"IMPLEMENTATION OF HYBRID EVOLUTIONARY ALGORITHM FOR BDD OPTIMIZATION BY FINDING OPTIMUM VARIABLE ORDERING"

Submitted in Partial Fulfilment of the Requirements for the award of the degree of

MASTER OF TECHNOLOGY

In

VLSI Design and CAD

Submitted By

SANISHA

Roll no. 601161022

Under the guidance of

Mrs. Manu Bansal, Assistant Professor, ECED, T.U, Patiala

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

THAPAR UNIVERSITY, PATIALA

June 2013

CERTIFICATE

I hereby certify that the work which is being presented in this thesis entitled "**Implementation of Hybrid Evolutionary Algorithm for BDD Optimization by Finding Optimum Variable Ordering**" by me in partial fulfilment of requirements for the award of degree of Master of Technology in VLSI Design and CAD from Thapar University, Patiala, is an authentic record of my study carried under the guidance and supervision of Mrs. Manu Bansal (Assistant Professor), ECED.

The matter presented in this thesis has not been submitted in any other University or Institute for the award of degree.

Date: 21st June, 2013

Sanisha

SANISHA

Roll No. 601161022

It is certified that the above statement made by the student is correct to the best of my knowledge and belief.

Date: 21st June, 2013

Manu Bansal

Mrs. Manu Bansal

Assistant Professor

ECED, Thapar University

Countersigned by:

(Dr. Rajesh Khanna)

Professor and Head

ECED, Thapar University

Patiala-147004

Dean of Academic Affairs

Thapar University

Patiala-147004

ACKNOWLEDGEMENT

This report is completed with prayers of many and love of my family and friends. However, there are a few people that I would like to specially acknowledge and extend my heartfelt gratitude who have made the completion of this report possible. With the biggest contribution to this report, I would like to thank **Mrs. Manu Bansal** had given me her full support in guiding me with stimulating suggestions and encouragement to go ahead in all the time of the report.

I am also thankful to **Dr. Rajesh Khanna**, Professor and Head, Electronics and Communication Engineering Department, for providing us with the adequate infrastructure for carrying out the work.

I am also thankful to **Dr. Kulbir Singh**, P.G Coordinator, Electronics and Communication Engineering department, for the motivation and inspiration that triggered me for the work.

Lastly, I would also like to thank my parents for their years of unyielding love and encourage. They have always wanted the best for me and I admire their determination and sacrifice.

SANISHA

ABSTRACT

Digital integrated circuits, often represented as Boolean functions, can be best-manipulated graphically in the form of Binary Decision Diagrams (BDD). Reduced-ordered binary decision diagrams (ROBDDs) are a data structure for representation and manipulation of Boolean functions. A major problem with BDD-based manipulation is the need for application-specific heuristic algorithms to order the input variables before processing. The order selected on input variables influences the size of the BDD, and with that, the average computation time and storage requirement, varying it from linear to exponential. Therefore, finding a good variable order for OBDDs is an essential part of OBDD-based *CAD* tools.

In the current problem of variable ordering, a number of algorithms exist in this context each with its own advantages and disadvantages. The technique that has been proposed and used in this thesis work is a Modified Memetic Algorithm (MMA) based technique that finds an optimal input variable order with an aim to reduce node count for a multi-input multi-output MIMO boolean function. An intrinsic feature of this algorithm of exploiting all available knowledge about the problem under study is coupled with global space exploration of genetic algorithm. Hence, as a result, this technique provides optimal BDD sizes for most of the benchmark circuits in lesser iterations, thus reducing computation time as well. Comparison tables have been presented to show the effectiveness of the proposed algorithm as compared to other previously implemented *state-of-art* techniques for variable ordering of shared OBDDs. These tables illustrate that using the Modified Memetic Algorithm MMA results in 24.03% average reduction in number of nodes as compared to original size and 2.75% average improvement with respect to the best known dynamic algorithm, i.e. Sift Algorithm.

TABLE OF CONTENTS

CHAPTER CONTENTS PAGE NO.

CHAPTER CONTENTS **PAGE NO.**

CHAPTER	CONTENTS	PAGE NO.

LIST OF ABBREVIATIONS

ABBREVIATION	MEANING
BB	Branch and Bound
BDD	Binary Decision Diagram
DAG	Directed Acyclic Graph
EA	Evolutionary Algorithm
GA	Genetic Algorithm
HGA	Hybrid Genetic Algorithm
LS	Local Search
MA	Memetic Algorithm
MIMO	Multiple Input Multiple Output
MMA	Modified Memetic Algorithm
MUX	Multiplexer
NP	Non-Polynomial in Time
OBDD	Ordered Binary Decision Diagram

ABBREVIATION	MEANING
PSO	Particle Swarm Optimization
PTL	Pass Transistor Logic
ROBDD	Reduced Ordered Binary Decision Diagram
SDD	Shared Binary Decision Diagram
WIN	Window Permutation Algorithm

LIST OF FIGURES

LIST OF TABLES

1. Introduction

Boolean function manipulation is an important component of many logic synthesis algorithms including logic optimization and logic verification of combinational and sequential circuits. Synthesis, verification, and testing algorithms of VLSI circuits manipulate large number of switching functions. Therefore it is important to have efficient methods to represent and manipulate such functions. A large class of problems in the area of VLSI CAD can be solved by adopting efficient underlying data structures. During the last decade, several methods based on Decision Diagram have been proposed and successfully used in many industrial applications. A Boolean function that describes a digital circuit can be represented as a Binary Decision Diagram (BDD) which is a directed acyclic graph with respect to an order and satisfying a set of properties. The number of its non- terminal nodes gives their size. For multiplexer based design styles such as Pass Transistor Logic (PTL), a smaller number of nodes directly transfers to a smaller chip area. A BDD can be implemented either in canonical form, reduced order (ROBDD) or in any other specific order (OBDD). If all identical nodes are shared and all redundant nodes are eliminated, the OBDD is said to be reduced (an ROBDD).

The size of a BDD is strongly dependent on the order of input variables. It is not unusual for the size of an OBDD to increase exponentially with the circuit size using one variable order, but linearly using another variable order. Since the memory requirement and processing time increase as the OBDD size increases, it is important to keep the size of the OBDD as small as possible. Hence a number of heuristics and algorithms have been developed to determine the optimal ordering for input variables.

1.1 Introduction to Binary Decision Diagram (BDD)

BDD is a multi-rooted (shared) directed acyclic graph (DAG) which is used to represent a set of Boolean functions [1]-[2]. Each node in the DAG represents a Boolean function F and has an associated variable x_i and pointers to two other nodes (functions) in the DAG. The node F is written as the tuple (x_i, G, H) where x_i is called the top variable of the function F, G is the positive cofactor of F with respect to x_i i.e. G = F_{xi}, and H is the negative cofactor of F with respect to x_i i.e. H = $F_{xi'}$. Shannon decomposition is carried

out in each node and node F is thus represented as $F = x_i G + x_i' H$. The constant functions 0 and 1 are represented by terminal nodes. BDDs are used for hardware verification. BDDs are also used by techniques for logic synthesis. These applications benefit from optimization of BDDs with respect to different criteria, i.e. objective functions. In this, BDD optimizations happen at a deep and abstract logic level. In a design flow, this is an early stage before the step of technology mapping. After BDD optimization, it is often possible to directly transfer the achieved optimizations to the targeted digital circuits. Given below are few examples of BDD optimization and their applications:

i.) In BDD-based functional simulation, a simulator evaluates a given function by direct use of the representing BDD. A crucial point here is the time needed to evaluate a function. Hence, BDD optimizations are desired to minimize evaluation time [3]-[4].

ii.) One way to synthesize a circuit for a given function is to directly map the representing BDD into a multiplexor circuit. It is known that optimizations of the BDD (e.g. BDD size, expected or average path length in BDDs) directly transfer to the derived circuit (e.g. area and delay minimization). With that, the targeted applications are in the field of logic synthesis.

1.2 Ordered Binary Decision Diagram (OBDD)

Ordered binary decision diagram (OBDD) was proposed by Bryant [2]. An Ordered Binary Decision Diagram (OBDD) is a pair (π,G) where π denotes the variable ordering of the OBDD and G is a finite DAG i.e. $G = (V,E)$ where, V denotes the set of vertices and E denotes the set of edges of the DAG, with exactly one root node and the following properties:

i.) A node in V is either a non-terminal node or one of the two terminal nodes in $\{\mathbf{1}, \mathbf{0}\}$.

ii.) Each non-terminal node v is labelled with a variable in X_n, denoted var(v), and has exactly two child nodes in V which are denoted then(v) and else(v).

iii.) On each path from the root node to a terminal node the variables are encountered at most once and in the same order.

Thus, OBDD [2] is a restricted decision graph in which the ordering of variables is consistent on all paths of the graph. The order selected for decision variables can significantly affect the number of nodes required in a BDD. The BDD in Fig. 1.1 (a) uses variable order a < b < c < d, while the BDD in Fig.1.1 (b) represents the same function, with variable order a < c < d < b. Both figures have same function but different number of nodes. Because the BDD is ordered, the DAG can be levelized with all nodes with a particular top variable at a given level. With this ordering restriction, sub graphs can be shared and the resulting diagram remains canonical. That is, the diagram is unique for a given logic function and for a given variable order.

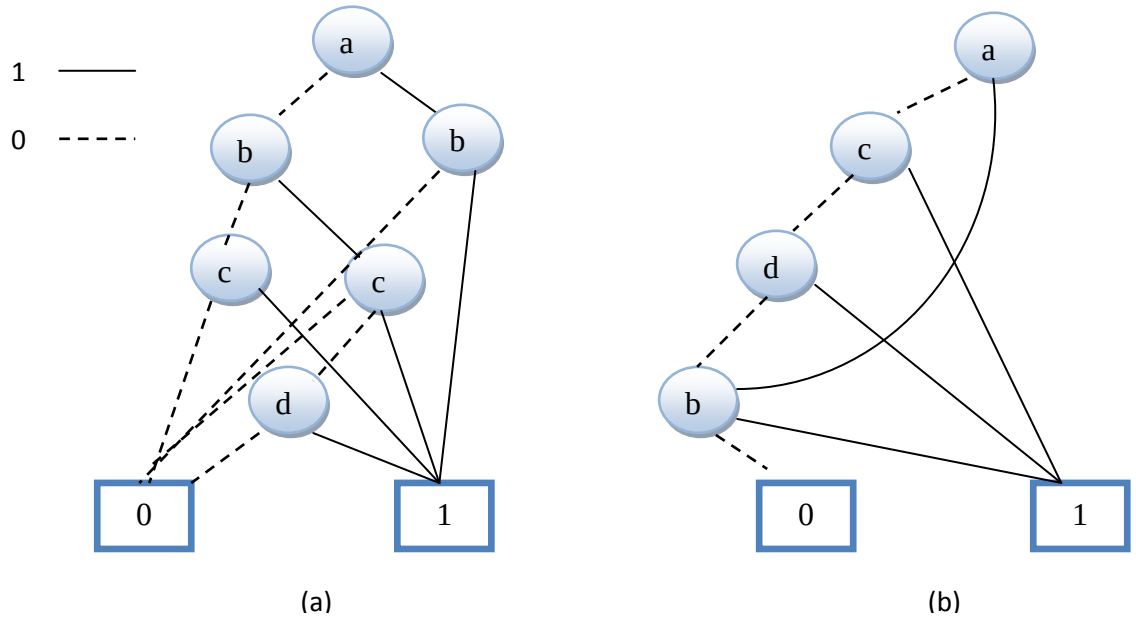

Fig. 1.1 Different OBDDs for same Boolean function F with different variable order

F = ab + a'c + bc'd

1.3 Reduced Ordered Binary Decision Diagram (ROBDD)

The ROBDD representation is defined by imposing restrictions on the BDD specification of Akers [1] such that the resulting form is canonical. A BDD is reduced if it contains no isomorphic sub graphs, and every vertex has distinct children. Reduced, ordered BDDs [2] are canonical representations of Boolean functions: for a fixed variable order, two functions are equivalent if and only if their BDDs are identical. An OBDD is converted to an ROBDD using the following reduction rules:

a. Merging equivalent nodes
b. Merging isomorphic nodes
c. Eliminating redundant tests

