, each of which consisting of three operations, an example chromosome can be [3, 1, 2, 2, 3, 1, 1, 3].

As a crossover operator Precedence Preservative Crossover (PPX) ,which was developed by Bierwith in 1996 [43], is used. For this operator to work, a string which is as long as a chromosome is used and filled with 1 and 2 values randomly. 1 means select operation from parent1 and 2 means select operation from parent2. Each operation placed in the offspring is deleted from both parents.By this way, the operator passes the presedence relations from both parents to the child. Only one child is generated as a result of crossover.

The used mutation operator only sligthly alters the offspring. This operator randomly picks a gene, deletes and reinserts it from a randomly selected position. Since an indirect representation is used, a decoding procedure is required. The permutations can be decoded into active, semi-active or non-delay schedules.

Semi-active: In order to create a semi-active schedule, the algorithm always schedules the next operation in the permutation. Figure 6.3. shows the procedure to create semi-active schedules:

1. Build the set of all beginning operations A.
2. Select operation o_{ik}^{*} from A which occurs leftmost in the permutation and delete it from A.
3. Append operation o_{ik}^{*} to the schedule and calculate its starting time.
4. If a job successor operation $o_{i,k+1}^{*}$ of the selected operation o_{ik}^{*} exists, insert it into A.
5. if A is not empty go to step 2, else terminate.

Figure 6.3. The procedure for decoding a chromosome into a semi-active schedule [26]

Active: In order to create active schedules, the above procedure is used with a modification in step 2. This algorithm is also known as Giffler & Thomson algorithm [44]. This procedure creates schedules in which no operation can be proessed earlier without delaying any other operations. All active schedules are also semi-active schedules. Figure 6.4 shows the modification that should be made to the procedure in Figure 6.3.

1. Determime the operation o' from A with the possible earliest completion time.
2. Determine the machine m' on which o' is processed. Build the set B from all operations in A which are processed on m'.
3. Delete the operations in B which do not start before the completion of o'
4. Select operation o_{ik}^{*} from B which occurs leftmost in the permutation and delete it from A.

Figure 6.4. The procedure for decoding a chromosome into an active schedule [26]

Non-delay: In order to create non-delay schedules a more rigid criteria for selecting operations from the conflict set (set B) should be used. Again step 2 of Figure 6.3 should be modified. The required modification is seen in Figure 6.5. This procedure creates schedules in which no machine is kept idle, if there is an imminent operation that can be processed immediateley. Non-delay schedules are both active and semi-active.

Optimal schedules are usually active ones, so evolutionary algorithms perform searches in the space of active schedules. On the other hand non-dely schedules has a better mean quality but they may not contain the optimal schedule. In order to search in between, the hybrid schedular is used in this algorithm.

1. Determime the operation o' from A with the possible earliest starting time time.
2. Determine the machine m' on which o' is processed. Build the set B from all operations in A which are processed on m'.
3. Delete operations in B which sart later than operation o'
4. Select operation o_{ik}^{*} from B which occurs leftmost in the permutation and delete it from A.

Figure 6.5: The procedure for decoding a chromosome into a non-delay schedule [26]

Hybrid: In this procedure a parameter is used to define the bound on the length of time that a machine can be kept idle. This parameter is δ. If δ is set to zero, the hyprid dcheduler becomes a non-delay scheduler and creates schedules in which no idle time of machines is allowed. On the other hand, if δ is set to 1, the hybrid scheduler creates active schedules. By setting δ to a value between 0 and 1, the search space can be changed. For $\delta = 0$, the largest search space is achieved and the best quality solution can be found but computation takes more time. If δ increases, the solution quality and computation time is expected to decrease. Figure 6.6 shows the modification that should be made to procedure in Figure 6.3 in order to obtain a hybrid scheduler.

Rescheduling Process: In order to solve a non-deterministic scheduling problem, the problem is decomposed into a series of deterministic problems. The decomposition operation takes place at every arrival of new jobs. If new jobs arrive at time t, the operations which have been started before t are deleted from all the jobs and all the permutations of the population. New release times and machine setup times are calculated for jobs and machines. All the jobs which has no remaining operations are totally removed from the system. The operations of new jobs are randomly inserted to the permutations.

1. Determime the operation o' from A with the possible earliest completion time.
2. Determine the machine m' on which o' is processed. Build the set B from all operations in A which are processed on m'.
3. Determime the operation o'' from B with the possible earliest starting time time.
4. Delete operations in B in accordance to parameter δ such that $B := \{o_{ik} \in B | t_{ik} < t'' + ((t' + p') - t'')\}$
5. Select operation o_{ik}^{*} from B which occurs leftmost in the permutation and delete it from A.

Figure 6.6. The procedure for creating a hybrid scheduler [26]

6.3.2. FLEX : Anticipation and Flexibility in Dynamic Scheduling

For solving a non-deterministic, dynamic optimization problem, many deterministic sub-problems are generated and solved to optimality. For each sub-problem only the front part of the problem is executed and the latter and usully the larger part is subject to rescheduling. Solving each sub-problem optimally may not lead to the globally optimal solution. That is because, this approach does not take “the effect of previous sub-problems on the latter ones” into account. The goal of this algorithm [27] is to take these effects into account in order to create both good and flexible schedules. A flexible schedule can adopt to changing environments better.

This algorithm described in this part resembles to the previous one in some aspects. It uses the same representation and decoding procedure as the previous algorithm which are the permutation representation and hybrid decoding. It also applies Precedence Preserving Crossover (PPX) and the mutation operator used in the previous algorithm.

In a dynamic problem, the solution quality depends on the solution quality obtained in each sub-problem. Since it is known that only the front part of a schedule will be implemented and the rest will be rescheduled, this algorithm searches for flexible schedules that will easily adapt after the arrival of new jobs. For this purpose, fitness function is modified for incresasing the early utilization of machine capacity. With other words this means decreasing the early idle times by penalizing them in the fitness function. That is because any idle time in the fixed schedule will be lost forever and will not be used after rescheduling. The fitness function is modified by adding a term which penalizes early idle times. Both terms in the fitness function are normalized to the interval [0, 1]. A weighting factor α is used to define the effectiveness of both terms in the fitness.

$$f_k = (1-\alpha)\hat{T}_k + \alpha\hat{P}_k \tag{6.5}$$

In this equation [27] $\hat{T}$ is the fitness term and $\hat{P}$ is the flexibility term which penalizes early idle times. These terms are normalized with the following equations:

$$\hat{T}_k = \begin{cases} \dfrac{\overline{T}_k - \min_l\{\overline{T}_l\}}{\max_l\{\overline{T}_l\} - \min_l\{\overline{T}_l\}} & : \ \max_l\{\overline{T}_l\} > \min_l\{\overline{T}_l\} \\ 0 & : \ otherwise \end{cases} \tag{6.6}$$

$$\hat{P}_k = \begin{cases} \dfrac{P_k - \min_l\{P_l\}}{\max_l\{P_l\} - \min_l\{P_l\}} & : \ \max_l\{P_l\} > \min_l\{P_l\} \\ 0 & : \ otherwise \end{cases} \tag{6.7}$$

P is the term penalizing the idle times. The idea is to penalize early idle times, so earlier idle times have particular importance. In order to reflect this idea the penalty of a particular idle time is linearly decreased with time t. The weight is calculated with the following equation:

$$w(t) = \max\left\{0, \quad 1 - \frac{t}{\beta}\right\} \tag{6.8}$$

According to this equation any idle time accuring at time $t = 0$ is weigted with 1 and any idle time occuring after $t = \beta$ is weigted with 0 and has no effect. In this equation β is a user-defined parameter defining the length of the time that will be considered.

