CHAPTER 3
MATERIALS AND METHODS

3.1 INTRODUCTION

Games have played a central role in Artificial Intelligence (AI) research since the commencement of the computer era. Many pioneers in computer science expended time on algorithms for chess, checkers, and other games of strategy. A fractional list includes such luminaries as Alan Turing, John von Neumann, Claude Shannon, Herbert Simon, Alan Newell, John McCarthy, Arthur Samuel, Donald Knuth, Donald Michie, and Ken Thompson. One motive why game-playing is a thrilling motion for humans are that it combines intellectual activity with direct race: better thinking and learning usually results in winning more games. It can test out and refine intellectual skills by playing games against opponents, and evaluate progress based on the results of the competition.

The same inspiration accounts for much of the interest in Computer Game-Playing as a problem domain for Artificial Intelligence (AI): programs which think better, ought to play better, and so win more games. Thus it can check out and refine different theories of intellect by writing game-playing programs which symbolize these different theories, and then perform the programs against each other, and consider the more intelligent program to be the one which wins the most games.

In the past times, researchers have put a lot of effort into improving the strength of computer program for board games such as Othello, Checkers, and Chess -- not just because it is fun, but also because games are beneficial test beds for Artificial Intelligence. Many search algorithms and pattern recognition methods can be simply
tested on game playing programs. After years of hard work, many remarkable results have been obtained. One of the most striking examples is Deep Blue’s defeat of Garry Kasparov, the World Chess Champion, in 1997. Yet, for some games such as Shogi (Japanese Chess) and Go, computers still have considerable room for improvement and humans are still far greater. [105][106]

This thesis refers to that recognized link between winning games and intelligent behaviour as the competitive presentation metric for intelligence. Such a link is always advantageous for research, because it would mean researchers would not have to resort to descriptive evaluation measures and could instead use competition to evaluate research success. Many AI researchers working on games have expected that such a link does exist and have engrossed their energies exclusively on building strong game-playing programs. Game Playing is a significant problem for AI researchers which raise the following questions:

- How to assess good work in AI applied to board games in the absence of performance?
- Basic AI techniques such as learning, planning, and problem-solving should be useful for something; but if it does not induce performance, then what to do?
- How to tell whether the performance of a program on some game is due to the general success of the AI theory it represents, or merely to the individual skill of the researcher in analyzing a specific problem?
- If the latter has any little reason to believe the technique will transfer to other problems.
- Is it possible to find some game for which enhanced performance on that game would be linked essentially to increased understanding and automation of general intelligent processing?

The thesis in this chapter documents the construction of a program that plays –Board game. The program takes as input the rules of an explicit board game and analyses those
rules to construct for that game an efficient representation and an evaluation function, both for use with a generic search engine in terms of search algorithms. The strategic analysis performed by the program recounts a set of general knowledge sources to the details of the particular game. Among other properties, this analysis defines the relative value of the different pieces in a given game.

Although does not learn from experience, the values resultant from its analysis are qualitatively similar to values used by experts on known games, and are adequate to produce competitive performance the time the program actually plays each game it is given. This appears to be the program to have resultant useful piece values directly from study of the rules of different games. Trials show that the knowledge implemented in is useful on games that were unknown to its programmer in advance of the competition and make it seem likely that future programs which integrate learning and more sophisticated active-analysis techniques will have an obvious competitive advantage on this new problem. When playing the known board games against humans and specialized programs, has derived from more general values some strategies which are familiar to players of those games and which are hard-wired in many game-specific programs. [107]

The study of board games, card games, and other mathematical games of tactic is desirable for a number of reasons. In general, they have some or all of the following possessions:

• Games have definite rules and simple logistics, making it comparatively easy to implement a complete player, allowing more time and effort to be spent on the actual topics of scientific interest.
• Games have composite strategies, and are among the toughest problems known in computational complexity and theoretical computer science.
• Games have a clear precise goal, providing a clear-cut definition of success, and efforts can be focused on accomplishing that goal.
• Games allow measurable results, either by the grade of success in playing the game against other adversaries, or in the solutions to related subtasks.

The issues of problem solving that motivate the use of genetic approach also characterize some of the boundaries to its applicability. The upsurge in computer power, the wide range of applications and the ability to collect massive amounts of scientific data all require a computational framework, such as genetic algorithms, capable of automatically generating optimized solutions with a demonstration that is both influential and understandable. However, the genetic algorithm has some high computational cost to run, has difficulty scaling to larger and harder problem instances, and uses a typical representation that can also limit its use.

At the heart of these issues is the use of a population of alternate solutions and the methods used to create new alternatives. A candidate, or alternative, solution is assessed by means of a fitness function that allows it to be compared with formerly evaluated solutions. A selection method chooses better solutions from the population that then undergo variations to produce new alternatives. In this way, the population is accountable for guiding a parallel search through the solution space. However, the evaluation of each population member becomes increasingly computationally expensive and designing an actual process of variation on the direct and complex encoding of solutions is not instinctive. Also, the amount of data produced by a population based search over an enormous space of possible search space makes the prediction and analysis of algorithm behaviour difficult.

Genetic algorithm uses a population of variable-length problem specific genetic string within the metaphor of evolution to guide search. Search is an outline for problem solving where substitute solutions are evaluated. A goal of search is to find an assembling of alternative solutions that leads to good solutions. When inventory over the entire space of alternative solutions is not possible, heuristics like genetic approach use rules of thumb to guide the search for alternative solutions. Genetic algorithm belongs to a class of
algorithms which use the metaphor of Natural Selection to determine the ordering of alternative solutions during search.

Genetic algorithm is a modest and powerful technique which has been applied to a wide range of problems in combinatorial optimisation, automatic programming and model induction for a general introduction to genetic programming. The direct encoding of results as variable-length genetic string that allows genetic algorithms to provide solutions. It can be evaluated and also examined to understand their internal workings. In this way, a genetic algorithms induced solution can characterize a single value after evaluation as in optimization, an iteratively built algorithm or a data-driven model useful in data mining and knowledge discovery. While genetic algorithm can be applied successfully in all three cases, each of which merits its own scientific discipline, the focus of this thesis is on the inner workings of the algorithm itself.

3.2 RESEARCH PERSPECTIVE

The focus of this thesis is on understanding of the genetic population, the ways it can be measured and the role it plays on guiding the search process. Explicitly, the diversity of the population is analysed to discover key features and their relationships that make search successful. In the process of evolving a better understanding of genetic algorithm, a richer explanation of the dynamics of the algorithm arises to facilitate and motivate future enhancements.

The existing theoretical models of evolutionary algorithms are partial in use and applicability due to their complexity. Therefore, the majority of theoretical work has been derived from experimentation. The approach taken in this thesis is also based on the careful design, collection and analysis of experimental results. [108][109]

Most current approaches within including those retaining some forms of machine learning rely on previous human analysis of specific games. Human researchers do most
of the stimulating game analysis, which makes it difficult to assess the generalization and applicability of different methods. It also makes it difficult to determine empirically the success of research which highlights generality like work on learning. This option undermines the use of competition as a metric for evaluating progress in AI and poses procedural problems for the field.

Examining and redressing this problem is the central concern of this part of the thesis, and this chapter provides the background search algorithms and optimization approaches and motivation for what will follow. The chapter is broken into two sections. First Section motivates by means of examples that means by game-analysis, and argues that thoughtful and automating game analysis is two of the major goals behind studying intelligent game-playing in AI.

Second Section substantiates the claim that much current work in relies on humans, instead of programs, to perform a good portion of this analysis. Next Section summarizes the chapter and points out an important methodological problem confronting the field of which stems, in part, from the very likelihood that humans are performing some of the analysis instead of the programs. [110]

3.2.1 Game-Analysis

Roughly speaking, game-analysis comprises the set of processes which operate on an abstract representation of a given game, and lead to the development of a set of specialized search methods, heuristics, and strategies specialized for that board game. The resulting strategies can provide humans and computers with a competitive advantage, relative to a group of other players who may use different strategies.