Redundant nodes in the graph, i.e. nodes not needed to represent a boolean function, can be eliminated. ROBDDs allow for a good trade-off between efficiency of manipulation and compactness. Compared to other techniques to represent Boolean functions, e.g. truth tables or Karnaugh maps, ROBDDs often require much less memory and faster algorithms for their manipulation do exist. Fig 1.2 shows the truth table for a boolean function and the corresponding OBDD and ROBDD.

a	b	c	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

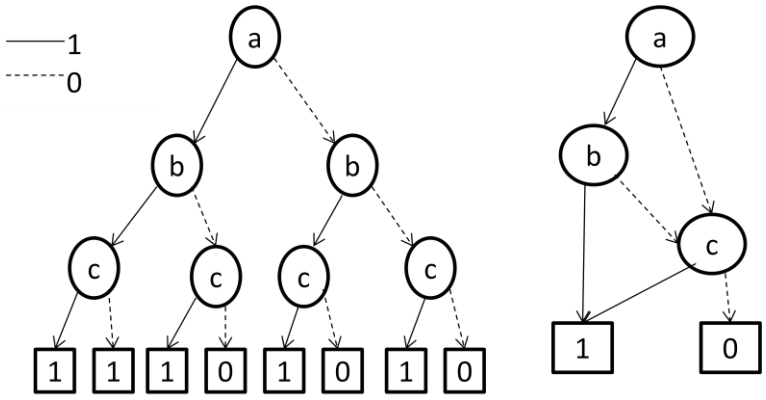

Fig. 1.2 Truth Table, Ordered BDD and corresponding ROBDD for the boolean function, f = ab+c

1.4 Applications of BDDs

Binary Decision Diagrams (BDDs) have achieved great popularity as data structures for representing Boolean functions in solving most of the combinational problems which arise in synthesis and verification of digital systems [1]-[3]. The two important properties of BDDs which form the basis for applications of BDDs in different areas are:

i.) BDDs are canonical. This property is useful when verifying the equivalence between two circuits [3]. Accordingly, two circuits are equivalent if and only if their BDDs are identical for a specific variable ordering.
ii.) BDDs are effective structures for the representation of large combinatorial sets.

1.4.1 Functional Synthesis: BDDs as MUX Circuits

Boolean functions, represented by BDDs can be realized using multiplexer based design styles such as Pass Transistor Logic (PTL). Pass Transistor Logic uses only two transistors to realize a multiplexor a wired OR of two MOS transistors. The advantages of PTL circuits as an alternative to static CMOS design are higher energy efficiency (low power), less circuit area and higher circuit speed. Each node in the corresponding BDD for a boolean function represents a 2x1 MUX. In 2x1 multiplexer when select line *s* is set to zero then output *y* is equal to *a* and when *s* is set to one then output is *b* which is shown in Fig 1.3 and is expressed logically as $y = s'a + sb$. The MUX representation is obtained by back propagation from leaf (terminal) nodes to the root nodes of the BDD, as shown in Fig 1.4. Hence, lesser node count is desirable in order to reduce the net area for realization and fabrication of a digital circuit.

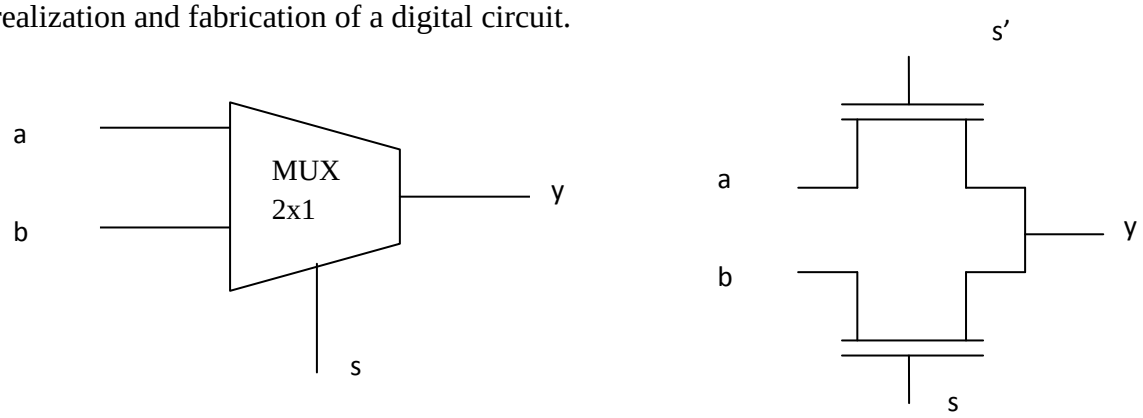

Fig 1.3 A 2x1 Multiplexer and corresponding transistor realization

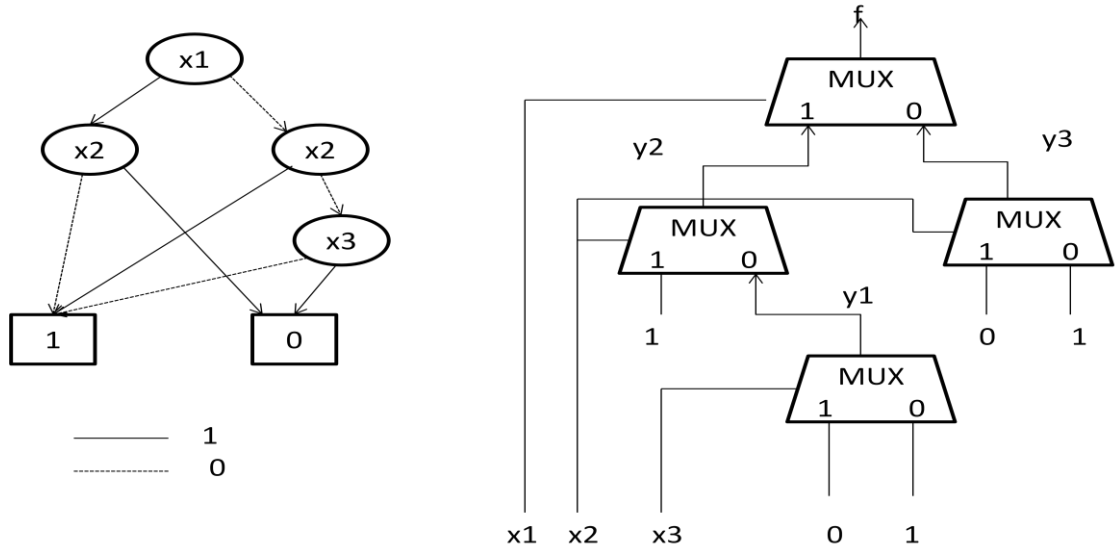

Fig 1.4 ROBDD and corresponding MUX representation for boolean function, f = x1x2'+ x1' (x2'x3'+ x2)

1.4.2 Functional Verification of Logic Circuits

Binary decision diagrams (BDDs) were originally invented for hardware verification to efficiently store a large number of states that share many commonalities [2]. The canonical property of BDDs makes them an efficient alternate approach for logic circuit comparison. The basic aim for hardware verification is to compare a new design to a *known good* design [5]. Two logic circuits are equivalent if and only if their compact representations in the form of BDDs are identical provided the same variable ordering is used in both representations as illustrated in Fig. 1.5. Fig. 1.5 (a) shows the BDD for boolean function f1 = ab+a'c+bc. On applying the reduction rules, i.e. eliminating redundant test, ROBDD for function f1 is obtained with variable order a < b < c as shown in Fig.1.5 (b). ROBDD for function f2 = ab+a'c with order a < b < c is shown in Fig.1.5 (c). The two figures being identical show that the two boolean functions f1 and f2 are identical.

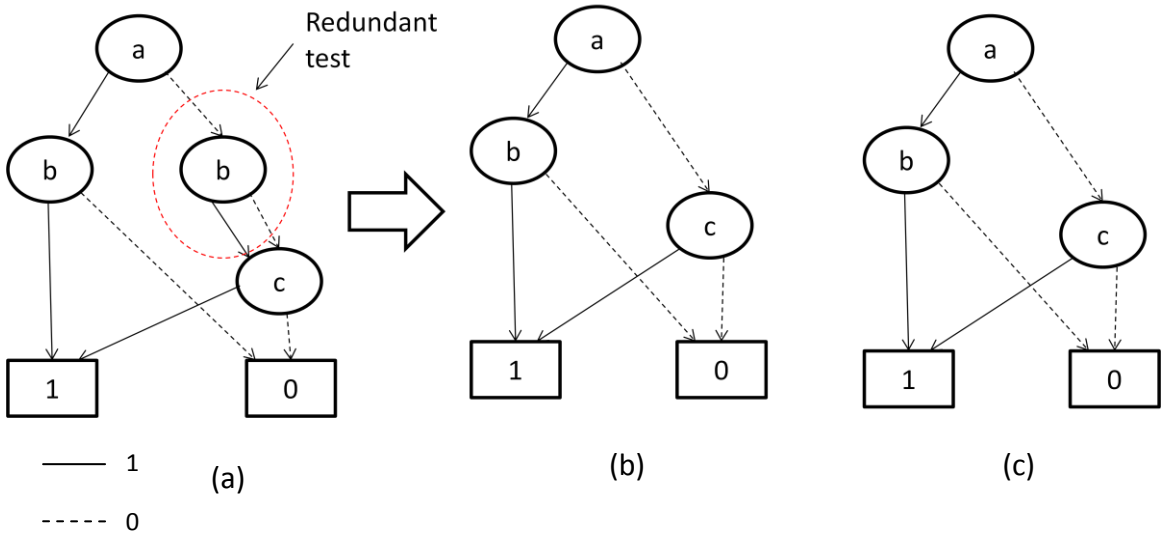

Fig. 1.5 Equivalence Verification of boolean functions f1 = ab+a'c+bc and f2 = ab+a'c

1.5 Objective of Thesis

The objectives of this thesis work are:

- To study the effect of variable ordering on BDD size corresponding to a given Boolean function.
- To explore different techniques and algorithms for reducing node count in a BDD.

- Exploration of the recently developed Evolutionary Algorithms (EAs) for altering BDD size by finding optimum variable orderings.
- Formulation of an improved hybrid Evolutionary Algorithm combining features of different algorithms with an aim to provide lesser BDD size in terms of reduced node count.
- Implementation of the modified hybrid Evolutionary Algorithm using C++ codes on a Linux platform. C++ Language allows incorporating the features of standard BDD package such as *Buddy-2.4*.
- Experimentation of the implemented algorithm with a number of Benchmark circuits taken from the LGSynth91 benchmark circuit suites.
- Analysis of the results obtained using proposed hybrid Evolutionary Algorithm.

1.6 Organization of Thesis

The thesis report is organized as follows:

- The thesis report begins with a brief introduction to Binary Decision Diagrams (BDDs) and its different types i.e. Ordered Binary Decision Diagram (OBDD) and Reduced Ordered Binary Decision Diagram (ROBDD).
- Chapter 2 provides an introduction to variable ordering problem in BDDs, different approaches and techniques developed so far for BDD optimization by reducing node count and comparison among these approaches.
- Chapter 3 provides an overview of different Evolutionary Algorithms which are being seen as a potential solution to variable ordering problem in BDDs.
- Chapter 4 provides a detailed description of the Modified Memetic Algorithm (MMA) that has been proposed in this thesis work, its important features, advantages flow chart of the proposed algorithm and its implementation for BDD optimization using standard BDD package.
- In Chapter 5 experimental results for different Benchmark circuits are given that demonstrates the efficiency of the proposed method by comparing it with other existing variable ordering methods.
- Chapter 6 gives the conclusion and future scope of the work proposed in this thesis.

2. Variable Reordering Algorithms

2.1 Variable Reordering Problem in BDDs

The size of BDDs and, with that, memory requirement and runtime of algorithms using BDDs often critically depends on the order chosen for the input variables, varying it from linear to exponential [6]. Various heuristics tend to find near optimal variable ordering for BDDs, as finding the best variable ordering is NP-hard. The BDD variable ordering algorithms can be broadly classified as *static variable ordering*, *dynamic variable ordering* and *evolutionary algorithms*.