By using this fitness function the idea is to generate schedules which are worse in quality but more flexible. Even if the performance of each sub-problem is reduced, the expectation is to increase overall performance by introducing flexibility into the fitness function.

In order to obtain good performance, the parameters δ, α and β should be defined carefully.

6.3.3. SGA: A Genetic Algorithm Approach to Dynamic Job Shop Scheduling Problem

This algorithm [25] uses the direct representation of schedules as chromosomes. The lenth of a chromosome is equal to the number of operations in the problem. Each operation is represented with its starting time in the schedule. The operations are arrenged in their index order. Using a direct representation makes it easier to encode a schedule into a chromosome and gets rid of the problem of false competition. Indirect representations usually suffer from this problem which is the competition of different representations of the same schedule.

The algorithm uses Time Horizon Exchange (THX) crossover and mutation [45] which are proposed in this algorithm. THX crossover is different from other crossover operators since it operates on the schedule level not on the chromosome level. As a standard crossover operator, THX operator selects a crossover point randomly which is a point in time. The operator uses this point as a scheduling decision point in G&T algorithm (see figure 6.7). For the first part of the child schedule which is before the decition point, the temporal relations between the operations are inherited from one parent. For the remaining part which is after the

decision point, relations are inherited from the other parent. The second child is obtained by reversing the parental roles.

Step 1:

Let C contain the first schedulable operation of each job
Let $r_{jm} = 0$ (r_{jm} is the earliest possible starting time of job j on machine m)for all opearations (j, m) in C.

Step 2:

Compute $t(C) = \min_{(j,m) \in C} \{r_{jm} + p_{jm}\}$ and let m^* denote the machine on which the minimum is achieved.

Step 3:

Let G denote the set of all conflicting operations on (j, m^*) On machine m such that $r_{jm^*} < t(C)$

Step 4:

Randomly selaect an operation from G and schedule it

Step 5:

Delete the operation from C; Include its immediate successor in C, update r_{jm} in C and return to step 2 until all operations are scheduled.

Figure 6.7. The Giffler and Thomson algorithm (G & T algorithm) [25]

The other important operator in this algorithm is THX mutation which is based on disjuctive graph. The idea of this operator is to find a critical block and reverse the operations on it. The critiacl block is a set of successive operations on the crital path which operate on the same machine. Such an operator may only create better solutions only if any of the operations is the first of a job or the last operation of a job.

Step 1:

Let C contain the first schedulable operation of each job;
Let $r_{jm} = r_j$ for all operations (j, m) in C.

Figure 6.8. The modification made to G&T algorithm to solve deterministic JSSPs [25]

For determnistic job shop scheduling problems only the first step of Giffler and Thomson algorithm is modified in order to consider job release times (see figure 6.8).

Step 1:

Let C contain the first schedulable operation of each job;
Let $r_{jm} = \max(r_j, a_m)$ for all operations (j, m) in C.
Where r_j is the release time of job *j* and a_m is the setup time of machine *m*.

Figure 6.9. The modification made to G&T algorithm to solve non-deterministic JSSPs [25]

In order to solve sthocastic JSSPs, the problem is decomposed into a series of deterministic sub-problems. Whenever a new job enters to the shop floor, the release time of this job is asumed to be the starting time of the new problem. The best schedule up to this point is executed and the operations which are executed are removed from the system. If all operations of a job are executed at this release time, this job is totally removed from the system. New release times are calculated for the remaining jobs. If a job has an operation which is being processed at the time of new arrival, this operation is executed and removed from the system and the completion time of this operation is set as the new release time of this job. Otherwise, the release time of the new job becomes the release time of the job. Some machines can be busy at the time of new job arrival. These machines becomes available at he completion times of these jobs. If a machine is idle at the time of new job arrival, the setup time is set to the job arrival time. Otherwise it is assignred to the completion time of the processed operation.

For sthocastic optimization problems Giffler and Thomson algorithm is modified in order to take account of both job release times and machine setup times (see figure 6.9).

The rescheduling for sthocastic JSSPs can take two forms. In one approach, the whole population can be restarted as solving a new deterministic problem from scratch. The other approach is to modify the population and continue the search from where it remains at the time of job arrival. Since only a few operations will be removed from the old problem, the new

problem resembles to the old one with a high probobility. In order not to loose this infoprmation, the second approach is used here and the population is modified at each arrival of a new job. For this purpose the operations of new jobs are randomly scheduled among the old operations. In order to implement this idea step 4 of Giffler and Thomson algorithm is modified (see Figure 6.10).

Step 4:
Randomly select an operation from C.
If the operation is from the new jobs, schedule it;
else schedule the operation in G from the old problem with the earliest starting time reported in the individual of the adapted population.

Figure 6.10. The modification made to G&T algorithm to adapt the population to the current problem [25]

6.3.4. OBGT: Order-Based Giffler and Thomson Genetic Algorithm

Order-based GT [46] is a genetic algorithms which combines order-based operators with GT (Giffler and Thomson) and ND (non-delay) methods. An important characteristic for JSSPs is to preserve order of scheduled operations. Uniform-order based crossover and order-based scramble mutation operators are applied for this purpose.

This algorithm uses a direct representation to encode population members. With other words the chromosome representation contains the schedule itself. Each chromosome includes a permutation of operations on each machine and each operation includes its starting time. This representation makes it easier to apply order-based operators.

Uniform order-based crossover operater creates offsprings by passing ordering relationships from both parents. This operator creates a templete whose length is equal to a chromosome length and fills it with 0's and 1's randomly. In order to create the first offspring,

the genes whose index contain 0 in the templete are copied from parent1 to the offspring and the remaining empty places are filled from the second parent by preserving the ordering relationships. The second offspring is generated by copying the members from parent2 ,whose index is filled with 1 in the templete, to offspring and filling the other locations from parent1.