3.3 GAME-SPECIFIC ENGINEERING APPROACH
An example of a game-specific response would be to compute a list of games and current champion programs which play these games. This list might comprise the following advice: If the game is Chess, then use the newest version of Deep Thought. If the game given to the researcher happens to be on this list, this advice would be enormously helpful. Unfortunately, if the game falls outside the list, this advice would be of little use. It is becoming a common perception in Artificial Intelligence that such a list of assistance is all that many researchers in have to offer. [111]

3.3.1 Game-Tree Search

It is an answer which more precisely reflects the lessons learned by current methods advocates the use of a brute-force search method (e.g., minimax with pruning and singular extensions), joint with extremely fast routines for updating board positions. This practice has proven effective on several games, and some toolkits have been developed to make it simpler to put on these techniques to a variety of games. However, this approach presumes that the researcher has a good evaluation function, which requires specific knowledge of the game. [112]

The burden on game analysis thus moves to the researcher, who must pick an appropriate set of features and weights for this function and it is the epitome of this thesis. Although there are some universal approaches to learning weights, this approach has offered very little explicit guidance about the construction of suitable features. However, it is now important to start understanding the importance of some features which may be essential in a variety of games, such as mobility.

Knowledge-Based Search has learned that exhaustive search may not be suitable for all games. Therefore we may also advise our researcher as follows: first, find some human who can analyse the game at expert level, then determine an suitable set of goals and sub goals, and priorities for these goals, and finally write a knowledge-based search program which exploits these. But as in game-tree search, it is really never said much about
explicit about how to find beneficial sub-goals, so again the researcher must do the
difficult game analysis on his own.

3.3.2 Database Enumeration

While knowledge engineering methods rely on human analysis by definition, such
analysis might be of much less importance when programs construct their own database
by enumerating a large set of conceivable positions. On the contrary, the human
researcher must achieve an extensive analysis of a game to determine at least the
following:

- How to enumerate positions methodically?
- How to avoid creating impossible and symmetric positions?
- Given the above, are there adequate resources to solve the problem?

By the time the human has replied these questions, it could well be argued that all the
game-analysis has already been done. Perhaps more importantly, this method is
applicable only to games that are small enough to be analysed in this way, so the answer
to the third question is likely to be negative.

3.3.3 Unsupervised Game-Learning

A much smaller proportion of learning work has measured how programs might become
strong players while trusting neither on active analysis nor on skill with experts. Most of
these methods can be considered as self-play, in which either a single player or a
population of players evolves during competition on large numbers of contests. A related
technique developed for basic playing programs which learned to predict the expected-
outcome of positions if played by random players which can also be regarded as a form
of self-play. [113] This is an actual for constructing evaluation functions for some games.
In principle, approaches based on self-play could give fully satisfactory and general advice to the hypothetical researcher, for example:

Use an off-the-shelf self-training technique. When given the game rules, choose an appropriate representation for states and strategies in the given game. When programs play against each other many times, and by the time of factual competition, select the strongest evolved program. It would seem that this advice requires minimal game-analysis on the part of the human. The key issues with this approach are as follows:

1. How much time is needed to evolve a strong player?
2. How effective is the training method at evolving good strategies on different types of games?
3. How much game-analysis must the human execute in order to plan an appropriate representation?

With respect to the first and second questions, these are at present unanswered. While the methods appear useful for certain types of problems; presented several examples of important strategic concepts which appeared difficult to derive without some active analysis.

Unfortunately, these questions are all very hard to answer, because the games, representations, learning methods, and amount of knowledge engineering have varied with each learning system. With respect to the third question in specific, discusses the “fixed representation trick”, in which many developers of learning systems spend much of their time finding a picture of the game which will allow their systems to learn how to play it well. It has produced tremendously strong backgammon program using a training scheme which is claimed to be “knowledge-free”, yet so far this method has been established only for one specific game (backgammon), with which the author was acquainted. [114]
3.4 GAME PARAMETERS FOR PROGRAM GENERATION

The probabilities devoted to the choice points, and the parameters which are used by the sub-tree constriction testers, are thus the parameters of the generator. Interestingly, they can be used (directly) to affect the circulations of syntactic structures generated like larger or smaller piece definitions, and also indirect influences on semantic aspects of generated games.

Rule Complexity is one property of interest is the intricacy (length) of the rules in a game. This can be controlled by means of a small set of parameters in the generator which are checked in order to choose between making a game module more complex, or leaving it as it is. This allows components to be produced with subjectively long descriptions, though longer descriptions are exponentially less likely than shorter ones. By varying these parameters, it can thus change the overall predictable complexity of the components to which they are associated. Instances of such parameters are those attached to the clauses, which control the likelihood of adding another disjunction to these definitions.

3.4.1 Decision Complexity

It is another statistical property of a game which can be determined in this way is the degree to which a game permits players to make choices, instead of assigning arbitrary values to these choices as part of the game definition. For example, pieces in Shogi promote to precisely one type of new piece each, whereas pawns in Chess endorse to any one of a set of choices, to be decided by the player at the time of promotion. This property can easily be varied to create different types of games, by modifying the delivery attached to the rule which decides, for example, whether a promotion or initial-setup choice should be arbitrary or not.

3.4.2 Search Complexity
It is a related statistical property of board game. It is essentially the size of the search space in a particular game. This can be adjusted, without touching the rule or decision complexities discussed above, by altering the probability distribution on board size, as larger board sizes will incline to allow more possible movements for each piece, and thus more likely moves in each position in the game. Of course, the parameters mentioned above also affect the size of the search space, such as growing the probability that a piece has different types of movement available. [115]

Certain parameters, in fact, have dramatic significances on the search space. For example, the presence of an rule, which allows the opponent to make a promotion decision before starting his move, grows the number of possible moves available to him in such a position: if a player had 0 normal move movements in a position, but he first has to promote an opponent’s piece to one of other pieces, then the total branching factor for that position is 20. If he had also to promote whichever piece he moved to one of pieces, the branching factor would increase to 20. At the opposite extreme of affecting search complexity, the presence of must capture restrictions has the effect of dramatically dropping the size of the search space.

3.4.3 Locality

A final property of notice is locality, which determines the portion of a board which can be traversed by a piece in one leap, without regard for the other squares on the board. The fewer localities, the more pieces on one side of the board can straight affect the status of pieces on another side of the board. It is possible that this affects the grade to which a program could reason about separate aspects of the board individually. Locality is affected by the modules constraining the limitations on riders and hoppers, the module which generates direction vectors, and the parameter, as a cylindrical board allows pieces to move from one side to the other with a direct leap.
3.4.4 Game Complexity Metrics

The preceding paragraphs discussed the manner in which qualitative properties of a game are influenced by precise combinations of parameters. Our understanding of the generator could be enhanced by a more systematic study of the relation between the low-level parameters and high-level properties. This effort has not yet been undertaken, in part because of the trouble of quantifying the complexity of large games. For example, the length of the rules upsurge the potential for obscure interactions between rules to influence strategies, but it is also likely for a game with a long description nevertheless to be tactically simple. Even the obvious metric on the size of the search-space of the game (e.g. the game tree) can not necessarily be measured or even projected reliably for large games. [116][117]

3.4.5 Consistency Checking

Deciding whether a generated game can possibly be won generally requires a level of analysis beyond that implemented in the game generator program. However, the current board game programs do perform a simple analysis to evade some of the common problems which would otherwise produce a high proportion of trivial games. Eventually, it is up to the programs to decide whether or not a game is trivial or even winnable, which certainly an aspect of game analysis is traditionally left to humans.

This showed that defining whether a symmetric chess-like game could possibly end in a win for one player is NP-hard. An open question is whether this board game problem is also in NP, and thus NP-Complete. The difficulty in answering stems from the fact that the problem is not necessarily in NP, because solution orders to random symmetric chess-like games may be of span exponential in the size of the game description. This means it could not verify a proposed sequence in polynomial time in just the game description.

For example, consider a game on a million by million boards where a player wins if a piece can reach a square on the opposite side of the board. As the board size is
represented in decimal notation in the game description, the board size component of the
game description is only 7 digits long, while the maximum solution length is an
exponential function of this. There are set of questions

a  Is a given move legal from a given position?
b  Is a goal is achieved in a given position, can both be answered in time polynomial in
   the length of the game description?

These observations can be determined by inspection from the domain theory for
symmetric board games. From this it follows that the question would be in NP if it were
possible to determine a polynomial bound on the length of the shortest move sequence to
achieve a goal from a given position. At present the existence of such a bound does not
appear likely.