Static ordering [6]-[7] is the node order determined before generating BDDs, and they are only adopted to get the initial order of the input variables. Motivation for these heuristics comes from the observation that BDDs tend to be smaller when related variables (topologically closer variables/inputs) are close together in the order. These variables determine sub circuit's outputs and should usually be close together in the order. Once determined, the order is maintained throughout all the subsequent processing. On the other hand, variables are reordered periodically in BDD package using Dynamic ordering heuristics [8]-[10] in order to reduce the number of nodes. In dynamic ordering, variable order is automatically changed by the BDD package, transparent to the user. The package determines the appropriate points at which to stop the processing, reorder the variables, and then resume the processing. Also, various evolutionary algorithms [11]-[12], [14], [17] and their hybrid versions [13], [15]-[16] have been proposed and implemented to achieve almost best results. Whereas most of these algorithms have been implemented in the standard BDD packages such as Buddy, CUDD etc., the algorithm proposed in this thesis has not yet been implemented in any such package, to the best of our knowledge.

2.2 Static Variable Ordering Techniques:

i.) **Depth First Traversal Algorithm [6]:** It is based on a depth-first traversal through a circuit from the output to inputs. It includes drawing a circuit diagram from a netlist description, and then using the order appearing in the diagram. This

algorithm is based on the observation that the inputs whose connections are topologically close to each other in the circuit should be near in the variable order. In the case of tree-structured circuits, a resulting variable order is one of the best orders. However, we cannot obtain good variable orders in some circuits, since most circuits are not tree structured. This algorithm has been developed for single output circuits. For multiple output circuits, the ordering algorithm is applied by considering a dummy output to which each of the primary outputs is connected. That is, the circuit traversal from each output is done in the order of priority of the outputs. The order of the output is determined such that the output which seems to have more complex logic has higher priority. This algorithm has the disadvantage that the variable order generated for the highest priority output node is used for other output nodes, even if it doesn't give the best result for them. Also, this algorithm cannot be used for larger circuits.

ii.) **Variable Interleaving [7]:** These algorithms were developed by Fujii et al. for multiple output circuits. While conventional algorithms use variable appending, the new algorithms use variable interleaving and are based on circuit topology. The principle of the algorithm is to merge a variable order for an output into a variable order for outputs with higher priority maintaining the order for the output as much as possible. In the conventional algorithms, a newly ordered variable is always appended to the end of the ordered variable set, while in the new algorithm, a newly ordered variable is inserted into an appropriate position of the ordered variable set. We call these variable merging methods variable appending and variable interleaving, respectively. The order of outputs is determined in the depth first traversal way. The conventional algorithm is different from the new algorithms only in the variable merging method. These algorithms have been found to be much more effective than conventional algorithms and apply to wider classes of circuits.

2.3 Dynamic Variable Ordering Techniques:

The idea behind dynamic variable reordering techniques is to have the OBDD package determine and maintain the variable order of the OBDD. This variable order is changed automatically by the OBDD package, transparently to the user, as operations are

performed [8]. Because the variable order within the OBDD is no longer static, this technique is referred to as dynamic variable ordering. When using dynamic variable ordering, a total order is defined for all variables before and after each package operation; however, the order is periodically adjusted by the OBDD package, as a consequence of an operation, to find a better order. Logically, the variable order changes in-between the package operations. Thus, it maintains all advantages provided by the ordered BDD data structure, such as canonicity and efficient recursive algorithms. Different algorithms implementing this technique are:

i.) **Window Permutation Technique [8]-[9]:** This technique minimizes the size of an OBDD using adjacent variable exchange. The window permutation algorithm proceeds by choosing a level i in the DAG and exhaustively searching all k! permutations of the k adjacent variables starting at level i. This is done using k! - 1 pair wise exchanges followed by up to k (k - 1)/2 pair wise exchanges to restore the best permutation seen. This is then repeated starting from each level until no improvement in the DAG size is seen. A level is marked after the permutation at the level is known optimal. Because the swap of two adjacent variables is efficient, the window permutation algorithm remains practical for values of *k* as large as 4 or 5**.** The window permutation algorithm is limited in its ability to find good variable orders. A limitation of the window permutation algorithm appears to be that several moves can be required to move a variable a long distance and these moves can be blocked by an intermediate up-hill move. Hence, it works well for less number of variables only. The complexity increases with the increase in the number of inputs.

ii.) **Sifting Algorithm [8]:** This algorithm is based on finding the optimum position for a variable, assuming all other variables remain fixed. If there are *n* variables in the DAG (excluding the constant level which is always at the bottom), then there are *n* potential positions for a variable, including its current position. Among these *n* positions, the sub-goal employed by the shifting algorithm is to find the spot which minimizes the size of the DAG. The shift algorithm has the advantage that a variable can move a long distance in the ordering. Note that the DAG-size can increase significantly after the first few variables swaps, and then eventually reduce below the starting point. This allows a type of uphill move to

be taken - the acceptance of the entire sequence of pair-wise swaps is based on the best position seen regardless of any increase in the intermediate DAG size.

iii.) **Linear Sifting Algorithm [10]:** It combines the efficiency of shifting and the power of linear transformations. Linear transformations replace one variable, x_i, with a linear combination of variables. A linear combination is obtained by taking the exclusive or of the arguments or its complement. Shifting is a local search algorithm that iteratively improves the variable order by a series of swaps of adjacent variables. Each variable is considered in turn and is moved up and down in the order so as to take all positions successively. The variable is then returned to one position where the minimum size of the BDD was recorded. The process then continues with another variable. The effectiveness of shifting stems from its ability to move a variable to any position in the order in a short time. Its time efficiency relies on the ability to quickly swap adjacent variables. It is indeed possible to perform such a swap by accessing only the nodes labelled by the two variables being exchanged. Each variable is considered in turn, and, as in sifting, it is moved up and down in the order. Let x_i, be the chosen variable, and let x_j, be the variable immediately following it in the order. One basic step of linear sifting consists of the following three phases:

a. Variables x, and x_j are swapped; let the size of the BDD after the swap be s1.
b. The linear transformation $x_j \rightarrow x_i, \equiv x_{j,}$ is applied; let the resulting size of the BDD be *s2*.
c. If s1 <= *s2* then the linear transformation is undone. This is obtained by simply applying the transformation again, since it is its own inverse.

The net effect of the three-phase procedure is that x_i is moved one position onward in the order, and possibly linearly combined with x_j. Conversely, if x_j is the variable being moved, it is first swapped and then x_i is combined with it. Linear sifting is slower than normal sifting in most cases, because the cost of a linear transformation is comparable to the cost of a variable swap. If we assume that the graphs manipulated by the two algorithms are approximately of the same size, then linear sifting will be approximately three times slower than standard sifting.

2.4 Evolutionary Algorithms used for Variable Ordering in BDDs:

i.) **Scatter Search Algorithm [11]:** Scatter search offers a reasonable compromise between quality (BDD reduction) and time. On smaller benchmarks it delivers almost optimal BDD size with less time than the exact algorithm. For larger benchmarks it delivers smaller BDD sizes than genetic algorithm or simulated annealing at the expense of longer runtime. Scatter search was introduced by Glover in 1977. Scatter search is very aggressive and attempts to find high quality solutions fast. The algorithm operates on a set of solutions, the reference set, by combining these solutions to create new ones. The main mechanism for combining solutions is such that a new solution is created from the linear combination of two other solutions. Unlike a "population" in genetic algorithms, the reference set of solutions in scatter search tends to be small. It chooses two or more elements of the reference set in a *systematic* way with the purpose of creating new solutions. The genetic algorithms and simulated annealing also result in fairly small BDD sizes. They are both population-based optimization that bear some similarity with scatter search. In some cases, the genetic algorithm can obtain BDD sizes almost as good as scatter search. In some other cases, simulated annealing can obtain BDD sizes close to that of scatter search. The large runtime for scatter search is the trade off between quality and speed. For CPU run time, a bigger diversification set actually does not increase the search time too much, but increasing the size of the reference set makes it harder to converge, resulting in a longer execution time.

ii.) **Genetic Algorithm Based Technique [12]-[14]:** This is an excellent multi-objective optimization technique. Genetic Algorithms (GA) are stochastic optimization based on principle of natural selection and natural genetics. They start with an initial population (solution space) consisting of a set of randomly generated solutions. Based on some reproductive plan especially, the crossover and mutation, they are allowed to evolve over a number of generations. After each generation, the chromosomes are evaluated based on some fitness criteria. Depending upon the selection policy and fitness value, the set of chromosomes for next generation are selected. Finally, the algorithm

terminates when there is no improvement in solution over a fixed number of generations. The best solution at that generation is accepted as the solution produced by GA. The formulation of Genetic Algorithm for any problem involves the careful and efficient encoding of the solutions to form chromosomes, crossover and mutation operators and a cost function measuring the fitness of the chromosomes in a population. Genetic Algorithm based technique for BDD optimization was first proposed by Drechsler in 1996 [12]. After that various modified techniques based on this algorithm have been proposed and implemented [13], [15] in order to further reduce node count as well as computation time for BDD optimization.

iii.) **Branch and Bound Algorithm [14]:** Branch and Bound (BB) based greedy algorithmic approach is an excellent optimization technique for multi-objective problems and has a finite but usually very large number of feasible solutions. A BB algorithm searches the complete space of solutions (exact method) for a given problem for optimum solution. This is done by an iteration process which has three main components: selection of the *solution set* for *bound* calculation, and *branching*. Branching, with its recursive application, defines a tree structure (the *search tree*) whose *nodes* are obtained from the previous level by a *splitting* procedure *i.e.* subdivision of the solution space of the nodes into two or more subspaces to be investigated in a subsequent iteration. The next step of the BB technique is a procedure that computes only the lower bounds of the *solution set* by calculating the bounds for each of the solutions, within the given *solution set*. This step is called bounding. The lower bounds are calculated by setting the objective function of the proposed problem based on the fitness. The key idea of the BB algorithm is: if the *lower* bound for some tree nodes (set of candidates) A is greater than the lower bound for some other node B in the same level, then A may be safely discarded from the search [14]. This step is called *pruning*, and is usually implemented by maintaining a global variable *m* (shared among all nodes of the tree) that records the minimum lower bound seen among all solutions examined so far. Any node whose lower bound is greater than *m* can be discarded. Otherwise, the bounding function for the subspace is calculated and compared with the current best solution.

iv.) **Genetic Tabu Hybrid Strategy [16]:** This technique combines the global space search of genetic algorithm with the local neighbourhood search and the gradual global optimizing of tabu search. This algorithm tries finding an optimum solution in the neighbourhood of the current solution while avoiding the problem of sub-optimal solution trap. Its basic principle is to pursue LS whenever it encounters a local optimum. The algorithm is partitioned into two sections. Initially, a guess set of N feasible candidate solutions each representing a variable order, constructing a population is chosen randomly in the search space. Each parent is allowed to generate more than one child to form a family. The children belonging to same family become the objects being selected in the first level where TS i.e. Tabu Search is used to choose the optimal child as parent for the next generation. The second-level selection performs competition among different families providing a measure of the fitness of every family and basic genetic operations are applied. Average fitness of every family is calculated and the number of children NoC allocated to each family for the next generation is made proportional to their fitness values and is given as

$$NoC = (No\ x\ NoCo\ x\ F)\ /\ S$$

Where, N_o is the initial number of families,

NoC_o is the initial number of children produced in each family,

F is the fitness of a family,

S is the sum of all families' fitness.

The total number of children in next generation is kept constant. Thus fitter individuals get better survival chance and eventually only one optimal family survive. The average value of object functions belong to each family is designated as the fitness F of each family. The efficiency of this algorithm is completely dependent on the neighbourhood structure and initial solution set. Also, the convergence speed decreases due to swapping operations on neighbourhood solutions.

v.) **Particle Swarm Optimization Technique [17]:** Particle Swarm Optimization (PSO) is a population-based stochastic technique inspired by social behaviour of bird flocking or fish schooling. In PSO, the potential solutions, called 'particles' fly through the problem space by following the current optimum particles. The algorithm requires a number of particles forming a population of potential solutions that can explore the entire range of available positions, learn from their past experiences, learn from others experience and finally converge near the solution, which may be the best or a suitably good solution after satisfying a definite termination criterion. The particles sense their proximity to a good solution using a parameter known as the fitness function. Each particle has a fitness value, which is evaluated by the fitness function (the cost function to be optimised), and has a velocity that directs its flight. During every generation (iteration), each particle is updated depending upon its initial position, the position of global best solution and its velocity. Significant reductions in node count as well as delay have been observed using this algorithm as compared to other evolutionary approaches, such as GA. The only limitation is the timing overhead as the number of variables increase.