Procedure OBGT Algorithm

1. Generate initial populatioın (50% ND and 50% GT)
2. WHILE not convergence
   - 2.1. FOR all individuals in the population $i = 1....n$
     - i. Order-based crossover with "P_c"
       (First parent is individual i, second parent random (U(0,1)).
     - ii. Order-based mutation with "P_m"
     - iii. Apply GT algorithm to repair both offspring
     - iv. Replace the worst individual(individual n)
       with best offspring (if improve exists)
     - v. Bubble the best offspring (if improve exists)
       to its position according to fitness

Figure 6.11. OBGT Algorithm [46]

Order-based mutation operator selects a sub-list of elements by randomly picking two points x and y. The operator simply permutes the genes between these locations.

After order-based operators are applied to the members, Giffler and Thomson algorithm is used to repair illegal offsprings.

In this algorithm, the initial population is generated by GT and ND aalgorithms. The GT algorithm creates the half of the initial population and the other half is generated by the ND algorithm. As a result the initial population includes both active and non-delay schedules.

At each genearation all population members have a chance to reproduce. There is no selection procedure. The population is always sorted according to fitness. All members from 1 to n are selected in their order of fitness. The first parent is the one whose turn is being processed and the second parent is selected randomly from the whole population. The selected parent first recombined with a predefined probobility Pc and then mutated with a predefined mutation probability Pc.

After recombination and mutation, the generated offsprings can be infeasible ones so both of them are repaired by the GT algorithm. Then the best of these two offsprings is selected to enter the main population. It replaces the worst member only if its fitness value is better than the worse one. Then the population is again sorted according to fitness. Some of the members can be replaced before recombination hence not all members have a chance to reproduce but most of them have this chance.

6.3.5. HCA: Heuristically-Guided GA

This algorithm [47] uses an indirect representation to encode chromosomes. The length of a chromosome is equal to the number of operations in the system. Each gene in the chromosome contains a *(Method, Heuristic)* pair. In this pair, method is the algorithm that will be used to create the conflict set in the GT algorithm. It can take one of two values: active, non-delay. Heuristic is the code of the heuristic that will be used to select one of the operations in the conflict set that will be scheduled next.

The heuristic can take one of the twelve values: WSPT, WLWKR, WTWORK, EGD, EOD, EMOD, MST, SOP, POPNR, PSOP, PWKR, RND. The description of these abbreviations are listed in Table 6.1.

This algorithm is a parallel genetic algorithm with five sub-populations arranged in a ring topology. Migration only occurs at every 5th generation. The best member of each sub-population migrates to the next one.

Table 6.1. Heuristics used for dynamic job shop problem

Rule	**Description**	**Priority of operation (j,m) at time t**
WSPT	Weighted shortest procesing time	w_j / p_{jm}
WLWKR	Weighted least work remaining	w_j / R_j
WTWORK	Weighted total work	w_j / P_j
EGD	Earliest global due date	$1/d_j$
EOD	Earliest operational due date	$1/\lfloor r_j + (d_j - r_j)R_j / P_j \rfloor$
EMOD	Earliest modified operational due date	$1/\max(r_j + (d_j - r_j)R_j / P_j, t + p_{jm})$
MST	Modified slack time	$-(d_j - R_j - t)$
SOP	Slack per operation	$w_j \lfloor 1 - (d_j - R_j - t)/n_j \rfloor / p_{jm}$
POPNR	Lowest ratio of the imminent operation of the weighted value of remaining operations	$1/\left(p_{jm} / \sum_i w_i * p_{im} \right)$
PSOP	Weighted smallest sum of (next processing time + SOP)	$w_j / (p_{jm} + (1 - (d_j - R_j - t)/n_j)/ p_{jm})$
PWKR	Weighted smallest ratio of the processing time to work remaining	$w_j / (p_{jm} / R_j)$
RND	Choose a random operation	Equal priority

The notatioın used in Tble 6.1 is shown in Table 6.2.

Table 6.2. The notation used in Table 6.1

Notation	**Description**
P_j	Total processing time of job *j*
R_j	Remaining processing time of job *j*
p_{jm}	Processing time of job *j* on machine *m*
w_j	Weight of job *j*
d_j	Due date of job *j*
r_j	Release time of job *j*
n_j	Number of remaining jobs for job *j*

At the initialization phase all sub-populations are initialized randomly. Rank based selection is used to select parents, recombination is only allowed between the members of the same sub-population. Uniform crossover and mutation are applied. Uniform crossover operator selects a crossover point randomly and replaces the tail part of the parents to generate offsprings. Since it only replaces gene pairs, the (Method, Heuristic) pairs are not destroyed by crossover operation. The mutation operator mutates a heuristic into another randomly with a predefined probability. If mutation is applied to a gene pair, the corresponding method is also mutated with a probability of 0.5.

6.4. Dynamic Job Shop Scheduling Problem Generator

In order to evaluate and compare performance of different algorithms which have been designed to solve dynamic job shop scheduling problem, an experimental environment, which is often used for simulating manifacturing environments in the literature, is implemented and used (see Holthaus & Rajendran (1997) [48]).

This environment consists of 6 machines and obeys the job shop scheduling problem model which means an ideal system. None of the machines can replace each other and it is assumed that no machine break downs occur in this system. The passing time of jobs between different machines is neglegted.

Jobs arrive to the system continually. Each job consits of at least 4 operations and at most 6 operations resulting in 5 operations on average. The machine order of operations within a job is generated randomly by using a uniform probability distribution. Each job does not have to pass all machines and can not pass from the same machine twice. The processing times of jobs are also generated randomly by sampling from a uniform distribution within the range [1, 19] resulting in a mean processing time of 10 for each operation and fifty (5 * 10) for each job. The inter-arrival times of jobs are generated from an exponential distribution with mean Λ. In this system each job has a weight. Weights are uniformly distributed in the interval [0, 1].

Modelling the arrival process of jobs is a key concept in this environment. The mean inter-arrival time Λ defines workload of the system whic is the number of jobs waiting to be processed in the system. Different utilization rates are used to define Λ and the workload of the system. The mean inter arrival time depends on the mean processing time $\overline{P}$, number of machines m and the utilization rate U. The equation for the mean inter arrival time is as follows:

$$\Lambda = \frac{\overline{P}}{(mU)} \tag{6.9}$$

The generator is initialized with a seed and it is designed in such a way that it always creates the same problem with the same seed. For the experiments in this study, three different utilization rates are used to create test cases with different difficulty levels. The cases are grouped into three categories as easy, moderate and hard. The easy cases are generated with a utilization rate of 0.65. The moderate and hard test cases are generated with utilization rates 0.75 and 0.9 respectively. Ten cases are generated for each type of problem.

6.5. Performance Measures

In order to compare the performance of algorithms four weighted metrics are used. These are the weighted flow time, weighted tardiness, weighted lateness and weighted earliness + tardiness.

