SUDOKU FORMATION THROUGH APPLICATION OF SOFT COMPUTING

A project report submitted
In partial fulfilment of the requirements for the award of the degree of

Bachelor of Technology
In
Computer Science & Engineering

Submitted By:

Suman Banerjee (17600116012)
Tanya Anand (17600116007)
Sourav Roy (17600116016)
Riya Saha (17600116031)

Under the guidance of

Mr. Sandeep Bhowmik, Dept. of CSE

Department of Computer Science & Engineering
Hooghly Engineering & Technology College, Hooghly

Affiliated to
Maulana Abul Kalam Azad University of Technology,
West Bengal

2019-20

HOOGHLY ENGINEERING AND TECHNOLOGY COLLEGE

Vivekananda Road, Pipulpati, P.O. & Dist-Hooghly (WB), Pin-712103

(Affiliated to Maulana Abul Kalam Azad University of Technology)

CERTIFICATE

This is to certify that the project entitled **"SUDOKU FORMATION THROUGH APPLICATION OF SOFT COMPUTING"** has been submitted for the fulfilment of the requirement for the award of the degree of Bachelor of Technology in Computer Science & Engineering by following B.Tech (CSE) final year students under the supervision of **Mr. Sandeep Bhowmik**, **Assistant Professor, CSE Department,** during a period from **January, 2020** to **June, 2020.**

Student Name (with Roll No.)

1. Tanya Anand (17600116007)
2. Suman Banerjee (17600116012)
3. Sourav Roy (17600116016)
4. Riya Saha (17600116031)

_____

Project Guide,
Department of CSE

_____

Mr. Dibyendu Samanta
Coordinator, Department of CSE

_____

Prof.(Dr.) Sumanta Bhattacharyya
Principal, HETC

PREFACE

Our goal was to assimilate the existing knowledge of Soft Computing of industrial interest into one consistent, self-contained volume accessible to engineers in practice and to motivated non-specialists with a strong desire to learn Soft Computing. Our intent was to provide a detailed presentation of some of the areas of Soft Computing which we have found to be of greatest practical utility in our own industrial experience, while maintaining a sufficiently formal approach to be suitable both as a trustworthy reference for those whose primary interest is further research, and to provide a solid foundation for professionals and others readers first learning the subject.

Throughout the project, we emphasize the relationship between various aspects of Soft Computing and its utility in generating Sudoku. Computer source code (e.g. Java code) for algorithms has been intentionally omitted in the documentation, in favor of algorithms specified in sufficient detail to allow direct implementation without consulting secondary references. We believe this style of presentation allows a better understanding of how algorithms actually work, while at the same time avoiding low-level implementation-specific constructs (which some readers will invariably be unfamiliar with) of various currently-popular programming languages.

Each topic has been discussed thoroughly to provide a self-contained treatment of one major topic. Collectively, however, it has been designed and carefully integrated to be entirely complementary with respect to definitions, terminology, and notation. Furthermore, there is essentially no duplication of material across chapters; instead, appropriate cross-chapter references are provided where relevant. To facilitate the ease of accessing and referencing results, items have been categorized and numbered to a large extent, with the following classes of items jointly numbered consecutively in each chapter: Definitions, Examples, Facts, Notes, Flowcharts, Algorithms, Implementations, and Methodologies.

At the end of the project, we have included a list of papers which has been listed in the reference section, each entry of which is cited at least once in the body of the documentation. Almost all of these references have been verified for correctness in their exact titles, volume and page numbers, etc. Finally, an extensive Index prepared which begins with a List of figures and tables.

Our intention was not to introduce a collection of new techniques and protocols, but rather to selectively present techniques from those currently available in the public domain. Essentially all of the algorithms related to Soft Computing, included have been verified for correctness by independent implementation, confirming the test vectors specified.

ACKNOWLEDGEMENT

It gives us immense pleasure to announce the partial completion of our project on **"SUDOKU FORMATION THROUGH APPLICATION OF SOFT COMPUTING"** and we are pleased to acknowledge our indebtedness to all the persons who directly or indirectly contributed in the development of this work and who influenced our thinking, behaviour and acts during the course of study.

We are thankful to our respected departmental Coordinator **Mr. Dibyendu Samanta** who granted all the facilities of the college to us for the fulfilment of the project.

We are thankful and express our sincere gratitude to our project guide **Mr. Sandeep Bhowmik** who gave his valuable time to us for the sake of our project. He helped us each and every aspect of our project both academically and mentally.

Finally the team expressed their gratitude to our respected principal **(Prof.)Dr. Sumanta Bhattacharyya** without his support our project would not have seen the light of success.

Student Name (Roll No.)	**Signature**
1. Tanya Anand (17600116007)	……
2. Suman Banerjee (17600116012)	……
3. Sourav Roy (17600116016)	……
4. Riya Saha (17600116031)	……

4th Year, 8th Semester
Department of Computer Science & Engineering
Hooghly Engineering & Technology College
Academic Year: 2019-20

ABSTRACT

The objective of the work was to formulate a Sudoku generating algorithm. The algorithm implements a hierarchy of four simple logical rules commonly used by humans. A variant of our algorithm generates a random Sudoku, which agrees well with the classifications given by the existing puzzle setters. Using evolutionary approach to ensure that all required constraints are met, we aim to optimize the efficiency of our algorithm through reuse of prior calculations and incremental/decremental modifications at each step.

Four difficulty levels are established, each pertaining to a range of numerical values returned by the solving function. The construction of a Sudoku puzzle begins with the generation of a solution. This is accomplished by means of a random number based function within the program. Following each change in the grid, the fitness is evaluated as the program solves the current puzzle. The procedure loops until either the problem is completely solved or the logical techniques of the program are insufficient to make further progress.

Selection of the optimal parameters for machine learning tasks is challenging. This project gives a brief introduction about Genetic algorithm (GA) which is one of the simplest random-based Evolutionary Algorithms and one of the techniques of Soft Computing.

This project studies the algorithm involved in generating Sudoku puzzles with the help of Genetic algorithm (GA). We consider the problem of generating well-formed Sudoku puzzles. Sudoku can be regarded as a constraint satisfaction problem. But when solved with the help of Genetic algorithm it can be handled as a multi-objective optimization problem.

The objectives of this study are:

1. To test if Genetic algorithm optimization is an efficient method for solving Sudoku puzzles,

2. Can Genetic algorithm be used to generate new puzzles efficiently?

The last of these objectives is approached by testing of puzzles that are considered difficult for a human solver are also difficult for the Genetic algorithm. The results presented in this documentation seem to support the conclusion that these objectives are reasonably well met with Genetic algorithm optimization.

Uniqueness is guaranteed due to the fact that the algorithm never guesses. If there is no sufficient information to draw further conclusions, then an arbitrary choice must be made. For obvious reasons, puzzles lacking a unique solution are undesirable, due to the fact that the logical techniques possessed by the program enables it to lead to an optimal solution.

INDEX

FIGURE INDEX

TABLE INDEX

1. INTRODUCTION

1.1 OVERVIEW OF SOFT COMPUTING

Soft Computing, an older term for Computational Intelligence, is the use of approximate calculations to provide imprecise but usable solutions to complex computational problems. The approach enables solutions for problems that may either be unsolvable or just too time-consuming to solve with current hardware. It is a set of artificial intelligence techniques which provides efficient and feasible solutions in comparison with conventional computing. They are basically integrated techniques to find solutions for the problems which are highly complex, ill-defined and difficult to model. Real world problems deal with imprecision and uncertainty can be easily handled using such techniques. Soft Computing provides a set of techniques which are hybridized and finally useful for designing intelligent systems.

Soft Computing provides an approach to problem-solving using means, other than computers. With the human mind as a role model, Soft Computing is tolerant of partial truths, uncertainty, imprecision and approximation, unlike traditional computing models [1]. The tolerance of Soft Computing allows researchers to approach some problems that traditional computing can't process. Soft Computing uses component fields of study in:

1. Fuzzy Logic
2. Machine Learning
3. Probabilistic Reasoning
4. Evolutionary Computation
5. Genetic Algorithms
6. Artificial Neural Networks

These disciplines are applied on the problem individually or as hybrids of techniques. As a field of mathematics and computer study, Soft Computing has been around since the 1990s. The inspiration was the human mind's ability to form real-world solutions to problems through approximation. Soft Computing contrasts with possibility, an approach that is used when there is not enough information available to solve a problem. Soft Computing is used where the problem is not adequately specified for the use of conventional math and computer techniques. It has numerous real-world applications in domestic, commercial and industrial situations.

Soft Computing is dedicated to system solutions based on Soft Computing techniques. It provides rapid dissemination of important results in Soft Computing technologies, a fusion of research in evolutionary algorithms and genetic programming, neural science and neural net systems, and fuzzy set theory and fuzzy systems. Soft Computing encourages the integration of Soft Computing techniques and tools into both everyday and advanced applications. What is important to note is that Soft Computing is not a melange. Rather, it is a partnership in which each of the partners contributes a distinct methodology for addressing problems in its domain. In this perspective, the principal constituent methodologies in Soft Computing are complementary rather than competitive. Furthermore, Soft Computing may be viewed as a foundation component for the emerging field of conceptual intelligence.

1.1.1 TECHNIQUES OF SOFT COMPUTING

1.1.1.1 FUZZY LOGIC

The term fuzzy refers to things which are not clear or are vague. In the real world, many times we encounter a situation when we can't determine whether the state is true or false, their fuzzy logic provides a very valuable flexibility for reasoning. In this way, we can consider the inaccuracies and uncertainties of any situation. Fuzzy logic is an approach to computing based on "degrees of truth" rather than the usual "true or false" (1 or 0), that is, the Boolean logic on which the modern computer is based. Fuzzy logic seems closer to the way our brains work. We aggregate data and form a number of partial truths which we aggregate further into higher truths and which in turn, when certain thresholds are exceeded, cause certain further results such as motor reaction.

Applications:

1. It is used in the aerospace field for altitude control of spacecraft and satellite.
2. It has used in the automotive system for speed control, traffic control.
3. It is used for decision making support systems and personal evaluation in the large company business.

1.1.1.2 NEURAL NETWORKS

A neural network is a series of algorithms that endeavours to recognize underlying relationships in a set of data through a process that mimics the way the human brain operates. Neural networks can adapt to changing input; so the network generates the best possible result without need to redesign the output criteria. The concept of neural networks, which has its roots in artificial intelligence, is swiftly gaining popularity in the development of trading systems. A neural network works similarly to the human brain's neural network.

Applications:

1. Artificial neural networks are extensively used in applications involving
2. Biometric Pattern Recognition
3. Classification
4. Prediction
5. Forecasting
6. Data clustering Problems

1.1.1.3 EVOLUTIONARY COMPUTATION

Evolutionary computation is a general name for a group of problem-solving techniques whose principles are based on the theory of biological evolution, such as genetic inheritance and natural selection. Evolutionary computation is usually implemented on computer systems that are used to solve problems, implementing techniques such as evolutionary algorithms, differential evolution, genetic algorithms and harmony search. Techniques in this field are used on problems that have too many variables for traditional algorithms to consider and in times where the approach to solving a particular problem is not well understood.

Applications:

1. Engineering Design
2. Data Mining
3. Bio Informatics
4. System Modelling and Identification
5. Artificial Life

1.1.1.4 MACHINE LEARNING

Machine learning focuses on the development of computer programs that can access data and use it to learn for themselves. The process of learning begins with observations or data, such as, direct experience, or instruction, in order to look for patterns in data and make better decisions in the future based on the examples that we provide. The primary aim is to allow the computers learn automatically without human intervention or assistance and adjust actions accordingly.