2.5 Comparison among Different Variable Ordering Techniques:

Table 2.1 Summary of Variable Ordering Algorithms

Algorithm	**Static/ Dynamic**	**Basic Principle**	**Applicability**	**Complexity**
Depth First Traversal	Static	Variables close in circuit topology are kept close in order.	In Smaller circuits only	Increases with increase in variables.
Variable Interleaving	Static	For multi-output circuits, variables for different outputs are interleaved instead of appending.	For large multi-output circuits. Can be extended for single output	Increases with increase in variables.

Algorithm	**Static/ Dynamic**	**Basic Principle**	**Applicability**	**Complexity**
Window Permutation	Dynamic	Variable exchange at all levels with the adjacent variables.	In smaller circuits.	Increases with increase in inputs.
Sifting Algorithm	Dynamic	Finding optimum position for a variable, keeping others fixed, and setting it at the position which reduces size of BDD.	In small and large circuits. Efficient than Window Permutation.	Linearly increase with inputs.
Linear Shifting	Dynamic	Combines shifting and linear transformation of variable.	Used for all circuits. But comparatively slower than Sifting Algorithm.	Increases with inputs.
Scatter Search	Evolutionary, Dynamic	Combines different variable orders from a given reference population to create improved order that reduces BDD size. Iterational process.	Highly efficient for smaller circuits. Takes longer time as the circuit size increases.	Increases with inputs.
Branch and Bound	Evolutionary, Dynamic	Dividing solution space into subsets and finding lower bounds i.e. fitness value of each solution in the solution set. The set with high bound value is discarded. Iterational process.	Efficient for all circuits. But provide number of solutions for a single problem.	Linear increase with inputs.

Algorithm	**Static/ Dynamic**	**Basic Principle**	**Applicability**	**Complexity**
Genetic Algorithm based	Evolutionary, Dynamic	Imitates the Darwin theory of evolution. Selects solutions, find fitness and generate best result after applying genetic operators i.e. crossover, mutation. Iterational process.	Works for all circuits.	Linear increase with inputs.
Genetic Tabu search based	Evolutionary, Dynamic	Combines global space search of GA with local search of Tabu search. Tries to find optimum solution in the neighbourhood of the current solution. Iterational Process.	Works for all circuits.	Depends on initial solution set and number of inputs.
Particle Swarm Optimization	Evolutionary, Dynamic	Stochastic technique inspired by social behaviour of bird flocking with solutions trying to follow the global best or optimum solution with every iteration by updating their current values	Works for all circuits.	Time complexity increase with inputs.

3. Evolutionary Algorithms

Evolutionary Algorithms EAs are population-based metaheuristic optimization algorithms. These are stochastic search methods that mimic natural biological evolution and/or the social behaviour of species. Such algorithms have been developed to arrive at near-optimum solutions to complex and large-scale optimization problems which cannot be solved by gradient-based mathematical programming techniques [18]. Every EA has three main components:

i.) **Population** of individuals representing candidate solutions to the underlying optimization problem.

ii.) **Fitness function** or objective function that determines the environment within which the solutions live and which determines the strength of an individual as an optimum solution to the given problem.

iii.) **Evolutionary operators** leading to the evolution of population of individuals that are better suited to their environment than the individuals that they were created from.

Thus, Evolutionary Algorithms work on populations of individuals instead of single solutions. In this way the search is performed in a parallel manner resulting in the faster convergence of these algorithms. A detailed description of some of the widely used EAs is given as under.

3.1 Simple Genetic Algorithm (SGA)

Genetic Algorithms GA [19] are a family of computational models inspired by evolution. These algorithms encode a potential solution to a specific problem on a simple chromosome-like data structure and apply recombination operators to these structures so as to preserve critical information. Commencing with a random population of solutions (chromosomes), these algorithms evaluate the fitness of each chromosome against a problem specific objective function to determine the best and worst solutions. Genetic algorithm is a search method that has broad practicability on the basis of natural selection and population genetics.

To emulate the Darwin theory of survival of the fittest, best chromosomes of the present generation exchange information through genetic operators such as crossover, inversion or/and mutation to produce offspring chromosomes. These offspring chromosomes are evaluated and allowed to reproduce if they are better than the previous generation. Hence, worst results are eliminated and best results are allowed to evolve with iterations. This process is continued for a large number of iterations until a best-fit (near-optimum) solution is obtained.

Main parameters affecting the performance of GAs are population size, number of generations (iterations), crossover rate and mutation rate. Larger population size (number of chromosomes) and large number of generations increase the likelihood of obtaining a global best solution, but increase the computation time as well. Crossover between two parents to produce an offspring is a natural evolution process; hence its rate normally ranges between 0.6 and 1.0. However, mutation being a rare process that resembles a sudden change in the offspring, its rate is kept low.

```
/*Algorithm GA */
Formulate initial population
Randomly initialize population
Repeat
    Evaluate objective function
    Find fitness function
    Apply genetic operators
        Reproduction
        Crossover
        Mutation
Until stopping criteria
```

Fig 3.1 Basic Genetic Algorithm [19]

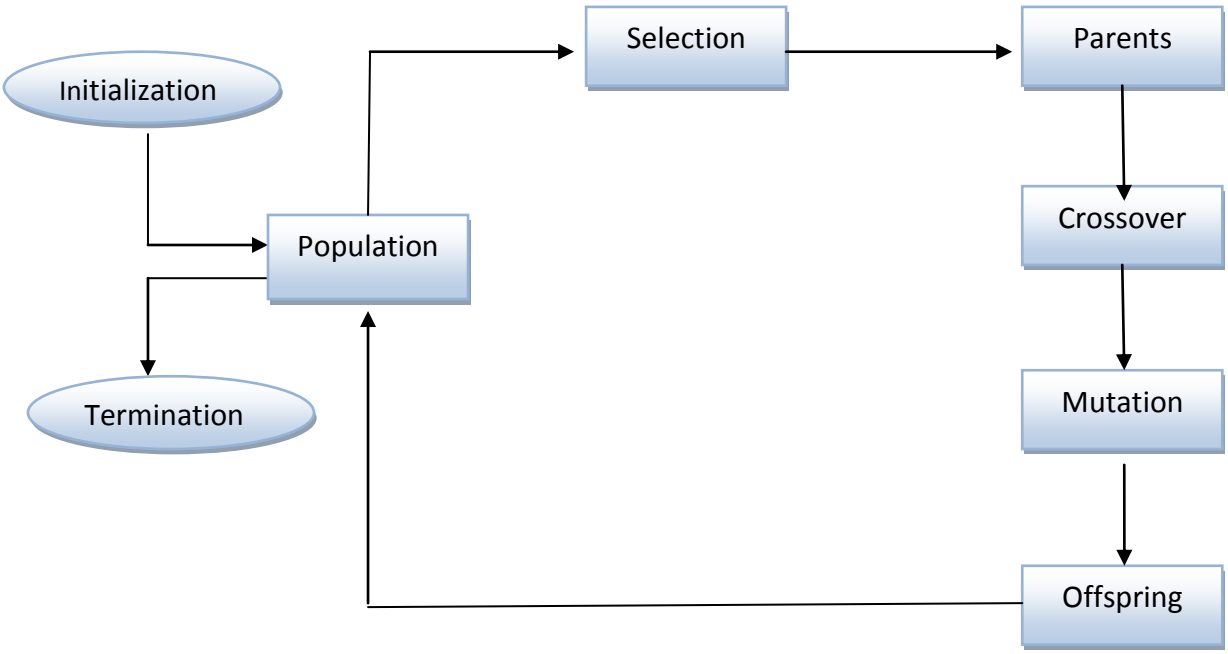

Fig. 3.2 Operation Flow of Genetic Algorithm

3.2 Hybrid Genetic Algorithm (HGA)

This algorithm is an improved version of the basic Genetic Algorithm. Whereas in a basic GA, the important parameters such as remaining population size, crossover rate and mutation rate are fixed, in this hybrid version the parameters influencing the algorithm are allowed to change dynamically depending upon the fitness of the next generation of population [13]. Also, the basic genetic crossover operation is modified so that invalid solutions are avoided [13]. In this algorithm, mutation rate, remaining population size and remaining number of generations are assigned values depending upon the improvement found in the individuals. A complete flow chart of this algorithm is as shown below in Fig 3.3.

Main parameters affecting the performance of GAs are population size, number of generations (iterations), crossover rate and mutation rate along with constants to decrease or increase remaining generations, population size and mutation rate depending upon the quality and results of next generation of population.

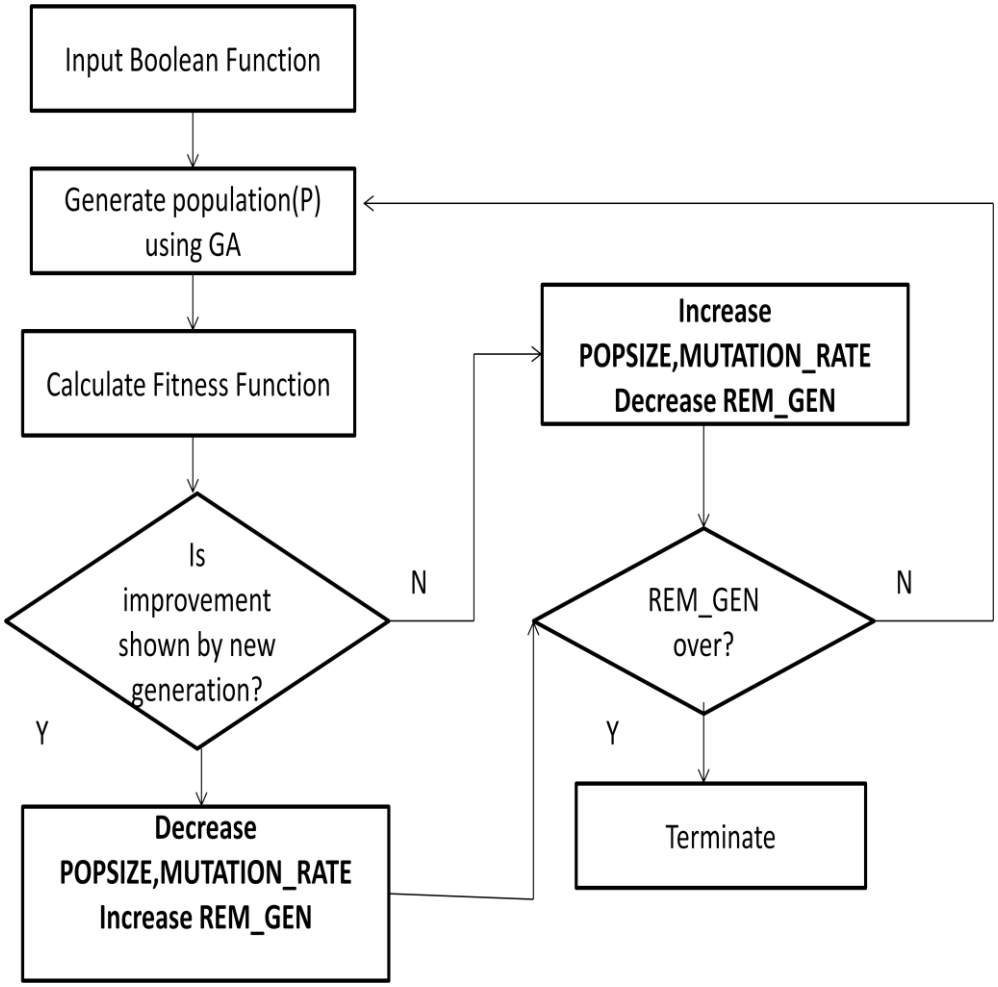

Fig. 3.3 Flow Chart for Hybrid Genetic Algorithm

3.3 Particle Swarm Optimization Algorithm (PSO)

Particle swarm optimization algorithm (PSO) was first proposed by Dr. Kennedy and Dr. Eberhart in 1995 [21]. It is a new intelligent optimization algorithm developed in recent years, which simulates the migration and aggregation of bird flock when they seek for food. [21] PSO is a computational method that optimizes a problem by iteratively trying to improve a candidate solution to a given problem with regard to a given measure of quality or fitness. PSO optimizes a problem by having a population of candidate solutions, called particles, and moving these particles around in the search-space according to simple mathematical formulae over the particle's current position and velocity without using any complicated evolution operation. Each particle's movement is influenced by its local best known position and is also guided towards the

global best known position in the search-space, which is updated every time better positions are found by other particles. Thus, throughout the process, three important values are monitored for each particle: particle's current position *cBest*, its best known position in previous iterations *pBest* and its current velocity *V*.

