CHAPTER 3
MATERIALS AND METHODS

3.1 INTRODUCTION:

This chapter presents materials and methods as applicable to game playing and AI. The methods particularly are various search techniques as applied to AI and games. Various soft computing techniques are also discussed. The implementation of these techniques, particularly genetic algorithms and other search techniques in the test bed games is explained. Detailed discussion on test bed games Go-Moku and Othello is also given. Various patterns and shapes which are important from programming and game playing aspect of the game are also discussed. Both the games have been implemented using object oriented language Java with GUI and mouse interface. The implementation aspect of the game is also discussed. Various screenshots of the game play are also shown and discussed to explain game GUI.

3.2 SEARCHING IN GAMES:

To carry out search by means of a computer program, several elements of the problem and search strategy must be defined. Search requires the following elements [1]:

- A defined problem, which often requires to solve an intelligent or complex model of the problem and the representation of possible solutions,
- A goal state that defines the ideal solution, where heuristic methods frequently use an evaluation function to grade or rank the non-goal state solutions,
- Transformation operators which are capable of varying an existing solution into an another alternative solution, and
• A strategy for searching the space of possible solutions using the representation and transformation operators.

The application of various transformation operators on a solution creates a region of solutions. These newly created solutions are then compared to the goal state. When knowledge of the problem domain is available, heuristic algorithms can define an evaluation function that allows the scoring and ranking or grading of solutions. The ranking of solutions determines which of the solutions in the neighborhood region are better.

While search requires the above items, much of artificial intelligence and research focuses on the last item, namely, finding good search strategies [1].

Search problem can be defined as a tuple \((S,N,V)\) where \(S\) is the set of states \(s\), \(N\) is the neighborhood operator, and \(V\) is the value propagation function. The states in \(N(S)\) are the successors of the state \(s\), and \(s\) is called as the predecessor of the states in \(N(S)\). The goal is to reach one of the terminal states from the start state through intermediate states. A search problem indirectly defines a directed graph, with states as nodes and arcs between every node and its successor nodes. There are two basic search methods, forward search and backward search.

Games can be considered as a special class of graph search problems. The states are defined by the set of positions, and \(V(s)\) is defined by the set of moves from position \(s\). The terminal positions have one of the three values win, draw or loss.

When we look at games as search problems, then our goal is not necessarily to solve them. The state-space of many games is so large that their solution is still far out of reach. Instead of solving a game, we generally try to find heuristics which estimate the exact values of the positions. But even if a game is not solvable, we can use forward search and backward search as a kind of preprocessing for tournament game playing. For example, we can use backward search to build endgame databases, which can improve the values
found by forward search. From the other end we can use forward search to build opening books, which improve and accelerate game play during the first few moves into a game which use pre-calculated heuristics values. Thus combination of backward and forward search can be used for preprocessing of game play.

Go-Moku is a deterministic two-player, zero-sum game with perfect information. The game can be represented by a directed graph of nodes in which each node represents a possible board state. The nodes are linked by branches, which represent the moves that are made between board states. The directed graph is called a game tree in which some nodes may be connected by multiple paths [20].

The search tree is that part of the game tree that is analyzed by a player. A search tree has one root node which corresponds to the position under analysis. The legal moves in this position are represented by branches which expand the tree to nodes at a distance of one ply from the root. Same way, nodes at one ply from the root can be expanded to nodes at two plies from the root, and so on. When a node is expanded “d” times, and positions up to “d” moves ahead have been analyzed, the node is said to be investigated up to depth “d”. The number of branches expanding from a node is called the branching factor. When a node is expanded the new nodes one ply deeper are called child nodes.

The nodes at the same depth and having the same parent are called siblings. The nodes which are not expanded are called leaf nodes. The expansion of leaf node stops when any of following conditions occurs:

- The corresponding position may be final or concluding and the node is then denoted as a terminal node,
- There may not be enough resources to expand the leaf node any further,
- Expansion may be considered inappropriate or needless.

The process of expanding nodes of a game tree and to finding the right moves are called searching. For simple games like the game of Tic-Tac-Toe, it is possible to expand the
complete game tree so that all leaf nodes correspond to final positions where the result is known, which can then be used to construct a strategy that guarantees optimal play. Theoretically, this can also be done for any board game. But in practice, the game tree is too large to expand completely because of limited computational resources. A simple estimate for the size of the game tree in Go, which assumes an average branching factor of 250 and an average game length of only 150 ply leads to a game tree of about $250^{150} \approx 10^{360}$ nodes which is impossible to expand fully.

As full expansion of game trees is impossible under nearly all conditions, leaf nodes generally do not correspond to final positions. Evaluation functions are used to predict the worth of the underlying tree which is not expanded. In theory, a perfect evaluation function with a one-ply search i.e an expansion of only the root node would be sufficient for optimal play. In practice, however, perfect evaluations are difficult to construct and for most interesting games they cannot be computed within a realistic time. Practically, full expansion and perfect evaluation are both unrealistic, so most search based programs use a balanced approach where some positions are evaluated directly while others are expanded further. Thus, balancing the complexity of the evaluation function with the size of the expanded search tree is known as the trade-off between knowledge and search [21][22][23].

3.3 OVERVIEW OF SEARCHING TECHNIQUES:

The problems that require searching for solution basically use search algorithms which are categorized broadly as: Uninformed search algorithms, and informed search algorithms [24].

3.3.1 Uninformed Search Algorithms:
These search algorithms are also known as blind or unsighted search algorithms as they do not have additional information about states beyond that provided in the problem definition. They can only generate successors and can distinguish whether successor state is a goal state or not.
Breadth first search (BFS) is uninformed simple search strategy in which root node is expanded first and then, all the nodes at the same level are expanded first and then only the next level are node are expanded. The space complexity is $O(b^{d+1})$. Thus, exponential complexity problems can not be solved by uninformed search methods due to high time and memory requirements [24].

Depth first search (DFS) is also uninformed simple strategy in which root node is expanded first and then that node is expanded it its deepest level, then only the next node at the same level is tried up to its deepest level. The advantage is that the memory requirement of DFS is very small compared to that of BFS. But the drawback is that it may get stuck in very long or an infinite path which may not terminate at all. This problem can be reduced by allowing the search to determined depth limit $\ell$. But if depth limit $\ell < d$ may introduce incompleteness. Same way if $\ell > d$, leads to non-optimal search. The space complexity is $O(b\ell)$.

Thus, informed search strategies are ineffective as they do not have the supplementary knowledge about the problem state [24].

3.3.2 Informed Search Algorithms:

These methods are also known as Heuristic search algorithms. These methods use problem specific knowledge in addition to the definition of the problem and can find solutions more efficiently.

Best first search is the general method in which a node is selected for expansion based on an evaluation function $F$. The node, which is having lowest evaluation, which normally is a measure of distance between current state and goal state, is selected for expansion. The method normally uses a function, called heuristic function, denoted $h(n)$, which takes node $n$ as an input parameter.
h(n) = estimated cost of cheapest path from node n to a goal node.

It is through the heuristic function, the search algorithm gets additional information about the problem. If n is a goal node, then h(n) = 0.

When it comes to internal representation of moves and position, we are mostly concerned with speed. Finding a good evaluation function is a matter of implementing knowledge about the game in question. The basic idea for this is quite simple. Since we cannot expect to work our way through the entire game tree, we have to find out a way of turning positions into values without further searching the game tree. That is the sole purpose of the evaluation function. It provides us with a interim value for a position, which by force has to be rough. Ultimately it is a static process which does not take into account how the game might develop from a given position. Instead, it is an attempt to assign a value by simply looking at current position of the board. If this process goes wrong, the alpha-beta search will select moves which may not direct to good positions, which can then be exploited by the opponent. There is no way of having a guarantee that the evaluation function will be right, as it can not be developed in one unique way. Mostly a big part of writing game playing programs consists of watching the program play and fine-tuning the evaluation function accordingly. Apart from a few common sense ideas, evaluation functions are therefore mostly based on heuristics. There is one characteristic regarding evaluation functions which concerns speed: Evaluating a position can be quite a complex process, with various aspects of the position requiring scoring. Therefore calculating such an estimated value for each position separately will certainly repeat some of the work done previously, and hence it will be somewhat slow.

By forcing an evaluation function to provide a value by looking only at the current position it is difficult to take into account.

Some aspects that might be appropriate to evaluating a position are discussed. The degree of importance may vary from game to game. To judge a position it is normally important to do these evaluations for both players [25].
• **Material:**

  For example, in chess, that would be the number of pieces, where each piece gets its own value, while in go, it would be a count of pieces on the board, and similarly for Othello. This is not equally useful in all games, however: In Othello, for example, it is not really the number of pieces in one’s own color that is important, but whether one holds specific fields or positions, for example corner positions. Quite often the player with the better position will have fewer pieces on the board. There are other games where the number of pieces may be irrelevant.

• **Space:**

  In some games it is possible to divide the board into areas of influence, where a given player controls a number of fields. This is particularly relevant for go. In chess, one can count the number of fields threatened by one player for this purpose, and in Othello the number of pieces which cannot be taken by the opponent - a connected group of pieces surrounding a corner. One can just calculate the size of these regions, or attach some sort of weight to them if not all fields are equally important.

• **Mobility:**

  Having many different moves available can be an advantage in a game of Othello being a particular example. For chess there is some doubt as to whether or not this is a useful measure, some people have tried and discarded it while others have retained the principle.

• **Tempo:**

  In games such as go there is a question of which player has the initiative that is the ability to make moves which attack further as opposed to having to make defensive moves whose main purpose is damage limitation. In most of the cases, having the initiative means that in reply, the other player has to make a defensive move to avoid worse, leaving the initiative with the original player.
• **Threats:**

The threats often force the player to make a bad move in order to avoid further loss at that point of time. Can one of the players capture or threaten to capture a piece? In Go-Moku, does one of the players have a number of pieces lined up already? In Othello, is a player threatening to take a corner?

• **Shape:**

This is actually about various pieces on the board connecting to each other. In chess, for example, a line of pawns advancing is much stronger than, say, pawns sharing a column. In go, shape is about ‘territory to be’—a few well-placed stones outline a territory which the player can defend when threatened. Judging shape can be very difficult, and usually shape is formed by a number of moves made, where every one such move improves the position only incrementally, but where the resultant position can be a lot stronger. Shape is also usually a long-term target. An evaluation function partially based on shape will have to be based on something other than the simple addition of piece-based evaluation functions.

• **Known patterns:**

In many games there are patterns which come up over and over again. This is particularly true for go, where there are many libraries of sequences of moves relating to a small area. In chess, a bishop capturing a pawn on the border is often trapped. In Othello, it is sometimes beneficial to sacrifice one of the corners if one can then force ownership of another corner. It might be worth while to program such things clearly in order to avoid making a bad move, or to follow the moves from a library if certain constellations are reached. In practice it is typically difficult to reliably recognize positions where such patterns should be applied, and to adjust the moves identified to the current situation.
The above criterion results in a variety of components that might make up the actual evaluation function. Typically these components are weighted and then added up, where the weights are determined based on heuristics. The reason for this is that summation is a simple process for combining numbers, which is fairly fast. Typically an evaluation function consists of a number of components each with a weight, that is, given a position \( p \) we have,

\[
f = w_1f_1 + w_2f_2 + \ldots + w_nf_n
\]

where, \( w_i \) are weights and \( f_i \) are the various aspects of the game.

We can think of the weights as allowing us to give varying importance to the different parts of the evaluation function.

Typically to improve an evaluation function, there are then two alternatives:

- Make the evaluation function more sophisticated by introducing further functions.
- Adjust the weights \( w_i \) to get better results.

Most of the time is typically spent on the issue of adjusting the weights of the game. It has to take into consideration about the features, patterns and other things. Following are the different ideas that can be used to fine-tune the weights [25].

- **Deducing constraints:**
  
  In games such as chess, every piece is given a material value. Clearly a rook, say, is more powerful than a pawn, and the material value should reflect that. By analyzing typical games, it can be possible to deduce constraints that these values should satisfy.

- **Hand tweaking:**
  
  This is what happens most often in practice. Programmers watch their implementation execution play and then try to judge which parameters should be changed, and how much. They perform the change and watch the effect again.
This produces reasonable results fairly quickly, but requires that the programmer has enough knowledge about the game to analyze what is going wrong and what is going right.

- **Optimization techniques:**

  Rather than using only human judgments to experiment any parameters involved, one can use general optimization techniques. One example for these is hill climbing: Small changes are made to the parameters, and changes are only kept if they improve the performance. This requires some sort of measure to judge performance, for example the percentage of won games against some opponent. This tends to be sluggish and risks being stuck in positions where each small change makes the performance inferior, but where a big change might bring giant gains, such situations are known as ‘local optima’.