Applications:

1. Virtual Personal Assistants
2. Online Fraud Detection
3. Video Surveillance
4. Social media Services

1.1.1.5 GENETIC ALGORITHMS

A genetic algorithm is a heuristic search method used in artificial intelligence and computing. It is used for finding optimized solutions to search problems based on the theory of natural selection and evolutionary biology. Genetic algorithms are excellent for searching through large and complex data sets. They are considered capable of finding reasonable solutions to complex issues as they are highly capable of solving unconstrained and constrained optimization issues. The most commonly employed method in genetic algorithms is to create a group of individuals randomly from a given population. The individuals thus formed are evaluated with the help of the evaluation function provided by the programmer. Individuals are then provided with a score which indirectly highlights the fitness to the given situation. The best two individuals are then used to create one or more offspring, after which random mutations are done on the offspring. Depending on the needs of the application, the procedure continues until an acceptable solution is derived or until a certain number of generations have passed.

Applications:

1. Robotics
2. Automotive Design
3. Optimized Telecommunications
4. Games

1.1.1.6 PROBABILISTIC REASONING

Probabilistic logics offer a rich framework to represent and process uncertain information, and are linked to statistics and machine learning in a natural way. Knowledge can be extracted from data, expressed in a suitable probabilistic formalism like Bayesian networks, and used for uncertain reasoning by applying inference mechanisms. Completeness of knowledge can be achieved by

presupposing additional assumptions like conditional independence of variables, like in most probabilistic networks. In both ways, a full probability distribution is generated from partial knowledge, on the base of which probabilities for arbitrary queries can be computed. Probabilistic reasoning is a method of representation of knowledge where the concept of probability is applied to indicate the uncertainty in knowledge.

Probabilistic reasoning is used:

1. When we are unsure of the predicates
2. When the possibilities of predicates become too large to list down
3. When it is known that an error occurs during an experiment

1.1.2 THE IMPORTANCE OF SOFT COMPUTING

An intelligent machine relies on computational intelligence in generating its intelligent behaviour. This requires a knowledge system in which representation and processing of knowledge are central functions. Approximation is a 'soft' concept, and the capability to approximate for the purposes of comparison, pattern recognition, reasoning, and decision making is a manifestation of intelligence. The journals of Soft Computing has more demand in the International market because this computing works on the real time application areas.

The perfection of fuzzy logic (FL), neural computing (NC), genetic computation (GC) and probabilistic reasoning (PR) has a vital consequence. In many cases, an issue can be sought out most perfectly by using fuzzy logic, neural computing, genetic computation and probabilistic reasoning together rather than using the one and only method. A striking example of a particularly effective combination is what has come to be known as "neuro-fuzzy systems." Such systems are becoming increasingly visible as consumer products ranging from air conditioners and washing machines to photocopiers and camcorders. Less visible but perhaps even more important are neuro-fuzzy systems in industrial applications. What is particularly significant is that in both consumer products and industrial systems, the employment of Soft Computing techniques leads to systems which have high MIQ (Machine Intelligence Quotient). In large measure, it is the high MIQ of Soft Computing-based systems that accounts for the rapid growth in the number and variety of applications.

Soft Computing tried to build intelligent and wiser machines. Intelligence provides the power to derive the answer and not simply arrive to the answer. Purity of thinking, machine intelligence, freedom to work, dimensions, complexity and fuzziness handling capability increase, as we go higher and higher in the hierarchy . The final aim is to develop a computer or a machine which will work in a similar way as human beings can do, i.e. the wisdom of human beings can be replicated in computers in some artificial manner. Soft Computing is likely to play an important role in science and engineering, but eventually its influence may extend much farther. In many ways, Soft Computing represents a significant paradigm shift in the aims of computing –a shift which reflects the fact that the human mind, unlike present day computers, possesses a remarkable ability to store and process information which is pervasively imprecise, uncertain, lacking in categoricity and approximations in containing high quality engineering solutions. In Soft Computing, the problem or task at hand is represented in such a way that the “state” of the system can somehow be calculated and compared to some desired state. The quality of the system’s state is the basis for adapting the system’s parameters, which slowly converge towards the solution. This is the basic approach employed by genetic algorithm, or more specifically Evolutionary computing.

1.1.3 WHAT IS HARD COMPUTING?

Traditional computing techniques based on principles of precision, certainty and rigor. The problems based on analytical model can be easily solved using such techniques. Real world problems which deal with changing of information and imprecise behaviour cannot be handled by hard computing techniques. Hard Computing is the ancient approach employed in computing with desired result and developing an accurately declared analytical model. The outcome of Hard Computing approach is a warranted, settled, correct result and defines definite management actions employing a mathematical model or algorithmic rule.

Hard Computing is achieved using sequential programs that use binary logic. It is deterministic in nature. The input data should be exact and the output will be precise and verifiable.

Advantages:

1. Accurate solutions can be obtained
2. Faster

Disadvantages:

1. Not suitable for real world problems
2. Cannot handle imprecision and partial truth

Many contemporary problems do not lend themselves to precise solutions such as –Recognition problems (handwriting, speech, objects, and images), Mobile robot coordination, forecasting, combinatorial problems etc and this is when Soft Computing comes into play. There are many problems related to the computational geometry, for example, finding the shortest path in a graph, finding closest pair of points given a set of points etc. is basically is a task of Hard Computing. And there are many such examples can be given. So, these are the concept of Hard Computing. In contrast to analytical methods, Soft Computing methodologies mimic consciousness and cognition in several important respects: they can learn from experience; they can universalize into domains where direct experience is absent; and, through parallel computer architectures that simulate biological processes, they can perform mapping from inputs to the outputs faster than inherently serial analytical representations.

The proliferation of Soft Computing techniques is remarkable in every field of physics, material sciences, computer chemistry, statistics, etc. This project outlined the various Soft Computing techniques and different applications areas where these techniques have been applied. The robustness, cost effectiveness, simplicity are the few characteristics of the Soft Computing.

1.1.4 SOFT COMPUTING vs. HARD COMPUTING

1. Hard Computing is nothing but the conventional computing; it needs an accurate stated analytical model and sometimes needs a lot of calculation time whereas the Soft Computing deals with uncertainty, biased truth and quite exact to achieve traceability, robustness and fewer cost solutions.
2. The Hard Computing depends on binary logic, crisp systems, and collection of programs. Coming to the Soft Computing it depends on fuzzy logic, neural computing, evolutionary computation, and probability reasoning.

3. Hard Computing has the feature of being accurate and the Soft Computing is near to the accuracy.
4. The Hard Computing needs the list of instructions to be written whereas the Soft Computing can have its own collection of instructions.
5. Hard Computing and Soft Computing are deterministic and stochastic respectively.
6. Hard Computing needs accurate input information, Soft Computing deals with noisy information.
7. Hard Computing follows a sequential process and Soft Computing permits parallel calculations.
8. Hard Computing generates accurate answers and Soft Computing furnishes approximate solutions.

The conventional computing approach i.e., Hard Computing is effective when it comes to solving a deterministic problem, but as the problem grows in size and complexity, the design search space also increases and it becomes difficult to solve an uncertain and imprecise problem by Hard Computing. So, Soft Computing has emerged as the solution to the Hard Computing which also provides a lot of benefits such as fast computation, low cost, elimination of the predefined software, etc.

1.1.5 APPLICATIONS OF SOFT COMPUTING

Rapid advancements in the application of Soft Computing tools and techniques have proven valuable in the development of highly scalable systems and resulted in brilliant applications, including those in biometric identification, interactive voice response systems, and data mining. Although many resources on the subject adequately cover the theoretic concepts, few provide clear insight into practical application. Soft Computing is a promising tool that can provide problem resolution methods, optimization approximation methods including search methods. Many real world problems cannot be solved using Hard Computing techniques that deal with precision and certainty due to the fact that either these real-world problems are difficult to model mathematically or computationally expensive or require huge amounts of memory.

The development of Soft Computing techniques has attracted the interest of researchers from different disciplines over the past two decades. Soft Computing techniques are applied in various domains such as bioinformatics, biomedical systems, data mining, image processing, machine control, robotics, time series prediction, wireless networks, etc.

Wireless Communication: Applications of Soft Computing in wireless communication covers broad area of resource allocation, Handoffs, networking optimization, power control, prediction etc. Artificial Neural Network and Fuzzy Logic methods of Soft Computing involves in resource allocation i.e. by using ANN technique, bandwidth allocation schemes utilization for mobile networks maximizes and minimizes bandwidth allocation for individuals. An algorithm based system can be used to achieve fast and reliable solutions for dynamic resource allocation. Evolutionary Computing method of Soft Computing uses a Soft Computing algorithm to achieve power control in WWAN. The algorithm helps in reducing mobile terminal power consumption and increasing the cellular network capacity. Security in WLAN is also enhanced by using Artificial Neural Network and Fuzzy Logic methods.

Communication Systems: In Communication Systems, Soft Computing can be effectively applied to obtain the solutions that have not been able to solve by Hard Computing. Artificial Neural Network

and Fuzzy Logic combined together and give Neuro-Fuzzy approaches that used for data compression and for equalizer.

Consumer Appliances: The field of consumer appliances activities is associated to practical product development. There is a huge scope of fuzzy logic, neural networks which have already brought artificial intelligence in home appliances. These techniques are used for various applications like washing machines, heaters, refrigerators, microwaves and many more.

Robotics: Robotics is an emerging field which is based on human thinking and behaviour. Fuzzy Logic and Expert System techniques integrate in a way to develop useful real world applications. Neuro-fuzzy approach learns obstacle avoidance and wall-following behaviour on a small size robot. Present day intelligence is considered to be interactive information processing among humans and artificial objects. Intelligence is human like information processing and adaptation to environment by learning, evolution and prediction. Soft Computing is widely used in this field.

Transportation: Soft Computing is applicable in constructing intelligent vehicles and provide efficient environment to each other i.e. to machines and drivers. Intelligent vehicle control requires recognition of the driving environment and planning of driving that is easily acceptable for drivers. The field of transportation deals with passengers, logistics operations, fault diagnosis etc. Fuzzy Logic and Evolutionary Computing are often used in elevator control systems.

Healthcare: Health care environment is very much reliant to on computer technology. With the advancement in computer technology, the use of Soft Computing methods provide better and advance aids that assists the physician in many cases, rapid identification of diseases and diagnosis in real time. Soft Computing techniques are used by various medical applications such as Medical Image Registration Using Genetic Algorithm, Machine Learning techniques to solve prognostic problems in medical domain, Artificial Neural Networks in diagnosing cancer and Fuzzy Logic in various diseases.

Data Mining: Data Mining is a form of knowledge discovery used for solving problems in a particular area. Data sets may be gathered and implemented collectively for purposes others that those for which they were originally created.

Therefore, we see the human mind is incomparable to anything. Human mind does not work on hard and fast rules i.e., it behaves according to the situation. Soft Computing follows the same approach. It is a breakthrough in science and engineering fields as it can solve problems that have not been solved by conventional approaches. It yields rich knowledge which enables intelligent systems to be constructed at low cost. Today we have microwave ovens and washing machines that can decide how to perform the tasks optimally. To build the machines, having human like knowledge, intelligence is required. To understand the concept of “Soft Computing” in a much broader sense, in our project we will try to generate a Sudoku using Soft Computing and in due course of action we will find out how we use appropriate algorithms, which will be dynamic in nature and would adjust to the environment accordingly. We all know that, Sudoku is a well-known puzzle that has achieved international popularity in the latest decade. Recently, there are explosive growths in the application of meta-heuristic algorithms for solving as well as generating Sudoku puzzles. Sudoku is a puzzle game designed for a single player, much like a crossword puzzle. The puzzle itself is nothing more than a grid of little boxes called “cells”.

1.2 AN OVERVIEW OF SUDOKU

Sudoku is a Japanese word consisting of two parts, *su* and *doku*, where the first part means number and the second means single. It has been originally called Number Place which indicates the nature of the game, in which numbers should be placed in appropriate places in a grid. Indeed, it is a logical game which has attracted both young and old people.