With iterations, position of the global best particle *gBest* is calculated and is taken as the best fitness. Accordingly, remaining particles update their velocities to catch up with the best particle. Based on the current velocity, previous best position and global best position, each particle's position is updated according to the following equations:

$$\text{New } V = \omega \text{ X current } V + \text{c1 X } rand(\,) \text{ X } (pBest - cBest) + \text{c2 X } Rand(\,) \text{ X } (gBest - cBest)$$

$$\text{New position } cBest = \text{current position } cBest + \text{New } V;$$

$$V_{max} \geq V \geq -V_{min,}$$

where, ω is an inertia weight employed for improvement to control the impact of previous velocities on current velocity, c_1 and c_2 are two positive constants called learning constants, *rand( )* and *Rand( )* are two random functions in the range [0,1], V_{max} is the upper velocity limit and V_{min} is the lower velocity limit. [18] This eventually moves the swarm towards the best solution. Fig. 3.4 explains the working of this algorithm through a flow chart.

The main parameters affecting the performance of PSO algorithm are population size i.e. number of particles, number of generation cycles (iterations), the maximum change in velocity of a particle and inertia constant ω [18].

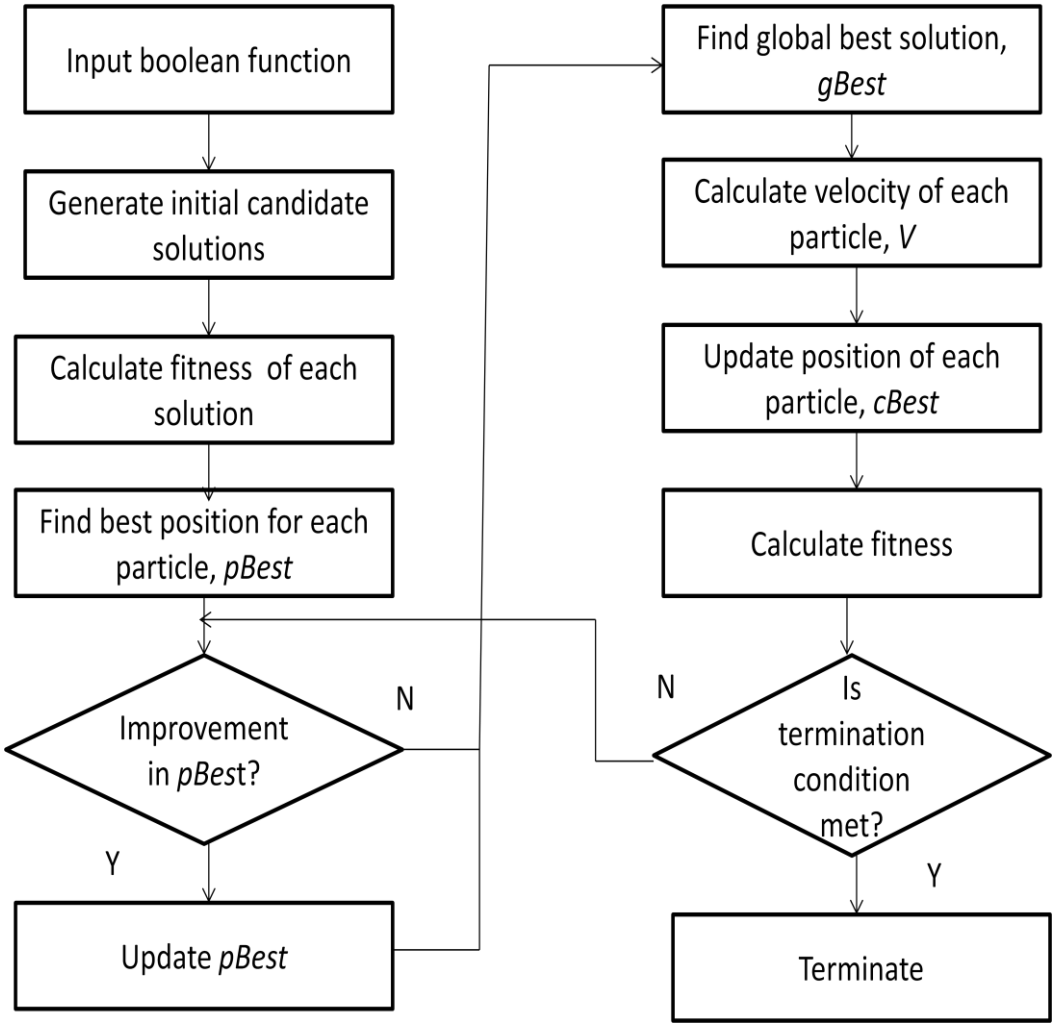

Fig. 3.4 Flow Chart for Particle Swarm Optimization algorithm

3.4 Basic Memetic Algorithm (MA)

MAs are inspired by Dawkins' notion of a meme [22]. These are based on "Universal Darwinism", a theory coined by Richard Dawkins in 1983. These are population-based optimization algorithms in which the solutions are subjected to processes of competition and mutual cooperation in a way that resembles the behavioural patterns of living beings from the same species. MAs are similar to GAs, but the elements forming a chromosome are called memes, and not genes. Meme is the basic unit of cultural transmission or imitation passed on by non-genetic operators.

The unique aspect of MA is that all chromosomes in a generation are allowed to gain some experience, through a local search (LS) [23], before being involved in the

evolutionary process. On a randomly created initial population, a local search is performed on each chromosome in order to improve its experience and thus obtain a population of local optimum solutions. The population then undergoes basic genetic operations to produce new generation, which is again subjected to local search. Various approaches have been proposed [20] to perform local search that include pair-wise swap operations among the memes of a chromosome as shown in Fig. 3.5 or adding a constant number to each meme. The performance of each chromosome is evaluated again after performing LS. The change is retained if the performance improves else it is reversed. Main parameters influencing the performance of MA are population size, number of generations, crossover rate and mutation rate along with local-search mechanism. Fig. 3.6 provides a description of this algorithm in the form of flow chart.

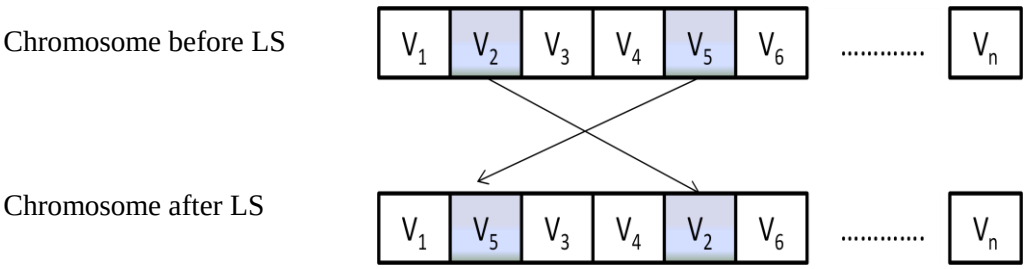

Fig. 3.5 Local Search LS using pair-wise swapping

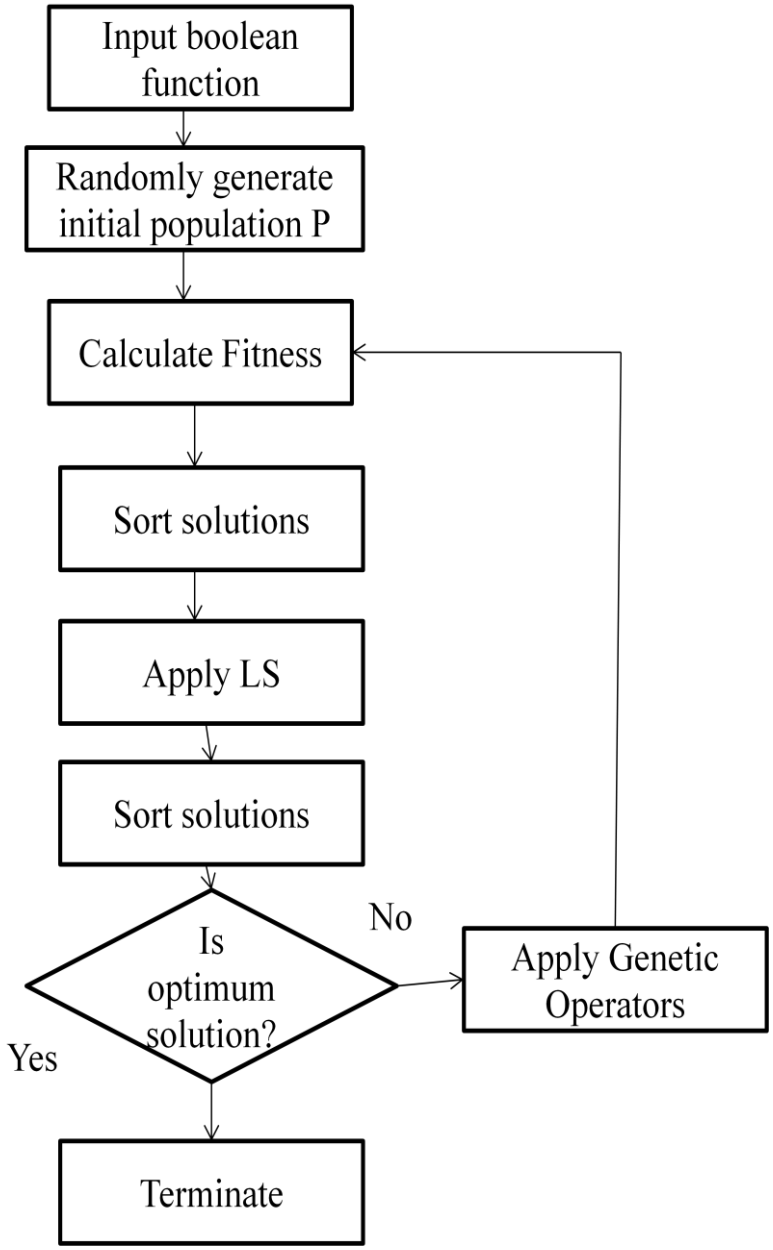

Fig. 3.6 Flow Chart for Basic Memetic Algorithm

4. Modified Memetic Algorithm for BDD Optimization

In this chapter, Modified Memetic Algorithm based approach that has been used to identify a good variable ordering in a shared BDD (SDD) is presented. This chapter explains the functioning of basic mechanisms that are used to implement this algorithm. The algorithm incorporates global search exploration feature of GA and local search exploitation of MA with various hybrid operations in a way that provides momentum to the search operation and at the same time prevents the solution to land in local optimum situation.

4.1 Solution or Meme Representation

The operation of MMA begins with a population (P) of randomly generated chromosomes representing potential solutions to the variable ordering problem in BDDs. A chromosome, representing a Boolean function of *n* variables, itself is a set of *n* memes (*i*1, *i*2, *i*3, *i*4, *i*5,…… *i*n), each represented by an integer number between 1 to *n* without repetition. For example, let us consider a combinational logic circuit consisting of 7 variables (inputs), then according to above, the chromosome can be represented by any of these forms:

3	4	6	1	7	5	2
011	100	110	001	111	101	010

Fig 4.1 Solution or Meme Encoding in Chromosome Form

This chromosome identifies an ordering of the input variables to be used in constructing the shared BDD for the multi-output function. The input variables are ordered according to the numbers appearing in the chromosome and the corresponding SDD is generated.