A* algorithm is the most widely-known form of best-first search. It evaluates the nodes by combining, the cost to reach the node \( g(n) \) and the cost to get from the node to the goal \( h(n) \).

So, \( f(n) = g(n) + h(n) \).

So, \( f(n) \) gives us the estimated cost of the cheapest solution through node \( n \).

A* algorithm is optimal if the heuristic function \( h(n) \) is an admissible heuristic i.e. \( h(n) \) never overestimates the cost to reach the goal node. The main drawback of A* algorithm is space complexity, because it keeps all generated nodes in memory, A* usually runs out of memory. This is the reason, why A* is not practical for many large state space problems. The improvements MA* (memory-bounded A*) and SMA* (simplified MA*) use all available memory, but they are not capable of providing optimal solution.

**3.3.2.1 Local Search Algorithms:**

If the path to the goal is not important, where ordering of the steps is irrelevant, local search methods can be used. These methods work using a single current state and generally move only to neighbors of that state. The path followed is not retained. Local
search algorithms are not systematic, but have two advantages: they use very little memory, and importantly they can find reasonable solution in large or infinite state space for which systematic algorithms are not fit.

The local search algorithms are useful for solving pure optimization problems, in which the aim is to find the best state according to an objective function.

3.3.2.2 Hill-climbing Search:

It is simply a loop that repeatedly moves in the direction of increasing value- that is uphill. It terminates when it reaches a peak where no neighbor has a higher value [1]. Figure 3.1 shows the basic hill-climbing search method.

![Figure 3.1 Basic hill-climbing search](image)

It does not maintain a search tree. It often get stuck because of local maxima – a peak that is higher than each of its neighbors, but lower than global maxima, ridges – it results in a sequence of local maxima, where all available actions point down the hill, plateau – it is an area of the state space landscape, where the evaluation function is flat. It can be local maximum or a shoulder from which it is possible to make progress. Figure 3.2 shows a fitness landscape which is very complex. The basic hill climbing algorithm will not be able to find the best value for the parameter under consideration.
Figure 3.2 Non monotonic fitness landscape

Figure 3.3 shows one more examples where the hill climbing method will not give optimized value of parameter, because the fitness value is random and is not co-related in any way with nearby values [1].

Figure 3.3 Random fitness landscape

The success of hill-climbing depends very much on the character of the state-space landscape- if there are few local maxima and plateau, random start hill-climbing will find a good solution very quickly. NP-hard problems typically have exponential number of local maxima.

The problem of getting stuck in local minima or maxima can be solved by introducing an idea known as momentum. If the search is consistently improving in one direction, then it
should continue in that direction for a little while, even when it looks that things are not improving any more. It takes a couple of steps to work out that the things are getting worse, and then only it reverses the direction. Figure 3.4 shows how the concept of momentum is applied to hill climbing and how the optimized value of the parameter under consideration is found.

Hill climbing is very much dependent on the initial value of the guess. If the initial guess is not on the slope toward the best parameter value, then the hill climber may move totally in the wrong direction and may end up climbing a smaller peak. Most hill climbing algorithms use multiple different start values distributed across the whole landscape. As shown in Figure 3.5, the correct optimum is found on the third attempt.
3.3.2.3 Simulated Annealing Search:

Annealing is a physical process where the temperature of a molten metal is slowly reduced, allowing it to harden in a highly ordered way. Reducing the temperature suddenly leads to internal stresses, weaknesses, and other undesired effects. Slow cooling allows the metal to find its lowest energy pattern.

As a parameter optimization technique, annealing uses a random term to represent the temperature. Initially, it is high, making the behavior of the algorithm very random. Over a time it reduces, and the algorithm becomes more predictable.

From above, it is clear that this method tries to combine hill climbing with randomness. There are two different methods to introduce randomness with hill climbing.

- Direct method
- Boltzmann probability

In direct method, initially a random value in larger range is used, as the method progresses, the range of random number goes on reducing.

In Boltzmann probability method, instead of picking the best move, it picks a random move. If the move improves the situation, it is always accepted. Otherwise, the algorithm accepts the move with some probability less than 1. The probability decreases exponentially with the bad quality of the move. Bad moves are likely to be allowed at the start and their probability of allowing decreases as time progresses. Thus, algorithm will find a global optimum with probability approaching 1.

Just like hill climbing method, momentum can be used to optimize the simulated annealing search method using trial and error to decide the amount of momentum, step size etc.

3.3.2.4 Local Beam Search:

Instead of keeping only one node in memory, the local beam search keeps track of k states rather than just one. It begins with k randomly generated states. At each step, all
the successors of all k states are generated. If any one is a goal, the search terminates. Otherwise, it selects k-best successors from the list and repeats the work. The important thing is all parallel search thread share useful information. It can suffer from concentration in a small region of state space. A variant of local beam search is called stochastic beam search – which is like hill climbing, uses k successors not from best criteria, but k successors are chosen randomly, with the probability of choosing a given successor being an increasing value of its value. Stochastic search bears some similarity with the process of natural selection.

3.3.2.5 Evolutionary Algorithms:

Evolutionary algorithms (EAs) are population-based metaheuristic optimization algorithms that use biologically inspired mechanisms like mutation, crossover, natural selection, and survival of the fittest in order to refine a set of solution candidates iteratively. The advantage of evolutionary algorithms compared to other optimization methods is their black box type character which makes only few assumptions about the primary objective functions. The definition of objective functions generally requires lesser insight to the structure of the problem space than the manual construction of an acceptable heuristic. EAs therefore perform consistently well in many different problem categories.

These algorithms work on the principles given by scientist named Darwin. The Darwinian evolution uses the principles of competition, inheritance, and variation within a population. These concepts are used to define a category of iterative enhancement metaheuristic search methods. Evolutionary algorithms use a population of solutions and genetic operators to carry out search. In this mode, evolutionary machine learning can meet and eventually go beyond the capabilities of human expertise [26].

The theory of Darwin can be condensed into ten observations and deductions

1. The individuals of a species possess great fertility and produce more offspring than can grow into adulthood.
2. Under the absence of external influences (like natural disasters, human beings, etc.), the population size of a species roughly remains constant.

3. Again, if no external influences occur, the food resources are limited but stable over time.

4. Since the individuals compete for these limited resources, a struggle for survival ensues.

5. Especially in sexual reproducing species, no two individuals are equal.

6. Some of the variations between the individuals will affect their fitness and hence, their ability to survive.

7. A good fraction of these variations are inheritable.

8. Individuals less fit are less likely to reproduce, whereas the fittest individuals will survive and produce offspring more probably.

9. Individuals that survive and reproduce will likely pass on their traits to their offspring.

10. A species will slowly change and adapt more and more to a given environment during this process which may finally even result in new species.

In particular, the evolutionary algorithm employs the following items:

- A population of contender or candidate solutions called individuals,
- A fitness function that evaluates and assigns each individual a score, or fitness value,
- Transformation operators that produce offspring or child individuals from parent individuals, implementing the concept of inheritance through stochastic variation, and
- A stochastic selection method for selecting individuals with better fitness to produce children.

Figure 3.6 shows basic cycle of evolutionary algorithms.
The configuration parameters of evolutionary algorithms are shown in figure 3.7.

1. Its basic parameter settings like the population size or the crossover and mutation rates,
2. Whether it uses an archive Arc of the best individuals found and, if so, which pruning technology is used to prevent it from overflowing,
3. The fitness assignment process and the selection algorithm,
4. The choice of the search space and the search operations,
5. The genotype-phenotype mapping connecting the search space and the problem space.
Figure 3.7 Configuration parameters of evolutionary algorithm

Some of the other important issues in Evolutionary algorithms are:

Population Initialization & Re-initialization: The simplest approach is to generate random individuals for the initial population. However, it is possible and often advisable to use heuristics to construct better individuals. The heuristics are typically knowledge-based and problem dependent. Apart from being generated at the start of the search, new individuals could also be introduced during the search by heuristics instead of using reproduction operators. This is known as re-initialization in which a part or whole population is reinitialized if the search gets stagnant.

Generational vs. Steady-State: The replacement strategy could be either a generational or a steady-state approach, also referred to as generational or steady-state selection. The selection of the approach is mainly driven by the fitness assignment strategy. If the fitness assignment strategy requires a large number of individuals in the population to estimate the fitness of an individual, the generational approach should be employed. On the other hand, the steady-state selection could be used if the fitness of an individual is
not affected by other individuals in the population. Inappropriate approaches could lead to expensive computational time and degraded performance of the search.

Exploration vs. Exploitation: This is also known as diversity vs. intensification of the population. The reproduction method is mainly the driving force for exploration whereas exploitation is taken care by the replacement strategy. In EAs, it is very difficult to obtain good results in terms of both exploration and exploitation at the same time. There is usually a trade-off between these two aspects. Therefore, balancing well this trade-off could lead to high performance algorithms.

Elitism: This term means that the best individuals are always included in the next population. It is clear-cut in the single-objective framework where there is only one best individual at any time. However under the multi objective framework, there could be always more than one best individual. The number of best individuals could increase dramatically with respect to the number of objectives of the problem. With a limited population size, it is non-trivial to identify which best individuals should be kept. An external archive or favoring best solutions with at least one best objective value could be the answer. Laumanns et al. also argued that the usefulness of elitism strongly depends on the mutation rate.

Duplication: It occurs in the population if there is at least a pair of distinct individuals having the same objective values. The problem with allowing duplication is that all individuals in the population could converge too quickly to a single point in the objective space which is not desirable. To deal with this problem, a tough approach it to firmly not allow any duplication while in a soft approach is to allow the replacement strategy to eliminate duplication.

Mating Restriction: The idea of controlled mating comes from natural selection where mating only seems to happen between similar individuals. Deb and Goldberg argued that mating between unlike individuals will likely lead to weak offspring called lethals. However, mating between too similar individuals, known as in-breeding, could affect
adversely the progress of the search. Mating restriction could be performed on either the genotype or phenotype of individuals.

Evolutionary algorithms are often categorized into four main branches that are primarily distinguishable by their usually used representation and operators [27]:

- **Genetic Algorithms** use a bit-string in a fixed length genome and two-parent crossover. A bit-string requires a mapping to a representation that can be evaluated and assigned fitness by the heuristic evaluation function. The genetic algorithm conventionally relied on various forms of crossover as the main operator.

- **Evolutionary Strategies** introduced by Rechenberg are a heuristic optimization technique based in the ideas of adaptation and evolution, a special form of evolutionary algorithms use a real-valued vector using a direct encoding of the candidate solution. Offspring are produced using Gaussian mutation operator. They usually use vectors of real numbers as solution candidates. Both the search and the problem space are fixed-length strings of floating point numbers.

- **Evolutionary Programming** employs a finite-state machine and mutation operators. Mutation operator is the only way of introducing variation. In this sense, the population in evolutionary programming is like an environment of species, where each individual is a different species.

- **Genetic Programming** uses a computer program or executable structure and two-parent crossover. Genetic programming is an evolutionary algorithm that represents solutions as programs [40][41]. In GP, usually some inputs or situations and corresponding output data samples are known or can be produced or simulated. The goal then is to find a program that connects them or that exhibits some kind of desired behavior according to the specified situations, as shown in figure 3.8.
These classifications correspond to common or early implementations. In practice, many implementations use components from different branches and make the classifications less accurate.

The genetic algorithm is a variant of stochastic beam search in which successor states are generated by combining two parent states, rather than by modifying a single state [28]. It holds the same correspondence with natural selection as it is there in stochastic beam search. In genetic algorithms, we deal with sexual reproduction. In this thesis a detailed discussion on genetic algorithms and its application to game playing aspect is discussed. The detailed discussion of genetic algorithm is given in section 3.5.

### 3.4 ADVERSARIAL SEARCH:

In the competitive environments, where the agents’ goals are in conflict with each other, make them become adversarial search problems, which are often known as games. The agents are usually limited to a small number of actions whose outcomes are defined by accurate rules. A game can be formally defined as a type of search problem with the components as: initial state, successor function, terminal state and utility function or objective function. The initial state and the legal moves for each side define the game tree for the game.
Many techniques have been developed in the last century for searching game trees. The base of most game-tree search algorithms is mini-max algorithm. Mini-max algorithm is in theory important; no modern game-playing engine uses it directly. Instead, most of the game-playing engines use some form of $\alpha\beta$ search algorithm which comes in many flavors. Perhaps the most successful flavor of $\alpha\beta$ is the iterative deepening principal variation search \cite{1}\cite{24}.