The Sudoku game is one of the most interesting challenging games which have been well known around the world. It is spread through different kinds of media, from newspapers to internet sites as well as mobile applications. Beside the popularity of the game, there are factors that attract many researchers to apply different metaheuristics algorithms to generate Sudoku puzzles. The difference between researcher's results lies in their algorithms' abilities to reach Sudoku optimal solution in reasonable time, so the rate of success and the number of iterations and elapsed times are the most important factors in comparing their performance [1]. Sudoku has been claimed to be very popular and even addictive because it's easy to play as it doesn't require general knowledge, linguistic ability or even mathematical skills, with simple rules but very challenging game with different levels.

1.2.1 GENERAL TERMS & RULES

1. **Cell:** A cell is a position on the Sudoku board. Cells contain cell values and pencil-marks (denoted by (n, {a, b, ...,j}) with n, the value of the cell and {a, b, ..., j}, the set of pencil-marks).
2. **Row**: A horizontal alignment of 9 cells in the Sudoku grid.
3. **Column**: A vertical alignment of 9 cells in the Sudoku grid.
4. **Box**: One of the nine 3×3 square groups of cells which together comprise the Sudoku grid.
5. **Group**: A row, column, or box on the Sudoku grid which must contain each digit from 1-9 exactly once.
6. **Given**: A cell whose answer is provided at the beginning of the puzzle.
7. **Pencil-Marks:** Pencil-marks are values placed in a cell that serve to represent the possible choices for the value of a cell.
8. **Cell Value:** The only value a cell can have in order to produce a solvable Sudoku board.
9. **Neighbouring Cell:** Two cells are neighbouring if the two cells are contained within the same row, same column or same box.

Sudoku puzzle with order 9×9 are composed of a 9 × 9 boxes (cells) namely of 81 positions as shown in Figure below, they are divided into nine 3×3 sub-blocks. When the Sudoku game is prepared to play, there are some pre-filled (fixed) numbers which are not allowed to changed or moved during the process of solving Sudoku puzzles, then you will start to fill other unfilled boxes (unfixed) in such way so that each row, column and 3×3 sub-blocks contain each integer {1,2,3,4,5,6,7,8,9} once and only once. Filling all the cells in Sudoku will be the solution [2].
Finally, in solving Sudoku puzzles, we deal with puzzles that have unique solution. We can summarize rules to solve Sudoku into three rules:

Rule 1: Each row should contain values from 1 to 9 without repeating any number.

Rule 2: Each column should contain values from 1 to 9 without repeating any number.

Rule 3: The sub-block should contain only values from 1 to 9 without repetition.

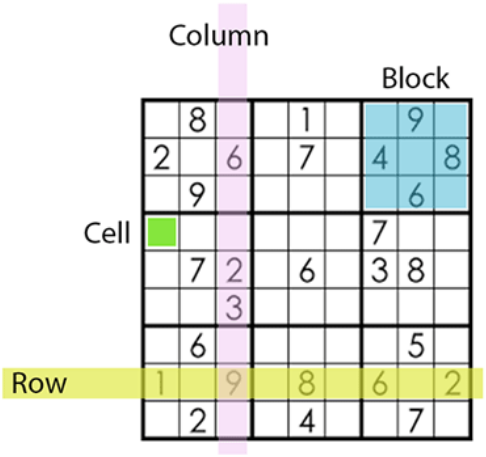

Figure 1: Structure of Sudoku

The general problem of generating Sudoku puzzles on n2×n2 boards of n×n blocks is known to be NP-complete. This gives us some indication of why Sudoku is difficult to solve, although on boards of finite size, the problem is finite and can be solved by a deterministic finite automation that knows the entire game tree. The popularity of Sudoku stimulated many Sudoku solvers to be born. Most of Sudoku solvers were developed using 'algorithmic' approaches [2,5].

1.2.2 CHALLENGES IN SUDOKU FORMATION

A proper Sudoku puzzle should contain a unique solution. Our aim for this challenge is not to generate a Sudoku solver algorithm but instead to create an algorithm to be used by a puzzle setter to produce a well-posed Sudoku grid: a grid with a unique solution [3].

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Figure 2: 9×9 Sudoku

Sudoku puzzles are often given a difficulty level such as "Beginner – Intermediate – Advanced – Expert". But the question arises that how an algorithm could be created to estimate the difficulty level of a Sudoku grid? Should different algorithms be used to generate Sudoku grids for a specific difficulty level? And also we know the fact that Sudoku has *6,670,903,752,021,072,936,960* possible permutations, therefore, it will take years to arrive at a unique solution.

This is not the only problem to deal with, but first of all, the simple fact is that the more difficult the puzzle, the more time it needs to be solved but time to be solved is intimately correlated to both number of clues (givens) and average alternatives to be investigated per empty cell [3]. Therefore, it seems impossible to derive a program depending on each and every difficulty level of a program so, we need such a program which can be adaptive in nature, that is, a program which will run according to the difficulty level of the game and will give us the required grid.

The second difficult task is to make sure that our program is *NP-complete*. Now the question is, what do we mean by NP-complete programs?
NP-complete problems are the hardest problems in NP set. A decision problem L is NP-complete if:
1) L is in NP (Any given solution for NP-complete problems can be verified quickly, but there is no efficient known solution.
2) Every problem in NP is reducible to L in polynomial time.

NP-complete problem, any of a class of computational problems for which no efficient solution algorithm has been found. Many significant computer-science problems belong to this class—e.g., the travelling salesman problem, satisfiability problems, and graph-covering problems. Solving the generalized Sudoku problem is NP-complete. Researchers have tried to propose heuristic algorithms or use existing local search methods.

Unlike the number of complete Sudoku grids, the number of minimal 9×9 Sudoku puzzles is not precisely known. (A minimal puzzle is one in which no clue can be deleted without losing uniqueness of the solution.) However, statistical techniques combined with a puzzle generator show that about (with 0.065% relative error) 3.10×10^{37} minimal puzzles and 2.55×10^{25} non-essentially equivalent minimal puzzles exist [4].Since there is little hope in providing polynomial time algorithms for NP-complete problems, the focus shifted towards understanding the nature of the complexity forbidding fast solutions to these problems. There has been considerable work in this direction, especially for the Boolean satisfiability problem SAT (or k-SAT), which is NP-complete for $k \geq 3$ [4]. Completeness means that all problems in NP (hence Sudoku as well), can be translated in polynomial time. We need to emphasize, however, that the dynamical properties characterize both the problem and the algorithm itself. For this reason, one compares the dynamical properties across problems of varying hardness using the same algorithm. Nevertheless, since there are problem instances that are hard for all known algorithms, the appearance of transient chaos with long lifetime should be a universal feature of hard problems.

The difficulty lies in the difference between the puzzles which can be solved by simple comparisons and those which require speculative reasoning at some point in the solution [4]. We define speculative reasoning to be anything involving the supposition of a particular possibility in a cell. A solver must then either work towards a contraction (ruling out that possibility) or a full solution. Our metric could rely on the number of entries given in the initial puzzle. This supposes a correlation between the density of the initial conditions and the difficulty of the puzzle. From experience, however, it is possible to reach a point at which a significant proportion of the grid is filled but the next step is hard to take. Similarly, a particular arrangement of digits in a sparse grid can still be simple to solve. For these reasons we will not take this idea further.

2. LITERATURE REVIEW

The basis of Sudoku can be dated back to 18th century when great Swiss mathematician Leonard Euler introduced the idea of Latin Squares in 1783.The first Sudoku was published in a puzzle magazine in USA, 1979. Having a very challenging and addictive nature, it has spread wildly all over the world. Sudoku is a logic-based combinatorial puzzle game which is popular among people of different ages. Due to this popularity, computer software(s) are being developed to generate and solve Sudoku puzzles with different levels of difficulty. Several methods and algorithms have been proposed and used in different software(s) to efficiently solve Sudoku puzzles. Various search methods such as stochastic local search have been applied to this problem.

In these works, chromosomes with little or no information were considered and obtained results were not promising. Others impose additional constraints, such as not permitting the same numeral to appear more than once in diagonals or in special sets of 9 cells that have the same colour, etc. For larger puzzles such as 16 x 16 or 25 x 25, GA or other stochastic search method may be effective. Methods for speeding up evolutionary computations through implementations on graphics processing units (GPU) may be also effective. As for manually solving the puzzles the solution integrated with the help of computer would be valuable in the research. Many methods regarding the solving Sudoku puzzles with the help of computer have been put forward but generating a Sudoku puzzle has always been a difficult task. For these basic puzzles, methods such as back-tracking, which counts up all possible combinations in the solution, and meta-heuristics approaches, have been effective in many researches.

Sudoku can further be classified into two categories:

1. Probabilistic Sudoku

A Probabilistic Sudoku is when a starting grid can be filled in different ways and more than one way reach to the objective (i.e. an empty grid is a probabilistic Sudoku).Whenever you talk about the probability of an event, you have to talk about the sample space (or equivalently, the measure) with respect to which you're taking the probability; otherwise, you run into potential paradoxes. And when we're talking about a Sudoku puzzle, the most natural sample space for determining your probability is the space of *correctly filled grids*; by this measure, the probability of 3 being in e.g. the first place is just the number of correctly filled grids with 3 in that space divided by the total number of correctly-filled grids.

2. Deterministic Sudoku

A Deterministic Sudoku consists of an initial grid, partially filled, which admit a unique solution; this is a big advantage for the player that at each step will always have a valid move to do, unlike the probabilistic version that in some steps can admit more possible moves that could lead to several different solutions or to an invalid grid.

The development of an algorithm capable of constructing Sudoku puzzles of varying difficulty entails the preceding formation of a puzzle solving algorithm. Therefore, various methodologies have been used in generation of Sudoku puzzle. Few have been listed below:

2.1 Generating a Sudoku using Non-Evolutionary Algorithms

2.1.1 Constraints Satisfaction Problem

Thomas Bridi introduces the overview of generating a Sudoku by using a Constraint Satisfaction Problem. As we have seen previously in the definition of Sudoku, we deal with constraints, then it seems logical to map the problem as a constraints satisfaction problem and it works perfectly [6].

CSPs are mathematical problems defined as a set of objects whose state must satisfy a number of constraints or limitations. CSPs represent the entities in a problem as a homogeneous collection of finite constraints over variables, which are solved by constraint satisfaction methods. CSPs are the subject of intense research in both artificial intelligence and operations research, since the regularity in their formulation provides a common basis to analyze and solve problems of many unrelated families. CSPs often exhibit high complexity, requiring a combination of heuristics and combinatorial search methods to be solved in a reasonable time. The Boolean satisfiability problem (SAT), the Satisfiability Modulo Theories (SMT) and Answer Set Programming (ASP) can be roughly thought of as certain forms of the constraint satisfaction problem. Examples of simple problems that can be modelled as a constraint satisfaction problem: Eight Queens Puzzle, Map Colouring Problem, Sudoku, Futoshiki, Cross Sums, and many other logic puzzles.

This is an example of the algorithm for the puzzle generation in pseudo language:

given G an empty grid;
do
update the constraints of each cell;
C := random selected cell from G;
PossibleValues := values satisfying the constraints of C;
V := random element from PossibleValues;
set V as value of C;
while G does not admit solution;

Where “update the constraints of each cell” and “G does not admit solution” consist to apply the constraints described by the rules of the game. In order to make a more attractive puzzle, we must make it more difficult to solve; to do this we have to modify the “G does not admit solution” function taking advantage of more sophisticated techniques to detect more complex Sudoku.

In our case we do not need backtracking because this algorithm is designed to work on a large amount of puzzles, the use of backtracking is not necessary and also goes to degrade the performance in terms of memory usage, so in case of failure the algorithm will restart from an empty grid; for simplicity we will call this algorithm Naive Forward Checking because it uses the forward checking constraint propagation but not the backtracking.