4.2 Fitness Function Calculation

Fitness function is calculated to determine the quality of each solution. Fitness of each individual is evaluated in terms of the number of nodes in the corresponding SDD that represents the fitness function for the problem, as determined by the standard BDD package *Buddy-2.4.*

$$Fitness\ Function, F\ =\ Number\ of\ nodes\ in\ SDD$$

4.3 Local Search Operation for Meme Improvement

All solutions are sorted in ascending order and the best and the worst *S* solutions among them undergo the local-search (LS) mechanism [18]. The LS mechanism used for optimizing the best *S* solutions performs pair-wise swap of memes in a chromosome, as depicted in Fig. 4.2. , until either a better local optimum result is obtained or for maximum

$$M = n\,X\,(n\text{-}1)\,/\,2$$

where, n is the number of input variables.

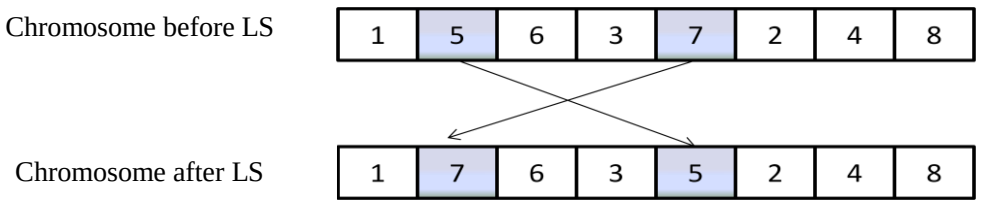

Fig. 4.2 Local Search LS mechanism for improving the performance of best *s* solutions

The LS mechanism used for optimizing the worst *S* solutions is different. These solutions are made to mimic the global best solution. Any worst solution *w* is obtained by randomly selecting a subsequence from the best solution to replace the corresponding position in worst *wth* solution, while keeping the other positions unchanged. However, if the constraint violation occurs, then the memes are randomly relocated to form a new feasible solution as illustrated in Fig. 4.3. The new solution is retained if the fitness improves, else it is reversed.

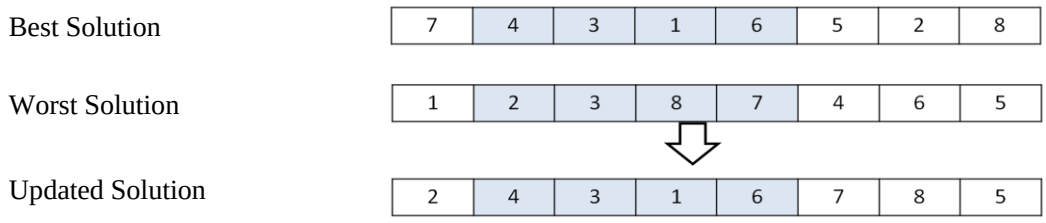

Fig. 4.3 Local Search LS mechanism for improving the performance of worst *s* solutions

4.4 Genetic Operators Used

The new local optimum solutions obtained after LS operation are again sorted and allowed to evolve by crossover and mutation operations. This process continues until an optimum variable order is obtained. The GA technique incorporated in this algorithm is hybrid GA as proposed in [13] in which important parameters such as population size, number of generations, crossover rate, mutation rate are not fixed, rather, these are made dynamic and change depending upon the improvement found in every generation.

4.4.1 Crossover Technique for generating new population

A modified alternating crossover mechanism is adopted to generate next generation as shown in Fig. 4.4.

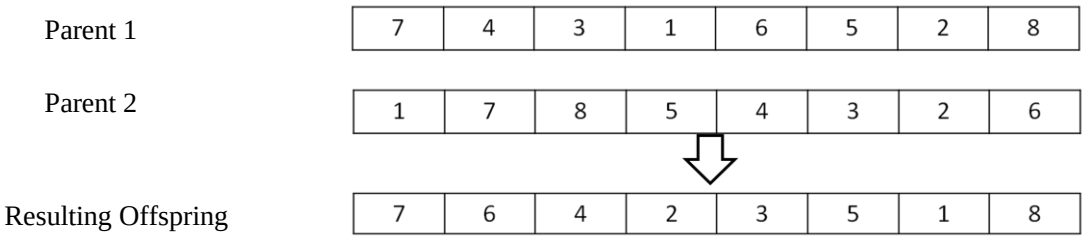

Fig. 4.4 Modified Alternating Crossover Mechanism

This crossover mechanism enhances the performance of the algorithm as compared to conventional crossover schemes. A new solution is generated from two randomly selected parents in such a way that memes are copied alternately from these parents. This technique is a hybrid version of the alternating crossover mechanism as presented in [24], the difference being that memes are copied from opposite ends of both the parents without allowing the constraints violation.

4.4.2 Mutation Operation

This operation is used to introduce variety in solution. In other words, mutation is defined as a process of randomly disturbing genetic information. This is done to prevent all solutions in a population to fall into a local optimum of solved problem. Mutation operation depends upon the type of encoding scheme used to represent the chromosomes. For example, for binary encoding a few randomly chosen bits are switched from 1 to 0 or from 0 to 1. Since integers are used to represent memes in this algorithm, the mutation operation incorporated is similar in a way that memes are complimented and replaced with their corresponding integer compliments as shown in Fig. 4.5. Also other memes are modified so as not to violate constraints.

$$\text{Compliment of integer } n \;=\; \text{ELEMENTS} - n \;+ 1$$

where, ELEMENTS represent the total number of input variables.

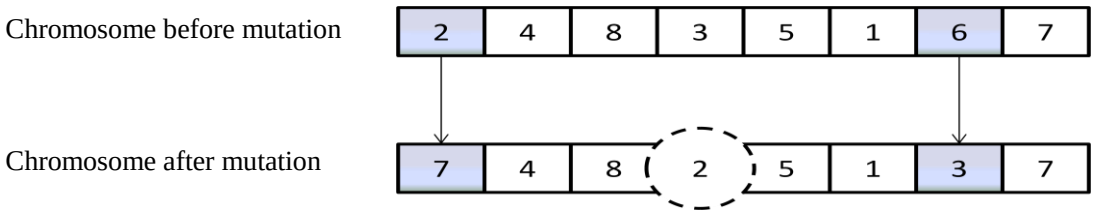

Fig. 4.5 Mutation Operation incorporated in MMA

4.5 Parameter Settings and Flow Chart

This algorithm takes a population size of 20 chromosomes per generation with 80% crossover rate and 4% mutation rate. The generation size begins with 20 generations and constants δ_1, δ_2, γ_1, and γ_2 as given in [13] have been fixed at 10%. The constant μ for change in mutation rate is set as 0.05. The constants MAX_POP and MAX_GEN are kept to be 50 and the constant MIN_POP is fixed at 5. For Simple Genetic Algorithm SGA, population size and number of iterations is kept fixed at 50, other parameters being same.

A complete flowchart of the proposed algorithm is presented below in Fig. 4.6.

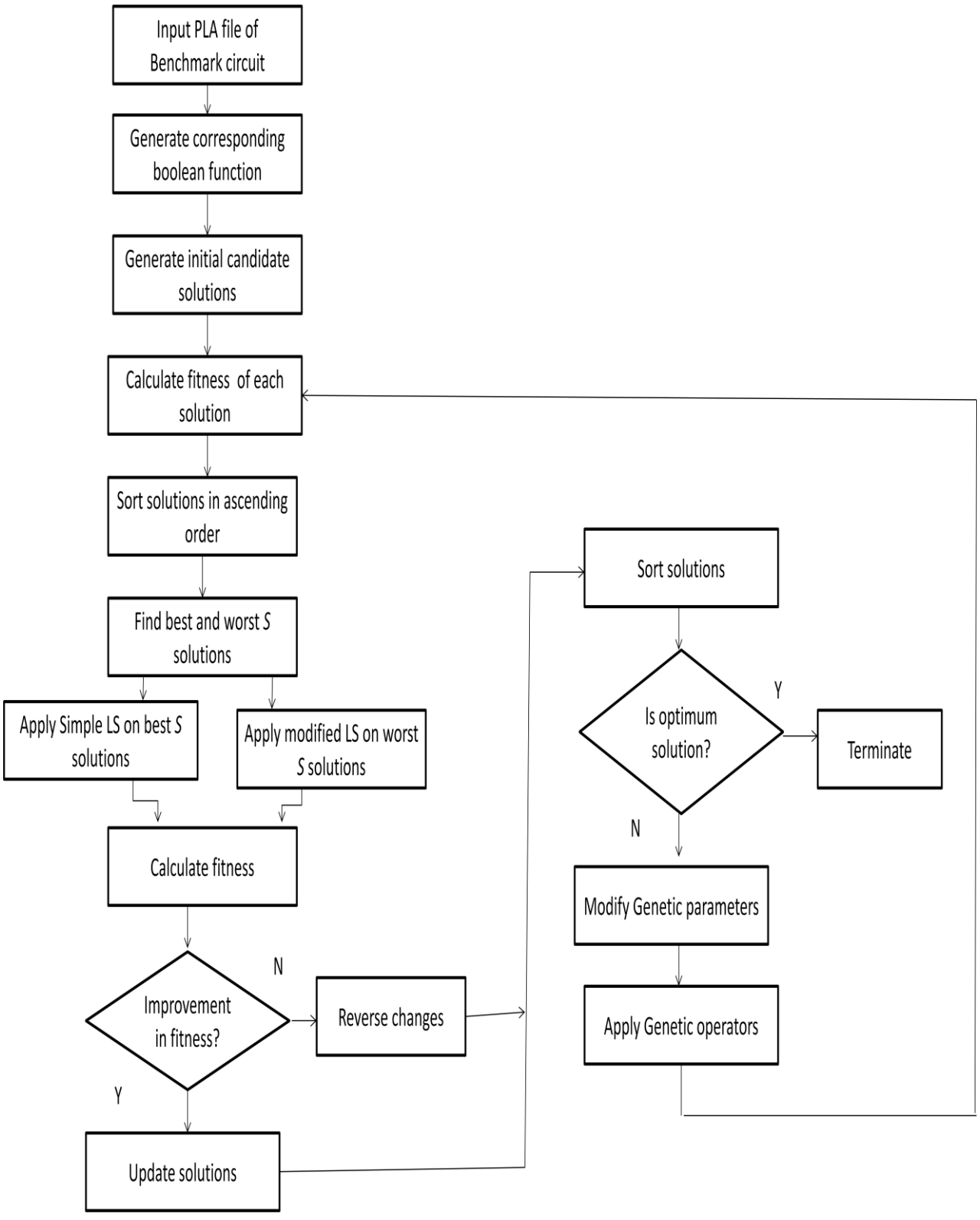

Fig. 4.6 Flow Chart for proposed Modified Memetic Algorithm

4.6 Overview of BDD Package

A BDD package has three main components: the BDD algorithm component, the dynamic variable ordering component and the garbage collection component. The common features of these components are described as under:

4.6.1 BDD Algorithm

This component computes the resultant BDDs for different Boolean functions. The implementation of these algorithms is typically based on depth-first traversal. The unique tables are hash tables with the hash collisions resolved by chaining. A separate unique table is associated with each variable to facilitate the dynamic-variable-reordering process. The computed cache is a hash-based direct mapped (1-way associative) cache. BDD nodes support complement edges where for each edge, an extra bit is used to indicate whether or not the target function should be complemented. The advantage of this encoding is that a function and its complement can be represented by the same BDD and use this extra bit in the reference edge to interpret the BDD either in the positive or the negated form. Implementation-wise, this extra bit is typically encoded in the least significant bit of the address pointer (the reference edge) to avoid incurring extra memory cost. This encoding exploits the property that address pointers in modern machines are always at least 4-byte aligned, which means the least significant bit is always 0. Thus it can be used to encode the complement information.