3.4.1 Mini-Max Search Algorithm:

In this method, the nodes of a game tree are categorized in two types - max and min nodes. In max type node, the player to move tries to maximize the score. The root node, which is by definition ply 0, is a max node by rule, and similarly all nodes at an even ply are max nodes. In min type of node, the opponent player tries to minimize the score. Nodes at an odd number ply are min nodes. Starting from evaluations at the leaf nodes, and by choosing the highest value of the child nodes at max nodes and the lowest value of the child nodes at min nodes, the evaluations are propagated back up the search tree, which eventually results in a value and a best move in the root node. Figure 3.9 shows the simple 2-ply game tree with tree width =2.

The strategy found by mini-max is optimal in the sense that the mini-max value at the root is a lower bound on the value that can be obtained at the boundary spanned by the leaf nodes of the tree under search. But, since the evaluations at leaf nodes, which are not final positions, this does not guarantee that the strategy is also optimal for a larger tree or for a tree of similar size after some more moves. When game playing engines use a deeper ply search, normally the play becomes stronger. Due to approximations and heuristics in the evaluation function, it is never accurate until the very end of game.

The number of positions that has to be searched by this algorithm is $W^D$, where $W$ is the width of the tree i.e. average number of moves possible in each position, and $D$ is the depth of the tree \cite{1}. This is very inefficient and would take lot of time.
3.4.2 αβ search Algorithm:

The mini-max search algorithm requires examining exponential number of game states. The basic idea of pruning is used to compute the correct mini-max decision without even looking at every node in the game tree. The Alpha-beta pruning applies branch and bound techniques to the mini-max algorithm. Although mini-max search can be used directly, it is possible to determine the mini-max value of a game tree much more efficiently using αβ search. The αβ search algorithm uses two limits, α and β, on the value of score during the search. The lower bound value - α represents the worst possible score for max. Any sub-tree of value below α is not attractive for investigating, which is called α cut-off. The upper bound value - β, represents the worst possible score for min. If in a node a move is found that results in a score greater than β value, the node does not need to be investigated further because min will not play this line, which is called a β cut-off. Figure 3.10 explains the algorithm.
When a search process decides not to further analyze some parts of the tree, which the mini-max search algorithm would investigate by default, this is called pruning. Some pruning, such as αβ pruning, can be done carefully without changing the mini-max result. However, it can also be interesting to prune nodes that are just not likely to change the mini-max result. When a pruning method is not guaranteed to preserve the mini-max result it is called forward pruning. Forward pruning discards some seemingly unpromising branches to reduce the size of the game tree. The depth of the search tree explored strongly influences the strength of the game-playing program. So sometimes exploring the best moves more deeply is better than considering all moves. Many techniques have been developed to perform forward pruning. Two normally used forward-pruning methods are null-move pruning and multi-cut pruning.

The efficiency of αβ search depends very much on the order in which the nodes are investigated. In the worst case scenario, the number of nodes visited by αβ is equal to the full mini-max search tree. In the best case, where we investigate only one node at max node, and all nodes at min node, when the best moves are always investigated first the number of nodes visited by αβ search approaches the square root of the number of nodes in the full mini-max tree. In the ideal case and assuming a constant branching factor b, this then provides a branching factor of 1 at even plies and b at odd plies, which results in a tree with an average branching factor of √b compared to b for the mini-max search tree. This means that with a just right move ordering and a limited amount of time, αβ can look ahead twice as deep as mini-max search without any extra risk [1] [24].
Figure 3.10 αβ pruning on simple 2-ply game tree with tree-width = 2

Good move ordering leads to huge reduction in nodes to be investigated. Much research effort has been targeted into finding good techniques for move ordering. The various move-ordering techniques can be characterized by their dependency on the search and their dependency on game-specific knowledge. Some of the well-known search-dependent move-ordering techniques are the transposition table, which stores the best move for previously analyzed positions, the killer heuristic, which selects the most recent moves that generated a cut-off at the same depth, and the history heuristic, which orders moves based on a weighted cut-off frequency as observed in investigated parts of the search tree. In principle, these techniques do not use game specific knowledge.

A large number of improvements to the Alpha-Beta algorithm have been developed. Many of them are used in practice and can dramatically improve the search efficiency. Here we discuss some major enhancements in alpha-beta cutoff algorithm which are based on one or more of the four principles: move ordering, minimal window search, quiescence search and forward pruning.
The simplest implementations of αβ search examine the tree up to a predefined depth. It is difficult to predict the time to finish the search. When playing under tournament conditions, this is a problem. Iterative deepening solves the problem by starting with a thin search and gradually increasing the search depth - usually by one ply per iteration until time runs out. Although this may seem inefficient at first, it actually turns out that iterative deepening can improve performance over plain fixed-depth αβ. The reason for this is that information from previous iterations is not lost and is re-used to improve the quality of the move ordering.

Iterative deepening was originally created as a time control mechanism for game tree search. It handles search depth problem depending on the estimate of the amount of time the search will take. A simple fixed depth is inflexible because of the variation in the amount of time the program takes per move. So David Slate and Larry Atkin introduced the notion of iterative deepening: it starts from 1-ply search, repeatedly extend the search by one ply until we run out of time, then report the best move from the previous completed iteration. It seems to waste time since only the result of last search is used. But fortunately, due to the exponential nature of game tree search, the overhead cost of the preliminary D-1 iterations is only a constant fraction of the D-ply search. Besides providing good control of time, iterative deepening is usually more efficient than an equivalent direct search. The reason is that the results of previous iterations can improve the move ordering of new iteration, which is critical for efficient searching. So compared to the additional cut-offs for the D-ply search because of improved move order, the overhead of iterative deepening is relatively small. Many techniques have proved to further improve the move order between iterations. Iterative deepening was originally created as a time control mechanism for game tree search. It handles the problem that how it should choose the search depth depends on the amount of time the search will take. A simple fixed depth is inflexible because of the variation in the amount of time the program takes per move. So David Slate and Larry Atkin introduced the notion of iterative deepening: start from 1-ply search, repeatedly extend the search by one ply until we run out of time, then report the best move from the previous completed iteration. It seems to waste time since only the result of last search is used. But fortunately, due to the
exponential nature of game tree search, the overhead cost of the preliminary D-1 iterations is only a constant fraction of the D-ply search. Besides providing good control of time, iterative deepening is usually more efficient than an equivalent direct search. The reason is that the results of previous iterations can improve the move ordering of new iteration, which is critical for efficient searching.

So compared to the additional cut-offs for the D-ply search because of improved move order, the overhead of iterative deepening is relatively small. Many techniques have proved to further improve the move order between iterations.

Another ordering technique is the transposition table, which is used to store results of previously investigated nodes. It is important to store this information because nodes may be visited more than once during the search. The results stored in the transposition table usually contain information about the value, the best move, and the depth to which the node was examined. Due to limitations in the available memory resource, it is not possible to store all node entries in the memory. In most search engines the transposition table is implemented as a hash table with a fixed number of entries. Addressing the relevant entry in a table takes place through a conversion to a sufficiently large number – which is called the hash value [24].

The transposition table can be used to suggest a probable candidate for best move when a like position occurs again. But it can neither order the left over moves of revisited positions, nor give any information on positions not in the table. So the killer move heuristic is frequently used to further improve the move ordering. The idea of the killer move heuristic is that different positions encountered at the same search depth may have like characters. So a good move in one branch of the game tree is a good bet for another branch at the same depth. The killer heuristic typically includes the following procedures:

1) Maintain killer moves that seem to be causing the most cutoffs at each depth. Every winning cutoff by a non-killer move may cause the replacement of the killer moves.
2) When the same depth in the tree is reached, examine moves at each node to see whether they match the killer moves of the same depth; if so, search these killer moves before other moves are searched.

The history heuristic, which is first introduced by Schaeffer, extends the basic idea of the killer move heuristic. As in the killer move heuristic, the history heuristic also uses a move’s previous usefulness as the ordering criterion. But it maintains a history for every legal move instead of only for killer moves. In addition to that it accumulates and shares previous search information throughout the tree, rather than just among nodes at the same search depth.

In the alpha-beta procedure, the narrower is the search window, the higher the possibility that a cutoff occurs. A search window with alpha = beta – 1 is called the minimal window. Since it is the narrowest window possible, many people believe that applying minimal window search can further improve search efficiency. Some alpha-beta refinements such as NegaScout and MTD are derived from minimal window search.

The idea of PROBCUT is based on the assumption that evaluations obtained from searches of different depths are strongly correlated.

NegaScout and Principal Variation Search are two similar refinements of alpha-beta using minimal windows. The basic thought behind NegaScout is that most moves after the first will result in cutoffs, so evaluating them precisely is of no use. Instead it tries to prove them inferior by searching a minimal alpha-beta window first. So for sub trees that cannot improve the previously computed value, NegaScout is superior to alpha-beta due to the smaller window. However, sometimes the move in question is indeed a better choice. In such a case, the corresponding sub tree must be revisited to calculate the precise minimax value.

A fixed-depth approximate algorithm searches all possible moves to the same depth. At this maximum search depth, the program depends on the evaluation of intermediate
positions to estimate their final values. But actually all positions are not equal. Some “quiescent” positions can be assessed accurately. Other positions may have a threat just beyond the program’s maximum search depth, and so cannot be evaluated correctly without further search.

The solution, which is called quiescence search, is increasing the search depth for positions that have potential and should be explored further. Although the idea of quiescence search is attractive, it is difficult to find out a good way to provide automatic extensions of non-quiescence positions.

### 3.5 GENETIC ALGORITHMS:

Evolutionary computing was introduced by I. Rechenberg in the work “Evolution strategies”. Genetic Algorithms was invented by John Holland and developed this idea in his book “Adaptation in natural and artificial systems” in the year 1975. Holland proposed GA as a heuristic method based on the concept of survival of the fittest. It was discovered as a useful tool for search and optimization problems.

Genetic algorithms are search methods that fairly accurate biological genetics i.e. simulate evolution in an attempt to find a solution or a goal for a particular problem. Genetic algorithms start with the creation of a set of completely random chromosome of alleles. These arrays of bits are translated into parameters of the solution to find out how well they estimated the solution. The function which finds out how well an individual chromosome is called fitness function or objective function [28].

Genetic Algorithms are global search techniques providing a powerful tool for optimization problems by imitating the mechanisms of natural selection and genetics. These operate on a population of likely solutions applying the principle of survival of the fittest to produce better and better approximations to a solution. In each generation, a new set of approximations is created by selecting the individuals according to their level of
fitness in the problem domain and combining them together using operators taken from natural genetics. Thus, the population of solutions is consecutively improved with respect to the search objective by replacing least fit individuals with new, better suited to the environment, just like as in natural evolution [32][33].

Genetic algorithms are a subclass of evolutionary algorithms where the elements of the search space $G$ are binary strings ($G = B^*$) or arrays of other elementary types. As shown in figure 3.11, the genotypes are used in the reproduction operations whereas the values
of the objective functions \( f \in F \) are computed on basis of the phenotypes in the problem space \( X \) which are obtained via the genotype-phenotype mapping [50].

According to Goldberg, genetic algorithms are different from other optimization and search procedures in four ways [28]:

- GAs operate with coded versions of the problem parameters rather than parameters themselves i.e., GA works with the coding of solution set and not with the solution itself.
- Almost all conventional optimization techniques search from a single point but GAs always operate on a whole population of strings i.e., GA uses population of solutions rather than a single solution for searching. This plays a major role to the strength of genetic algorithms. It improves the chance of reaching the global optimum.
- GA uses fitness function for evaluation rather than derivatives. As a result, they can be applied to any kind of continuous or discrete optimization problem.
- GAs use probabilistic transition operates while conventional methods for continuous optimization apply deterministic transition operates i.e., GAs does not use deterministic rules.

Each individual in the population set, also called chromosome is represented by a string. A string chromosome can either be a fixed-length tuple or a variable-length list. String chromosomes are normally bit strings, vectors of integer numbers, or vectors of real numbers. The string is formed by a number of sub-strings equal to the number of the problem’s variables. Thus, a chromosome is subdivided into genes. A gene is the GA’s representation of a single factor for a control factor. Each factor in the solution set corresponds to gene in the chromosome. Each variable is coded in an appropriate coding system like binary, integer, real-valued, etc. The population size and the chromosome size are kept constant during the whole search process.
For Binary genetic algorithm, if the chromosome has $N_{\text{var}}$ variables given by $p_1$, $p_2$, $p_3$, ….,$p_{N_{\text{var}}}$ then the chromosome can be written as an $N_{\text{var}}$ element row vector

$$\text{Chromosome} = \left[ p_1, p_2, p_3, \ldots, p_{N_{\text{var}}} \right]$$

The fitness function is function of chromosome parameters like:

$$\text{Cost} = f(\text{chromosome}) = f(p_1, p_2, p_3, \ldots, p_{N_{\text{var}}})$$

The performance of each individual chromosome in the population is evaluated through an objective function. The objective function models the dynamic problem and produces as output a fitness value. For calculating fitness, the chromosome has to be first decoded and the objective function has to be evaluated. The fitness not only indicates how good the solution is, but also corresponds to how close the chromosome is to the optimal one. The fitness value determines how good the respective individual is with respect to the objective of the problem [31].