It is intuitive to understand that, in this way we will not have the security of generating more complex Sudoku but only to recognize it, then the next step is to understand how it is possible to get closer to the optimal solution of our problem, thanks to artificial intelligence.

2.1.2 Inverse Problem

Domenico Bloisi and Luca Iocchi [7] proposed the idea of generating Sudoku as an inverse problem and it is necessary to invert the methods that are used to solve Sudoku puzzles. Before this can be done, some discussion of the forward methods is an obvious prerequisite. First of all, they constructed a more robust mathematical model of a Sudoku board which provided a framework on which we can build the forward and inverse methods. Then, they defined some forward methods that are commonly used, and deduced inverse methods for each one.

Fortunately, when the methods are reduced to a single operation, their descriptions became quite simple. So, for each method, we provide a concise verbal description followed by a rigorous mathematical definition followed by a list of preconditions required to perform a simple action, with the exception of the single candidate method, this action will be to remove a pencil-mark from a cell. Then, the inverse function is defined similarly, where the action will be to add a pencil-mark to a cell.

With the exception of the single candidate method, the inverse method is not proper – in general, one can utilize an inverse method to remove too much information from the board. However, the inverse method may be applied to undo the forward method in every case. It important to note that these are not functions – at any given point, it may be possible to execute a method on the board in a large number of ways – so, it is no surprise that we are unable to properly invert these methods.

In general, the study of inverse problems has brought fresh insight to many topics. Often, an inverse problem may seem more complicated at first glance, but lead to interesting simplification of the corresponding forward problem. The authors have not seen any previous study of the inverse Sudoku problem, which suggests that there could be benefit to looking at the problem in this way. The inverse methods which we have examined seem to be exactly as simple (or complicated) as their corresponding forward methods, which may provide interesting insight to Sudoku.

2.1.3 Digging Hole Strategy & Transformation Method

In Digging Hole strategy, an already solved Sudoku puzzle is taken as input. Then a number of values from cells are removed in some specified sequence based on the level of difficulty, and then the generated puzzle is checked for having a unique valid solution or more. In the second method, new instances are generated based on transformation(s).

Here an existing Sudoku instance of certain difficulty level is taken as input. Then different kinds of transformation are applied on it, and as a result new Sudoku instances are generated [10].

The Digging Hole Strategy and Transformation Method can be represented in the form a flowchart mentioned below.

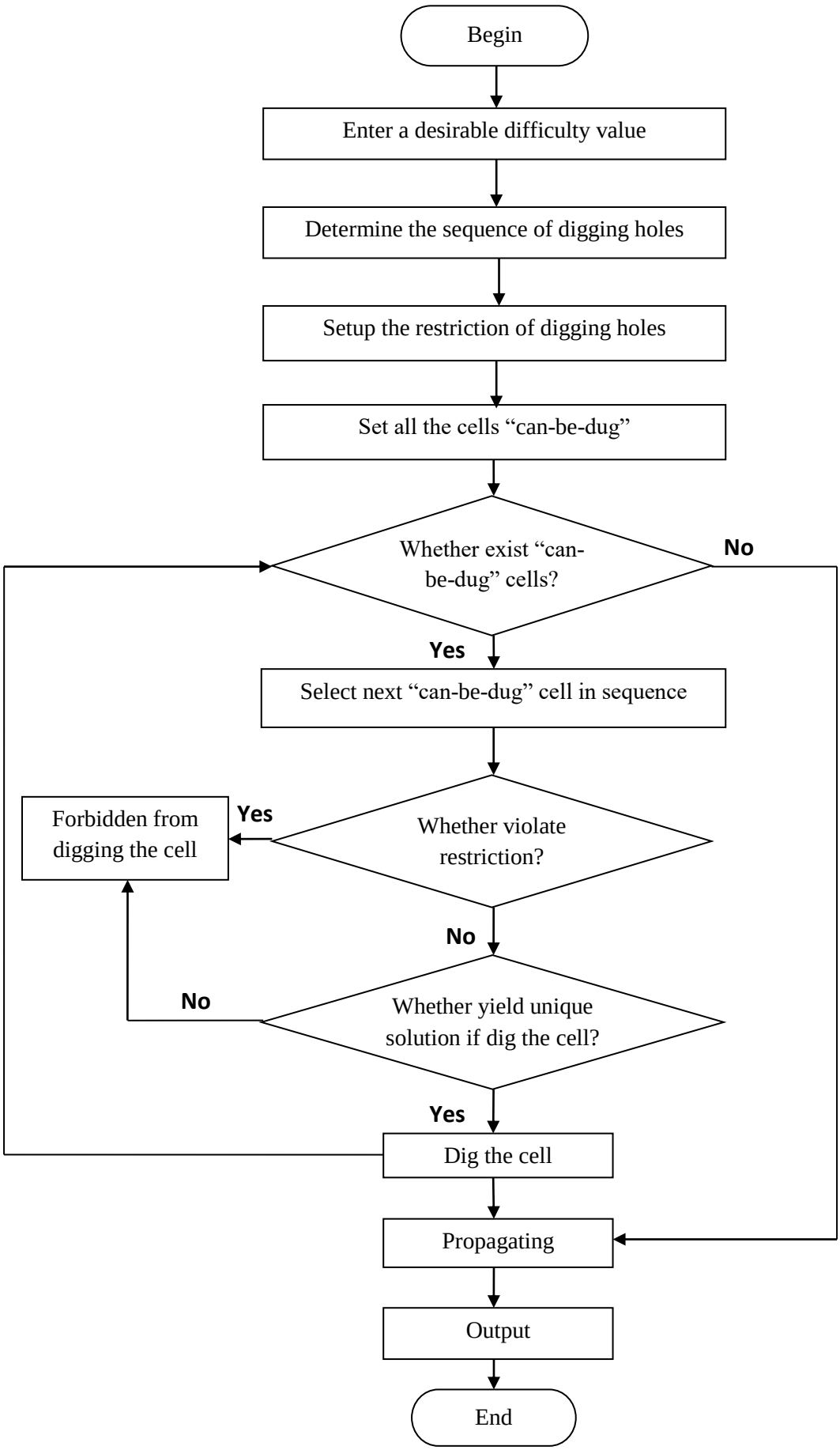

Figure 3: Flowchart of Digging Hole Strategy

2.2 Generating a Sudoku using Evolutionary Algorithms

2.2.1 Backtracking

David Martin, Erica Cross and Matt Alexander researched on the deterministic approach of cracking a Sudoku. In their technique, the construction of a Sudoku puzzle begins with the generation of a solution. Rather than starting with an empty grid and adding numbers, the program begins with a complete Sudoku, produced by a random-number-based function within the program. The advantage is obvious; rather than starting with a puzzle lacking a unique solution and hoping to stumble upon one with a unique solution, the program begins with a puzzle having a unique solution and maintains the uniqueness [5].

Once a completed Sudoku has been created, the puzzle is developed by working backwards from the solution, removing numbers one by one (at random) until one of several conditions has been met. These conditions include a minimum difficulty rating (to ensure that the puzzle is hard enough) and a minimum number of empty squares (to ensure that the puzzle is far from complete). Following each change in the grid, the difficulty is evaluated as the program solves the current puzzle. If the program cannot solve the current puzzle, then one of two possibilities must be true: either the puzzle does not have a unique solution, or the solution is beyond the grasp of the logical methods possessed by the algorithm. In either case, the last solvable puzzle is restored and the process continues.

In theory, there may occur a situation in which removing any given number will not yield a puzzle that is solvable (by the algorithm) and has a unique solution so in such a case, the puzzle creator has reached a “dead end”, and cannot make further progress toward a higher difficulty rating. If there is not sufficient information to draw further conclusions, such as the event that an arbitrary choice must be made (which must invariably occur for a puzzle with multiple solutions to be solved) the solver simply stops. But it may result in an error for Sudoku puzzles which either have no solution or multiple solutions but it does not differentiate between the two.

There are three major drawbacks of the standard backtracking scheme:

1. One is *thrashing*, i.e., repeated failure due to the same reason. Thrashing occurs because the standard backtracking algorithm does not identify the real reason of the conflict, i.e., the conflicting variables. Therefore, search in different parts of the space keeps failing for the same reason.
2. The other drawback of backtracking has to perform *redundant work*. Even if the conflicting value of variables is identified during the intelligent backtracking, they are not remembered for immediate detection of the same conflict in a subsequent computation.
3. Finally, the basic backtracking algorithm still *detects the conflict too late* as it is not able to detect the conflict before the conflict really occurs, i.e., after assigning the values to the all variables of the conflicting constraint. This drawback can be avoided by applying consistency techniques to forward check the possible conflicts.

2.2.2 Genetic Algorithm (GA)

This project studies the problems involved in generating Sudoku puzzles with Genetic Algorithms (GA). Sudoku is a number puzzle that has recently become a worldwide phenomenon. Sudoku can be regarded as a constraint satisfaction problem. When solved with genetic algorithms it can be handled as a multi-objective optimization problem.

The three objectives of this study was:
1. To test if genetic algorithm optimization is an efficient method for solving Sudoku puzzles,
2. Can GA be used to generate new puzzles efficiently, and
3. Can GA be used as a rating machine that evaluates the difficulty of a given Sudoku puzzle.

The last of these objectives is approached by testing if puzzles that are considered difficult for a human solver are also difficult for the genetic algorithm. The results presented in this project seem to support the conclusion that these objectives are reasonably well met with genetic algorithm optimization [8,9].

In 2006, T.Mantere and J.Koljonen published a paper that describe how genetic algorithms can solve a puzzle used like a genome and also by applying swap operations; if given a blank puzzle, the algorithm fills up the grid generating a valid and solved Sudoku, but that is unplayable, the next step would be to remove the numbers from the grid to get a Sudoku with a feasible solution and the fewest number [9].

The result was that by applying the mutation, the average difficulty of the population tends asymptotically to a certain value with some small deviation from time to time; this asymptote depends on the fitness function but also by the number of techniques used by the solving algorithm. Another test was made to calculate the percentage of successfully mutated grids and the percentage of mutation that produced a better solution, this test was done on starting population of about 40200 Sudoku grids and the result was that in average only the 8.24% of starting grid ported successfully to a deterministic puzzle before the fail of the naive forward checking algorithm [10]; a possible way to increase this percentage could be to implement backtracking at least for the puzzle generation from genotype; the second result was that the percentage of mutation leading to a more difficult Sudoku is 35.88%. About performance: this algorithm takes 3 hours for generating and mutating 10,000 puzzles.

3. METHODOLOGY

Traditional generate-and-test solution strategies work very well for the classic form of the puzzle, but break down for puzzle sizes larger than 9x9 is quite a difficult task. An alternative approach is the use of genetic algorithms (GAs). GAs has proved effective in attacking a number of NP-complete problems, particularly optimization problems such as the Travelling Salesman Problem. In this study, we will concentrate on Sudoku generation using evolutionary methods. The main reason for generating Sudoku with GA is to learn more about capabilities of GA in constrained combinatorial problems, and hopefully to learn new tricks to make it more efficient also in this field of problems.

3.1 GENETIC ALGORITHM

Genetic Algorithms was first proposed by John Henry Holland and his students and colleagues at the University of Michigan, most notably David E. Goldberg and has since been tried on various optimization problems with a high degree of success.

Genetics Algorithm is a part of Evolutionary Computation. The principle under genetic algorithms is the same principle of the evolutionary theory of Darwin: in an environment, individuals that survive and reproduce, are those with better characteristics, so their genes are partially passed on to the next generation. In evolutionary computation, an individual is a possible solution of a problem, a population is a set of individuals, there is a fitness function that describe how many solutions are near to the optimal solution of the problem, every individual is obtained starting from a genotype, a genotype is the representation of all the variables that can make an individual different from another [11].

Nature has always been a great source of inspiration to all mankind. Genetic Algorithms (GAs) are search based algorithms based on the concepts of natural selection and genetics [11]. GA's are a subset of a much larger branch of computation known as *Evolutionary Computation*.