4.6.2 Dynamic Variable Ordering

Since the size of a BDD graph is sensitive to the order selected on input variables, dynamic variable reordering is an essential part of all modern BDD packages. This component aims to dynamically establish a good variable order as the computation progresses. Typically, when a variable reordering algorithm is invoked, all top-level operations that are currently being processed are aborted. When the variable reordering algorithm terminates, these aborted operations are restarted from the beginning. The dynamic variable reordering algorithms are generally based on the *sifting* algorithm; i.e., the variable orders are changed by exchanging nodes in one level with nodes in the adjacent level. This process is illustrated in Fig. 4.7. This figure shows the sifting process for exchanging the orders of the variable *a* and the variable *b*. Each node is tagged with a

number so that these can be reffered easily. Let *f* be the function representing a BDD in terms of its cofactors.

$$f = v'.f|v<-0 + v.f|v<-1$$

where, *v* is the variable in b's root node and the 0-cofactor *f |v<-0* is recursively defined by the reachable sub graph of b's 0-branch child. Similarly, the 1-cofactor *f |v<-1* is recursively defined by the reachable sub graph of b's 1-branch child. Using this definition, the BDD in Fig. 4.6(a) can be represented as

$$a'.(b'.f0 + b.f1) + a.(b'.g0 + b.g1)$$

By rearranging this formula, we get

$$b'.(a'.f0 + a.g0) + b.(a'.f1 + a.g1)$$

New BDD after the performing sifting operation is shown in Fig. 4.6(b). Implementation-wise, node 4 and node 5 are created to represent the new children of node 1. Node 1 is updated to reference these new children and is relabelled with variable b. As for node 2 and node 3, they remain unchanged and if there are no references to them, they will be garbage collected later. Note that because node 1 might be referenced by others, it is important that node 1 is reused with its reachable graph representing the same function. Without this reuse, a new node will have to be created in place of node 1 and all the references to node 1 will be needed to be relocated and updated in order to reference this new node, which is very inefficient. The node-reuse technique allows the sifting algorithm to be a local operation involving only the node and its children.

1

0

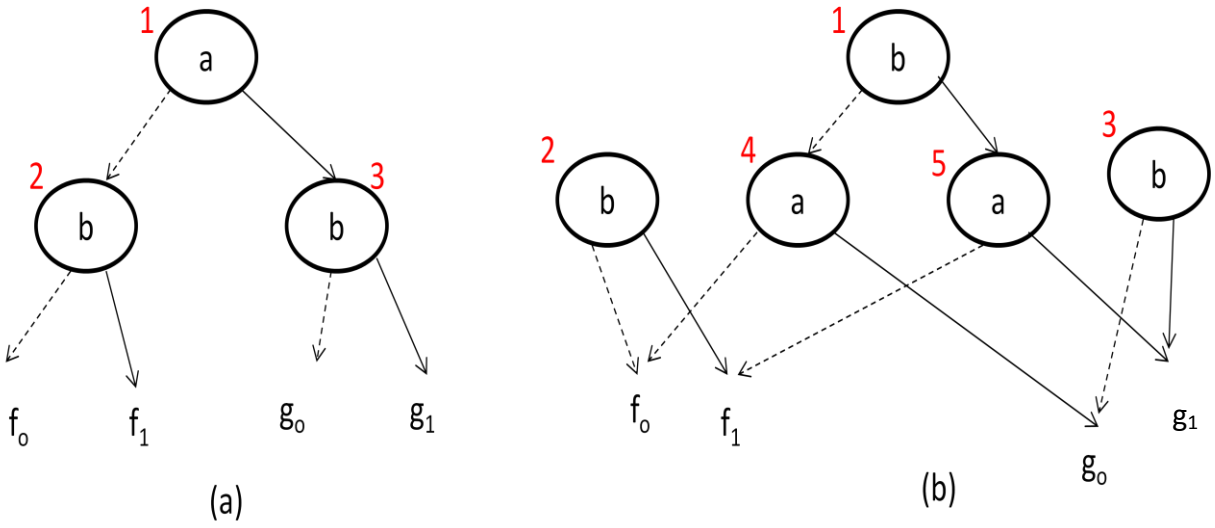

Fig 4.7 Shifting process for Dynamic Variable Reordering

4.6.3 Garbage Collection

BDD computations are inherently memory intensive because after all, it is all about traversing and constructing graphs. Furthermore, in verification, many intermediate BDD results are created to arrive at a simple final answer—true or false. Thus, it is important to have a good garbage collector to automatically remove BDD nodes that are no longer useful. A BDD node is referred to as *reachable* if it is in some BDD that external user has a reference to. As external users free references to BDDs, some BDD nodes may no longer be reachable (*deaths*). These nodes are referred to as *unreachable* BDD nodes. Hence all the *unreachable* nodes are garbage collected by the package leading to the fast manipulation and construction of BDDs by reducing the memory requirement.

5. Experimentation and Results

In this chapter, experimentation results by applying the proposed MMA with a number of Benchmark circuits taken from the LGSynth91 benchmark circuit suites have been presented. The MMA based technique, as defined above, is implemented with C++ codes and experimented by running on a Linux Red Hat Enterprise Tool. The result of experimentation for different Benchmark circuits has been shown in Table 5.1 and Table 5.2.

5.1 Comparison with Evolutionary Algorithms

The Table 5.1 compares the results of MMA based program with the other evolutionary algorithms i.e. simple genetic algorithm (SGA), hybrid genetic algorithm (HGA) and the latest particle swarm optimization algorithm (PSO) [17]. However, results with HGA are different from those in [13] as parameter settings are different.

Table 5.1. Comparison of MMA results with other Evolutionary Algorithms

Circuit Name	**#i**	**#o**	**Original size**	**SGA (within 10 runs)**	**HGA (within 5 runs)**		**PSO [17]**	**MMA (Proposed) (within 2 runs)**	
					Nodes	*It*		*Nodes*	*It*
5xp1	7	10	88	68	68	15	68	**68**	8
9sym	9	1	33	33	33	15	----	**33**	8
b12	15	9	91	68	67	16	62	**59**	9
bw	5	28	118	106	106	15	100	**106**	8
clip	9	5	254	96	95	19	112	**93**	8
con1	7	2	18	15	15	11	16	**15**	6
inc	7	9	73	62	61	17	79	**61**	9
misex1	8	7	47	37	37	14	37	**36**	8

Circuit name	**#i**	**#o**	**Original Size**	**SGA (within 10 runs)**	**HGA (within 5 runs)**		**PSO [17]**	**MMA (Proposed) (within 2 runs)**	
					Nodes	*It*		*Nodes*	*It*
misex2	25	18	140	99	95	18	89	**86**	14
misex3c	14	14	844	527	456	16	462	**445**	8
rd53	5	3	23	23	23	16	----	**23**	6
rd73	7	3	43	43	43	16	----	**43**	6
rd84	8	4	59	59	59	16	----	**59**	6
sao2	10	4	154	97	95	18	91	**85**	7
sqrt8	8	4	42	35	35	14	33	**33**	7
squar5	5	8	38	37	37	17	37	**37**	6
vg2	25	8	1087	222	255	18	182	**151**	9
Z5xp1	7	10	69	68	68	14	----	**68**	8

where, *It* represents the number of iterations.

5.2 Comparison with Dynamic Algorithms

Table 2 performs comparison with the reordering heuristics that have been incorporated in the BDD package *Buddy-2.4*, such as Window Permutation WIN2, WIN2ite, WIN3 and Sifting algorithm[8], [9]. Percentage improvement observed by using MMA as variable reordering algorithm w.r.t original size and as compared to Sifting algorithm is also indicated. The results indicate that in 55.55% cases, our MMA based approach has resulted in lesser number of node counts for the SDDs. Also, for all circuits, number of iterations to generate optimum variable order is lesser than the corresponding HGA.

Table 5.2. Comparison of MMA results with Dynamic Algorithms

Circuit name	Original size	Win2	Win2 Ite	Win3	Sift	MMA (Proposed)	% improvement w.r.t. Original size	% improvement w.r.t. Sift Algo
5xp1	88	87	82	82	82	**68**	22.72	17.07
9sym	33	33	33	33	33	**33**	0.00	0.00
b12	91	86	86	65	65	**59**	35.16	9.23
bw	118	113	112	106	106	**106**	10.17	0.00
clip	254	178	108	105	105	**93**	63.37	11.42
con1	18	18	18	18	18	**15**	16.67	16.67
inc	73	75	69	68	68	**61**	16.43	10.29
misex1	47	43	42	41	41	**36**	23.40	12.19
misex2	140	126	111	86	83	**86**	38.57	-3.60
misex3c	844	750	719	490	454	**445**	49.66	1.98
rd53	23	23	23	23	23	**23**	0.00	0.00
rd73	43	43	43	43	43	**43**	0.00	0.00
rd84	59	59	59	59	59	**59**	0.00	0.00
sao2	154	149	109	91	92	**85**	44.81	7.60
sqrt8	42	40	39	37	37	**33**	21.43	10.81
squar5	38	38	38	38	38	**37**	2.63	2.60
vg2	1087	864	113	104	103	**151**	86.11	-46.70
Z5xp1	69	68	68	68	68	**68**	1.45	0.0
Average							***24.03***	***2.75***

From the data in Table 5.2, it is observed that MMA resulted in an average reduction of 24.03% in SDD sizes. Also the proposed MMA provides a 2.75% average improvement in node count of Benchmark circuits as compared to the best known dynamic algorithm i.e. Sift Algorithm.

5.3 Graphical Analysis of results

Fig. 5.1 shows the differences in runtime of HGA and MMA in finding the optimum variable ordering for different BDD circuits. It provides a graphical view of the reduction in number of iterations done by the proposed MMA in achieving better results as compared to those achieved by using HGA. The figure shows that as the number of input and output variables in the circuit increases, the runtime of MMA remains half that of HGA even when all the performance influencing constants are kept same in these algorithms.

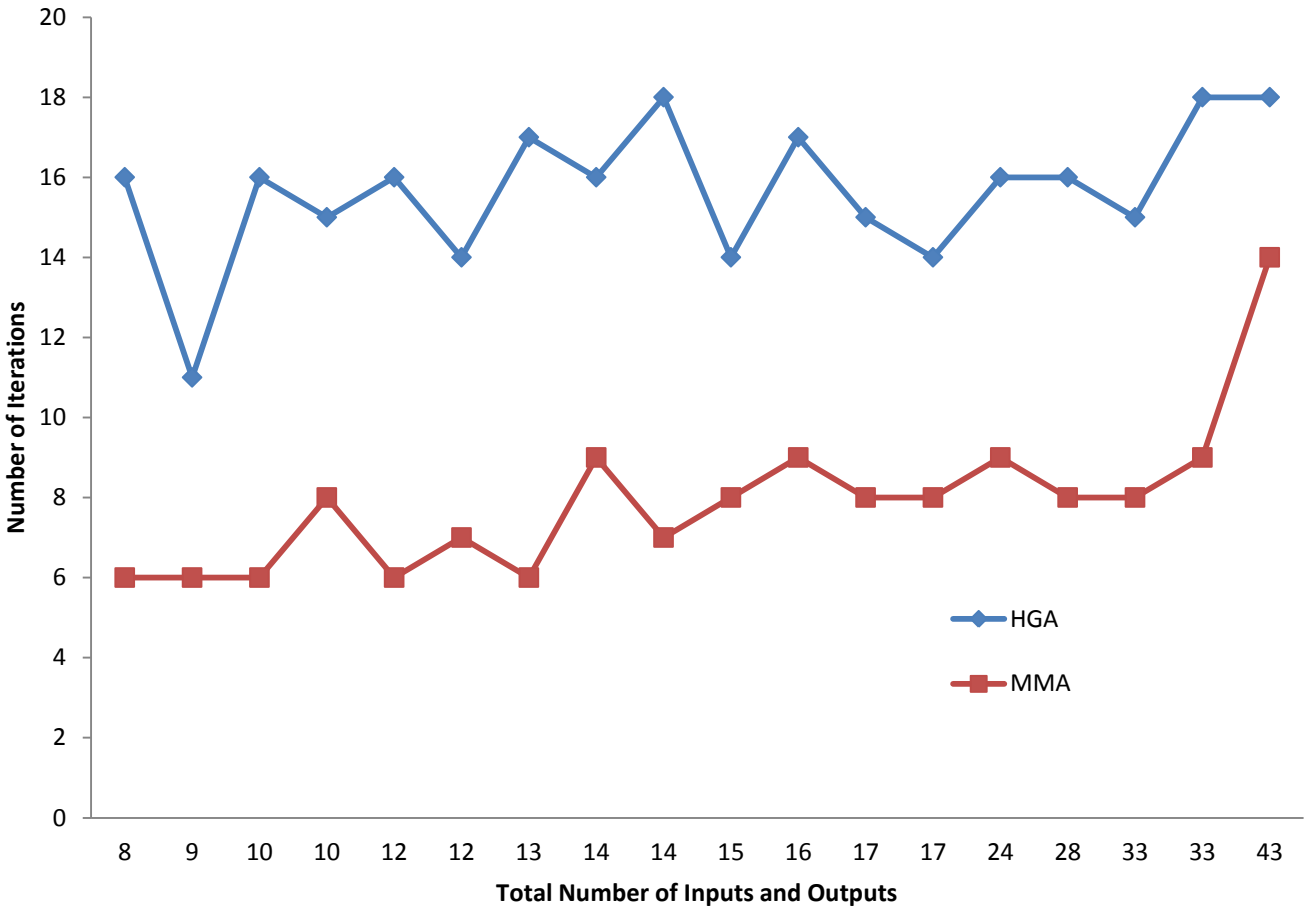

Fig. 5.1 Graph showing runtime differences in HGA and MMA

A graphical view of reduction in BDD sizes in terms of node count by using different optimization algorithms has been shown in Fig. 5.2. Accordingly, it can be observed that in most of the circuits, the results provided by the proposed MMA are better as compared to other *state-of-art* algorithms.