The two important aspects of population used in genetic algorithms are [1]:

1. The initial population generation.
2. The population size.

The size of the population is dependent on the complexity of the problem. Normally, the initial population is a random initialization. The size of the population gives rise to few problems too. The larger the population is, the easier it is to explore the search space. But it has been established that the time required by a GA to converge is $O(n\log n)$ function evaluations where $n$ is the population size. We say that the population has converged when all the individuals are very much similar and further progress may only be possible
by mutation. Large population is quite useful. But it requires much more effort in terms of computational cost, memory and time.

**Breeding in GA:**

The breeding process is the heart of the genetic algorithm. During this process, new individuals are created.

The breeding cycle consists of three steps [28]:

- Selecting parents,
- Crossing the parents to create new individuals - offspring or children and,
- Replacing old individuals in the population with the new ones.

### 3.5.1 Selection:

Selection is the process of choosing two parents from the population for the purpose of crossing. The purpose of selection is to put emphasis on fitter individuals in the population in hopes that their offspring will have higher fitness. Chromosomes are selected from the initial population to be parents for reproduction. The problem is how to select these chromosomes. According to Darwin’s theory, the best ones continue to exist to create the offspring. Selection is a method that randomly picks chromosomes out of the population according to their evaluation function. The higher the fitness function, the more chance an individual has to get selected. It is the pressure of selection that drives the GA to ultimately improve the population fitness over generations. Figure 3.12 explains basic selection process. The convergence rate of GA is mostly determined by the proper selection process.

The selection function can be any increasing function, but we will concentrate on fitness-proportionate selection, whose selection function is the probability function

\[
P_s (x_i) = \frac{\text{f} (x_i)}{\sum_{k=1}^{n} \text{f} (x_k)}
\]

on the population \{x_1, ..., x_n\}. 
The selection types can be categorized as:

- Proportionate selection – the selection takes place on the fitness value of individual relative to the fitness value of other individual.
- Ordinal-based selection - the selection takes place not only on their fitness value but on the overall rank of fitness in a population.

![Figure 3.12 Basic selection process in GA](image)

Too strong selection will lead to sub-optimal solution and reduces the variety needed, while too weak selection will lead to the result of too slow evolution.

There are various selection methods.

### 3.5.1.1 Roulette Wheel Selection:

It is one of the traditional GA selection techniques. The principle of roulette wheel selection is a linear search through a roulette wheel; each individual is assigned a piece of the roulette wheel, the size of the piece is proportional to the individual’s fitness. The wheel is rotated \(N\) times, where \(N\) is the number of individuals in the population. On each rotation, the individual under the wheel’s marker is selected to be in the group of parents for the next generation. This is only a fairly strong technique, because there is no guarantee of best individual getting selected, but has a greater chance of selection. It is easy to implement Roulette Wheel Selection method. The rate of evolution is dependent on the variance of fitness values in the population set of individuals [32].
3.5.1.2 Random Selection:

This technique randomly selects a parent from the population. It is a little more disruptive, on average, than roulette wheel selection.

3.5.1.3 Rank Selection:

The Roulette Wheel method has problems when the fitness values differ by a very large amount and may lead to very quick convergence. Under this situation, rank selection method slows down the convergence by preserving the variety in the population. It ranks the population and every chromosome receives fitness from the ranking. The worst has fitness 1 and the best has fitness N. It preserves variety and thus leads to a successful search. In effect, potential parents are selected and a tournament is held to decide which of the individuals will be the parent.

3.5.1.4 Tournament Selection:

It is the most balanced strategy in terms of getting convergence and also maintaining variety in the population. This strategy provides selective pressure by holding a tournament competition among individuals. The best individual from the tournament is the one with the highest fitness, which is the winner. Tournament competitions and the winner are then inserted into the mating pool. The tournament competition is repeated until the mating pool for generating new offspring is full. It is the fitness difference that provides the selection pressure. This method is more optimal and leads to optimal solution.

3.5.1.5 Boltzmann Selection:

Simulated annealing is a method of maximizing or minimizing a function. This method simulates the process of metallurgy, where in slow cooling of molten metal to achieve the minimum function value in a minimization problem. In this strategy, a continuously varying temperature controls the rate of selection according to a predetermined schedule. The temperature starts out at high value,
which means the selection pressure is low. The temperature is gradually lowered, which gradually increases the selection pressure, thus allowing the GA to narrow down more closely to the best part of the search space while maintaining the appropriate degree of variety. The probability of best individual selected and put in the mating pool is very high.

3.5.1.6 Stochastic Universal Sampling:

This method provides zero bias and minimum spread. The individuals are mapped to adjacent segments of a line, such that each individual’s segment is equal in size to its fitness exactly as in the case of roulette-wheel selection. Here, equally spaced pointers are placed over the line, as many as there are individuals to be selected. This method ensures selection of an offspring, which is closer to what is deserved than the Roulette Wheel selection method.

3.5.2 Crossover (Recombination):

Crossover is the process of taking two parent solutions for mating and producing from them a child. After the selection process, the population is full of better individuals. Reproduction actually creates a copy of good strings but does not create new ones. Crossover operator is applied to the mating pool with the hope that it creates a better offspring [34]. Crossover is a recombination operator that proceeds in three steps:

i. The reproduction operator selects randomly a pair of two individual strings for the mating.

ii. A cross location is selected at random along the length of the string.

iii. And then, the position values are swapped between the two strings form the cross point.

There are various crossover techniques used and they are as follows:

3.5.2.1 Single Point Crossover:

The traditional genetic algorithm uses single point crossover method. In this method, a crossover point is selected randomly, where the two mating
chromosomes are cut once at corresponding points and the sections after the cuts are interchanged.

Figure 3.13 explains single point crossover.

\[
\begin{array}{c}
\text{Parent 1} & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\
\text{Parent 2} & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \\
\end{array}
\]

\[
\begin{array}{c}
\text{Child 1} & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\
\text{Child 2} & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
\end{array}
\]

Figure 3.13 Single point crossover

3.5.2.2 Two Point Crossover:

In this method, instead of crossing over at one point only, the cross over takes place at two points and the contents between these points are exchanged between two parents. The addition of more points of crossover leads to disorder of building blocks, but the advantage is that it allows systematic search of problem space. Two point crossover is generally considered better than single point crossover. Figure 3.14 explains two point crossover.

\[
\begin{array}{c}
\text{Parent 1} & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\
\text{Parent 2} & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{c}
\text{Child 1} & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\
\text{Child 2} & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\
\end{array}
\]
3.5.2.3 Multi-Point Crossover (N-Point crossover):

It is an extension of two point crossover method. There are two ways in this crossover. One is even number of cross-sites and the other odd number of cross-sites. Random selection of cross sites around a circle takes place and information is exchanged.

3.5.2.4 Uniform Crossover:

This method is quite different form one Point or two Point crossover methods. Each gene in the offspring is created by copying the corresponding gene from one or the other parent according to the randomly generated crossover mask. The crossover mask is of the same length as of the parents. As can be seen from figure 3.11, while producing child 1, when there is a 1 in the mask, the gene is copied from the parent 1 else from the parent 2. While producing child 2, when there is a 1 in the mask, the gene is copied from parent 2, when there is a 0 in the mask; the gene is copied from the parent 1.

Figure 3.15 explains uniform crossover.

| Parent 1 | 1 0 1 1 0 0 1 1 |
| Parent 2 | 0 0 0 1 1 0 1 0 |
| Mask     | 1 1 0 1 0 1 1 0 |
| Child 1  | 1 0 0 1 1 0 1 0 |
| Child 2  | 0 0 1 1 0 0 1 1 |

Figure 3.15 Uniform crossover

3.5.2.5 Three Parent Crossover:
In this technique, instead of two parents, as the name suggests, three parents are randomly chosen. Each bit of the first parent is compared with the bit of the second parent. If both are the same, the bit is taken for the offspring otherwise; the bit from the third parent is taken for the offspring.

Figure 3.16 explains three parent crossover.

| Parent 1 | 1 1 0 1 0 0 0 1 |
| Parent 2 | 0 1 1 0 1 0 0 1 |
| Parent 3 | 0 1 1 0 1 1 0 0 |
| Child    | 0 1 1 0 1 0 0 1 |

Figure 3.16 Three parent crossover

3.5.2.6 Crossover with Reduced Surrogate:

This is implemented by restricting the location of crossover points such that crossover points only occur where gene values differ.

3.5.2.7 Ordered Crossover:

This method is used when problem is of order based. Two parents with two random points of crossover are selected. The crossover takes place in parent one according to the ordering of bits in parent two. Similarly crossover takes place in parent two according to ordering of bits in parent one. Figure 3.17 explains ordered crossover.

| Parent 1 | 4 2 1 3 6 5 | Child 1 | 4 2 3 1 6 5 |
| Parent 2 | 2 3 1 4 5 6 | Child 2 | 2 3 1 4 5 6 |

Figure 3.17 Ordered crossover

3.5.3 Mutation:
The next operation after crossover is the mutation. The mutation operator prevents the algorithm to get stuck in a local minimum [28]. Mutation plays the role of recovering the lost genetic materials as well as for randomly unsettling genetic information. We can say that it is an insurance policy against the irreversible loss of genetic material. Traditionally, mutation has been considered as a simple search operator. If crossover is thought to take advantage of the current solution to find better ones, mutation is believed to help for the exploration of the whole search space. Mutation is generally viewed as a background operator to maintain genetic diversity in the population. It introduces new genetic structures in the population by randomly modifying some of its building blocks. Mutation generally helps to escape from local minima’s trap and maintains variety in the population. For different kinds of representation, there are different forms of mutation. Mutation of a bit involves flipping a bit, changing 0 to 1 and vice-versa. The probability of mutation is normally considered as 1/L, where L is the length of chromosome. Various mutation techniques are:

3.5.3.1 Flipping:

Bit flipping involves changing 0 to 1 and 1 to 0 based on a mutation chromosome generated. The parent chromosome under consideration bits are flipped based on the bits of a randomly selected mutation chromosome. Following figure 3.18 explains bit flipping.

<table>
<thead>
<tr>
<th>Parent</th>
<th>1 0 1 1 0 1 0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mutation Chromosome</td>
<td>1 0 0 0 1 0 0 1</td>
</tr>
<tr>
<td>Child</td>
<td>0 0 1 1 1 1 0 0</td>
</tr>
</tbody>
</table>

Figure 3.18 Flipping of bits

3.5.3.2 Interchanging:

Two random positions of the string are chosen and the bits corresponding to those positions are interchanged. Following figure 3.19 explains interchanging—where the bits interchanged are shown in bold.

| Parent | 1 0 1 1 0 1 0 1 |
Materials and Methods

3.5.3.3 **Reversing**:

A random position is chosen and the bits next to that position are reversed and child chromosome is produced. Following figure 3.20 explains reversing.

![Figure 3.20 Reversing of bits](image)

All genetic operators are implemented with a certain predefined probability.

**3.5.4 Genetic algorithm cycle:**

![Figure 3.21 Genetic algorithm cycle](image)

In addition to moving ahead search by exploitation, genetic algorithms offer exploration of the search space by providing large and undetermined jumps in the search space. Exploration is achieved by combining elements from two existing solutions to
create a third, which may represent a new point in the search space. This new point actually is a window of hope and allows the method to actually search space that may otherwise be left untouched if the search method is purely exploitive in operation. Figure 3.21 explains the genetic algorithm cycle.

By maintaining a set of solutions in parallel, and through the explorative form of genetic algorithms, they are able to dig for the search space in quite a few locations at the same time. This makes genetic algorithms less local in its nature, giving them the quality of robustness. We can define robustness as consistency in finding optimal or if not optimal near optimal solutions, and the consistency in the quality of these solutions. It is precisely this reason that has encouraged my research into genetic algorithms and its applications in game playing. Figure 3.22 show the flowchart of genetic algorithm.