For measuring performance of algorithms in static environments, the most widly used measure is the makespan. That is the time span required to execute the best schedule found. The makespan of a schedule is the comletion time of its final operation. Therefore, the schedule which has the least completion time is the best schedule found by an algorithm and it defines the performance of this algorithm. This measure is not proper for dynamic optimization since the makespan or the completion time of a schedule depends on the release time of the last job.

For this reason, instead of makespan, the mean flow time is used as an objective function and performance measure for dynamic optimization. The mean flow time for a schedule is the average of the job flow times. As a result, this objective function is used to minimize the time for procecessing operations in the system and complete the jobs as soon as possible. Mean flow time of a schedule is calculated with the following equation [26]:

$$\overline{F} = \frac{1}{n}\sum_{i=1}^{n}\left(C_i - r_i\right) \tag{6.10}$$

In this equation n is th number of jobs in the system. C_i is the completion time of job i and r_i is the release time of job i. In this work, each job has a weight and weighted flow time is used as an objective function and performance measure.

$$\overline{F} = \frac{1}{n}\sum_{i=1}^{n}\left(\left(C_i - r_i\right) * w_i\right) \tag{6.11}$$

Another measure is the weighted tardiness. Tardiness is an objective function which is based on the lateness of each job. For tardiness only the late jobs are considered ansd the aim is to minimize the amount of time that a job is processed after its due date. The goal is to finish jobs with the minimum lateness. This objective is proper, if the due dates of a manufacturing system is critical and it is important to deliver each job on time. The weighted tardiness is calculşated with the following equation:

$$\overline{T} = \frac{1}{n}\sum_{i=1}^{n}\left(\max\left(0,\left(C_i - d_i\right)\right) * w_i\right) \tag{6.12}$$

The third measure used for this comparison is the weigted lateness. This measure is also based on late jobs but it also includes the jobs that are finised before their due date. The goal is to minimize lateness by finishing jobs before their due dates as much as possible. The equation for this measure is as follows.

$$\overline{L} = \frac{1}{n}\sum_{i=1}^{n}\left(\left(C_i - d_i\right) * w_i\right) \tag{6.13}$$

The last measure used in this work is the total earliness + tardiness. The goal of this objective function is to finish each job just on time, neither later nor earlier than its due date. Any job which finishes later or earlier than its due date increases this measure and decreases the fitness (goodness) of the schedule. This measure is calculated with the following equation:

$$\overline{ET} = \frac{1}{n}\sum_{i=1}^{n}\left(\left(\max\left(0,\left(C_i - d_i\right)\right) + \max\left(0,-1*\left(C_i - d_i\right)\right)\right) * w_i\right) \tag{6.14}$$

6.6. Parameter Settings

A population of 100 individuals is used for all algorithms. All algorithms except OBGT uses generational replacement strategy and rank based selection with linear ranking and elitism is applied for the best individual.

For production scheduling re-scheduling (PSRS), the crossover and mutation rates are taken to be 0.8 and 0.2 respectively. Another important algorithm specific paramater is δ which defines the characteristics of the hybrid schedular. For different utilization rates, different δ values which are recommended in the original paper are used. For easy problems δ is set to 0.7, for moderate problems it is set to 0.5 and for hard problems, δ is taken to be 0.4.

For anticipation and flexibility (FLEX) algorithm, the same crossover and mutation rates are used as in the PSRS algorithm. The algorithm specific parameters for this algorithm are δ, α and β. α is the parameter defining the relative importance of flexibility term and the objective term. With other words it is the weighting factor. α and δ values are varied for different utilization rates. For easy cases, the used α and δ values are 0.4 and 0.5 respectively. For moderate cases, α and δ are set to 0.5 and 0.25 respectively. For the hard cases 0.7 and 0.5 values are used for α and δ. For all cases β is set to 90.

Order-based Giffler and Thomson (OBGT) uses a selection and replacement stragegy similar to GENITOR [6]. It applies μ+1 elitist delection, generates one individual and replaces the worst individual in the population. OBGT does not use a selection strategy. The crossover and mutation rates used for this algorihm are 0.7 and 0.01 respectively.

For SGA algorithm, THX crossover is applied with a probopbility of 0.7 and THX mutation is applied with a probability of 0.01.

6.7. Experimental Results

Dynamic job shop scheduling problems can be stochastic or deterministic. In this part, the algorithms are tested and compared on both type of environments. The experiments are made for four different objective functions: weighted flow time, weighted tardiness, weighted lateness and weighted earliness + tardiness. The results are reported in terms of these four functions. Each experiment is made for three different utilization rates in order to see the effect of increasing workload on the performance of differnet algorithms. In addition to solution quality, computational efforts of the algorithms are also reported and compared in terms of the average execution time and number of fitness evaluations.

6.7.1. Stochastic Job Shop Scheduling Results

For this experiment, utilization rates are varied to see the effect of changing workload on the performance of different algorithms. For each utilization rate ten different test cases are generated by using different seeds. The results are the averages of ten executions for each algorithm. For each performance metric, the objective function is changed and the experiments are repeated on the same test cases.

Each case starts with ten jobs and 1 or 2 jobs are added at every change. If the inter-arrival time of a the first added job is very close to 0, then another job is added to the system. For each sub problem, the algorithms are run for 20 generations to optimize the sub-problem. With other words, jobs are added at every twenty generations. Each case consists of 500 jobs.

In order to obtain the true utilization rates, first and last 100 jobs are not included in the performance of the algorithms.

Table 6.3 shows the average performances of algorithms in terms of weighted flow time, for various utilization rates. According to weighted flow time values, the best performer algorithm is the PSRS algorithm for all utilization rates. As utilization rate increases, all of the algorithms degrades in performance. The most heavily affected algorithm is the FLEX algorithm. This result may be due to early idle times. The main goal of this algorithm is not to process jobs as soon as possible. This algorithm tries to find a schedule which has the least idle time for the early parts of scheduling. This may result in more waiting on the other parts of the scheduling process and the effect becomes more apperant as the number of jobs in the system waiting processing increases.

Table 6.3. Weighted flow time values of different algorithms for various utilization rates

Algorithms	**Weighted Flow Time**		
	Utilization Rates		
	0.65	**0.75**	**0.9**
PSRS	49.671	58.581	76.990
FLEX	52.383	64.527	117.282
OBGT	54.366	69.407	98.138
SGA	56,530	68,362	95,272

Table 6.4 shows the average execution times and evaluation counts of algorithms. The algorithm which has the highest time computational effort is the OBGT algorithm. That is because, this algorithm makes more number of evaluations and always sorts the population according to fitness after each crossover and mutation. The algorithm which requires the least computational effort is PSRS since it has less evaluation count and the PPX crossover and mutation used by this algorithm works faster than other operators. This algorithm works even faster than FLEX which uses the same operator nad makes same number of evaluations. That is because, FLEX computes and penalizes the early idle times.