3.4.6 Generalizing existing features

The first and most noticeable approach to finding class-wide knowledge for board games
is to examine the knowledge currently used by programs which play precise games in the
class. By isolating the assumptions relevant to a game-assumptive concept, it was in
some cases possible to simplify the concept to apply to the class as a whole. Three
significant and general features emerged in this way: mobility, centrality and promotion.
[118]

The concept of mobility is used in some manner in almost all game-playing programs. The
common factor in most mobility features is that they compute a set of properties
which are essential, but not always sufficient, for a player to have a legal move in a
position. Both Reversi and Checkers programs comprise terms in their evaluation
functions which count the moves available to each piece possessed by each player in the
current position. Note that this is not enough to guarantee that the player really has those
moves accessible. One example comes from Checkers: although many pieces have
thinkable movements, a player might be forced to capture an enemy piece, in which case he may in reality have only one legal move. Another example comes from chess: a player may be attributed with a large number of queen moves, although none of them might actually be legal if the player’s king were in check.

When demanding to develop a general concept of mobility which could be used in this class, it was essential to determine whether mobility was always needed, all else being equal, or if its rationality assumed some properties of the game. For example, in both Reversi and Checkers, a player wins by seizing all of the enemy pieces. Thus it seemed possible that for positive goals of this form, in which a player wins by weakening the enemy, mobility might be useful. In negative goals, in which a player wins by fading himself, it seemed that mobility might be damaging. One example of such a negative game is lose-chess. In this game, each player must make a capture whenever possible, and the goal is to have no more moves, in the language of symmetric chess-like games. Another negative game is the ordinary version of Reversi, in which a player wins by having the most pieces on the board at the end of the game, but mobility often declines with each piece a player captures. After examining successful strategies in both the win and lose versions of these games, the opposite conclusion was reached. [119]

That is, mobility appears to be treasured for either type of goal, all else equal. Some sign for this is that in both win and lose versions of Chess and Reversi (Othello), the openings are almost equal regardless of the final goal. That is, both players endeavor for increased mobility in the opening, as this gives them greater control. With greater control, they then go on to achieve advanced goals. Conversely, it turns out that efforts to reach the final goal in spite of reduced mobility often result instead in early defeat. For example, a player who directly tries to give away all pieces in lose-chess quickly winds up with only a few moves available. Although this means the player has almost attained the final goal, the opponent is then in such control of the game that he can force the first player to arrest the entire opponent’s remaining pieces. [120]
3.5 SEARCH, EAs AND GENETIC ALGORITHM

Evolutionary algorithms are heuristic search techniques within the broadly defined domain of artificial intelligence. Problem solving is a common application domain of artificial intelligence, where search is the framework for problem solving with computers. How does an evolutionary algorithm implement and carry out search? This chapter focuses on describing the process of search, the elements required to carry out search and different search strategies. Traditional search is defined first, followed by mutual heuristic methods. The evolutionary algorithm is then described, showing how it addresses the task of problem solving by means of search. After introducing and describing genetic approach, the chapter concludes by discussing two important research issues: scalability and representation.

3.5.1 Problem Solving and Search

Most human activity can be defined as a form of compound problem solving. So problem solving by a computer program is measured to be an instance of artificial intelligence, where the task consists of managing information and searching for solutions. Given an illustration of a problem and a description of an ideal solution, the goal of search is to find a solution equivalent to or close to the ideal solution. In many cases, decision making and problem cracking are formulated as optimization problems. The role of the search method is to find the best solution among possible alternatives, optimizing for solution quality.

While exploring the possibility of learning machines, designed an algorithm that was able to evaluate the quality of a computer program, make random changes to it and then re-test for improvements. Since then, many improvements have been made in the field of artificial intelligence. [121]

3.5.2 Requirements of Search
To carry out search by means of a computer program, numerous elements of the problem and search strategy must be defined. Search requires the following basics:

- A defined problem, often requiring intelligent or complex behaviour to solve.
- A model of the problem and the demonstration of possible solutions.
- A goal state that defines the perfect solution, where heuristic approaches often use an evaluation function to rank the non-goal state solutions.
- Transformation operators that is capable of altering an existing solution into an alternative solution.
- A policy for searching the space of possible solutions using the representation and transformation operators.

The submission of the transformation operator(s) on a solution creates a neighbourhood of solutions. These new solutions can then be equated to the goal state. When knowledge of the problem is available, heuristic algorithms can outline an evaluation function that allows the scoring and ranking of solutions. The ranking of solutions regulates which, if any, of the solutions in the neighbourhood are better.

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

3.5.3 Algorithms to Perform Search

With the ability to create and evaluate a possible solution, a random search strategy can be defined. A random search method recurrently generates solutions, evaluates them and generates more. Random search uses no knowledge of formerly generated solutions and typically stops when a solution meets a particular criterion, such as being equal to the goal state or having quality above a threshold.
Transformation operators allow a solution to be transformed into one or many new solutions, depending on the number and type of operators. In this way, the neighbourhood of the current solution is generated. The neighbourhood signifies all the solutions that can be generated by the application of the operators on the current solution. Search strategies regulate the operators used, the size of the neighbourhood searched and which solutions are selected to continue the search from. In traditional search, where classically a goal state is defined, the operators can define a search tree. A node in the tree represents a solution, and its successors, or children, are defined by the operators. Strategies can then be defined to search the tree. Two such blind search strategies are depth-first and breadth-first search. [122]

Breadth-first search produces all the successors of the root node in the search tree. Next, it generates all the successors of those nodes and continues until a solution is found. Depth-first search generates a single inheritor of the root node. Next, it generates one successor to that node and continues until a maximum depth is reached. At this point, it backtracks to the preceding root node and, if possible, generates another successor.

For problem domains with large search trees, blind search methods become inefficient and unrealistic. This is mainly true when a search problem is cast as a decision problem like whether or not a solution exists with quality above a threshold that is undecidable in polynomial time of the size of the problem instance. That is, if the decision problem is in the complexity class, the optimisation or search problem must also be at least as hard.

Though, blind search methods classically do not use any knowledge of the problem domain, which is often readily accessible. It is mutual for problems to become intractably huge in size; estimated methods called heuristics are typically used. Heuristics define rules-of-thumb about the problem structure, such as a way to breed a feasible solution in a combinatorial optimisation problem or a search strategy that finds good solutions in a reasonable amount of time. Heuristics can increase search efficiency by only considering
a subset of all thinkable solutions or by only generating solutions that are likely to be healthier than the current solution.

A simple heuristic search method is hill-climbing. Hill-climbing (or steepest descent) begins with an arbitrarily generated solution and continues to produce neighbors until a better solution is found. This new solution becomes the current solution and the algorithm begins producing its neighbors. If there are no refining solutions in the current neighbourhood, the method will stop. The current solution may or may not be a global optimum in this case. Stochastic hill-climbing is a variant of hill-climbing that proceeds in the same way but can also accept neighbors that are equal to the current solution. The multi-start hill-climbing method attempts to increase performance by performing several trials or runs of hill-climbing that each start from a different random initial solution. The operator used in heuristic techniques stereotypically results in a small random change to the current solution. This type of operator, often called mutation, requires little or no domain knowledge. If all the neighbors of the existing solution are created before selecting the best, the method is called neighbourhood search. Variable neighbourhood search adaptively increases the size of the neighbourhood when no refining solution is found.

Another simple heuristic search method is best-first. This method generates all the neighbors of a solution, selects the best one, produces all its neighbors, etc. When no better solution is found in a neighbourhood, the search returns to the previous neighbourhood and selects the second best, and continues until no enhancements can be found. Beam search is similar to best-first, but does not remember the complete previous neighbourhood. Instead, beam search recollects a fraction of the best solutions in all previous neighborhoods to return to when no improvements are found.

These methods are also called local search procedures as they only accept refining solutions that are in the local neighbourhood of the current solution. By using information of the domain, heuristic evaluation functions can be defined to convey an estimation of
the quality of a current solution or the likelihood that the search will continue toward a goal state. Heuristic evaluation functions are also called cost functions, objective functions or fitness functions. The projected solution quality reported by these functions is often called the *fitness* of the solution. With this measure of solution quality, the search strategy has additional information with which to guide search. Metaheuristics methods characterize search strategies that use heuristic evaluation functions to guide search. Metaheuristics are iterative improvement algorithms that repeatedly transform solutions, selecting better *or* worse new solutions in order to improve the overall fitness of the final solution obtained. Metaheuristics methods adopt that the heuristic evaluation function, representation and operators combine to lead search on a path from the current solution to a better one, hopefully to the goal state or a global optimum.