Let \( H \) be schema template made of 1s, 0s and *s, where * is used as a wildcard. \( H = 1 * * 0 * 0 \) is a schema. The number of non * or defined bits in a schema is called its order and denoted by \( o(H) \). The greatest distance between two defined bits is the defining length \( d(H) \). Let \( S \) be the set of all strings of length 1. Let \( m(H, t) \) be the number of instances of \( H \) at time \( t \). Let \( f(x) \) represent the fitness of chromosome \( x \), and \( f(t) \) represent the average fitness at time \( t \), or

\[
\bar{f}(t) = \frac{\sum_{x \in S} f(x)}{n}
\]

Where \( n = | S | \). Let \( \hat{u}(H, t) \) represent the average fitness of instances of \( H \) at time \( t \), or

\[
\hat{u}(H, t) = \frac{\sum_{x \in H} f(x)}{m(H, t)}
\]

If we completely ignore the effects of crossover and mutation, then the expected value is
If we consider the effects of crossover and mutation, we can get a good lower bound on \( E(m(H, t+1)) \). We get the schema theorem proved by Holland,

\[
E(m(H, t+1)) = n \frac{\sum_{x \in H} f(x)}{\sum_{x \in S} f(x)} = \frac{\sum_{x \in H} f(x)}{f(t)} = \frac{\hat{u}(H, t)m(H, t)}{f(t)}
\]

3.5.5 Search Termination:

In GA, the convergence criteria of algorithms can be like following conditions [34] [35]:

- **Maximum generations:**
  
  The algorithm stops after specified number of generations is over.

- **Elapsed time:**
  
  To give response within certain time limit this condition is used. If the specified number of generations is over, naturally the process will terminate.

- **No change in fitness:**
  
  It is useful when there is no change in fitness values for a specified number of generations.

- **Stall generations:**
  
  It is useful when there is no improvement observed in objective function for a number of sequential generations.

- **Stall time limit:**
  
  It is useful when there is no improvement in objective function during an interval of time.
3.5.6 Constraints:

If the genetic algorithm deals with only the objective function, without information about the specification of variable, then it is called unconstrained
optimization problem. While in the case of constrained optimization problem, information about variables under consideration is provided either as equality or inequality relation.

3.5.7 Advantages and limitations of Genetic algorithm:

Some of the advantages of Genetic Algorithm are [28][31]:

- Parallelism
- Solution space is wider
- The fitness landscape is complex
- Easy to discover global optimum
- The problem has multi objective function
- Only uses function evaluations.
- Easily modified for different problems.
- Handles large, poorly understood search spaces easily
- They are resistant to becoming trapped in local optima
- They perform very well for large-scale optimization problems
- Can be employed for a wide variety of optimization problems

Some of the limitations of Genetic Algorithm are:

- The problem of identifying fitness function
- Definition of representation for the problem
- Premature convergence occurs
- The problem of choosing the various parameters like the size of the population, mutation rate, cross over rate, the selection method and its strength.
- Have trouble finding the exact global optimum
- Require large number of fitness function evaluations
- Configuration is not straightforward
3.6 NATURAL OPTIMIZATION ALGORITHMS:

Genetic algorithm is not the only optimization algorithm that models the natural processes. There are other optimization algorithms which are used for global optimization.

3.6.1 Particle Swarm Optimization:

Particle Swarm Optimization (PSO) was formulated by Edward and Kennedy in 1995. The idea behind the algorithm was inspired by the social behavior of animals, such as bird flocking or fish schooling [36].

It similar to GA in that it begins with a random population. But it does not have operators such as crossover and mutation. The rows in population matrix are called as particles. The particles contain the variable values and are not binary encoded. Each particle of the population moves around the cost surface with a velocity.

In the initialization phase of Particle Swarm Optimization, the positions and velocities of all individuals are randomly initialized. In each step, first the velocity of a particle is updated and then its position. Therefore, each particle p has a memory holding its best position best(p) ∈ G.

The particles update their velocities and positions based on the local and global best solutions. The algorithm updates the vector known as velocity vector for each particle and adds that velocity to the particle position or values. The update of velocity vector is decided by the best global solution related with the lowest cost ever found by a particle and the best local solution related with the lowest cost in the current population. The advantage of PSO is that it is easy to implement and there are few parameters to adjust. The PSO is able to tackle the problem of local minima. As the generations pass, the particle swarming becomes apparent [37].
3.6.2 Ant Colony Optimization:

Ants can find the shortest path to food by laying a certain type of chemical path as they walk. Other ants simply follow the chemical path to find out the food. Ants that which are able to find the shorter path will create a strong path of chemical faster than the ants who are choosing a longer path. Since stronger type of chemical path attracts ants in a better way, more and more ants choose the shorter path until eventually all ants have found the shortest path. The first ant colony optimization algorithm was designed to solve the traveling salesman problem, because this most likely resembles finding the shortest path to a food source [30]. The Ant colony optimization (ACO) algorithm performed better only after coupling it with a local optimizer. One problem with this algorithm is premature convergence to a less than optimal solution because of too much too much virtual chemical was laid down. To come out of this problem a phenomenon of chemical evaporation is used. This algorithm is natural for Traveling Salesman Problem (TSP). It starts with a number of ants that follow a path around the different cities. Each ant deposits a chemical along the path it moves on. Initially, each ant is assigned a city selected randomly. The next city is selected based on the weighted probability which is a function of the strength of the chemical laid on the path and the distance of the city. ACO is found to do well on a Traveling Salesman Problem (TSP) with 30 cities.

3.7 LEARNING:

Learning AI has the potential to become accustomed to each player, learning their tricks and techniques and providing a reliable challenge. It has the potential to produce more realistic characters in terms of behavior, which can learn about their surroundings and use it to the best effect. There are different learning techniques, starting from very simple number alteration through to complex neural networks. Learning techniques can be classified based on when the learning actually occurs, what is learned and the effects of learning on behavior.
3.7.1 Online or Offline Learning:

When learning takes place while the player is playing, it is called on-line learning. It allows the character to become accustomed dynamically to the players style and provide more consistent challenges. As a player plays more, his characteristic qualities can be better predictable by the computer, and the behavior of characters can be tuned to the playing styles. But, it faces the problems of predictability and testing.

The majority of learning in game AI is done offline, either between levels of the game or mostly at the time of development in development studio before the game leaves. This is carried out by processing data about real games and trying to work out strategies or parameters from them. The learning algorithms in games are usually applied offline; it is uncommon to find games that use any kind of online learning. Learning algorithms are more and more being used offline to learn deliberate features of multiplayer maps, to produce accurate pathfinding and movement data.

3.7.2 Intra-behavior Learning:

This is the simplest kind of learning in which a small region of character behavior is altered. It does not change the whole quality of the behavior. These techniques are easy to control and also easy to test. The front line for learning AI in games is learning of behavior. Over a time, an increasing amount of character behavior may be learned, either online or offline. Some of this may be to learn how to choose between ranges of different behaviors. The learning can be supportive in decision making.

3.7.3 Parameter Landscape:

The simplest learning algorithms are those that calculate the value of one or more parameters. Numerical parameters are used throughout the development of AI. These values can very often have a large effect on the behavior of a character. A small change in a decision making probability, can lead an AI into a very special approach of play. Most commonly, this is done offline.

A common way of understanding parameter learning is the fitness or energy value. For each value of the parameter there is some energy value. This energy value often called a
“fitness value” in some learning techniques which represents how good the value of the parameter is for the game. We can visualize the energy values by plotting them against the parameter values. Figure 3.23 show the fitness value for a one dimensional parameter. The purpose of parameter learning system is to find the best values of the parameter. The energy landscape model usually assumes that low energies are better, so we try to find out the valleys in the landscape. While, fitness landscapes are usually the opposite, so they try to find the peaks in stead of valleys.

It is very clear that the difference between energy and fitness landscapes is an issue of terminology only: the same techniques apply to both. We need to simply swap searching for maximum (fitness) or minimum (energy).

### 3.7.4 Action Prediction in Learning:

Many a times it is useful to be able to predict what players will do next. Normally, humans are bad at behaving randomly. Psychological research has been carried out over decades and shows that humans cannot precisely randomize their responses, even if they specifically try to do. Mind magicians and expert poker players make use of this research. They can often easily work out what the human will do next or think next based on a fairly small amount of experience of what we have done in the past. The action forecast can be in terms of:
• Choice - In the context of game, it means the choice the player will make given a list of choices.

• Raw Probability - The simplest way to predict the choice of a player is to keep a count of the number of times he chooses each option. This will then form a raw probability of that player choosing that action again. When the choice is made only once, then this kind of prediction may be all that is possible. If the probabilities are gained from many different players, then it can be a good indicator of which way a new player will behave.

• String Matching - When a choice is repeated several times, a simple string matching algorithm can provide good estimate. The sequence of choices made is stored as a string, it can be a string of numbers or objects, not just a string of characters. In the left-and-right game this may look like LRRLRLLLLRLRRLRR, for example. To predict the next choice, the last few choices are searched for in the string, and the choice that normally follows is used as the prediction. How many last choices we use is termed as window size. For example, a window size of 2 means that we look at the last two choices and predict what can be the next choice by looking at the pattern.

• N-Grams - In an N-Gram we keep a record of the probabilities of making each move given all combinations of choices for the previous N moves. Here, N is one greater than the window size parameter, so 3-Gram will be a predictor with a window size of 2. By far the most common application of N-Gram prediction is in warfare games. Sword combat games and any other combo-based fight games involve timed sequences of moves. Using an N-Gram predictor allows the AI to predict what the player is trying to do as they start their sequence of moves. It can then select an appropriate contradiction.
3.7.5 Machine Learning in Games:

In chapter 1, a mention was made about artificial intelligence techniques in game playing. As was explained in Chapter 1, many game playing programs highly depend on knowledge to increase the accuracy of their evaluation function. Domain experts can provide some of this knowledge, but however there remains a significant gap between the positional judgment of the best humans and the ability of knowledge engineers to summarize that opinion in the form of a heuristic evaluation function.

A totally different approach is to allow a machine learn its own domain specific evaluation function. If a human can learn to master the game, perhaps so can be a machine. A program might for example learn to evaluate a position, or learn to evaluate a move given a position. The decision learning techniques are as follows:

- Naïve Bayes Classifiers
- Decision Tree Learning
- Supervised learning
- Reinforcement learning

3.7.5.1 Naïve Bayes classifiers:
Bayesian learning simply calculates the probability of each hypothesis or assumption, given the data, and makes predictions on that basis. So, prediction is based on all the hypothesis, which are weighted by their probabilities. The model is naïve because it assumes that the attributes are conditionally independent of each other, given the class. It is simple to implement and provides a good baseline for any more complicated techniques. It uses the concept of conditional probability. It uses the Bays rule as:

\[
P (A |B) = \frac{P(B|A)P(A)}{P(B)}
\]

3.7.5.2 Decision Tree Learning:
In decision trees, a series of decisions which generate an action to be taken based on a set of observations. At each branch of the tree some part of the game is considered and a different branch is chosen. Ultimately, the series of branches lead to an action.

ID3 is the well known decision tree algorithm. ID3 stands for “Inductive Decision tree algorithm 3” or “Iterative Dichotomizer 3.” It is simple to implement, relatively efficient decision tree learning algorithm. The basic ID3 algorithm uses the set of observation i.e. action examples. Observations in ID3 are usually called “attributes”. The algorithm starts with a single leaf node in a decision tree and assigns a set of examples to the leaf node.

It then splits its current node so that it divides the examples into two groups. The division is chosen based on an attribute, and the division chosen is the one that is likely to produce the most efficient tree. When the division is made, each of the two new nodes is given the subset of examples that applies to them, and the algorithm repeats for each of them. It uses the concepts of entropy and information gain. Entropy is a measure of the information in a set of examples. Information gain is simply the reduction in overall entropy.

3.7.5.3 Supervised learning:

The algorithm relies on the availability of examples. The examples can be hand built or generated from experience. In game playing, such an example could be the best move from a certain position, or the position's utility value. These types of examples are used in the back propagation step to generate the error term. The error term then controls the learning process. This is called as supervised learning. Fundamentally, an example is a pair of signals - an input signal, and a signal containing the desired output. The learning task is to learn a function that given the input, produces the preferred output. But, this method can be used only when a large amount of labeled training data is available. Care need to be taken when selecting training data, as a supervised learning system has the limitation that it can do only as well as the examples from which it learns.
3.7.5.4 Reinforcement learning:

Reinforcement learning is the name given to a range of techniques for learning based on experience and practice. This method of learning has been studied by animal psychologists for many years. In its most universal form a reinforcement learning algorithm has three components: a searching strategy for trying out different actions in the game, a reinforcement function that gives feedback on how good each action is, and a learning rule that associates the two jointly.