Table 6.4. Average execution time and evaluation counts for various utilization rates

Algorithms	Weighted Flow Time					
	Execution Time			Number of Evaluations		
	U = 0.65	U = 0.75	U = 0.9	U = 0.65	U = 0.75	U = 0.9
PSRS	65,800	121,000	454,700	104,000	104,000	104,000
FLEX	78,000	198,000	563,000	104,000	104,000	104,000
OBGT	156,400	342,100	1102,600	204,000	204,000	204,000
SGA	70,900	142,900	432,000	114,000	114,000	114,000

Table 6.5 shows the weighted tardiness values of different algorithms for different utilization rates. Tardiness is the amount af time that jobs passes their due dates. Early jobs are not considered in tardiness. The results reflects the results of weighted flow time. The algorithm which processes the jobs as soon as possible has the least tardiness. As a result PSRS has the least tardiness value and SGA has the highest. The tardiness values also increases as utilization rate increases. Again the most affected algorithm is the FLEX algorithm which penalizes early idle times. SGA algorithm works worse than OBGT, but it is affected less from the changes in the utilization rate. As the workload of the system increases, SGA deals with this problem better than OBGT.

Table 6.5. Weighted tardiness values of different algorithms for various utilization rates

Algorithms	Weighted Tardiness		
	Utilization Rates		
	0.65	0.75	0.9
PSRS	2,725	7,640	23,147
FLEX	3,553	11,580	41,900
OBGT	5,187	15,993	41,058
SGA	5,262	15,290	35,828

The execution times and fitness counts listed in Table 6.6 resembles the results of the previous experiment. Only the objective function changes, so the computational effort of algorithms do not change much. Small changes occur, this differences may be due to the number of crossover and mutation operators applied during execution.

Table 6.6. Average execution time and evaluation counts for various utilization rates (U)

Algorithms	Weighted Tardiness					
	Execution Time			Number of Evaluations		
	U = 0.65	U = 0.75	U = 0.9	U = 0.65	U = 0.75	U = 0.9
PSRS	65,200	124,400	409,500	104,000	104,000	104,000
FLEX	79,400	144,800	433,800	104,000	104,000	104,000
OBGT	160,300	351,100	1107,600	204,000	204,000	204,000
SGA	76,300	158,300	408,800	114,000	114,000	114,000

Table 6.7. Weighted lateness values of different algorithms for various utilization rates

Algorithms	Weighted Lateness		
	Utilization Rates		
	0.65	0.75	0.9
PSRS	-16,385	-6,697	31,730
FLEX	-13,909	-0,748	37,932
OBGT	-9,763	5,090	37,118
SGA	-10,131	6,079	32,158

Table 6.8. Average execution time and evaluation counts for various utilization rates

Algorithms	Weighted Lateness					
	Execution Time			Number of Evaluations		
	U = 0.65	U = 0.75	U = 0.9	U = 0.65	U = 0.75	U = 0.9
PSRS	64,700	119,400	456,600	104,000	104,000	104,000
FLEX	77,700	150,100	434,800	104,000	104,000	104,000
OBGT	169,000	343,800	1183,300	204,000	204,000	204,000
SGA	75,100	206,700	442,900	114,000	114,000	114,000

Table 6.7 shows the average weighted lateness values for different algorithms. A negative lateness value means that jobs usually finish before their due dates. For utilization rate 0.65 which is a relaxed and easy situation, all algorithms are able to finish jobs before their due dates and all of them have negative lateness values. For an easy test case the

workload of the system changes between 10 and 20 jobs. When the utilization rate increases to 0.75, the jobs in system changes between 20 and 30. This is a moderate case and the lateness of algorithms increases. Since there are more jobs waiting to be processed or competing for the scarce resources, the system has difficulty in comleting all the jobs on time. Again the best performer algorithm is PSRS for all cases. The worst algorithm is the SGA algorithm for the easy case, but for higer workloads it works better and less affected than others.

Table 6.9. Weighted earliness + tardiness values of different algorithms for various utilization rates

Algorithms	**Weighted Earliness + Tardiness**		
	Utilization Rates		
	0.65	**0.75**	**0.9**
PSRS	7,123	11,548	25,642
FLEX	11,302	18,127	48,239
OBGT	9,424	19,430	42,696
SGA	12,231	19,352	39,941

Table 6.9 shows the weighted earliness + tardiness values of algorithms. This measure shows the ability of algoriths to finish jobs just on time neither early nor late. The best performer algorithm is the PSRS algorithm. According to other metrics FLEX performs better than OBGT ;but for this case OBGT is better than FLEX which means that FLEX algorithm finishes jobs earlier than their due date. This result may be due to penalizing idle times.

Table 6.10. Average execution time and evaluation counts for various utilization rates

Algorithms	**Weighted Earliness + Tardiness**					
	Execution Time			**Number of Evaluations**		
	U = 0.65	**U = 0.75**	**U = 0.9**	**U = 0.65**	**U = 0.75**	**U = 0.9**
PSRS	72,700	119,700	427,800	104,000	104,000	104,000
FLEX	93,600	163,000	435,600	104,000	104,000	104,000
OBGT	182,300	368,200	1151,000	204,000	204,000	204,000
SGA	78,100	177,600	427,200	114,000	114,000	114,000

Table 6.10 shows the average execution times and evaluation counts for the last experiment. Results reveal that as shift length increases, average excecution time of algorithms increases while the average evaluation counts remains the same. This increase in the execution time is a result of the increase in the workload of the system. As the number of jobs in the system increases, the algorithms deal with longer schedules. Evaluating, mutating and recombining these schedules takes more time.

6.7.2. Deterministic Job Shop Scheduling Results

For this experiment, utilization rates are varied to see the effect of changing workload on the performance of different algorithms in a deterministic environment. For each utilization rate ten different test cases are generated by using different seeds. Each case consists of 30 jobs which have exponentially distributed inter-arrival times. Since the problem is deterministic, all the jobs are known in the initialization of the problem. The results are the averages of ten executions for each algorithms and each algorithm is run for 1000 generations for each case. For each performance metric, the objective function is changed and the experiments are repeated on the same test cases.