When operators persuade small changes to the solution that result in small fluctuations to solution quality, the fitness landscape is said to be *smooth*. The term strong connection describes the property where minor changes to the solution result in small changes to its fitness. This property can also apply to large changes in solutions. Strong connection indicates that there is a correlation between the size of solution alteration and the size of change in the resulting solution quality. Weak causality describes the case where there is little or no connection between the size of the changes in solutions and fitness values. Fitness landscapes were recommended using the family of correlated landscapes. When the landscape is not smooth, local search methods are likely to get stuck with sub-optimal solutions.

In order to avoid sub-optimal solutions, meta-heuristics use a diversity of techniques to be robust to non-smooth, or rough, fitness landscapes. Most metaheuristics search policies allow non-improving solutions to be designated during search. Two popular methods are simulated annealing and tabu search. Simulated annealing is like to a hill-climbing method that begins with a random current solution. However, in simulated annealing, a parameter defines the likelihood that an inferior scoring solution is accepted.
Over time, the temperature is reduced, allowing the acceptance of worse solutions with smaller probability. [123]

Tabu search produces the neighbourhood of a current solution and considers any of the new solutions that are not on a list of previously visited ones, the tabu list. Thus, the exploration is temporarily banned from re-visiting solutions, preventing cycling. In this way, if all improving solutions are on the tabu list, then this algorithm can also visit non-improving solutions. Tabu search can also comprise an explicit method of directing the search toward similar or dissimilar new solutions, called strengthening and diversification. Both simulated annealing and tabu search have been applied to an extensive range of real-world problems. Another class of metaheuristics search strategies is those based on the theory of Natural Selection. These are described in the following section.

One such symbol that was previously proposed is that of Beam Search. Beam search is similar to the best-first heuristic but only allows a limited capacity of memory of formerly visited solutions. Thus, quotas of the search space may be forgotten and optimal solutions may be missed. In genetic programming, the population and selection methods provide the basis for the metaphor. These elements provide a kind of memory of previous search and the ability to select the most promising aspects to continue search with.

Another possible metaphor is that of hill-climbing. Hill-climbing is a memory-less version of beam search. While comparisons are made between genetic programming search quality, hill-climbing and random search methods, to characterize the search behaviour is a different matter. It is clearly not the case that genetic programming is a random search. The population, selection and variation of better solutions produce a search by the trial-and-error of new solutions, but a metaphor of random search would suggest that the search continues with no direction from previous solutions. Hill-climbing, on the other hand, may be an applicable metaphor as it captures the sense that
while a population does exist, convergence and operator behaviour are likely to prevent the population from returning to a previous state.

### 3.6 Conceptual Evolutionary Algorithm

**Tab 3.1 Basic Evolutionary Algorithm**

<table>
<thead>
<tr>
<th>Step</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization: create an initial population</td>
</tr>
<tr>
<td>Evaluation: evaluate fitness values of individuals in the population.</td>
</tr>
<tr>
<td>while stopping criteria not met do</td>
</tr>
<tr>
<td>Selection: choose high fitness value parents to fill the mating pool.</td>
</tr>
<tr>
<td>Variation: apply variation operators on parents to create offspring.</td>
</tr>
<tr>
<td>Evaluation: evaluate fitness values of parents and offspring.</td>
</tr>
<tr>
<td>Replacement: replace parents with higher quality offspring.</td>
</tr>
<tr>
<td>end while</td>
</tr>
<tr>
<td>output the best individuals in the population</td>
</tr>
</tbody>
</table>

#### 3.6.1 Individual Representation

Within the EA framework, individuals are considered at two levels: the genotypic level and the phenotypic level. The genotype of an individual, also referred to as the chromosome, is a string of genes of finite length. The chromosomes could be in any form such as binary, integer/alphabet, real-valued. The most common forms are binary representation and integer/alphabet. In the binary representation, each gene in encoded individuals that takes a value of either 0 or 1 and the worth of genes is important. In the permutation representation, each gene in encoded individuals takes a notable value and the position of genes is important. The choice of chromosome illustration, which depends on the problem as well as the EA itself, is important. An incompatible representation could lead to unnecessary computational overhead and low performance of the EA.

The genotype represents the characteristics (decision variables) of individuals but not the quality of individuals. The quality of individuals, the phenotype, is acquired by decoding the chromosome. The phenotype is also known as the objective worth of individuals. It is
vital to understand the differences between genotype and phenotype representations because of their different purposes and association to different stages during the evolutionary process.

3.6.2 Evaluation of Individual Fitness

The evaluation of an individual is the course which obtains the objective values, the phenotype, of the individual by decoding its genotype. The phenotype of individuals could be used to straight compare individuals based on its dominance and at different stages of the algorithm.

However, it is very mutual that the algorithm takes one step further in evaluating individuals' quality by conveying fitness to each individual. This process is re-ferred to as the individual’s fitness evaluation to distinguish from the individual's objective values evaluation. The fitness evaluation derives the fitness of an individual from its objective values. While computing the fitness of an individual, the fitness evaluation might or might not consider other individuals in the population. The fitness evaluation policy is one of the core features of which could discriminate one EA from others. Fitness assignment strategies could be characterized into three types:

- Dominance-based: The fitness of individuals is determined by comparing individuals to others in the population based on dominance.

- Non dominance-based: The fitness of individuals is determined by applying transferring functions on the objective values of individuals which combine and/or modify the objective values.

- Hybridization: Based on some satisfaction on dominance condition, transferring functions are applied to the objective values of individuals to obtain the fitness value.
3.6.3 Parents Selection

The parents’ selection, or mating selection, is the process of selecting individuals to participate in the making of offspring. There are two common mating selections: stochastic selection and tournament selection. Stochastic selection selects parents at random regardless of parents' quality/fitness. Tournament selection applies an extra layer onto the stochastic selection. In tournament selection, parents are also drawn at random but selected parents then undergo a fitness comparison process. The maximum quality parent, i.e. the winner of the tournament, is designated to be in the mating pool. The size of the tournament is usually set to 2 (binary tournament). There are other schemes such as fitness proportional selection and truncation selection. In fitness proportionate selection, the probability of each individual being nominated for the mating pool is in proportion to its fitness. In truncation selection, each of the top individuals in the population with respect to fitness get proportional copies in the mating pool. The selection of parents is also one of the key features in EAs.

3.6.4 Replacement Strategy

The replacement strategy, which is also recognized as the environmental selection or survival selection, is the process of selecting individuals for the next generation based on the fitness of individuals. As opposed to the parents’ selection, which is usually stochastic, the replacement strategy which typically selects the best individuals based on their fitness is mainly deterministic. The replacement strategy is characterized into generational and steady-state selection schemes. The dissimilarity between these two schemes is at which point parents and offspring contend for survival. In the generational selection scheme, for each generation, the descendants population is constructed first which is based on the parent population.

The sizes of the offspring population and parent population are usually the identical size of the current population. The offspring population and the current population are pooled.
Individuals from consequential population are then selected for the next generation. However, the steady-state selection scheme allows offspring to compete for survivor and reproduction as soon as they are created. In other words, offspring are measured to replace their parents or other individuals in the current population instantly after offspring are constructed. Besides the fitness assignment and the parents’ selection, the replacement strategy is also one of the important facets of an EA.

### 3.6.5 Reproduction Mechanism

The reproduction machinery uses the genetic material, the genotype, of parents to create offspring. The purpose of the reproduction mechanism is to manipulate the gene structure of individuals in the existing population, create new individuals with the anticipation that better individuals could be obtained. There are a large number of genetic operators that can be used for the reproduction, but broadly these are divided into two categories:

**Recombination:** is usually a binary operator which combines the genetic material from two parents to produce the genetic material for one or two children. The recombination operator, which is usually a random operator, resolves which part of the genetic material from parents is inherited by the offspring. The standard behind the recombination is that “good” genes in parents are united in the genetic material of the offspring.

**Mutation:** is a unary operator which is applied to the genotype of one individual in order to marginally modify the gene structure of that solution. The mutation operator is maximum times a stochastic operator. The mutation operator attempts to introduce a few new features that might not be hereditary from the parents. The purpose is to add variety to the population and contribute so that the total search space can be possibly explored.