It functions on the principle of punishment and reward. Punishment and reward can be used to guide an animal's behavior. Application of this model onto games leads to an economical view of a game - By winning the game a reward is earned, while losing the game leads to punishment, or negative reward.

Q-learning algorithm is an example of reinforcement learning. Q-learning treats the game environment as a state machine. Q-learning is known as a model-free algorithm because it does not try to build a model of how the world works. It simply treats everything as states. Model-free algorithms, such as Q-learning, have a tendency to be notably easier to implement. Q-learning is named for the set of quality information, called Q-values, it maintains about each possible state and action. The algorithm keeps a value for every state and action it has tried. The Q-value represents a measure of how good it thinks that action is to take when in that state. The important parameters of the algorithm are learning rate and discount rate. The learning rate controls how much influence the current feedback value has over the stored Q-value. The learning rate range is \([0, 1]\). The discount rate control how much an action’s Q-value depends on the Q-value at the state it leads to. The discount rate range is \([0, 1]\).

In games, reinforcement or punishment is usually obtained after many moves, at the very end part of the game - the game either ends in a win, loss, or draw. The task of reinforcement learning is to use these delayed rewards to improve the quality of play.
This is really a difficult task, since it is often not clear which moves contributed to the outcome of the game. In a game that ends in victory, it is possible that there may still have been some bad moves, and playing very well except for one small mistake might lose the game. The most popular technique for learning from delayed reinforcements is Temporal-Difference learning (TD). TD-learning is often used to learn to evaluate positions.

3.8 TIRES OF GAME ARCHITECTURE:

The general game architecture consists of some tiers. There are four tiers of the game architecture [42]. Following figure 3.24 shows the four tiers of general game architecture.

1. I/O devices: A large amount of data from input and output devices such as keyboard, mouse, joystick, monitor or display unit are generated.
2. Device APIs: The data stream generated needs to be processed by one or more engines, such as graphics engine, input manager etc.
3. Main: Most of the CPU processing time is used on marshalling the processed data from the game engines and performing general purpose logic. The artificial intelligence and game play code provide a great challenge in tuning the parameters. These are responsible to incorporate game features.
4. Data: The data which is processed by various engines needs to be streamed out to the video console for creating colorful graphics. This process is called write streaming or rendering.

3.9 DETAILS OF GAME GO-MOKU:

Go-Moku is a zero sum, deterministic, perfect information, two-player board game. It is played on 15*15 or 19*19 boards. The objective of the board game is to get five stones or pieces in a row, be it horizontal, vertical or diagonal. In Go-Moku, simple rules lead to a highly complex game, played on the intersections of horizontal and vertical lines. Going from left to right the vertical lines are lettered from ‘a’ onwards; going from the bottom to the top the horizontal lines are numbered from ‘1’ onwards. Two players, black and white, move their pieces in turn by placing a stone of their own color on an empty intersection, called a square. Black starts the game. The player who first makes a line of five consecutive stones of his/her color (horizontally, vertically or diagonally) wins the game. The stones once placed on the board during the game never move again nor can they be captured. If the board is completely filled, and no one has five-in-a-row, the game ends in a draw. In some games there is a restriction that the black’s very first move must be in the center of the game board. White's first move can be anywhere. Black's second move must be outside the 5 by 5 area in the center of the game board. White's second move, and all moves after that, can be anywhere.

In a perfect-information game, all players have access to all information defining the game state and its possible continuations. A sudden-death game may end abruptly by the creation of one of a pre specified set of patterns. Go-Moku is an example of a sudden-death game. The game is terminated if one of the players has created a line of five stones with his or her color of piece. Sudden-death games need not always terminate through the creation of a sudden-death pattern. Go-Moku is declared a draw when all squares on the board are occupied without either player creating a winning pattern of consecutive five.

The state-space complexity of a game is defined as the number of legal game positions reachable from the initial position of the game. For each of the board position, there are
three possibilities- black, white or empty. For a Go-Moku board of 19* 19 size, Thus, an upper bound to the state-space complexity is $3^{361}$.

The main application of the state-space complexity of a game is that it provides a bound to the complexity of games which can be solved through complete listing.

The game-tree complexity is an approximation of the size of a mini-max search tree which must be built to solve the game.

3.9.1 Important Constructs in Go-Moku:

The special constructs in Go-Moku are also known as threats. These threats are very important for playing the game, as they force the opponent to move a piece at a particular location in order to avoid lose or losing situation immediately. The various threats are: Open Four, Four, Three, and Split Three etc shown in figure 3.25.

**Open four:**

There are several constructions in the game which happen frequently and therefore have been given names by players. An open four means an immediate win for the player who makes it unless the opponent player can make five in a row. There are two places where a stone can make five in a row, while the opponent can block only one.

**Four:**

This is almost the same as an open four. The only difference is that the four is already blocked on one side, so it is not an immediate win. But the opponent's next move is compulsory, so the four is frequently used in a sequence of moves which allows you to win.

**Three:**

The three consists of three pieces in a row. If the three is open, the player threatens to make an open four, so his opponent must do something against this.
Split Three:

Another form of the three is the split three. The split three is also three in a row, but between two pieces there is an empty place. Here the player also threatens to make an open four, so his opponent must stop this, either by putting a piece in the middle, or on one of the sides. The first option is most of the time the best, because if you block on the side, your opponent can create a four which gives your opponent an extra tempo.

In Go-Moku, a threat is an important structural idea; the main types have descriptive names: All the threats are explained in one figure 3.26 the four (figure -a) is defined as a
line of five squares, out of which the attacker player has occupied any four, with the fifth square empty; the open four (figure b) is a line of six squares, out of which the attacker player has occupied the four center squares, while the two outer squares are empty; the three (figure c and d) is either a line of seven squares of which the three center squares are occupied by the attacker player and the remaining four squares are empty, or a line of six squares with three consecutive squares of the four center squares occupied by the attacker player and the remaining three squares empty; the split three (figure e) is a line of six squares of which the attacker player has occupied three non-consecutive squares of the four center squares, while the other three squares are empty. A winning pattern, i.e., a line of five squares, all occupied by one player, is named a five.

Figure 3.27 Complicated Threats

Figure 3.27 shows some of complicated threats which can develop in the course of a game, which threaten to win in two or more moves. Three examples are shown, in each of which black threatens to play at the intersection of the two lines of black stones. In figure (3.27a), black threatens to create a double four, in figure (3.27b), black threatens to create a four-three, and in figure (3.27c), black threatens to create a double three. Each of these is a winning pattern. White can counter the threats shown above in 3, 4 and 5 possible ways, respectively [18].
3.9.2 Design Overview of Go-Moku:

The program of Go-Moku is designed and developed in Java with GUI and mouse interface. The program presents a choice to the user for different options. The board is shown as 15 * 15 size, where the user can interact with. The user can select the first player to move and accordingly the moves are displayed. The moves of computer are automatically chosen using a genetic algorithm. The computer move is also displayed in terms of row and column value to help the user. The panel also displays whose turn now it is. When the game is over, it displays who is the winner. The user can start a new game at any time.

Figure 3.28 Go-Moku game initial window

Figure 3.28 shows the initial window where the user can interact with it. The window shows initial board configuration. The board is represented in center and on both sides of it there is text field to display some information to user. At the bottom is information panel, where whose turn it is, is displayed. At the top is “New Game” menu which when clicked displays various options to the user. When the menu “New Game” is clicked, it
shows various menu options like – computer first, human first, human vs. human and Exit options.

Figure 3.29 shows the menu options.

![Menu options in Go-Moku window](image)

Figure 3.29 Menu options in Go-Moku window

Figure 3.30 shows the game window after some moves, where the human has selected the first i.e. black player move. It shows that the next move is of black i.e. human player and also shows that the last move of computer is X=7, Y=8 i.e. row 8 and column 7, numbering starts from top.
Following figure 3.31 explain the architecture [45] of the Go-Moku program.

Figure 3.31 Architecture of Go-Moku system
As shown in the architectural class diagram of the system, Go-Moku is the main class of the game program, where the main() method runs the application. The Go-Moku class internally uses the class WindowContent, which is responsible for GUI of the board, Genetic Algorithm and related functionality of the game like move generation, Graphics refresh, checking for game over and other functionality [44]. WindowContent class in turn uses the functionality of ComputerMover thread, Move class, Square class and Config class. Square class also uses the functionality of Config class. The detailed discussion of all the classes is given below.

The Config.java file contains Config class which has all the configuration parameters such as square height, width, and board size, population size for genetic algorithm. These parameters are used throughout the application.

<table>
<thead>
<tr>
<th>Config</th>
</tr>
</thead>
<tbody>
<tr>
<td>public static final int SQUARE_HEIGHT</td>
</tr>
<tr>
<td>public static final int SQUARE_WIDTH</td>
</tr>
<tr>
<td>public static final int BOARD_HEIGHT</td>
</tr>
<tr>
<td>public static final int BOARD_WIDTH</td>
</tr>
<tr>
<td>public static final long WAIT_BEFORE_NEXT_COMPUTER_MOVE</td>
</tr>
<tr>
<td>public static final int POPULATION_START_SIZE</td>
</tr>
</tbody>
</table>

The GoMoku.java file contains GoMoku class which has all the menu items and menu bar items. It is the main class to run the application. It also uses WindowContent object and also implements actionListener.

<table>
<thead>
<tr>
<th>GoMoku</th>
</tr>
</thead>
<tbody>
<tr>
<td>MenuItem computerStarts</td>
</tr>
<tr>
<td>MenuItem humanStarts</td>
</tr>
</tbody>
</table>
The Square.java file contains class Square. It contains the information about individual square of the board. It has parameters such as its position in the board, piece colour of the piece if occupied by any player, and mouse related parameters. The board is made up of squares. Actually, board is a two dimensional array of squares.

<table>
<thead>
<tr>
<th>Square</th>
</tr>
</thead>
<tbody>
<tr>
<td>private int xCoordinate</td>
</tr>
<tr>
<td>private int yCoordinate</td>
</tr>
<tr>
<td>public Color colorOfPiece</td>
</tr>
<tr>
<td>private boolean mouseOver</td>
</tr>
<tr>
<td>WindowContent parent</td>
</tr>
<tr>
<td>boolean mouseClicked</td>
</tr>
<tr>
<td>public void addPiece(Boolean)</td>
</tr>
<tr>
<td>public void removePiece()</td>
</tr>
<tr>
<td>public boolean isOccupied()</td>
</tr>
<tr>
<td>public void mousePressed(MouseEvent)</td>
</tr>
<tr>
<td>public void mouseReleased(MouseEvent)</td>
</tr>
<tr>
<td>public void mouseEntered(MouseEvent)</td>
</tr>
<tr>
<td>public void mouseExited(MouseEvent)</td>
</tr>
<tr>
<td>public void mouseClicked(MouseEvent)</td>
</tr>
<tr>
<td>public void update(Graphics)</td>
</tr>
<tr>
<td>public void paint(Graphics)</td>
</tr>
</tbody>
</table>
It has a method to add or remove a piece from given square position. Also it has a method to check whether a square is occupied or not. Mouse related events are also implemented. Whenever there is a change in the square a method refreshes the square.

The WindowContent.java file is the main file which contains the elements of the board such as board panel. It has parameters such as move count, who is black player, the board as an array of Square object. It also has all the parameters related with genetic algorithm operations for population, chromosomes, fitness values and weights for fitness functions. It also has methods to decide next move, make move, show move, new game start, update parameters, check for victory, and genetic algorithm related methods such as fitness evaluation of population, tournament selection, cross-over operation and mutation operation.

<table>
<thead>
<tr>
<th>WindowContent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Container boardPanel</td>
</tr>
<tr>
<td>TextField infoLabel</td>
</tr>
<tr>
<td>TextField weightLabel</td>
</tr>
<tr>
<td>TextField statusLabel</td>
</tr>
<tr>
<td>Square[][] board</td>
</tr>
<tr>
<td>int [][] pop</td>
</tr>
<tr>
<td>int [] fitValue</td>
</tr>
<tr>
<td>int [][] npop</td>
</tr>
<tr>
<td>int [][] weights</td>
</tr>
<tr>
<td>int moveCount</td>
</tr>
<tr>
<td>public boolean blackIsHuman</td>
</tr>
<tr>
<td>public boolean whiteIsHuman</td>
</tr>
<tr>
<td>private String black</td>
</tr>
<tr>
<td>private String white</td>
</tr>
<tr>
<td>public boolean show</td>
</tr>
<tr>
<td>boolean computerMove</td>
</tr>
<tr>
<td>boolean start</td>
</tr>
</tbody>
</table>
ComputerMoverThread is a thread [46] which is responsible for scanning a move for computer. Depending on whose turn it is, it scans for the next move of the computer. The methods scanOn() and scanOff() make scanOn variable ON and OFF respectively.
3.9.3 Window Content, New Game and Menu:

The board panel is displayed in the center of the window, the three Text Boxes are shown one on south, one on east and one on west side of the board panel [47]. The window shown is not resizable, nor can it be maximized. Whenever the user selects new game from menu and appropriate option, a new game starts. The initial board setup is displayed with move form computer if computer is the first player; otherwise it waits for the human to make the initial move. Whenever new game starts, it initializes the move count to zero, records the information such as who is the first player and also initializes the population for the genetic algorithm. It also initializes the weight values which are to be used in finding the fitness value of each chromosome in the population as the game progresses.