Table 6.11. Weighted flow time values of different algorithms for various utilization rates in deterministic environment

Algorithms	**Weighted Flow Time**		
	Utilization Rates		
	0.65	**0.75**	**0.9**
PSRS	47,438	52,661	61,978
FLEX	51,678	55,780	62,483
OBGT	45,926	50,325	56,586
SGA	46,949	52,104	55,352
HCA	46,543	51,668	55,809

Table 6.11 shows the weighted flow time values of different algorithms in a deterministic environment. The results reveal that the best performing algorithm is the OBGT algorithm This algorithm uses steady-state model and does not use a stochastic selection schema. It always replaces the worst individual and sorts the population according to their

fitness. By the way, the algorithm obtains better results than other stochastic algorithms. For the stochastic problem, this algorithm is not the best one. This may be due to lose of diversity in the population. In deterministic problem, only the jobs have release times; but the problem is totally known at the start of optimization. But in stochastic case, losing diverstiy is a problem since jobs are added continually and any solution which is very good for the previous sub-problem may be very bad for the next one. So the population should still have diversity. FLEX alorithm has the worst performance and this is an expected result. This algorithm penalizes early idle times in order to use this idle times after rescheduling. Since in deterministic problem, there is no rescheduling, the use of flexity component in the fitness function degrades the performance of the algorithm.

Table 6.12. Average execution time and evaluation counts for various utilization rates in deterministic environment

Algorithms	**Weighted Flow Time**					
	Execution Time			**Number of Evaluations**		
	U = 0.65	**U = 0.75**	**U = 0.9**	**U = 0.65**	**U = 0.75**	**U = 0.9**
PSRS	95,000	100,800	106,100	100,000	100,000	100,000
FLEX	96,300	102,500	104,900	100,000	100,000	100,000
OBGT	135,100	144,200	143,500	199,000	199,000	199,000
SGA	60,100	60,800	61,700	109,400	109,400	109,600
HCA	64,300	68,400	69,200	100,000	100,000	100,000

Table 6.12 shows the average execution times and evaluation counts for various utilization rates in deterministic environments. OBGT has the highest time overhead due to the high number of evaluations and sorting operations. SGA and HCA works much faster than PSRS and FLEX because the operators used for these algorithms works faster than the PMX crossover and mutation operators used by PSRS and FLEX. Computational effort is not affected by the utilization rate since the number of jobs included in each generation of the genetic algorithm is fixed and thirty only the inter-arrival times change.

Table 6.13 shows the weighted tardiness values of different algorithms in a deterministic environment. In this case OBGT and SGA algorithms give the best results according to tardiness objective. This means these two algorithms are better than others while completing

jobs before their due dates. On the other hand these algorithms are more affected than HCA from the changes in the utilization rate. As the workload of the system increases, HCA deals with this situation better than others and it is the best algorithm for the last case (utilization rate = 0.9)

Table 6.13. Weighted tardiness values of different algorithms for various utilization rates in deterministic environment

Algorithms	Weighted Tardiness		
	Utilization Rates		
	0.65	0.75	0.9
PSRS	6,394	11,230	14,086
FLEX	7,688	14,192	19,795
OBGT	3,815	6,673	9,041
SGA	3,811	6,127	9,067
HCA	4,128	7,262	8,878

Table 6.14. Average execution time and evaluation counts for various utilization rates in deterministic environment

Algorithms	Weighted Tardiness					
	Execution Time			Number of Evaluations		
	U = 0.65	U = 0.75	U = 0.9	U = 0.65	U = 0.75	U = 0.9
PSRS	95,500	102,700	102,600	100,000	100,000	100,000
FLEX	97,300	103,800	104,100	100,000	100,000	100,000
OBGT	137,300	147,700	144,100	199,000	199,000	199,000
SGA	59,100	60,600	62,400	109,300	109,800	109,700
HCA	65,200	69,800	69,500	100,000	100,000	100,000

Table 6.14 shows the execution times and evaluation counts of algorithms for various utilization rates. The results are similar to results shown in table 6.12. This means computational effort does not change between two objectives. That is an expected situation since the behaviour of algorithms does not change by changing the objective function, only goal of optimization changes.

The results in Table 6.15 reveals that the best performer algorithm according to weighted lateness is the SGA algorithm. This algorithm is also the best algorithm according to weighted tardiness results. This means SGA algorithm is better than others in completing jobs before their due dates, it also completes them earlier than their due dates. This may be a result of using THX operators. These operators work on schedule level instead of chromosome level and they are more sensitive to completion times of jobs. The worst performer algorithm is the FLEX algorithm and it is also the most heavily affected algorithm from the cahages in utilization rates. That is an expected result since this algorithm penalizes early idle times to create flexible schedules. It may assume that a schedule which has less early idle time, is better than other schedules even if it has more jobs which does not finish by their due dates..

Table 6.15. Weighted lateness values of different algorithms for various utilization rates in deterministic environment

Algorithms	**Weighted Lateness**		
	Utilization Rates		
	0.65	**0.75**	**0.9**
PSRS	-5,436	-0,654	5,785
FLEX	-1,754	3,397	16,812
OBGT	-5,816	-1,174	2,867
SGA	-6,822	-1,722	2,454
HCA	-4,507	-0,044	3,806

Table 6.16. Average execution time and evaluation counts for various utilization rates in deterministic environment

Algorithms	**Weighted Lateness**					
	Execution Time			**Number of Evaluations**		
	U = 0.65	**U = 0.75**	**U = 0.9**	**U = 0.65**	**U = 0.75**	**U = 0.9**
PSRS	95,500	100,600	104,000	100,000	100,000	100,000
FLEX	97,400	102,900	105,900	100,000	100,000	100,000
OBGT	138,700	149,400	144,400	199,000	199,000	199,000
SGA	58,700	60,800	61,900	109,400	109,400	109,500
HCA	64,900	68,700	69,200	100,000	100,000	100,000

When the results in table 6.16 are investigated, it is seen that the computational is not affected from the changes in the utilization rate or in the objective function.

Table 6.17. Weighted earliness + tardiness values of different algorithms for various utilization rates in deterministic environment

Algorithms	Weighted Earliness + Tardiness		
	Utilization Rates		
	0.65	0.75	0.9
PSRS	14,769	15,781	19,487
FLEX	16,542	18,632	24,674
OBGT	8,262	10,747	13,316
SGA	9,597	11,477	12,912
HCA	12,691	14,575	14,605

Table 6.18. Average execution time and evaluation counts for various utilization rates in deterministic environment

Algorithms	Weighted Earliness + Tardiness					
	Execution Time			Number of Evaluations		
	U = 0.65	U = 0.75	U = 0.9	U = 0.65	U = 0.75	U = 0.9
PSRS	99,300	106,200	105,800	100,000	100,000	100,000
FLEX	99,400	104,500	107,400	100,000	100,000	100,000
OBGT	141,700	151,800	146,300	199,000	199,000	199,000
SGA	60,900	62,300	62,600	109,500	109,300	109,500
HCA	66,000	68,200	70,000	100,000	100,000	100,000

The last objective, weighted earliness + tardiness, compares the algorithms according to their ability in completing jobs on time neither late nor early. The results are shown in table 6.17. According to this objective OBGT is the best performer algorithm. SGA is worse than OBGT. According to weighted tardiness and weighted lateness SGA is better which means it finishes job earlier than their due dates But for this measure finishing earlier or later both has a negative effect on the performance of algorithms, finishing just on time is aimed. The least affected algorithm from the changes in utilization rate is HCA and the worst affected algorithm is the FLEX algorithm.