It is noted that both recombination and mutation operators are genotypic depiction dependent. Different individual representations often require different operators. It is also noted that while operators for mutation are usually problem independent, many
recombination operators are problem specific. A few problem independent operators are presented below:

The simplest form of mutation operator is bit mutation used with a binary representation. Each gene in the genetic material of an individual is inverted between 0 and 1 with certain probability usually very low and in the range of 0.01 to 0.1 or 0.05. This probability is known as the mutation probability, which is reasonably small to prevent too much disruption of the inherited genetic material. For a permutation representation, the mutation operator usually exchanges two randomly selected genes in the permutation list at once or more times.

Recombination, or crossover, operators have various different forms. Two common forms of crossover for binary representation are k-point and uniform crossover. The simplest versions of k-point crossover are one-point crossover (figure 3.1) and two-point crossover (figure 3.2). In one-point crossover, a crossover point is selected at random then genes on one side of the crossover point are exchanged between individuals. In two-point crossover, two crossover points are selected at random and genes between two crossover points are exchanged between individuals.

![Figure 3.1: One Point Crossover](image1)

![Figure 3.2: Two Point Crossover](image2)
In uniform crossover (figure 3.3), every gene is exchanged between individuals with
definite probability, which is usually set to 0.5. With respect to the permutation
representation, crossover operators must preserve the genetic information i.e. all genes
must be in the genotype in whichever way the permutation list is modified.

![Figure 3.3: Uniform Crossover](image)

Oliver et al. presented cycle crossover, applied on two parents, which preserves the
absolute position of genes in the permutation list. A crossover point is selected at random
as the starting position of the cycle. The gene at that position of the designated parent is
copied to the genotype of the offspring. The next position of the cycle is the position, in
the selected parent, of the gene which appears at the current position in the other parent.

![Figure 3.4: Cycle Crossover](image)

The cycle repeats until the gene at the starting position of the cycle is met. Unfilled
positions of the offspring are filled by genes, in those positions, of the other parent. The
procedure of the cycle crossover is illustrated in figure 3.4.

Another crossover operator for permutation representation is uniform order based
crossover, applied on two parents. Uniform order-based crossover requires a random
binary template. Firstly, the offspring is filled with genes from a selected parent at positions indicated as 1 in the template. The remaining genes in the selected parent, which are arranged in the same order as appeared in the other parent, then fill empty positions in the offspring. The uniform order-based crossover is illustrated in figure 3.5.

<table>
<thead>
<tr>
<th>Parent</th>
<th>0</th>
<th>7</th>
<th>1</th>
<th>5</th>
<th>6</th>
<th>3</th>
<th>8</th>
<th>4</th>
<th>9</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parent 2</td>
<td>2</td>
<td>3</td>
<td>7</td>
<td>8</td>
<td>5</td>
<td>4</td>
<td>9</td>
<td>1</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Template</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Offspring</td>
<td>0</td>
<td>2</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>3</td>
<td>8</td>
<td>4</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>Offspring</td>
<td>2</td>
<td>0</td>
<td>7</td>
<td>8</td>
<td>5</td>
<td>4</td>
<td>6</td>
<td>1</td>
<td>3</td>
<td>9</td>
</tr>
</tbody>
</table>

Figure 3.5: Uniform Order-based Crossover

There are also other popular crossover operators for permutation representation such as partly mapped crossover, position based crossover, order crossovers. These crossover operators and their evaluation could be found in. Throughout this thesis, one-point crossover operator is employed for binary representation.

3.6.6 Stopping Criteria

Stopping criteria define situations in which the evolutionary search terminates and the best individuals are presented. Stopping criteria usually vary accordingly to the form of applications and the drive of the studies. For theoretical studies, in which the purpose is to explore the performance of newly proposed EAs for example, the stopping criteria are usually the number of evaluations or generations or the amount of execution time.
However, the latter is rarely used due to its little reliability and dependence on other factors such as hardware, operating systems and programming languages. For real-world application (especially in real-time application), in which the computational time is limited, it is sensible to set the amount of execution time as the stopping criterion. There are other measures such as on-line performance metrics which keep track of the improvement of population or best solutions until no enhancement after a certain amount of time or evaluations. The number of generations could be also the stopping criterion of real-world applications.

3.7 GAME TREE SEARCHING AND PRUNING

This section of the chapter mainly focuses on game tree searching and pruning aspects. It presents contextual knowledge on game playing programs: how to build a game tree and how to choose the next move. The following sections further introduces the most fruitful refinement of minimax search—the alpha-beta algorithm.

3.7.1 Game trees and Mini-max Search

Virtually all game playing programs use a game tree to represent positions and moves. Nodes do represent game positions, and the root node resembles to the current position of the game. The branches of a node represent the permissible moves from the position represented by the node. A node is called a leaf if it doesn’t have a heir. Using the rules of the game, researcher can evaluate a leaf as a win, lose, draw or a specific score.

But unfortunately the whole game tree size is tremendously huge for almost all interesting games. For example, checkers is $10^{20}$, and chess is $10^{40}$. The total number of nodes in game tree is roughly $W \times D$, where $W$ stands for the number of possible moves on average for each node, and $D$ is the typical game length. Therefore, no any practical algorithm can manage such a full tree due to lack of time.
One easy solution of such games is to stop creating the tree at a fixed depth, \( d \), and use an evaluation function to estimate the positions at \( d \) moves ahead of the root. This thesis uses the term ply, which was first introduced by Samuel, to denote the depth of a game tree. The nodes at the deepest layer will be leaves. Typically, the value of a leaf, estimated by the evaluation function, is represented the number in proportion to the chance of winning the game.

![Min-Max Search Tree](image)

**Figure 3.6 Min-Max Search Tree**

Game playing programs rest on game tree search to find the best move for the current position, assuming the best play of the opponent. In a two-person game, two players choose an alternate legal move and both of them instinctively try to maximize their advantage. Because of this reason, finding the finest move for a player must assume the opponent also plays his/her best moves.

In other words, if the leaves are evaluated on the viewpoint of player A, player A will always play his /her moves that maximize his score by increasing the value of the resulting position, while the opponent B plays moves that minimize the value of the resulting position. This reason is base behind the name MiniMax or Min-Max algorithm.

Figure 3.6 shows a Min-Max search in a game tree. Every leaf has a corresponding value, which is estimated from player A’s viewpoint. When a path is chosen, the value of the
child will be passed back to the parent. For example, the value for D is 6, which is the extreme value of its children, while the value for C is 4, which is the minimum value of F and G. In this example, the best sequence of moves found by the maximizing / minimizing procedure is the path through nodes A, B, D and H, which is called the main continuation. The nodes on the path are denoted as principal continuation (PC) nodes.

Table 3.2 Pseudo code of Negamax algorithm

```c
// pos : current board position
// d: search depth
// Search game tree to given depth, and return evaluation of root node.
int NegaMax(pos, d)
{
    if (d=0 || game is over) return Eval (pos);
    // evaluate leaf position from current player’s standpoint
    score = - INFINITY; // present return value
    moves = Generate(pos); // generate successor moves
    for i =1 to size of(moves) do // look over all moves
    {
        Make(moves[i]); // execute current move
        // call other player, and switch sign of returned value
        cur = -NegaMax(pos, d-1);
        // compare returned value and score value, update it if necessary
        if (cur > score) score = cur;
        Undo(moves[i]); // retract current move
    }
    return score;
}
```
For simplicity, users can alter the game tree values slightly and use only intensification operations. The trick is to maximize the scores by negating the returned values from the children instead of searching for minimum scores, and approximate the values at leaves from the player’s own viewpoint. This is called the NegaMax algorithm.

Since most game-playing programs examine large trees, game tree search algorithms are commonly implemented as a depth-first search, which necessitates memory only linear with the search depth. Table 2.2 is the pseudocode of NegaMax algorithm, implemented as a depth-first search, and Figure 2.3 illustrates the NegaMax procedure using the same game tree as Figure 2.1.

![NegaMax Search Tree](image)

Fig 3.6 Nega-Max Search Tree

A NegaMax search has to assess every leaf of the game tree. For a uniform tree with exactly W moves at each node, a d-ply NegaMax search will evaluate $W \times d$ leaves. This makes a deeper search of a “bushy” tree impossible. Fortunately, a improvement is talked in next section called Alpha-Beta pruning that can decrease the amount of work.
Materials and Methods

significantly: in the best case, to twice the depth it might reach using NegaMax search in the same amount of time.