3.9.4 Square, Board and Pieces:

The game board is implemented as a two dimensional array of squares. Each square has its X and Y co-ordinate value in the board. The square can be filled with white piece, black piece or can be empty. The player having the turn fills the square with his colour of the piece. Whenever there is a move either by human or computer player, that move is done on the board with a function such as addPiece(). It checks if the square is occupied or not. If the square is empty it makes a move on to that square on the board. The argument to addPiece() is also the colour of the piece to move to that place. There is also a method removePiece() which does exactly the opposite work of addPiece(). Normally, removing a piece is not needed in the game. It is needed only when a user selects a new game menu option from the menu. The removePiece() method is used to present a fresh board to the user. It essentially clears all rows and columns of the board and presents a fresh new game board.

3.9.5 Genetic Algorithm Player:

Genetic algorithm player moves are dependent on situations. In the opening move, normally it plays at places which are usually played in the beginning, and then the moves are normally attacking i.e. putting pieces near to each other in order to create line of 5,
and forcing the human player to defend. It also finds out the threats paused by the enemy and tries to defend wherever required.

A special thread is continuously running to generate computer moves. It keeps track of when it is the turn of a computer. Move generation of the computer player takes place automatically. When it is the turn of a computer, the method adopted for move generation is the randomized parallel search algorithm i.e. genetic algorithm. The selection of the next move of a computer goes through various operations of genetic algorithm.

In the program, a certain size of population is selected. More the population size better is the move. But, too large a population size leads to very much delay in selection of the next move [48]. Each chromosome in the population represents a one possible move by computer. Which chromosome will be selected for move that depends on the fitness value for that chromosome. Higher the fitness value better is the move. Chromosome is a bit string. Each chromosome in the population is made up of 8 bits. Figure 3.32 shows the encoding of the structure of chromosome [28].

<table>
<thead>
<tr>
<th>Bit</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>

Figure 3.32 Chromosome structure

Bits 7 to 4 represent the Y(row) value, and bits 3 to 0 represent X(column) value. Thus 4 bits are used for X and 4 bits are used for Y value. So, there are $2^4 = 16$ possible values for the X and Y value. For a board of 15 * 15 size, the valid value for X and Y are from 0 to 14 only. If a chromosome stores a value 15 for X or Y, which is outside the valid range of 0 to 14, then it needs a special adjustment to avoid problem as far as mapping the value to game board of size 15 * 15.

As the game board is two dimensional, the population is used as a two dimensional array. The first dimension of population represents the population size and the second dimension represents the chromosome length or size. Genetic algorithm requires
initialization of the population to start the process. The initialization of population is carried out using randomization of bits of chromosomes every time the human player starts a new game. As the chromosome is a bit string, every bit in the chromosome in the population set is randomly initialized. As far as the first move of computer is concerned all the chromosomes are equal from fitness value point of view if computer is the first player.

Fitness function operates using following main considerations:

- The possibilities the move gives us to build rows of five stones in the future.
- Building up long sequences of three and four.
- Identify winning situation.

The general form of fitness function for board game is like:

\[ f(s) = w1* f1 + w2*f2 +w3 * f3 + \ldots \]

Where, \( f1, f2, f3 \) etc are usually based on human strategic knowledge of the game in terms of threats and opportunities in the game, while \( w1, w2, w3 \) etc are weightage associated with each \( f \) in the formula.

To play better, In the Go-Moku game, it is very important to identify various threat patterns developed on the board. It is very important to assign the correct weight values to various threats. In evaluating the fitness values for each chromosome in the population, the weights assigned to each type of threat plays very important role. The threats can be Two, Three or Four.

Following table 3.1 shows the weight values used in the game program for various threats that may develop in the game. The weight values for player and opponent for the same type of threat are different. Using the weights mentioned in the program to play the computer genetic algorithm player in an attacking mode. If we reverse the values for
player and opponent, then the computer genetic algorithm player will play in defensive mode.

<table>
<thead>
<tr>
<th>Player/Threat</th>
<th>One</th>
<th>Two</th>
<th>Three</th>
<th>Four</th>
</tr>
</thead>
<tbody>
<tr>
<td>Player</td>
<td>4</td>
<td>16</td>
<td>600</td>
<td>6000</td>
</tr>
<tr>
<td>Opponent</td>
<td>2</td>
<td>10</td>
<td>400</td>
<td>4800</td>
</tr>
</tbody>
</table>

Table 3.1 Weights table normal threats

In addition to threats of above type, some structures can easily be converted into more dangerous threats and need immediate attention to avoid possible loss. So, these types of threats are assigned more weight values compared to normal threats.

In the program, method called as updateBoardValues() checks for all possible threats developed in the board. Actually, method fitness() is called every time a new move is to be decided for the computer. The fitness() method internally uses updateBoardValues() method to find out the fitness value for all the chromosomes of the population. Given a board position on the board, the updateBoardValues() method checks whether that position is occupied or not. If it is occupied, then the fitness value returned is 0, otherwise it calculates the fitness value for that position taking into the consideration of board situation and the weights assigned to threats mentioned in the table. The method has to search for all threats that may occur horizontally, vertically or diagonally (both). It also has to check whether it is open type or closed type threat. As open threat is normally more dangerous, higher weight is assigned to open threat.

The weights assigned to various other types of threats are shown in table 3.2

<table>
<thead>
<tr>
<th>Player/Threat</th>
<th>Open (2+1) closed on one side</th>
<th>Open (2+1) Split Three</th>
<th>Open (2+2) or Open(3+1) Split Four</th>
</tr>
</thead>
<tbody>
<tr>
<td>Player</td>
<td>4000</td>
<td>4500</td>
<td>6000</td>
</tr>
<tr>
<td>Opponent</td>
<td>4000</td>
<td>4500</td>
<td>5000</td>
</tr>
</tbody>
</table>

Table 3.2 Other threats
The next move selection from the computer goes through the process of iteration for number of generations. In each generation, a fitness value is found out for each chromosome in the population. After finding out the fitness value, selection process takes place. In the program, I have used tournament selection, in which a tournament selection is carried out amongst chromosomes in a population. This method finds out better chromosomes from fitness point of view and discards the inferior chromosomes, thus creates a new population of chromosomes with better fitness values. Once the new population is ready, a crossover operation takes place. There are various methods of crossover. In the program a two-point cross over is used to generate possibly better chromosomes. The two points for crossover are not fixed but they are generated randomly, so as to cover all board positions without any bias.

Following algorithm generates two points k and l randomly for crossover. The points k and l will have value between 0 and 7.

Algorithm for generating two random points, for crossover.

| Step 1. k = Random number less than 2 |
| Step 2. Do while count <> 0 |
| Count =0 |
| L= Random number between 0 and 7(mb) |
| If ((k=0) and (l=0)) or ((k=l) and l=mb)) |
| Count = count +1 |
| Step 3. End |

After cross over operation between best fit chromosomes, the next operation carried out is mutation. As the genetic algorithm has the tendency to get trapped in local maxima, mutation operation is carried out to prevent possible trap in local maxima. There are various methods of mutation. The bit flipping method is used in the program. If a certain
condition on similarity of bits of two chromosomes in a population is satisfied, then a randomly selected bit position is flipped i.e. toggled in both the chromosomes. Probability of mutation is kept very low.

The selection of random bit number is once again through random number generation. We can use similar algorithm for generating it. The mutation operation flips the bit from 1 to 0, and 0 to 1. As the population is stored as a two dimensional array, it is basically an operation on matrix to flip the concerned bit.

The operation of any genetic algorithm is very much dependent on the parameters of the algorithm.

Following table 3.3 lists the important parameters and their value.

<table>
<thead>
<tr>
<th>Genetic Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Population size</td>
<td>500</td>
</tr>
<tr>
<td>Chromosome size</td>
<td>8 bits</td>
</tr>
<tr>
<td>Number of generations</td>
<td>100</td>
</tr>
<tr>
<td>Mutation probability</td>
<td>1.25%</td>
</tr>
</tbody>
</table>

Table 3.3 Genetic algorithm parameters

Mutation operation is carried out once after every 10 iterations, that too on 1 bit out of 8 bits of chromosome.

So, mutation probability = \((1/8) * (1/10) * 100\% = 1.25 \%
\)
Figure 3.33 shows the outline of the genetic algorithm adopted in the program.

1. **[Start]** Generate random population of \( n \) chromosomes (suitable solutions for the problem)
2. **[Fitness]** Evaluate the fitness \( f(x) \) of each chromosome \( x \) in the population
3. **[New population]** Create a new population by repeating following steps until the new population is complete
4. **[Selection]** Select two parent chromosomes from a population according to their fitness (the better fitness, the bigger chance to be selected) The idea is to choose the better parents.
5. **[Crossover]** With a crossover probability cross over the parents to form a new offspring (children). If no crossover was performed, offspring is an exact copy of parents.
6. **[Mutation]** With a mutation probability mutate new offspring at each locus (position in chromosome).
7. **[Accepting]** Place new offspring in a new population
8. **[Replace]** Use new generated population for a further run of algorithm
9. **[Test]** If the end condition is satisfied, stop, and return the best solution in current population
10. **[Loop]** Go to step 2

Figure 3.33 Outline of genetic algorithm

Figure 3.34 shows the X marked board position, which is a threat of type Open (2+1) and needs immediate attention, a move is required at that position, otherwise it will be converted in open Four and it will be a sure win for the human player.
Figure 3.34 Threat (2+1) type structure in game play Go-Moku

Figure 3.35 shows the move generated by computer after finding the above threat. The latest move of computer is shown using an arrow at row 7 and column 5.

Figure 3.35 Identification of threat shown in figure 3.30 to avoid loss
3.9.6 Checking for Game Over:

Before allowing the next move of the player whose turn it is, the system must check for the possibility of game end. So, at the end of every move of any player, the system checks for possibility of game end. In the WindowContent class, a method called as victoryCheck() checks for possible game end. The method checks all board positions which are occupied, and checks for the pattern of consecutive five pieces having the same colour as of the player who played last move. If such a pattern is found it returns appropriate value and appropriate message of game over is displayed, and also stops the thread which is responsible for generating the next move of computer. If no such pattern is found, it allows the move of the next player and the game continues in normal way.

Figure 3.36 shows the Open (3+1) threat developed at the X mark position on the board. The computer must move to that position to avoid loss.

![Figure 3.36 Threat developed](image)

Figure 3.37 shows the board where the computer player did not move at the Open (3+1) threat at X position shown, but instead it moves to row 7 and column 10. This allows the human player to move at row 5 and column 4. So, a pattern of consecutive five pieces of
the white player is found out by the victoryCheck() method and it displays game over message and game stops there.

Figure 3.37 Computer loses and white (Human) won the game

In addition to that, program also keeps track of the information and statistics on the performance of the genetic algorithm player. It keeps information about time required to generate each move of the computer. It also records the data about fitness values for each generation, sum and average value of fitness for each generation of genetic algorithm. It also keeps record of the result of the game in a separate file. All this information can be used for analysis of various genetic algorithm parameters and its effect on overall play of the game program.

3.10 DETAILS OF GAME OF OTHELLO:

The exact start of Othello are not known, although it is now believed that it has originated from the Chinese game ‘Fan Mian’, which translates to ‘reverse’, hence the alternative name Reversi has come up. Until recently the game has been primarily played in Japan, where it is the second most popular game next to in 1890’s a game called Game of Go.
Reversi was being marketed in England by Jaques and Sons of London. This game was very similar to the modern game except for two differences:

- Each player was given a set number of discs at the start of the game. If a player ran out of discs, his opponent completed the remainder of the game.
- The game began with an empty board.

In 1975, Goro Hasegawa introduced the modern version of Othello which has been adopted across the world. In the modern version, players take from a central pool of discs and the initial layout of four discs is provided. Othello is fast growing in popularity with major international tournaments held every year. It has also been a popular test-bed for AI research.