In GAs, we have a *pool or a population of possible solutions* to the given problem. These solutions then undergo recombination and mutation (like in natural genetics), producing new children, and the process is repeated over various generations. Each individual (or candidate solution) is assigned a fitness value (based on its objective function value) and the fitter individuals are given a higher chance to mate and yield more "fitter" individuals. This is in line with the Darwinian Theory of "Survival of the Fittest" [11]. In this way we keep "evolving" better individuals or solutions over generations, till we reach a stopping criterion.

Genetic Algorithms are sufficiently randomized in nature, but they perform much better than random local search (in which we just try various random solutions, keeping track of the best so far), as they exploit historical information as well.

Genetic Algorithms have the ability to deliver a "good-enough" solution that is "fast-enough" [11]. This makes genetic algorithms attractive for use in solving optimization problems. The reasons why GAs are needed are as follows –

Solving Difficult Problems

In computer science, there is a large set of problems, which are *NP-Hard*. What this essentially means is that, even the most powerful computing systems take a very long time (even years) to solve that problem. In such a scenario, GAs proves to be an efficient tool to provide *usable near-optimal solutions* in a short amount of time.

Getting a Good & Fast Solution

Some difficult problems like the Travelling Salesperson Problem (TSP), have real-world applications like path finding and VLSI Design. Now imagine that you are using your GPS Navigation system, and it takes a few minutes (or even a few hours) to compute the "optimal" path from the source to destination. Delay in such real world applications is not acceptable and therefore a "good-enough" solution should also be delivered as "fast-enough" solution too.

3.2 BASIC STRUCTURE OF GENETIC ALGORITHM

The basic structure of a GA is as follows −

We start with an initial population (which may be generated at random or seeded by other heuristics), select parents from this population for mating. Apply crossover and mutation operators on the parents to generate new off-springs. And finally these off-springs replace the existing individuals in the population and the process repeats. In this way genetic algorithms actually try to mimic the human evolution to some extent.

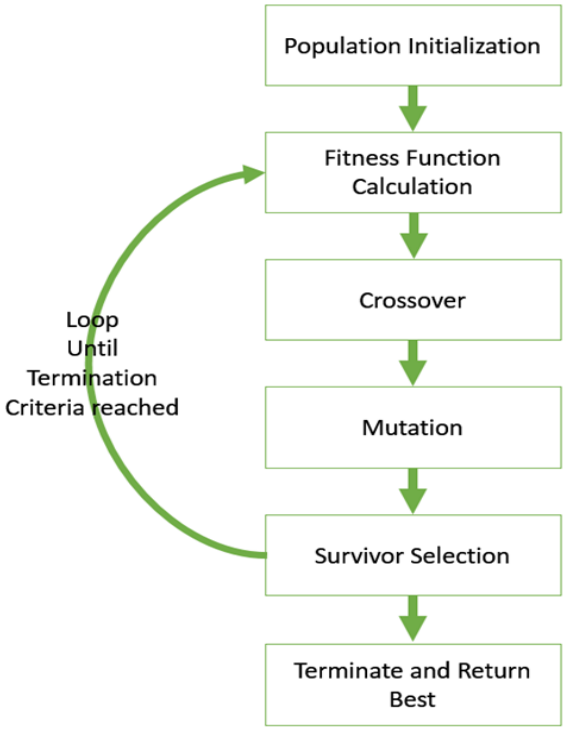

Figure 4: Basic Structure of Genetic Algorithm

The whole algorithm can be summarized as:

1. Randomly initialize populations P
2. Determine the fitness of population
3. Until conversions repeat:
   a) Select parents from population
   b) Crossover and generate new population
   c) Perform Mutation on new population

The general steps that are followed in Genetic Algorithm are as given below:

POPULATION INITIALIZATION

Population is a subset of solutions in the current generation. There are two primary methods to initialize a population in a GA. They are −

Random Initialization − Populate the initial population with completely random solutions.

Heuristic initialization − Populate the initial population using a known heuristic for the problem.

It has been observed that the entire population should not be initialized using a heuristic, as it can result in the population having similar solutions and very little diversity. It has been experimentally observed that the random solutions are the ones to drive the population to optimality. Therefore, with heuristic initialization, we just seed the population with a couple of good solutions, filling up the rest with random solutions rather than filling the entire population with heuristic based solutions.

FITNESS FUNCTION

The fitness function simply defined is a function which takes a *candidate solution to the problem as input and produces as output* how "fit" our how "good" the solution is with respect to the problem in consideration. Calculation of fitness value is done repeatedly in a GA and therefore it should be sufficiently fast [12]. A slow computation of the fitness value can adversely affect a GA and make it exceptionally slow.

In most cases the fitness function and the objective function are the same as the objective is to either maximize or minimize the given objective function. However, for more complex problems with multiple objectives and constraints, an algorithm designer might choose to have a different fitness function. A fitness function should possess the following characteristics-

1. The fitness function should be sufficiently fast to compute.
2. It must quantitatively measure how fit a given solution is or how fit individuals can be produced from the given solution.

In some cases, calculating the fitness function directly might not be possible due to the inherent complexities of the problem at hand. In such cases, we do fitness approximation to suit our needs.

CROSSOVER

Crossover is a genetic operator used to vary the programming of a chromosome or chromosomes from one generation to the next [13]. Crossover is sexual reproduction. Two strings are picked from the mating pool at random to crossover in order to produce superior offspring. The method chosen depends on the Encoding Method.
Different types of crossover are as follows:

Single Point Crossover: A crossover point on the parent organism string is selected. All data beyond that point in the organism string is swapped between the two parent organisms. Strings are characterized by Positional Bias.

Two-Point Crossover: This is a specific case of a N-point Crossover technique. Two random points are chosen on the individual chromosomes (strings) and the genetic material is exchanged at these points.

Uniform Crossover: Each gene (bit) is selected randomly from one of the corresponding genes of the parent chromosomes. Use tossing of a coin as an example technique.

The crossover between two good solutions may not always yield a better or as good a solution. Since parents are good, the probability of the child being good is high. If offspring is not good (poor solution), it will be removed in the next iteration during "Selection" [14].

MUTATION

Mutation is a genetic operator used to maintain genetic diversity from one generation of a population of genetic algorithm chromosomes to the next. It is analogous to biological mutation. Mutation alters one or more gene values in a chromosome from its initial state. In mutation, the solution may change entirely from the previous solution. Hence GA can come to a better solution by using mutation. Mutation occurs during evolution according to a user-definable mutation probability. This probability should be set low. If it is set too high, the search will turn into a primitive random search.

The classic example of a mutation operator involves a probability that an arbitrary bit in a genetic sequence will be changed from its original state. A common method of implementing the mutation operator involves generating a random variable for each bit in a sequence. This random variable tells whether or not a particular bit will be modified. This mutation procedure, based on the biological point mutation, is called single point mutation. Other types are inversion and floating point mutation. When the gene encoding is restrictive as in permutation problems, mutations are swaps, inversions, and scrambles.

SURVIVOR SELECTION

The Survivor Selection Policy determines which individuals are to be kicked out and which are to be kept in the next generation. It is crucial as it should ensure that the fitter individuals are not kicked out of the population, while at the same time diversity should be maintained in the population.

Some GAs employs *Elitism*. In simple terms, it means the current fittest member of the population is always propagated to the next generation. Therefore, under no circumstance can the fittest member of the current population be replaced. The easiest policy is to kick random members out of the

population, but such an approach frequently has convergence issues, therefore the following strategies are widely used.

TERMINATION CONDITION

The termination condition of a Genetic Algorithm is important in determining when an execution will end. It has been observed that initially, the GA progresses very fast with better solutions coming in every few iterations, but this tends to saturate in the later stages where the improvements are very small. We usually want a termination condition such that our solution is close to the optimal, at the end of the run. Usually, we keep one of the following termination conditions −

1. When there has been no improvement in the population for X iterations.
2. When we reach an absolute number of generations.
3. When the objective function value has reached a certain pre-defined value.

For example, in a genetic algorithm we keep a counter which keeps track of the generations for which there has been no improvement in the population. Initially, we set this counter to zero. Each time we don't generate off-springs which are better than the individuals in the population, we increment the counter.

However, if the fitness any of the off-springs is better, then we reset the counter to zero. The algorithm terminates when the counter reaches a predetermined value. Like other parameters of a GA, the termination condition is also highly problem specific and the GA designer should try out various options to see what suits his particular problem the best.

This is how genetic algorithm actually works, which basically tries to mimic the human evolution to some extent. So to formalize a definition of a genetic algorithm, we can say that it is an optimization technique, which tries to find out such values of input so that we get the best output values or results [14].The genetic algorithm repeatedly modifies a population of individual solutions. At each step, the genetic algorithm selects individuals at random from the current population to be parents and uses them to produce the children for the next generation. Over successive generations, the population "evolves" toward an optimal solution. You can apply the genetic algorithm to solve a variety of optimization problems that are not well suited for standard optimization algorithms, including problems in which the objective function is discontinuous, non-differentiable, stochastic, or highly nonlinear.

Their most interesting properties are:

1. Efficiency.
2. Simple programmability.
3. Extraordinary robustness regarding the input data.

The most important property is robustness, and this represents an emulation of nature's adaptive algorithm of choice. Mathematically, it means that it is possible to find a solution even if the input data do not facilitate finding such a solution. Besides the above-mentioned advantages, there is one major disadvantage: GAs has to be carefully designed. An unfavourable choice of operators might affect the outcome of the application. Therefore, a precise knowledge of the basics and the context is crucial for any problem solution based on GAs. In a more detailed sense, the GA represents a population-based model, which employs selection and recombination operators to generate new data points in a search space. There are several GA models known in the literature, most of them designed as optimization tools for several applications.

4. PROJECT PLANNING

The Sudoku game is one of the most interesting challenging games which have been well known around the world. A proper Sudoku puzzle should contain a unique solution. Our aim for this challenge is not to generate a Sudoku solver algorithm but instead to create an algorithm to be used by a puzzle setter to produce a well-posed Sudoku grid: a grid with a unique solution [13]. The general problem of solving Sudoku puzzles on n2 ×n2 boards of n×n blocks is known to be NP-complete. This gives some indication of why Sudoku is difficult to solve, although on boards of finite size, the problem is finite and can be solved by a deterministic finite automation that knows the entire game tree. The popularity of Sudoku stimulated many Sudoku solvers to be born. Most of Sudoku solvers were developed using 'algorithmic' approaches [15].

We are implementing Genetic algorithm as a way-out of Soft Computing in Sudoku formation. According to Genetic algorithm steps our own implementation functions are as follows:-

Random Matrix Generation

Random matrix of 9×9 containing nine sub blocks of 3×3 is generated here. They will be considered as the operating elements for the entire program.

Fitness Function

According to Sudoku game rule, Fitness here is calculated every time whenever any repetitions of a number is identified in a single matrix (row-wise, column-wise or sub-grid wise) and counter is incremented. And once the Fitness is calculated, then we will proceed further depending upon the Fitness Score. This function is repeatedly used after every step of the program.

Selection

Selection process is the most important function of the entire project. After we get the fitness of the initial matrices, we have to select lower valued matrices from the rest and proceed on to further steps. Selection is based upon the decreasing fitness value of the matrices. We will consider only those matrices for selection whose fitness value will be less. Our next functions will depend upon the result of selection.

Crossover

Crossover is done by swapping the elements of any two sub grids of any two different matrices. Crossover cannot be done in between two sub grids of a single matrix because it will not be fruitful for the other conditions of Sudoku rule which are row-wise and column-wise repetitions. Sub-grids are considered as offspring.

Mutation

Mutation is done by replacing any number of any random sub-grids of any single matrix by a random new number ranging from 1 to 4. As this is not a binary system, we can't do proper Mutation of Genetic Algorithm by alternating '1' and '0' only.