7. CONCLUSION

In this thesis, first, we present a complete and an extensive performance evaluation of leading evolutionary optimization techniques in dynamic environments. We have examined and implemented a set of 13 evolutionary optimization techniques on a common platform by using the moving peaks benchmark and by varying important problem parameters such as shift length, change frequency, correlation and the number of peaks in the landscape. Ten algorithms in the set can be classified into three distinct categories [4] according to their functionalities: i) approaches that maintain diversity throughout the run, ii) memory-based approaches, and iii) multi-population approaches. The performance study covers algorithms with different philosophies and characteristics in order to show their relative performance under different environmental changes. Standard evolutionary algorithm is also included in this study in order to see the improvement achieved by different approaches.

Two new techniques are also proposed, which were constructed by hybridizing several leading approaches. The first one is the hybridization of self organizing scouts with local search provided by crossover hill climbing method, which is called SOS+LS algorithm in this thesis. The second one is the combination of multinational GA method with random immigrants method, which is called MN+RI algorithm. The SOS+LS algorithm significantly outperforms the other 12 algorithms with respect to various problem parameters including severity of change, frequency of change, predictability of change and the problem complexity.

In this work, both the solution quality and computational efforts of algorithms are compared. In order to compare solution quality or the efficiency of algorithms, the results are reported in terms of offline error. Additionally, a new comparison metric which is based on signal similarity is proposed and used for performance evaluation of algorithms. Computational effort is reported in terms of average number of fitness evaluations and average execution time.

When the results of the algorithms are observed, the most effective ones are the multi-population approaches and hybrid approaches. These algorithms are capable of tracking local optima as well as global optima. On the other hand, the multi-population approaches take more time than other approaches. If accuracy of solution is more important than the time it takes to find the optimal solution, these algorithms can be considered. The most heavily affected algorithms from the dynamism are the memory based ones since they depend on old good solutions.

Based on the experimental study, it is also observed that the hybrid methods outperform the related work with respect to quality of solutions for various parameters of the given benchmark problems. Up to 24 percent improvement is obtained by the hybrid methods in the experiment related to shift length. In the experiments related to change frequency and correlation parameters, hybrid methods performs up to 22 percent better. In the last experiment, number of peaks in the landscape is varied. Hybrid methods show up to 24 percent improvement in this experiment.

In this thesis, we also perform the performance evaluation of algorithms which have been designed to solve dynamic job shop scheduling problems. Most research in the area of scheduling is centered on static scheduling problems. There are 5 algorithms in the literature for solving dynamic job shop scheduling problems. An extensive comparison of these algorithms on both deterministic and stochastic job shop problem is performed by varying the system workload. The algorithms are evaluated according to four objective functions: weighted flow time, weighted lateness, weighted tardiness and weighted earliness + tardiness.

The results are reported in terms of these four metrics. Based on the results we conclude that the best performing algorithm for stochastic job shop scheduling problem is the PSRS algorithm and the best algorithms for the deterministic case are the OBGT and SGA algorithms which give very similar results in deterministic environments. The performances of all algorithms degrade as the workload increases in both the stochastic and deterministic cases. And there is not an algorithm which works best for all cases. Different environment conditions require different algorithms with different characteristics.

REFERENCES

1. Branke, J., *Evolutionary Optimization in Dynamic Environments*, Kluwer, 2001.

2. Branke, J., *Evolutionary algorithms for dynamic optimization problems - a survey*, Institute AIFB, University of Kalsruhe, *Technical Report*, 387, 1999.

3. Jin, Y. and J. Branke, "Evolutionary optimization in uncertain environments-a survey," *IEEE Transactions on Evolutionary Computation*, vol. 9, no. 3, pp. 303-317, 2005.

4. Branke, J., *The moving peaks benchmark*, http:// www.aifb.unikarlsruhe.de/jbr/MovPeaks

5. Eiben, A. E. and J.E. Smith, *Introduction to Evolutionary Computing*, Springer, 2003.

6. Whitley, L. D. and J. Kauth, "GENITOR: a different genetic algorithm", *Proceedings of the Rocky Mountain Conference on Artificial Intelligence*, pp. 118-130, 1988.

7. Cobb, H. G. and J. Gerfenstette, "Genetic algorithms for tracking changing environments", *5th International Conference on Genetic Algorithms*, pp. 523-530, Morgan Kaufman, 1993.

8. Cobb, H. G., *An investigation into the use of hypermutation as an adaptive operator in genetic algorithms having continues, time-dependent non-stationary environments* , Naval Research Laboratory, Washington, USA, *Technical Report AIC-90-001*, 1990.

9. Vavak, F., T. C. Fogarty and K. Jukes, "A genetic algorithm with variable range of local search for tracking changing environments", *Parallel Problem Solving from Nature*, ser. LNCS, no. 1141. Springer Verlag Berlin, 1996.

10. Grefenstette, J. J., “Genetic algorithms for changing environments”, *Parallel Problem Solving from Nature 2*, pp. 137-144, Berlin, North Holland, 1992.

11. Mori, N., H. Kita and Y. Nishikawa, “Adaptation to a changing environment by means of the thermo-dynamical genetic algorithm”, *Parallel Problem Solving from Nature*, ser. LNCS, no. 1141, pp. 513-522, Springer Verlag Berlin, 1996.

12. Cedeno, W. and V. R. Vemuri, “On the use of niching for dynamic landscapes”, *International Conference on Evolutionary Computation*”, IEEE, 1997.

13. Branke, J., “Memory enhanced evolutionary algorithms for changing optimization problems”, *Congress on Evolutionary Computation CEC99*, vol. 3, pp. 1875-1882, IEEE, 1999.

14. Ramsey, C. L. and J. J. Grefenstette, “Case-based initialization of genetic algorithms”, *International Conference on Genetic Algorithms*, ser. LNCS, pp. 84-91, Morgan Kaufmann, 1993.

15. Ryan, C., “Diploidy without dominance”, *Third Nordic Workshop on Genetic Algorithms*, pp. 45-52, 1997.

16. Bendtsen, C. N. and T. Krink, “Dynamic memory model for non-stationary optimization”, *Congress on Evolutionary Computation*, pp. 145-150, IEEE, 2002.

17. Branke, J., T. Kaussler, C. Schmidt and H. Schmeck, “A multi-population approach to dynamic optimization problems”, *Adaptive Computing in Design and Manufacturing 2000*, pp. 299-308, Berlin, Springer Verlag, 2000.