3.7.2 The Alpha-Beta Algorithm

As mentioned in previous section, it is not essential to explore all the nodes to determine the minimax value for the root. It can be evidenced that large chunks of the game tree can be pruned away. Knuth and Moore showed that the minimax value of the root can be obtained from a traversal of a subsection of the game tree, which has at most leaves, if the “best” move is surveyed first at every node. [112]

McCarthy was the first to comprehend that pruning was possible in a minimax search, but the first systematic solution was provided by Brudno. A few years later, Knuth and Moore further refined it and proved its properties. The success of alpha-beta search is achieved by cutting away uninteresting portions of the game tree. For an example point of view, \( \max(6, \min(5, A)) \) and \( \min(5, \max(6, B)) \) are always equal to 6 and 5 respectively, no matter what values A and B are. Therefore, pruning of the sub-trees corresponding to A and B during game tree search can be done. To appreciate these cut-offs, the alpha-beta algorithm employs a search window (alpha, beta) on the expected value of the tree. Values outside the search window, i.e., smaller than alpha or larger than beta, cannot affect the outcome of the search. Table 2.4 shows the pseudo code for the alpha-beta algorithm in NegaMax form. In order to return the correct minimax value, alpha-beta search should be raised with an initial window of alpha = -\( \infty \) and beta = \( \infty \). [124]

3.7.3 Alpha-Beta procedure

1. If the level is the top level let alpha be -1 and let beta be +1.
2. If the limit of the search has been reached, compute the static value of the current position relative to the appropriate player. Report the result.
3. If the level is a minimizing level,
4. Until all the children are examined with Alpha-Beta or until the alpha is equal to or greater than beta.

5. Use the Alpha-Beta procedure, with the current alpha and beta value, on a child; note the value reported.

6. Compare the value reported with the beta value; if the reported value is smaller, reset beta to the new value.


8. Otherwise, the level is a maximizing level:

9. Until all the children are examined with Alpha-Beta or until the alpha is equal to or greater than beta,

10. Use the Alpha-Beta procedure, with the current alpha and beta value, on a child; note the value reported.

11. Compare the value reported with the alpha value; if the reported value is larger, reset alpha to the new value.


In order to exemplify the alpha-beta search process discussed in above section as an example. Figure2.5 shows the same game tree as Figure 2.3, in which one node (L) and one sub tree (the sub-tree of G) are pruned by alpha-beta search.

Figure 3.8 Alpha-Beta Search Tree
Table 3.3 Pseudo code for Alpha-Beta Algorithm

```c
// pos : current board position
// d: search depth
// alpha: lower bound of expected value of the tree
// beta: upper bound of expected value of the tree
// Search game tree to given depth, and return evaluation of root.
int AlphaBeta(pos, d, alpha, beta)
{
    if (d=0 || game is over)
        return Eval (pos); // evaluate leaf position from current player’s standpoint
    score = -INFINITY; // preset return value
    moves = Generate(pos); // generate successor moves
    for i =1 to size of(moves) do // look over all moves
    {
        Make(moves[i]); // execute current move
        //call other player, and switch sign of returned value
        cur = -AlphaBeta(pos, d-1, -beta, -alpha);
        //compare returned value and score value, note new best score if necessary
        if (cur > score) score = cur;
        if (score > alpha) alpha = score; //adjust the search window
        Undo(moves[i]); // retract current move
        if (alpha >= beta) return alpha; // cut off
    }
    return score;
}
```

The leftmost branches are navigated with the initial window \((-\infty, \infty)\). After having evaluated the left child of D, the middle child of D is examined with the window \((-\infty, -6)\) since the value for D is at least 6. At node E, alpha is updated to -6 after completing the
search of left child D. So the search window of the right sibling E is \((-\infty, 6)\). After E’s left child is visited, the new alpha is adjusted to 7, which is larger than beta, so it’s right sibling L is cut off. It can also look to this procedure as determining the value of min (6, max (7, -L)), or max (-6, - max (7, -L)) in NegaMax form. No matter what value L is, the result we get is always 6, or –6 in NegaMax form.

3.8 EVALUATION FUNCTIONS

This is the second most important constituent of the thesis after search algorithm. Most successful game-playing programs apply heuristic evaluation functions at terminal nodes to estimate the probability that the player to move will win. Characteristically a successful evaluation function is the combination of a weighted sum, of a number of distinct features. Each board feature measures a property of the board position. Thus building evaluation functions has two phases:

1. Selecting good board game features.
2. Combining them appropriately to obtain a single numerical value.

Selecting features is important as well as difficult. Researcher or game program developer has to avoid too few features as well as redundant ones. It also necessitates both expert game knowledge and programming skill because of the well-known compromise between the complexity of the evaluation function and the number of locations it can assess in a given time: a more precise evaluation function might actually result in mediocre play if it takes too long to compute. Feature combination is also critical and very unintuitive. It demands the need not to establish only a balance among varied strategies but also be aware of the interactions between related features.

Evaluation Features of different board games depend on the type and kind of the game along with its move making rules and playing strategies. There are some universally accepted groups of evaluation features which are in true sense applicable to almost all types of board games. This thesis proposes board game programs for Reversi and Game
of Checkers. The important feature for them is also Mobility, Territory and Territory-and-Mobility. The idea of the Mobility feature is basically derived from Chess. It focuses on the number of possible moves since the more possible moves a player has; the less likely it is that the player will run out of legal moves. Similar the territory feature is based on the concept of controlling more squares, since controlling more squares can offer more space for the pieces to move. The feature Territory-and-Mobility, suggested by Hashimoto, combines the merits of Mobility and Territory in a way which we describe below. [125]

3.8.1 Mobility

In many board games, the last player who is able to complete a move gets an advantage or some cases wins the game, so having more possible moves than the opponent is a crucial factor for winning. The mobility of a player is defined as the totality of the possible moves of all his/her pieces or discs, and the mobility feature is calculated by subtracting the opponent’s mobility from the player’s. The mobility feature may be useful for the opening stage where the territory sorting is not clear enough. But in the mid-game and endgame, the board is separated into several battlefields. So controlling your own areas and invading the opponent’s territories are more vital than enlarging your mobility. Another problem is that if player maximizes mobility, all the player’s pieces tend to get better chance of stay in the middle of game and/or capturing possibility for future moves. The main benefit of this feature is its evaluation speed. Typically the evaluator can calculate the mobility feature faster than the territory feature.

3.8.2 Territory

The number of possible moves is huge in the opening stage and decreases gradually. From the mid-game phase on, the board is separated into several arenas. Therefore, keeping one’s own territory as large as possible or captured positions intact is critical for winning the board game. In any board game a square or board position belongs to the
player who can reach it faster with one of his/her pieces. If both players can reach a square in the same minimum number of moves, this square is neutral, i.e., it doesn’t belong to either player. In game playing programs bitmap can be used, which can be designed and coded to implement this algorithm efficiently.

The territory feature can correctly assess enclosed and almost-enclosed areas. Its performance is rationally good in the endgame and in well-balanced situations. Though, the territory features assumes the dics or piece movement in all possible directions and defend against attackers approaching from different sides at the same time. Therefore, in unbalanced situations such as where one piece or disc is facing several, this feature evaluates the board too optimistically.

3.8.3 Territory and Mobility

Hashimoto et al. combined the concepts of Mobility and Territory to build a new evaluation function, called Territory-and-Mobility. Typically the TM feature is evaluated via three steps:

1. Use function to evaluate the each player’s territory
2. Count mobility in each player’s territory
3. Sum the results of 1 and 2 using a specific weight.

Hashimoto et al. believes that adding mobility to the territory feature allows the program to place all discs and pieces in a coordinated way. In accumulation, the calculation of territory becomes more precise by adding the mobility scores in Step 3. After testing many values, they suggest that setting the weight to specific value in evaluation function gives the best performance. [126]

3.8.4 Evaluation Function Construction
The main matter in board game playing program is to assess available move and select the best move depends upon the process of selection of move and describe how it can be combine features to create an evaluation function. [127][128]

Traditionally, an evaluation function is a linear combination of a number of features (F1, F2, …, Fn), i.e., a weighted sum:

\[
\text{Evaluation Function } F = W_1 * F_1 + W_2 * F_2 + … + W_n * F_n \tag{3.1}
\]

Here the coefficients (W1, W2, …, Wn) are evaluation weights which are formed by implementer or found by analysis & optimization process.