Loop Creation

As the whole process is based upon randomness, we can't assure the optimal solution in single time execution. Therefore, to overcome this boundary, a loop has been created and put down the Crossover and Mutation part of the program including their Fitness function calculation within that. When the Fitness Score of any single matrix will be '0' after executing the loop several times, then we can consider it as our optimal solution.

4.1 PERT Chart

A PERT chart is a graphic representation of a project's schedule, showing the sequence of tasks, which tasks can be performed simultaneously, and the critical path of tasks that must be completed on time in order for the project to meet its completion deadline. The chart can be constructed with a variety of attributes, such as earliest and latest start dates for each task, earliest and latest finish dates for each task, and slack time between tasks. A PERT chart can document an entire project or a key phase of a project.

Tasks can also branch out and travel their own paths rejoining the main path at some later point. Any milestones such as points of review or completion are indicated as well.

Activity	Optimistic (weeks)	Most Likely (weeks)	Pessimistic (weeks)
Determination of Objective and Scope	3	4	6
Analysis (Survey)	5	6	7
Study	5	6	8
Planning	4	5	8
Module Specification	3	4	6
Implementation	7	8	10
Document Compilation	5	6	8

Table 1: PERT Activity Time Estimates

4.2 PROBLEM DEFINITION

Our main problem is to generate a Sudoku puzzle. For that purpose, first we have to proceed towards generating a random solved Sudoku. In our format, we are implementing Sudoku as a version of 9x9 grids which has 3×3 sub grids. In that case, we are applying Soft Computing. In Soft Computing, there are different ways to generate a Sudoku problem. Among them, we have chosen Genetic Algorithm for our project.

Nevertheless, since there are problem instances that are hard for all known algorithms, the appearance of transient chaos with long lifetime should be a universal feature of hard problems. The larger the search space, the more challenging it is to find and verify an optimal solution [12]. Also, any case that has only one global optimum within a plethora (group) of sub-optimal solutions results in the proverbial 'needle in a haystack' scenario. Finally, finding the optimal solution may require the algorithm to perform an extensive search any time when it reaches a local optimum, which is generally impractical in terms of the time it would take to find the optimal solution. For problems such as the travelling salesman problem, the only way to verify that a solution is the global optimum is an exhaustive search of every possible solution. Many problems contain local optimum's that can 'trap' many searching algorithms (these local optimums are often called 'basins'). A searching algorithm can find a local optimum within a basin and unless the algorithm is designed to be able to escape said basin, it will likely never find a better solution than the one in its current location [12]. Without the capacity to escape, the algorithm could easily (and incorrectly) determine that it has found the global optimum. Local optimums are much easier to find compared to global optimums in the majority of cases.

Genetic Algorithm is such an algorithm which helps us to find an optimal solution with random preferences. A potential solution in a genetic algorithm fits a genotype that is defined at the outset of the program [8]. For example, a genotype for a travelling salesman problem may be a one dimensional array containing a permutation of the cities the salesman must visit. So if we proceed to our project with the Genetic Algorithm then we can find an optimal way of Sudoku generation and that will lead us to success.

In terms of experience, as we are novice in this field, we have tried different ways to compare their execution time(s) and complexities as well as regarding space. Genetic Algorithm implementation in Sudoku formation is a safe and simple way for a beginner to reach their destination [4,8]. Sudoku provides an example of a very large, variable search space that depends solely on the initial hint distribution [19]. The size of the search space for a specific Sudoku puzzle grows exponentially with each starting blank. A genetic algorithm has the ability to find optimal solutions within a large search space without requiring much user supplied information.

Soft Computing is probably the best way to get an optimal and unique solution for that type of problem which rarely finds a solution with hard computing or brute-force method or any optimization method(s). So, to make a universal way-out in Sudoku formation problem, application of Soft Computing (Genetic Algorithm) should be most preferable.

5. SOFTWARE AND HARDWARE REQUIREMENTS

HARDWARE REQUIREMENTS

1. Processor: Minimum Recommended 2 GHz or more
2. Storage space: Minimum 125MB for JRE
3. Memory (RAM): Minimum recommended 2GB or above

OPERATING SYSTEMS

1. Windows**:** 7 or newer
2. MAC**:** OS X v10.7 or higher
3. Linux**:** Ubuntu

SOFTWARE REQUIREMENTS

1. JDK 10.0.1
2. PDF Reader

6. IMPLEMENTATION

6.1 FLOWCHART

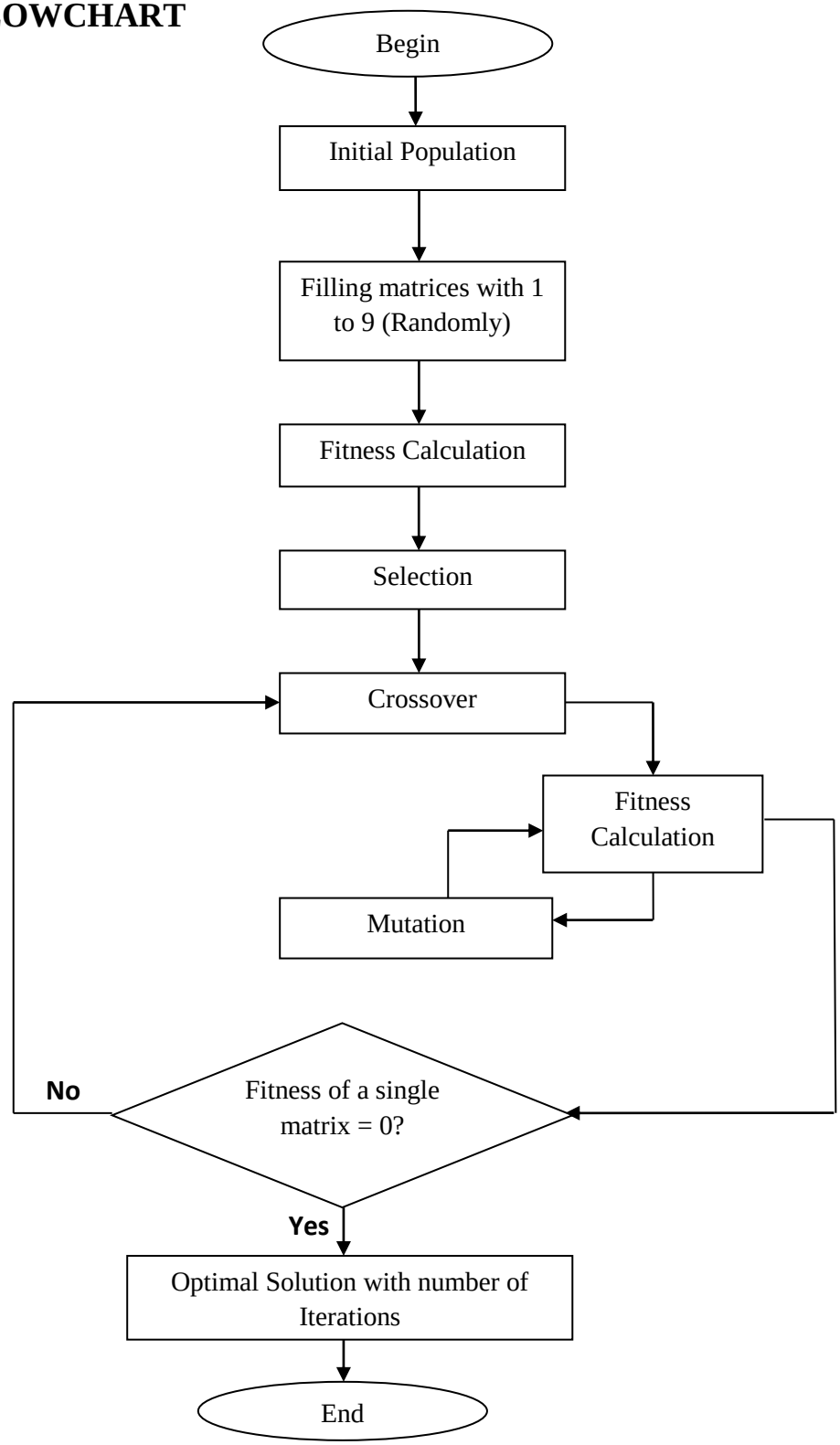

Figure 5: Flowchart of our approach

6.2 ALGORITHM / WORKFLOW

We all know that Sudoku is of 9x9 grid which has 3x3 sub grids and we are implementing it with the help of Soft Computing. Since, there are numerous ways to generate a Sudoku problem. Among them, we have chosen Genetic algorithm in our project. The entire program has been written in Java language and each sub-type under genetic algorithm is categorized under different functions to decrease the ease to understand it. Function calls are used to initiate those functions.

Initialization of Sudoku

1. According to Genetic Algorithm, initial Population is the first step. As per the definition of Genetic algorithm, in initial population, each and every element is considered as genes and offspring's are considered as chromosomes. If we have to implement this concept, then we need to identify our matrices as genes. Each individual in the population is characterized by a set of parameters, such as: initial configuration of the Sudoku grid, a generation, which represents the current state of the individual in one dimensional array, and the fitness value of the individual. So at first, we have to initialize the Sudoku with random numbers using rand() function. Later, defined a matrix which will resemble the initial configuration of the Sudoku grid and then initializing the population size and maximum iterations accordingly. The matrix consists of integer numbers ranging from 1 to 9 organized as per the Sudoku configuration but in random order. In many cases, we will be converting the matrices from 1D to 2D or vice-versa as per the program.

2. We have also introduced the concept of vector and cloning of array. The reason behind the concept of cloning is to ensure that any changes made inside the data while executing the program should not affect the original data and Vectors, on the other hand, is useful to expand the size of an array in advance or over the lifetime of a program. Also in our project, since we are dealing with random numbers therefore, Vectors are good options to deal with varying sizes.

3. Next, we have created a 1D array (Sudoku array) and kept the clone of the original matrix, only after converting it from 2D to 1D, into the array. Again, created a new 1D array (current Sudoku) and allocated it with the population size. Finally, we cloned the Sudoku array and represented it as an Initial gene. After that, randomly populated the current Sudoku with the cloned array and converted it into 2D array and printed it.

4. Under the function SudokuGenetic(), we will create an object(rand) using random(). The bestselection() function will be called and populate the current Sudoku with fit individuals and check if it is not equal to zero. Then traversing the entire population, and checking if rand.nextboolean() is true or false and accordingly it will call the mutation function and the crossover function. Since, our program is dealing with randomness, therefore, we have used this function (rand.nextboolean()) to keep the system in random format instead of having any predefined condition. Once the traversal is completed, the condition rand.nextboolean() is checked again. Now this time, if it is true then we will initialize the current Sudoku with the bestselection() from the population otherwise with the randomselection() from the population. Iteration is incremented. Even after the entire traversal, if we don't find any fit individual then we will repopulate the current Sudoku by cloning it and turning it into a gene and repeating the same process again by calling the fitness() function. Or else, if we find the final solution then we will print the Sudoku with its number of iterations.

Fitness Function

Fitness Function evaluates how close a given solution is to the optimum solution of the desired problem. It determines how fit a solution is.

1. In genetic algorithms, each solution is generally represented as a string of binary numbers, known as chromosomes. We have to test these solutions and come up with the best set of solutions to solve a given problem. Each solution, therefore, needs to be awarded a score, to indicate how closely it resembles the overall specification of the desired solution. This score (fitness value) is generated by applying the fitness function to the test, or results obtained from the tested solution.

2. Here we are declaring two arrays *initialGene[]* and *gene[]* where initialGene[] is the initial population. We are doing all our computation on the array gene[] so that the actual initial population is not altered. Fitness function and random functions are declared. The fitness value of gene has to be calculated i.e. among the huge initial population, we have to find out the fittest elements for better results.