18. Branke, J. and H. Schmeck, “Designing evolutionary algorithms for dynamic optimization problems", *Theory and Application of Evolutionary Computation: Recent Trends*, pp. 239-262, Springer, 2002.

19. Ursem, R. K., “Multinational GA multimodal optimization techniques in dynamic environments”, *Genetic and Evolutionary Computation Conference GECCO-2000*, pp. 19-26, Morgan Kaufmann, 2000.

20. Oppacher, F. and M. Wineberg, “The shifting balance genetic algorithm: Improving the ga in a dynamic environment”, *Genetic and Evolutionary Computation Conference (GECCO)*, vol. 1, pp. 504-510, Morgan Kaufmann, San Francisco, 1999.

21. Branke, J., E. Salihoglu and S. Uyar, “Towards an analysis of dynamic environments", *Genetic and Evolutionary Computation Conference - GECCO-2005*, pp. 1433-1439, 2005.

22. Morrison, R. W. and K. A. DeJong, “A test problem generator for non-stationary environments", *Congress on Evolutionary Computation*, vol. 3, pp. 2047-2053, IEEE, 1999.

23. Younes, A., P. Calamai and O. Basir, “Generalized benchmark generation for dynamic combinatorial problems", *GECCO Workshop on Evolutionary Algorithms for Dynamic Optimization*, pp. 25-31.

24. Bierwirth, C. and H. Copher, *Dynamic task scheduling with genetic algorithms in manufacturing systems*, Department of Economics , University of Bremen, Germany, *Technical Report*, 1994.

25. Lin, S. C., E. D. Goodman and W. Punch, “A genetic algorithm approach to dynamic job shop scheduling problems", *Seventh International Conference on Genetic Algorithms*, Morgan Kaufmann, 1997, pp. 481-488.

26. Bierwirth, C. and D. C. Mattfeld, “Production scheduling and rescheduling with genetic algorithms", *Evolutionary Computation*, vol. 7, no. 1, pp. 1-18, 1999.

27. Branke, J. and D. Mattfeld, “Anticipation in dynamic optimization: The scheduling case", *Parallel Problem Solving from Nature (PPSN VI)*, ser. LNCS, vol. 1917, pp. 253-262, Springer, 2000.

28. Stanhope, S. A. and J. M. Daida, “Optimal mutation and crossover rates for a genetic algorithm operating in a dynamic environment", *Evolutionary Programming VII*, ser. LNCS, no. 1447, pp. 693-702, Springer, 1998.

29. Collard, P., C. Escazut and A. Gaspar, “An evolutionary approach for time dependent optimization", *International Journal on Artificial Intelligence Tools*, vol. 6, no. 4, pp. 665-695, 1997.

30. Angeline, P. J., “Tracking extreme in dynamic environments”, *Sixth International Conference on Evolutionary Programming*, ser. LNCS, vol. 121, pp. 335-345, Springer, 1997.

31. Back, T., “On the behavior of evolutionary algorithms in dynamic environments", *IEEE International Conference on Evolutionary Computation*, pp. 446-451, IEEE, 1998.

32. Mori, N., S. Imanishi, H. Kita and Y. Nishikawa, “Adaptation to changing environments by means of the memory based thermo-dynamical genetic algorithms”, *International Conference on Genetic Algorithms*, pp. 299-306, Morgan Kaufmann, 1997.

33. Lewis, J., E. Hart and G. Ritchie, “A comparison of dominance mechanisms and simple mutation on non-stationary problems”, *Parallel Problem Solving from Nature*, ser. LNCS, no. 1498, pp. 139-148, Springer, 1998.

34. Jong, K. D., *An analysis of the behavior of a class of genetic adaptive systems*, Ph.D. dissertation, University of Michigan, Ann Arbor MI, 1975.

35. Morrison, R., “Performance measurement in dynamic environments", *GECCO Workshop on Evolutionary Algorithms for Dynamic Optimization Problems*, pp. 5-8, 2003.

36. Weicker, K. and N. Weicker, “On evolution strategy optimization in dynamic environments”, *Congress on Evolutionary Computation*, vol. 3, pp. 2039-2046, 1999.

37. Trojanowski, K. and Z. Michalewicz, “Searching for optima in non-stationary environments”, *Congress on Evolutionary Computation*, vol. 3, pp. 1843-1850, IEEE, 1999.

38. Gaspar, A. and P. Collard, “From gas to artificial immune systems: Improving adaptation in time dependent optimization”, *Congress on Evolutionary Computation*, vol. 3, pp. 1859-1866, IEEE, 1999.

39. Weicker, K., “Performance measures for dynamic environments”, *Parallel Problem Solving from Nature*, ser. LNCS, vol. 2439, pp. 64-73, Springer, 2002.

40. Simoes, A. and E. Costa, “Using genetic algorithms to deal with dynamic environments: A comparative study of several approaches based on promoting diversity”, *Proceedings of the Genetic and Evolutionary Computation Conference- GECCO 2002*, New York, USA, 2002.

41. Tsutsi, S., Y. Fujimoto and A. Ghosh, “Forking genetic algorithms: Gas with search space Division schemes”, *Evolutionary Computation*, vol. 5, no. 1, pp. 61-80, 1997.

42. Lozano, M., F. Herrera, N. Krasnogor and D. Molina, “Real-coded memetic algorithms with crossover hill-climbing”, *Evolutionary Computation*, vol. 12, no. 3, pp. 273-302, 2004.

43. Bierwirth, C., D. C. Mattfeld and *H.* Kopfer, “On Permutation Representations for Scheduling Problems”, *Parallel Problem Solving from Nature Conference (PPSN4)*, pp. 310-318, 1996.

44. Giffler, B. and G. L. Thomson, “Algorithms for Solving Production Scheduling Problems”, *Operations Research 8*, 487-503, 1960.

45. Lin, S., E. Goodman and W. Punch, “Investigating Parallel Genetic Algorithms on Job Shop Scheduling Problems”, *6th International Conference on Evolutionary Programming*, pp. 383-393,1997.

46. Vazquez, M. and L. D. Whitley, “A Comparison of Genetic Algorithms for the Static Job Shop Scheduling Problem**”,** *Parallel Problem Solving from Nature Conference 2000 (PPSN VI)*, pp: 303-312, 2000.

47. Hart, E. and P. Ross, “A Heuristic Combination Method for Solving Job-shop Scheduling Problems”, *Proceedings of the V Parallel Problem Solving From Nature (PPSN V),* Lecture Notes in Computer Science, Vol. 1498, pp. 845-854, Springer, 1998.

48. Holthaus, O. and C. Rajendran, "Efficient Dispatching Rules for Scheduling in a Job Shop", *International Journal of Production Economics*, Vol. 48, No. 1, pp. 87 - 105, 1997.