But there are two difficulties with this method. First, it is hard for humans to estimate these coefficients correctly, since they don’t use game tree search and evaluation functions. That was also the initial motivation for Samuel to propose ways to tune weights automatically in. Furthermore, this technique adopts that no correlations or redundancies between features exist. This assumption is clearly false since almost every pair of features is correlated to some degree. To solve this problem, Lee et al present a pattern classification approach. [129]

3.8.5 Automatic Feature Combination

Arthur Samuel, a novice Checkers player, is one of the earliest and most important researchers on Checkers learning programs. From 1947 to 1967, he proposed and experimented on many different methods of machine learning. In the next two sections, we introduce the two most important ones:

- Linear evaluation learning through self-play
- Nonlinear evaluation learning through book moves.
In linear evaluation learning, Samuel tuned the coefficients by positioning two copies of the Checkers programs named Alpha and Beta to play against each other. At the opening, Alpha and Beta are identical. The only difference is that Beta keeps its weights fixed while Alpha continuously tunes its weights during the experiment. If Alpha beats Beta, Beta adopts Alpha’s evaluation on the next round of experiments. Otherwise, Alpha tries other conducts to tune its weight. Sometimes manual intervention is necessary if the learning course gets stuck. When Alpha consistently defeats Beta, its evaluation function is considered as the stabilized final version. After this learning procedure, the final program can play a reasonably good game of checkers. As one of the first machine learning examples, Samuel’s procedure is a milestone in automatic evaluation function construction. But as Lee et al pointed out in, it is based on several incorrect assumptions. First, it erroneously assumes that all features in the evaluation function are independent, so it cannot capture the relationships between features. Second, it assumes that all the inaccurate evaluations are caused by the evaluation function, while sometimes the real reason is the limited horizon of the search depth or process.

Third, it assumes that when the evaluation function is excessively optimistic, the problem must come from optimistic features. This is clearly incorrect because it may be due to negative features are not negative enough. Finally, it assumes that Alpha’s evaluation must be better than Beta’s if player Alpha beats Beta. But when both two programs are naive, a win may be the result of luck or the opponent’s errors. [133]

3.8.7 Non-linear evaluation learning through book moves

In order to cope with these improper assumptions, Samuel familiarizes a new procedure to build nonlinear evaluation functions through book moves in. To handle non-linear interactions among features, Samuel devised signature tables. These are multi-dimensional tables where each dimension is indexed by the value of some feature. For each game position, the table cell indexed by its feature values contains the conforming
evaluation value. Samuel collected 24 features for the game of Checkers. Obviously applying this scheme straight could result in an impractically large table. Samuel dealt with this problem using two methods. First, he organized the tables using a three-level hierarchical association. At the first level, each table combines four features, and only interactions between those four features are considered. Each table in level one or two produces a value to index into tables in the higher level. Furthermore, Samuel restricted the feature values to (-1,0,1) or (-2,-1,0,1,2). This results in a final configuration with a reasonable number of cells.

To evade the incorrect assumptions in self-play, Samuel used book moves to train these cells. He collected a “book” of board positions and the equivalent moves played by human masters. For each cell, he counted how many times the corresponding feature combination was chosen in book moves, $A$, and how many times the corresponding grouping was a legal move but was not chosen in book moves, $D$. The cell value was then evaluated as $(A-D)/(A+D)$.

According to Samuel’s experiments, signature table learning through book moves considerably outperformed the self-play learning procedure. But there are a number of new problems with this approach. First, this approach is based on a problematic assumption that no other moves are as good as or better than book moves. Second, confining feature values causes some smoothness problems. Finally, the higher-level signature tables cannot handle inter-table associations. So the correlated features must be arranged into the same group at the first level. [134]

As a result, Samuel’s procedure needs extreme human tuning. The implementer has to put a lot of effort in determining how to confine the feature values and arranging the structure of signature tables. This is undesirable since the learning procedure may be affected by human errors.

3.8.8 Evaluation function learning
Lee et al introduced a new learning algorithm for evaluation function construction [129]. They used the game of Othello as their test territory and dramatically improve an Othello program BILL2.0 that has achieved at the world championship level. This approach also includes two stages: training and recognition.

In the training stage, a database of branded training positions is required. Lee et al took these positions from real games generated from BILL 2.0’s self-play. All winning positions of player were labeled as winning positions, and all positions of the losing player as losing positions. Of course, positions might be mislabeled, e.g., that Bill lost from a position in which an optimal player would win. [135] First, although BILL2.0 was still using a linear evaluation function, it had been carefully tuned and it was a world championship-level player. Moreover, the initial position of each game was generated by 20 arbitrary moves, after which the player that is ahead usually goes on to win the game. [136]

To acquire the training positions, two replicas of BILL 2.0 are used to play with each other from early positions. The initial positions were generated by 20 random moves. After that, each side played the residual 40 half-moves in 15 minutes and the last 15 moves were played using perfect endgame search. An entire of 3000 games were played and their positions were recorded as training data. [137]

For each training board, the four features were calculated and represented as a feature vector. Then, the mean vectors and covariance matrices for both categories, winning and losing, could be estimated. In the game of Othello, different strategies should be used for diverse stages of the game. Lee et al defined a stage as the number of discs on the board. [133] For a stage with N discs, they used training positions with N-2, N-1, N, N+1, and N+2 discs to generate a corresponding discriminant function. Using a series of slowly varying discriminant functions, the evaluation function provides a fine measure of game positions for different stages. To identify a new position, it first computes the features
and combine them to form the feature vector, \( x \). Then it can evaluate the position by substituting \( x \) into the final evaluation function. [138]

### 3.9 THESIS EMPLOYED GENETIC METHODS

The game playing programs tries to imitate human game playing approach in its own limited functioning scopes. Such competences can be well explored in an important domains like board games of zero-sum, deterministic, full-knowledge, alternate move and two player. Games of research for this thesis are Game of Checkers and Game of Reversi.

These games are played on an \( N \times N \) board for some given \( N \). Here work is done on Checkers and Reversi, also known as Othello, are popular games with a rich exploration history. Though a board game played on an 8x8 board, it differs widely from other board games as they are piece-capturing and a piece-placing game respectively with simple playing rules.

In Reversi the number of pieces on the board increases as the game progresses on, rather than decreasing as it does in board game of Checkers. Evolutionary genetic approach is employed to solve the board games of the thesis. Various genetic parameters are shown in Table 3.4.

### 3.10 GENETIC ALGORITHMS ESSENTIALS

A vital step in the application of GAs is the documentation of a “fitness function”, which is used to measure how close each chromosome comes to solving the problem at hand. The fitness function is also used to handpick those chromosomes that will participate in the creation of offspring. Individualities of the fitness function play a significant role in the conduct and success of the GA in finding a solution. Considerable research pains that have focused on the issue of epistasis, a distinctive of a fitness function in which the fitness of a chromosome hang on the interaction between gene values at different locations on the chromosome.
Materials and Methods

Table 3.4 Various Genetic Parameters

- **Population size**: This option defines the size of the population.
- **Parameters**: This option defines how many parameters are used in the evaluation function of the computer.
- **Number of games per match**: This option specifies how many games are played during a match between two Evaluation functions.
- **Survivor Probability**: This option defines the probability of the total no. of Evaluation Values carried forward to the next Generation. Here the value is 10%.
- **Crossover Probability**: This option defines the probability that crossover will occur; it works only with conjunction the **classic style algorithm**.
- **Mutation Probability**: This option defines the rate of mutation in terms of the number of bits that have being mutated.
- **Classic style fitness**: This option work only with the **classic style algorithm**. It defines the "fitness giver" Evaluation function at each generation.
- **Use fitness sorting every generation**: This option defines if one wants to sort the Evaluation function of the population according to their fitness, in some manners, just before all the mutation/recombination at each generation actually takes place.
- **Generations**: This option defines how many generations to simulate. The value – 1 means infinite.