It is mostly attractive because of the simplicity of its rules. After a minute of introduction, one may start playing a game, and still the game has enough complexity to leave a dedicated person with years to master. The difficulty to visualize dramatic changes of the disk patterns on the board makes the game of Othello quite a challenge for human players.

Formally speaking, Othello is a two-person zero-sum deterministic finite board game with perfect information. Two-person zero-sum games are characterized by the fact that either one player wins and the other loses, or neither player wins resulting in a draw or tie. It is not possible for both players to win.

Deterministic games are games where there are no random events based on event of chance or luck such as the roll of a dice. In addition to that Othello is a finite game, because there are a finite number of moves: at most 60. Othello has perfect information, because all the game information such as piece position is available to all players.
3.10.1 Rules of the Game:

Othello is played on an 8 × 8 grid, using dual-colored disks. Every disk has one white and one black side. Like Go-Moku and chess, it is deterministic, perfect information, zero-sum game of strategy between two players, black and white. The figure 3.38 shows the standard notations used in Othello [6].

![Othello Board](image)

Figure 3.38 Standard notations of Othello

Certain squares of the game board assigned special letters, this notation was developed by the inventor of the game, Goro Hasegawa. The notations are shown in figure 3.39 [6]. The B-squares are in the center of the edge, the C-squares are on the edge next to the corner, and the A-squares lie between the B-squares and C squares. The X-squares are diagonally adjacent to the corners, with the ‘X’ indicating danger.

![Othello Board](image)

Figure 3.39 Othello board and letter assignment

Black opens the game from the initial board configuration shown in figure 3.40.
A legal move for a player is a placement of a piece on the board which results in the capture or trap of one or more opponent’s pieces. A capture occurs when a player places a piece of his color in a blank (empty) square adjacent to a line of one or more pieces of the opposing color followed by a piece of his color. Captured disks are flipped to the captor’s color.

Play continues until neither player has a legal move, which generally happens when the board is totally filled. At the end of the game, the pieces of each color are counted, and the player with the most number of pieces is declared as the winner. If the number of pieces of both the player is same then the game ends in a draw. The beauty of Othello is that just one move on the board may change the game situation very significantly.

3.10.2 Basic Strategies of Othello:

Playing the game of Othello requires the player to understand some important strategies of the game. Some of the important strategies are discussed here [39].

Disc Difference (Greedy):

The most basic strategy is to maximize the number of own pieces at every move. This strategy is often used by beginner Othello players. Although this strategy is very important in the later stages of the game, it can become disastrous when used earlier in
the game. As the opponent can maximize on his positions by making a couple of good moves in middle or end part of the game which may significantly cost the game.

**Stable discs:**

As the ultimate goal of Othello is to end up with as many discs as possible, many players presume that one should flip as many discs as possible at each move in the hope of ending up with as many as possible. But, this intuitive strategy is in fact among the worst possible in most situations. However, certain discs are "stable" in the sense that they cannot under any situation be flipped by your opponent, so obtaining these discs is usually a good idea. Furthermore, they provide as a means for capturing opponent’s non-stable discs. Thus the stability of the player’s discs refers to the number of stable discs that the player possesses and their locations. It is frequently the case that good stability leads to a winning situation; however this does not essentially imply that controlling all four corners leads to a win. In Reversi, there exists a hierarchy of stable discs. Placing such discs help the player to maximize his winning chances.

Given the rules of the game, the only way for your opponent to take a corner is if you play in one of the squares next to a corner, i.e., the C-squares or X-squares. Playing the so-called "X-squares" is a danger strategy. The "X-squares" are b2, g2, b7, g7 which are diagonally opposite to a corner. If your opponent is able to capture the corner either immediately or eventually your position will often be inferior. Good players seldom play to an X-square before move 30 or 40.

**Mobility:**

The idea is to limit your opponent's moves while maximizing your own. Often this will ultimately lead to a situation in which your opponent is forced by lack of any other moves to make a unfortunate corner sacrifice. In the mid game, the player who has the least discs is winning. As an end result, good Othello players will often choose moves which flip as few discs as possible, preferably only one.

On the other hand, a player with good mobility has a large choice of moves, allowing him to direct the game towards an advantageous situation. Mobility is important during the
middle part of the game where both players are fighting for a good position, which will provide easy capture of corners.

**Centralize and not to build walls:**
As far as possible, the discs should be centered, to minimize moves of the opponent. If the wall is built, then it will give many choices to the opponent to move and will decrease your mobility. Building walls and going out of the moves normally go hand in hand.

**Opening Moves:**
The introduction of strong computer programs in the 1990’s had a spectacular effect on opening theory. For experts, looking to take an advantage wherever they can, this has usually intended devoting a greater percentage of their practice time to researching and memorizing book openings. There have even been cases of people playing the entire game using memorized moves. The opening move strategy emphasizes the importance of control of central area of the board. In the opening, it is often worth making a louder move in order to confine central discs and set up quiet moves in the future.

**Edge Moves:**
At the start of the game there are 60 empty squares on the board, and 28 of those squares are on the edges. Thus edge moves account for almost half of all the moves in a typical game. Quiet moves are usually better than loud moves, and this is true for edge moves as well. If your opponent has run out of moves, then a quiet edge move is often enough to decide the game.

**End Game strategies:**
Unlike most strategy games, in Othello the board becomes more crowded as the game goes on, which results in more discs being flipped on each turn. Rapidly changing fortunes in the endgame is one of the things that make Othello great game. Here the strategy of controlling the diagonal is very important- because it allows control of the X squares.
3.10.3 Game Complexity:

State-space complexity of any board game represents the number of possible board states in the game. For example, in Othello there are 64 board locations where each location can take one of three values, white, black or empty, giving approximately $3^{64} \approx 10^{28}$ total states. Game tree complexity represents the total number of nodes in a fully-expanded game tree. Othello has a game tree complexity of $10^{58}$. Based on that figure, the average branching factor is about 9.26. The branching factor rises slowly from 4 and peaks at 12 on the 30th move.

3.10.4 Design Overview of Othello:

The program of Othello is designed and developed in Java with GUI and mouse interface. The program presents a choice to the user for different options. The board is shown as 8 * 8 size, where the user can interact with. The user can select the first player to move and accordingly the moves are displayed. The moves of computer are automatically chosen using a genetic algorithm and adversarial search. The panel also displays whose turn now it is. When the game is over, it displays who is the winner. The user can start a new game at any time.
Figure 3.41 Othello initial GUI window

Figure 3.41 shows the initial window where the user can interact with it. The window shows initial board window. At the bottom is information panel, where whose turn it is, is displayed and a label where number of moves the computer counts forward is displayed. On the right, four text boxes are displayed which give information about counts of the move of black and white player. At the top is “New Game” menu which when clicked displays various options to the user. When the menu “New Game” is clicked, it shows various menu options like – computer first, human first, human vs. human and Exit options.

Figure 3.42 shows the menu options.
Figure 3.43 shows the game window after first move of computer (black), it shows that the next move is of white i.e. human player and also shows that number of whites and number of blacks on the board.
Figure 3.42 Othello window menu options

Figure 3.43 Othello game play
As the Othello is also a board game just like Go-Moku, the classes of the application are more or less same. In the case of Othello, as the next move must flip at least one piece of the opponent, there is a question of finding out legal moves. For that a separate class named as Othello is used, which deals with finding out legal moves and then executing that legal move.

In addition to that a class called as Search is used to determine the next move by searching a game tree using adversarial search method like mini-max algorithm or alpha-beta cut-off. It contains the methods which actually carry out the game tree search and determine the next move of the computer. The Evaluation class is responsible for evaluating game situations.

As Othello is a game of strategy, and there are certain rules for a legal move, a class called as Othello is created which contains all the methods for checking legal move or not and if it is a legal move, then finds out which pieces are to be flipped due to that move. It also makes a move on the board and updates the current position of the board after a move.
Table 3.4 summarizes the important classes and their main functionality in Othello game.

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>OthelloApp</td>
<td>It is the class which is having main() GUI window of the application. The application uses other classes.</td>
</tr>
<tr>
<td>WindowContent</td>
<td>It contains content of the window. It contains the variables related with board. The methods for making move and refreshing graphics are also here. It also contains methods for starting a new game. It uses other classes like Square, Othello, Config, Move etc</td>
</tr>
<tr>
<td>Square</td>
<td>It represents one board square and contains methods related with checking whether the position is occupied or not, if occupied what is the colour of the player</td>
</tr>
<tr>
<td>Search</td>
<td>It is responsible searching game tree using game tree search method and returns the move. It also uses Othello class to check validity of the move</td>
</tr>
<tr>
<td>Move</td>
<td>Represents one position on the board in terms of point (X,Y)</td>
</tr>
<tr>
<td>Config</td>
<td>Contains all configuration parameters of the game such as board size, square size, depth to look ahead in algorithm, genetic algorithm related parameters</td>
</tr>
<tr>
<td>FunctionEvaluator</td>
<td>Used for evaluating the game.</td>
</tr>
<tr>
<td>Tournament</td>
<td>For carrying out tournament on evaluating functions and compare the players.</td>
</tr>
</tbody>
</table>

Table 3.4 Summary of classes in Othello Program
3.10.5 Square, Board and Pieces:

The game board is implemented as a two dimensional array of squares. Each square has its X and Y co-ordinate value in the board. The square can be filled with white piece, black piece or can be empty. The player having the turn fills the square with his colour of the piece. Whenever there is a move either by human or computer player, that move is done on the board with a function such as addPiece(). It checks if the square is occupied or not. If the square is empty and legal as per the rules of game, which are checked by the class Othello in the game program, it makes a move on to that square on the board. The argument to addPiece() is also the colour of the piece to move to that place. There is also a method removePiece() which does exactly the opposite work of addPiece(). Normally, removing a piece is not needed in the game. It is needed only when a user selects a new game menu option from the menu. The removePiece() method is used to present a fresh board to the user. It essentially clears all rows and columns of the board and presents a fresh new game board. In addition to that, as the game has a feature of flipping the pieces of the opponent, a method called as turnPieces() is used to find out the pieces to be flipped with appropriate colour.

3.10.6 Computer Player:

Computer player moves are dependent on situations. In the opening move, normally it plays at places which are usually played in the beginning, and then the moves are normally following a strategy. The moves are evaluated as per the weights table. It uses a combination of adversarial algorithm and genetic algorithm to evaluate the move.

A special thread is continuously running to generate computer moves. It keeps track of when it is the turn of a computer. Move generation of the computer player takes place automatically. When it is the turn of a computer, the move takes place using game tree search method.
Fitness function operates using following main considerations:

- Every board position is assigned a weight value. It tries to maximize the fitness function value.
- Identify winning situation.

As shown earlier, the general form of fitness function for board game is like:

\[ f(s) = w_1 \cdot f_1 + w_2 \cdot f_2 + w_3 \cdot f_3 + \ldots \]

Where, \( f_1, f_2, f_3 \) etc are usually based on human strategic knowledge of the game in terms of threats and opportunities in the game, while \( w_1, w_2, w_3 \) etc are weightage associated with each \( f \) in the formula.

The weights \( w_i \), which are used for evaluating the board position are shown below in table 3.5.

```
<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>-3</td>
<td>11</td>
<td>8</td>
<td>8</td>
<td>11</td>
<td>-3</td>
<td>20</td>
</tr>
<tr>
<td>-3</td>
<td>-7</td>
<td>-4</td>
<td>1</td>
<td>1</td>
<td>-4</td>
<td>-7</td>
<td>-3</td>
</tr>
<tr>
<td>11</td>
<td>-4</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>-4</td>
<td>11</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>2</td>
<td>-3</td>
<td>-3</td>
<td>2</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>2</td>
<td>-3</td>
<td>-3</td>
<td>2</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>11</td>
<td>-4</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>-4</td>
<td>11</td>
</tr>
<tr>
<td>-3</td>
<td>-7</td>
<td>-4</td>
<td>1</td>
<td>1</td>
<td>-4</td>
<td>-7</td>
<td>-3</td>
</tr>
<tr>
<td>20</td>
<td>-3</td>
<td>11</td>
<td>8</td>
<td>8</td>
<td>11</td>
<td>-3</td>
<td>20</td>
</tr>
</tbody>
</table>
```

Table 3.5 Othello weights table

From the weights table it is very clear that it is symmetric, row 0 with 7, row 1 with 6, row 2 with 5 and row 3 with 4, similarly with columns also. The weight assigned to each
position is logical form the view of importance of the position. The weights assigned to ‘X’ position is lowest (-7), while weights assigned to stable disc i.e. corner position is highest i.e. 20. A weight assigned to ‘A’ position is high enough because it may force the opponent player to move to ‘C’ position which may allow us the possession of corner disc.