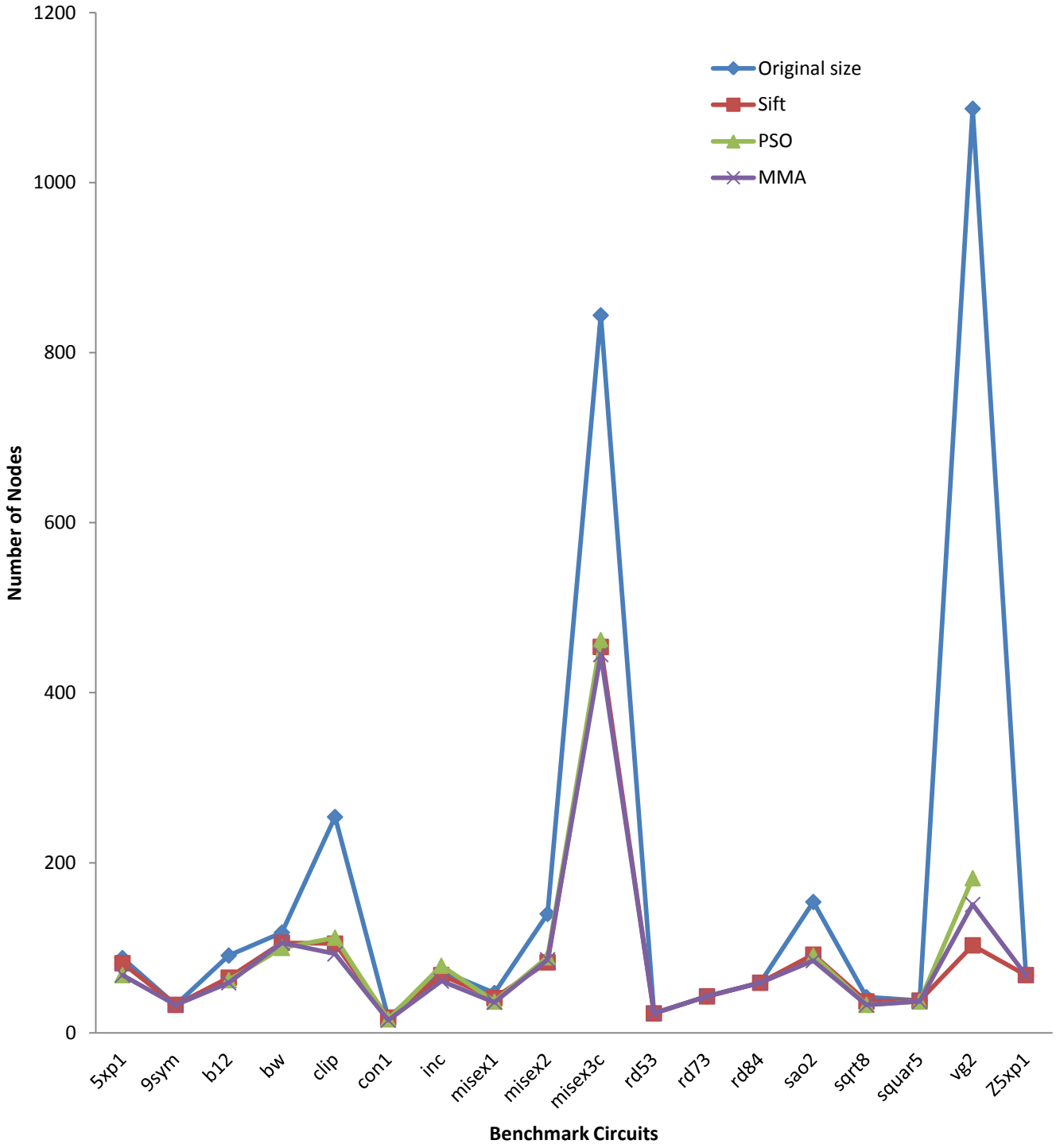

Fig 5.2 Graph showing reduction in BDD size in terms of node count using different algorithms

6. Conclusion and Future Work

6.1 Conclusion

Memetic Algorithm (MA) is a stochastic search algorithm based on Dawkins' theory of cultural evolution. It is an optimization technique based on strategic combination between global and local search operation. It is a method of practical success in a variety of optimization problems, particularly for best approximate solutions of NP-complete problem. In this thesis work, a Modified Memetic Algorithm (MMA) has been presented as one of the variable reordering algorithms for reducing the node count in shared BDDs for multi-output functions. The basic components of MA are modified and new refined features such as different local search (LS) operations for best and worst chromosomes, modified alternating crossover technique and modified mutation operations are incorporated in this new Hybrid Modified Memetic Algorithm (MMA) with an aim to enhance its performance in terms of quality of solutions, memory requirement and runtime.

Through experiments, it has been observed that the results have improved significantly compared to those obtained using the other Evolutionary as well as Dynamic Algorithms, in terms of both the node count reduction and the number of iterations. By using the proposed algorithm, an average improvement of 24.03% has been observed in the results when compared to original node count of the circuits and 2.75% average improvement w.r.t. the best known dynamic algorithm i.e. Sift algorithm. Thus, using this algorithm not only reduces the node count in SDDs as compared to other variable reordering algorithms, but at the same time, requires lesser iterations and population size to obtain the optimum result. Hence, the processing time is also reduced. This makes it an efficient variable reordering method.

6.2 Future Work

Future work will be concentrated on power optimization of logic circuits using this algorithm. Also performance improvement in terms of memory requirement will be another agenda in future work.

7. Paper Publication

1. Presented Poster Paper, "Reducing Node Count in Shared BDDs by finding Optimum Variable Ordering using Memetic Algorithm" , in *Seventeenth International Symposium on VLSI Design and Test(VDAT-2013*), Jaipur, INDIA.

References

[1] Sheldon B. Akers, "Binary Decision Diagrams," *IEEE Transactions on Computers*, vol. C-27, no. 6, pp. 509-516, Jun 1978.

[2] R.E. Bryant, "Graph based Algorithms for Boolean Function Manipulations," *IEEE Transactions on Computers*, vol. 35, no. 1, pp. 667-691, August 1986.

[3] K. Priyank, "VLSI Logic Test, Validation and Verification, Properties & Applications of Binary Decision Diagrams," *Lecture Notes*, Department of Electrical and Computer Engineering University of Utah, Salt Lake City, UT 84112, 1997.

[4] Y.Y. Liu, K.H. Wang, T.T. Hwang and C.L. Liu, "Binary Decision Diagram with minimum expected path length," *Design, Automation and Test*, pp. 708-712, 2001.

[5] Y. Iguchi, T. Sasao and M. Matsuura, "Evaluation of multiple-output logic functions using decision diagrams," *Asian South Pacific Design Automation Conference*, pp. 312-315, 2003.

[6] Masahiro Fujita, Hiranori Fujisawal and Yusuke Matsunaga, "Variable Ordering Algorithms for Ordered Binary Decision Diagrams and Their Evaluation," *IEEE Transactions on Computer-Aided Design*, vol. 12, no. 1, pp. 6-12, January 1993.

[7] H. Fujii, Goichi Ootomo and Chikahiro Hori, "Interleaving Based Variable Ordering Methods for Ordered Binary Decision Diagrams," *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pp. 38-41, 7-11 November 1993.

[8] R.Rudell, "Dynamic Variable Ordering for Ordered Binary Decision Diagrams," *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pp. 42-47, 7-11 November 1993.

[9] Eric Felt, Gary York, Robert Brayton and Alberto Sangiovami-Vincentelli, "Dynamic Variable Reordering for BDD Minimization," *Proceedings of EURO-Design Automation Conference*, pp. 130-135, 1993.

[10] C. Meinel and F. Somenzi, "Linear Sifting of Decision Diagrams," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol 19, no. 5, pp. 521-533, May 2000.

[11] William N.N. Hung, Xiaoyu Song, El Mostapha Aboulhamid and Michael A. Driscoll, " BDD Minimization by Scatter Search," *IEEE Transactions on Computer-*

Aided Design of Integrated Circuits and Systems, vol. 21, no. 8, pp. 974-979, August 2006.

[12] R. Drechsler, B. Becher and N. Gockel, "A genetic algorithm for variable ordering of OBDDs," *IEEE Proceedings on Computer and Digital Techniques*, vol 143, no. 6, pp. 364-368, 1996.

[13] N. Zhuang, M.S.T. Benten and P.Y.K Cheung, "Improved Variable Ordering of BDDs with Novel Genetic Algorithm," *IEEE International Symposium on Circuits and Systems*, vol. 3, pp. 414-417, 12-15 May 1996.

[14] Saurabh Chaudhary and Anirban Dutta, "Algorithmic Optimization of BDDs and Performance Evaluation for Multi-level Logic Circuits with Area and Power Trade-offs," *Proceedings of Seventeenth International Conference on Electronics, Circuits and Systems*, pp. 627-630, Dec 2010.

[15] Misagh Takapoo and M.B Ghaznavi-Ghoushchi, "IDGBDD: The novel use of ID3 to improve Genetic algorithm in BDD reordering," *International Conference on Electrical Engineering/Electronics Computer Telecomm. and Information Technology*, pp. 117-121, 19-21 May 2010.

[16] Wang Mingquan and Yu Haibin, "BDD Minimization Based on Genetic Tabu Hybrid Strategy," *Proceedings of the Sixth International Conference on ASIC*, pp. 948-952, 2005.

[17] A. Mitra and S. Chattopadhyay, "Variable ordering for shared binary decision diagrams targeting node count and path length optimization using particle swarm technique," *IET Computers and Digital Techniques*, vol. 6, no. 6, pp. 353-361 November 2011.

[18] Emad Elbeltagi, Tarek Hegazy and Donald Grierson, "Comparison among five evolutionary-based optimization algorithms," *International Journal of Advanced Engineering Informatics*, pp. 43-53, January 2005.

[19] Tom V. Mathew: Genetic Algorithm, Report submitted at IIT Bombay.

[20] Merz P and Freisleben B, "A Genetic local search approach to the quadratic assignment problem," *Proceedings of Seventh International Conference on Genetic Algorithms*, pp. 465-472, 1997.

[21] Lin Lu, Qi Luo, Jun-yong Liu and Chuan Long, "An Improved Particle Swarm Optimization Algorithm," *IEEE International Conference on Granular Computing*, pp. 486-490, 2008.

[22] Wikipedia search, "Memetic Algorithm," http://en.wikipedia.org/wiki/Memetic_algorithm.

[23] Miriam Pescador Rojas and Carlos A. Coello Coello, "A Memetic Algorithm with Simplex Crossover for Solving Constrained Optimization Problems," *World Automation Congress 2012*, pp. 1-6, 24-28 June 2012.

[24] Wolfgang Lenders and Christel Baier , *Genetic Algorithms for the Variable Ordering Problem of Binary Decision Diagrams* In: Alden H. Wright, Michael D. Vose, Kenneth A. De Jong and Lothar M. Schmitt (Eds.), *Foundations of Genetic Algorithms*, Japan: Springer, pp. 1-20, August 2004.