It is significant and striking to know that the GAs do not guarantee to find the best of the possible solutions for a problem, but they are generally good at finding acceptable solution in an acceptable time. This is a requirement of many board games having very high search space. GA assigns suitable coding (or representation) for the problem, it is required a fitness function, which assigns a merit to each encoded solution and it is important to define the selection and reproduction rules which are genetic operators. Each possible solution for a problem is represented by a set of parameters or genes. The genes are joined together to form a string of values or a chromosome. The most common
representation is the binary string form as it is simple and easy to be manipulated by the genetic operators. [139][140]

The most old-fashioned genetic operators are selection, crossover and the mutation. In the first case, two individual’s chromosomes of the inhabitants are selected based on some selection criteria and their chromosome strings are cut at a randomly chosen position. Resulting two tail segments are then swapped over to produce two new full length chromosomes. The mutation operator is generally applied to each descendent individually after crossover. It arbitrarily alters some genes with a small chance. Mutation is traditionally seen as a “background” operator; however, examples in nature show that asexual reproduction can evolve sophisticated creatures without crossover. [138]

To solve the board game described, it was used a binary representation with 32 bits for Checkers and 10 bits for Reversi where each bit (or each chromosome gene, in the language of GAs) represents one of the possible board square position or position family which can be selected in the radial direction.

When the value of the bit is 1, it means that the bit to which it refers is occupied by player’s disc and when the value of the bit is 0, the disc is not selected or occupied and when it is -1 it means it is been captured by player’s opponent. The fitness function offers the decidability. The greater the decidability values for a given distribution of points, the superior the fitness of the individual and the higher the probability of being chosen for reproduction.

The thesis basically describes the idea and the implementation in evolutionary board games. The game passes through various stages. The goal of this method is to improve the performance of the game in each of the stages. Evolutionary progression helps in building better board game playing program which helps in playing and exploiting winning return in every move of each stage. Genetic algorithm is used for the speciation of population, genetic operator parameters and it is effortlessly implemented for a
specified number of generations. A min-max algorithm with alpha beta pruning for maximum three ply depth is used to implement to find the associated fitness values. [141]

3.11 GAME IMPLEMENTATION THEMES

3.11.1 Game Tree

To find the following move of a player, a game tree is raised with a limited depth. Each node in a game tree characterizes the configuration of the board at some stage of the game. The worth of the terminal nodes (leaf nodes) in terms of fitness weight is measured with the assessor like min max algorithm.

The calculated values using the evaluation function of the terminal nodes propagate upward using min/max operations. The max operation selects the max value of all children nodes and the min operation chooses the min value among all the children at their level. The current configuration of the board is represented as a root node and the arc represents a move. At an odd number level, the max operation is used and vice versa. [142]

3.11.2 Evolutionary Genetic Algorithm

A collective theory of evolutionary genetic algorithm can accomplish better than the single and individual best school of thought. In the Game playing research community nowadays, in initial play of the program, a roulette wheel genetic selection is conducted to select the best player in generations. The moves are selected on the fitness values and games are played accordingly generations by generations.

The values tend to get better as better as ‘good’ moves have the tendency to give better moves- winning moves. Good individuals resulting from roulette wheel are kept as for next generations and select-crossover-mutate genetic cycle functions in its totality and gives genetically evolutionary values. [143]
3.12 DETERMINISTICS BOARD GAMES

The main focus of traditional processes for developing tactics for board games divides the game into various game stages like opening, middle, and endgame stages. For each stage, a different heuristic can be applied. For example, it is very challenging to define the most appropriate choice in the opening stage of a game, so the use of an opening book from games played by experts is advantageous.

Human players definitely possess an advantage over computers in the opening stage because it is difficult to quantify the relevance of the board configuration at an early stage. To be extra competitive from very early playing phase, an opening book can be very helpful but a huge opening book can make the program inflexible and without innovation. One of the important parts of game programming is to design the evaluation function which builds very good move making and disc placing by capturing good positions for games like Reversi for the middle stage of the game. [144]

These opening games are to form very strong piece or disc formation strategies which can help to build good board capturing or board mobility feature in mid game. In the middle stage, a game tree with a limited depth is created and a game feature based heuristic evaluation function is applied to estimate the relevance of each move.

The evaluation function is often a linear combination of game features and board square weights. These features are based on human knowledge, such as the number of important disc positions, the sum of free positions to make move(potential mobility), the piece differential between two players, the stable discs which can’t be flipped by opponent (stable discs) and other pattern-based features. Defining these components and assigning weights to them requires expert knowledge and a long trial-and-error tuning. These can compete with hand-tuning weights in terms of time and efficiency. In Reversi, the results of the game can be calculated in real-time if the number of empty spaces is less than 26. Here effectual and optimized search algorithm and good evaluators are badly needed to build strong end game base. Finally, in the end game, the number of pieces or possible
moves becomes reasonably small and deterministic calculation of the final moves is possible. [145]

Table 3.5 Pseudo code showing piece presence in 8x8 arrays of squares

```plaintext
for (i=0;i<8;i++)
    for(j=0;j<8;j++)
        switch (board[i, j])
            case wP:
                if (board[i+1,j] empty) generate move to (i+1,j)
                if (i==2 && board[i+1,j] empty && board[i+2,j] empty)
                    generate move to (i+2,j)
                if (j > 0 && board[i+1,j-1] contains black piece)
                    generate capture of (i+1,j-1)
                if (j < 7 && board[i+1,j+1] contains black piece)
                    generate capture of (i+1,j+1)
                break;
            ...
```

Table 3.6 Pseudo code showing piece presence score in 8x8 arrays of squares

```plaintext
for (i=0;i<8;i++)
    for(j=0;j<8;j++)
        score += value[square[i, j]];
        score += T[new square,piece] - T[old square,piece]
        score += T[new square,piece] - T[old square,piece]
```

To implement 8x8 arrays of squares and to keep track which piece is present in the square and counting the value following functions are used which are shown in table 3.5 and 3.6 respectively.

Following two tables show the algorithm to search game tree to given depth, return evaluation of root node and search game tree to given depth, return evaluation of root node are shown in following two tables 3.7 and 3.8 respectively.
Table 3.7 Pseudo code search game tree to given depth

```c
// search game tree to given depth, return evaluation of root node
double negamax(int depth)
{
    if (depth <= 0 || game is over) return eval (pos);
    else {
        double e = -infty;
        for (each move m available in pos) {
            make move m;
            e = max(e, -negamax(depth - 1));
            unmake move m;
        }
        return e;
    }
}
```

Table 3.8 Pseudo code showing piece movement

```c
// search game tree to given depth, return evaluation of root node
move rootsearch(int depth)
{
    double e = -infty;
    move mm;
    for (each move m available in pos) {
        make move m;
        double em = -negamax(depth - 1);
        if (e < em) {
            e = em;
            mm = m;
        }
        unmake move m;
    }
    return mm;
}
```

Note that above procedure this only finds the evaluation, but doesn't determine which move to actually make. The program has only need to find an actual move at the root of the tree. The board game implementation of Game of Reversi and Game of Checkers are shown one by one with their implementation details, genetic parameters, screen shots and collected results.

3.12.1 Evaluation Function Formation
Normally, an evaluation function is the linear sum of the values of applicable features selected by experts. The input of the evaluation function is the bit value resultant from the current configuration of the board and the output of the function is a value of quality value of move. Evaluation function co efficient are derived from the positional significant of the board squares.

These values are key values in determining some features of the board evaluation function that can be exhibited using genetic algorithm. The board games are very important problem domain for learning the evaluation function such as determining the architecture of the board game model and transformation of the board configuration into numerical form.

A board game program proceeds exactly in the same manner. But there are a few technical difficulties. How to know that a certain position in the game is "good" to make a move? That position has to be evaluation of a position, but "some" is not good enough for a program: it has the need to be able to articulate a clear-cut principle for the quality of a position that it can express in the firm framework of a programming language. Such evaluation functions depend on the nature of the game.

A mutual way to view evaluation functions is to encode the estimate of the quality of a position in a number. The higher the number it get from the evaluation, the better a position is for the program. The lower the number, the better the game goes for the opponent. Usually, one designs evaluation functions such that they yield positive values when things are going well for the program and negative numbers if the opponent is winning. Zero indicates a balance of power.

3.13 CONCLUSION

This chapter first described three popular evaluation features of the Board game. Besides, some evaluation function construction methods and their applications in Checkers and
Othello were introduced. There are two interesting questions left in this chapter. The first one is how important these features are in position evaluation. The second one is whether it can be combined using genetic optimization to the board games of this thesis effectively.

Genetic algorithm is an evolutionary algorithm that represents solutions as programs. The next chapter of the thesis present genetic algorithm as an original representation as bit-strings in a fixed length genome. A bit-string necessitates a mapping to a representation that can be evaluated and assigned fitness by the heuristic evaluation function. The remainder of this thesis will focus specifically on the canonical form of this employed genetic methods and their game tree theme based representation. Next chapter deals with their implementation and result analysis.