3. Now, for the calculation of fitness value, we will first initialize it with zero. Both the array initialGene[] and gene[] has to be converted to a 2D array. Since here we will be working with a 9x9 matrix, so we have to deal with 81 entries. Now to convert 1D array to 2D array, we are using the function *Convert1Dto2D()* which converts both the arrays from 1D to 2D using the *math.sqrt().* In math.sqrt(), we are taking the length of the array as parameter and this function computes its square root and returns the length of the 2D array.

4. The next step is to check the row and column errors of the obtained matrix for which we have created two arrays *rowError[]* and *colError[]* which are initially set to boolean value 'TRUE'. We have the required 2D array so we will check for errors to compute the fitness value which was initially zero. We are first traversing the whole matrix starting from index value 0 to the length of the array. Whenever any conflict is encountered between the row element and the column element, the fitness value is incremented by 1.

5. The box size will be determined using the math.sqrt() which is equal to the square root of the newGene's length.

a) We need to traverse the whole matrix for error checking. This will be done in two parts i.e. i and j will traverse the main matrix and k and l will traverse the sub-grids.

b) After traversing the main matrix, an array *boxError* is created whose length is equal to the length of newGene. Again, the sub-grids are traversed and occurrence of conflict is checked.

c) If there is any conflict between the elements of main matrix and sub-grids, fitness is incremented and the boxError of the newGene with the conflicting elements is set to boolean value 'TRUE'.

At last, the final fitness value is returned.

Crossover

After the calculation of fitness, we implemented the Crossover step of Genetic Algorithm in our project. In this step, we have created two integer type one-dimensional matrices (*gene1* and *gene2*) which is the local reference of the matrix(*gene*). Then we have initialized two integer variables (start and end) with random numbers respectively and made sure that the value of 'start' is always less than 'end'. Then we have created two one-dimensional matrices (*newgene1* and *newgene2*) with the same allocation memory as of 'gene1' which represents the two sub-grids of two different matrices. Then we performed the swapping operation between those two sub-grids.

We have selected any two sub-grids of different matrices in random order. The elements of those two sub-grids are swapped and the rest of the matrix remains unchanged. We can't apply this swapping between two sub-grids of a single matrix, because it will not help the problem of row-wise or column-wise repetitions. After swapping, we made the desired changes to the respective sub-grids of the original matrix(gene).

Mutation

Once the crossover is completed, we proceed further to the next step of Genetic Algorithm i.e. Mutation. In this step, we have selected an index of an element of any sub-grid of the matrix (gene) randomly. Then, we have replaced the selected number with a randomly system generated number within certain range. In Genetic algorithm, Mutation is done by replacing '0' by '1' and vice-versa. In our project, as we are working with integers ranging from 1 to 9, so we just can't interchange two numbers as it is done in the actual Mutation process of Genetic Algorithm. But, each individual in the population is subject to a mutation. So, we have implemented the Mutation step in a slightly different way for the sake of our purpose in order to obtain minimum fitness value. Thus, we have replaced the cell value with a valid integer number ranging from 1 to 9.

Selection Process

The Selection Policy determines which individuals are to be kicked out and which are to be kept in the next generation. It is crucial as it should ensure that the fitter individuals are not kicked out of the population, while at the same time diversity should be maintained in the population.

Some GAs employs *Elitism*. In simple terms, it means the current fittest member of the population is always propagated to the next generation. Therefore, under no circumstance can the fittest member of the current population be replaced. The easiest policy is to kick random members out of the population, but such an approach frequently has convergence issues, therefore the following strategies are used.

1. We will consider the fitness values of the matrices and we will go for those who have it minimized. Our optimum target is to minimize the fitness value. So, we will select the lower fitness valued matrices which are actually fitter.

2. If we get the fitness value of any matrix equals to zero, then we will consider it as our solution. Otherwise, selection process will be carried on by choosing the minimum fitness valued matrices for next round of the process. Therefore, we have created two functions named *Bestselection()* and *Randomselection().*

3. Under the Bestselection() function, we are considering a vector 'sudokus', whose first value is stored in minimum. After that, we will traverse the entire Sudoku one by one and compare the

calculated fitness values with the minimum. If we encounter a value which will be less than minimum, then we will store that value into the minimum. Selection is generally done according to the best selection function.

4. After best selection, we are left with unfit individuals which may contain few fit individuals. Now, our next task is to perform the random selection among all the individuals. The Randomselection() function will help us to separate unfit individuals from the population. Under this function, we are initializing a max and a sum variable with the help of which we will be randomly choosing individuals (fit) and populate the current Sudoku, and rest of the process will be carried out accordingly. In case, if we get zero in any of the values, it will automatically be considered as the best selection.

The program continuous to iterate until the fitness function of the best selection is not zero in which case that generation is chosen as the solution of the Sudoku problem.

This is how the different functions works in the program.

How our code works?

At first, our code begins with defining a two dimensional 9×9 matrix, say M, consisting of integers ranging from 0 to 9. After that, we will be initializing the population size and maxIteration as per our requirement. Though, we have defined the population size and maxIteration in our code, but we may make alterations in future and convert it into user input. Then, we will make a clone of the previously defined two dimensional matrix M into one dimensional array in order to prevent any changes in the original matrix. Once we get the one dimensional array, we will clone the array again therefore, setting it as the initial gene and invoking the initializeSudoku function and storing it in another array, named currentSudoku. The next step will be the calculation of the fitness value of the current sudoku and invoking the next functions accordingly.

Next, we will be populating the current Sudoku with the elements from Best Selection and check the fitness value of those elements that whether the value is equal to 0 or not. If the value is equal to 0, then the obtained matrix will be our optimal solution otherwise crossover and mutation function will be invoked and iterations will be incremented accordingly. Whenever crossover and mutation functions are executed, fitness value is calculated simultaneously.

6.3 PERT Chart

Activity	Time Taken (weeks)
Determination of Objective and Scope	3
Analysis (Survey)	5
Study	5
Planning	4
Module Specification	3
Implementation	7
Document Compilation	5

Table 2: PERT Activity Time as per our Implementation

7. RESULT AND DISCUSSION

7.1 PERFORMANCE EVALUATION

This project has shown that the Genetic algorithm is a feasible method to generate any genre of Sudoku puzzles. The algorithm is also an appropriate method to find a solution faster and more efficient compared to any stochastic methods. The proposed algorithm is able to generate such puzzles with any level of difficulties in a short period of time. Our approaches have revealed that the implementation of the Genetic algorithm in our project is leading us to a possible way out quicker with respect to the computing time to generate any puzzle. It can guarantee to find at least one optimal solution. However, this algorithm is efficient because the steps of it are based on random facts of the algorithm. In other words, the algorithm is good for fresh and inexperienced strategies to generate the puzzles. This algorithm checks all possible solutions to the puzzle until a valid solution is found which is a time consuming procedure resulting an inefficient solver. Further study needs to be carried out in order to optimize the Genetic algorithm. A possible way could be implementing partial steps of GA strategies. Otherwise by following all of the alternatives of GA, it may not be possible for us to reach our destination because our project is based on matrices, not people or binary elements.

7.2 ANALYSIS OF RESULT

Results of implementation of Genetic Algorithm(GA) on generating Sudoku puzzles were satisfactory, at least in the sense of obtaining an optimal solution in a reasonable amount of time. GA implementation in our project did not always generate a given Sudoku problem in single time execution. Allowing longer run times (more generations) matters here. We expect a minimum time execution, then the GA is able to jump out and find the best solution. After applying GA, we have waited here for a '0' fitness score of any of our matrices and that is the optimal solution. An "optimal" solution strategy, using all the solutions according to their fitness from a number of generations, had a better chance of finding the correct solution, but did not always do in a particular time. In general, changing the parameters according to fundamental GA steps (elements of matrices, sub grids of matrices) had a profound impact upon performance, but it was not obvious how to generalize the loop execution from one problem instance to the next.

As expected, larger fitness score of matrices ultimately reduced to null and we were up to the mark. In our project, variation in the fitness value can be seen. We have displayed the value stored inside the gene and printed it in the Sudoku format along with iteration number. Also, simultaneous fitness values in the output are displayed and when the fitness value drops to 0, then the optimal solution of the Sudoku is printed. Variations in the execution time can also be seen and with reference to the Figures provided, the execution time were 11m15s and 07m33s for Figure(6) and (7) respectively. When we get the desired optimal solution that will be captured as:

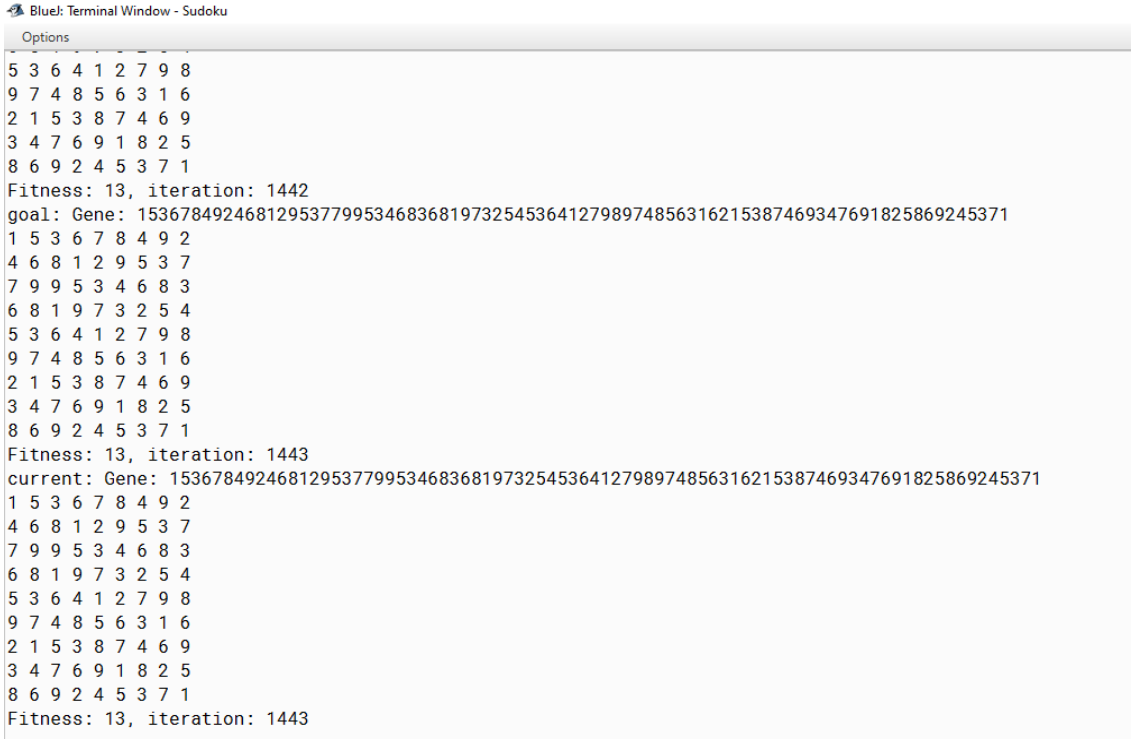

Figure 6(a): Intermediate Output Terminal 1

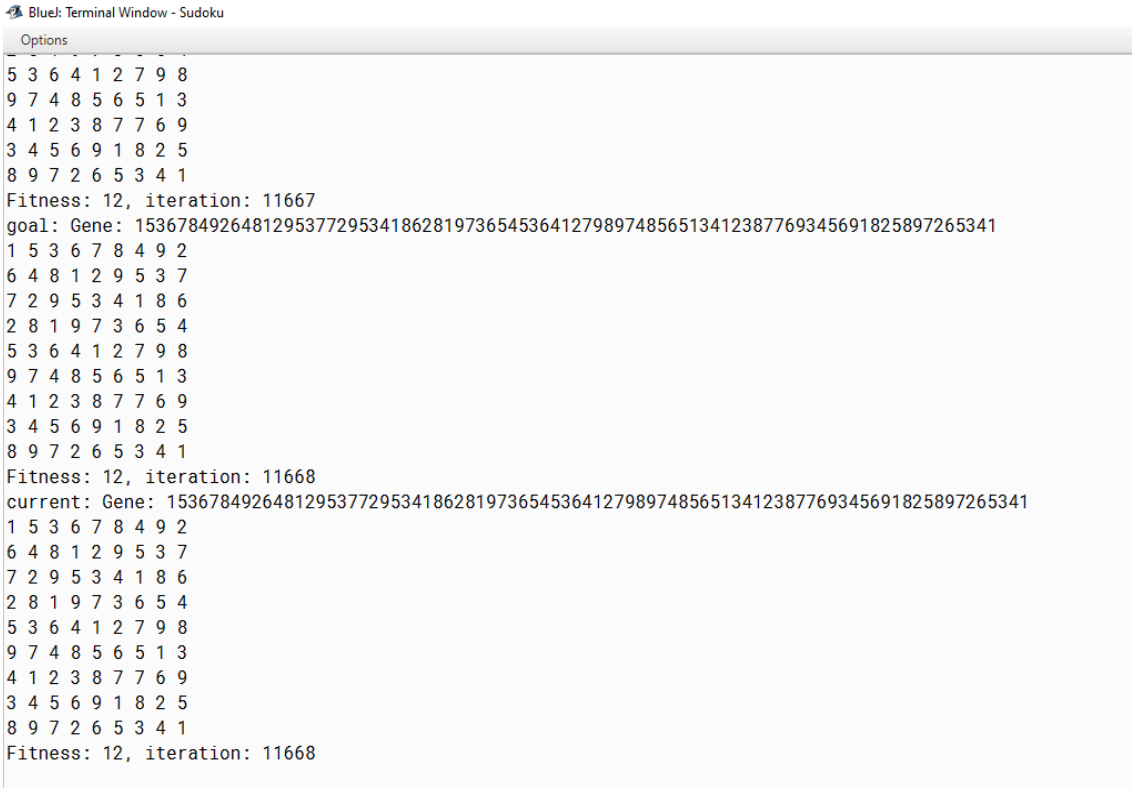

Figure 6(b): Intermediate Output Terminal 1

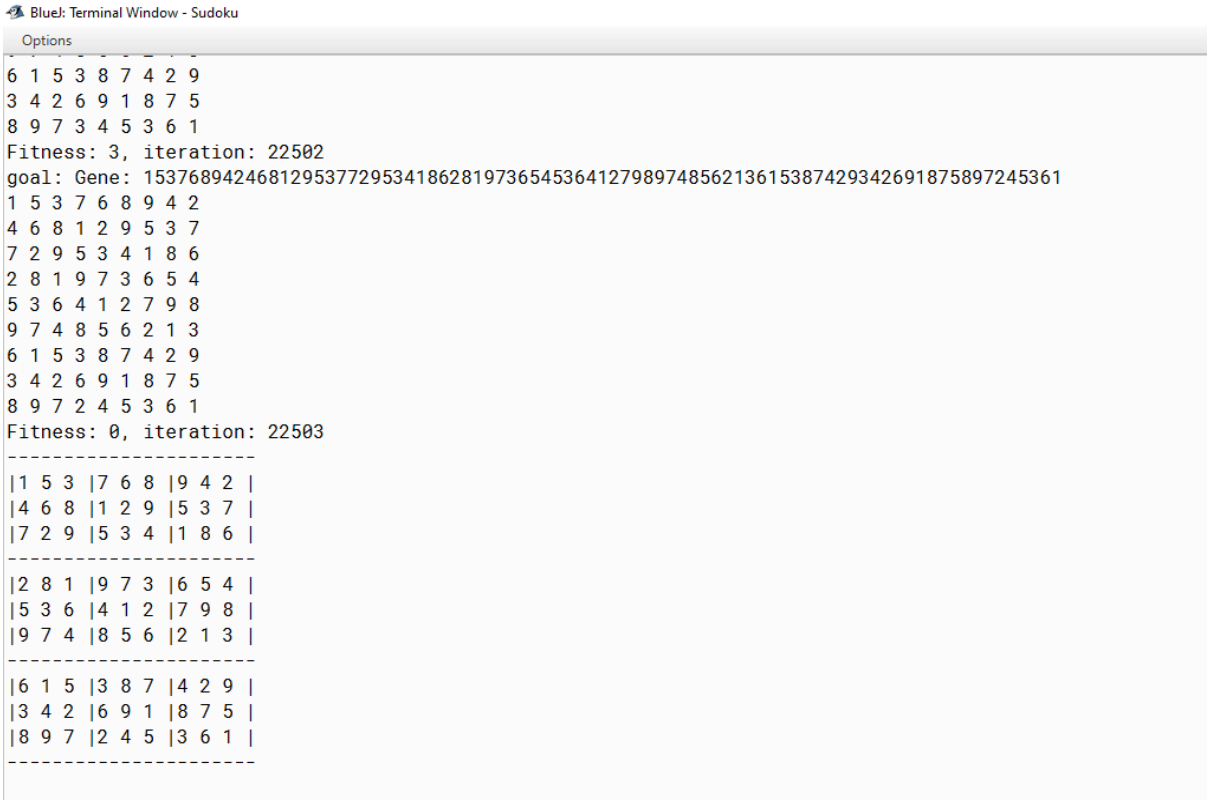

Figure 6(c): Output of Optimal solution 1

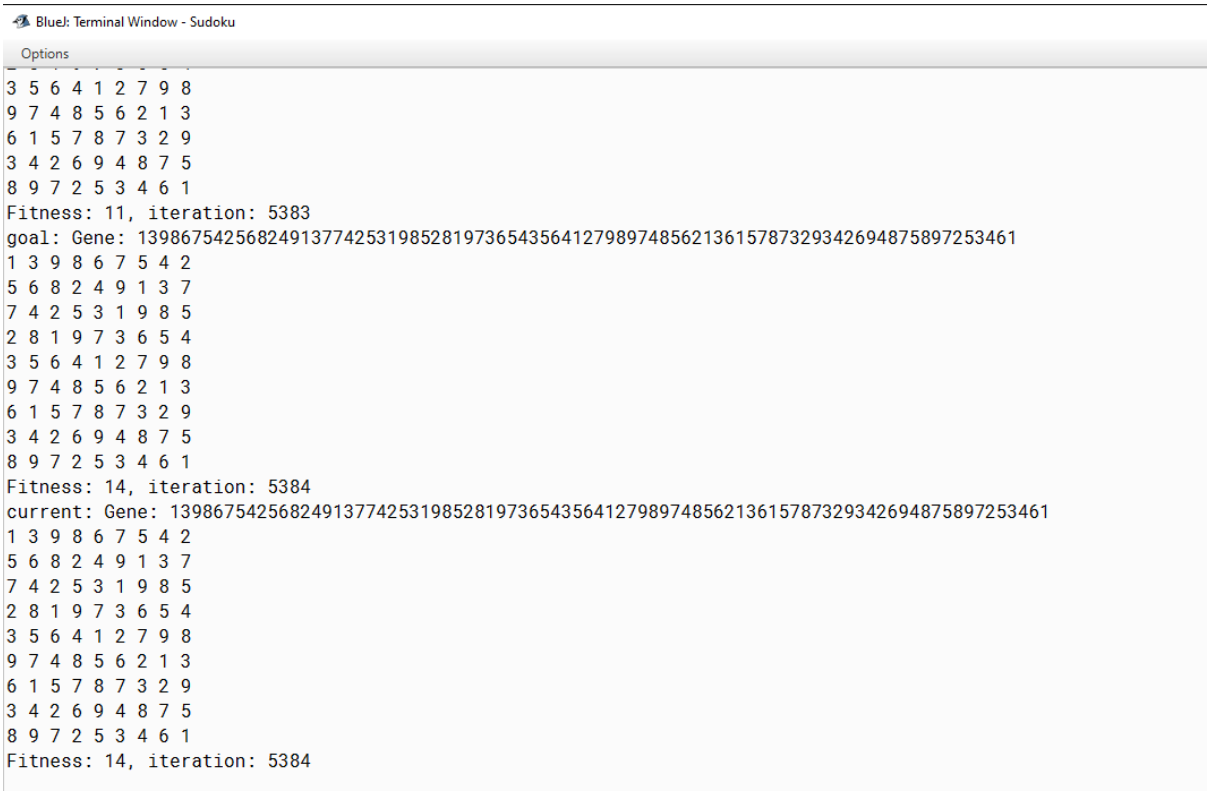

Figure 7(a): Intermediate Output Terminal 2

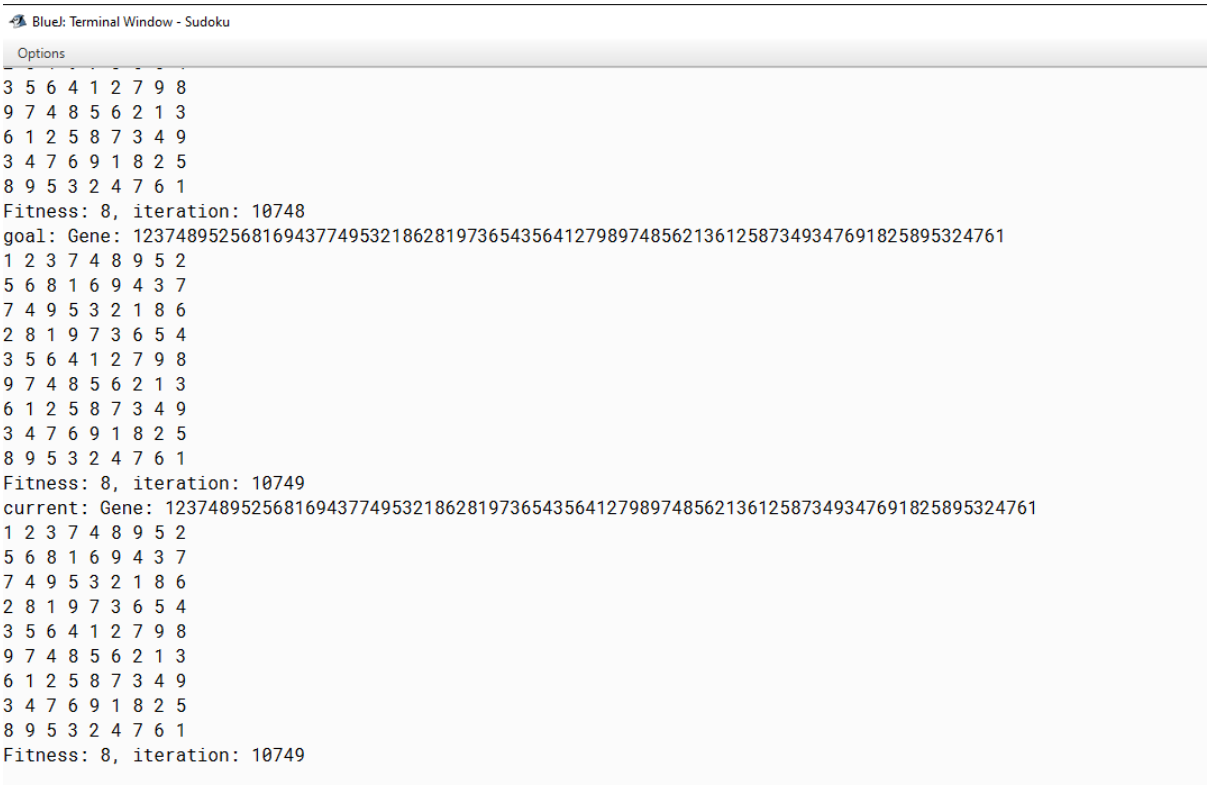

Figure 7(b): Intermediate Output Terminal 2

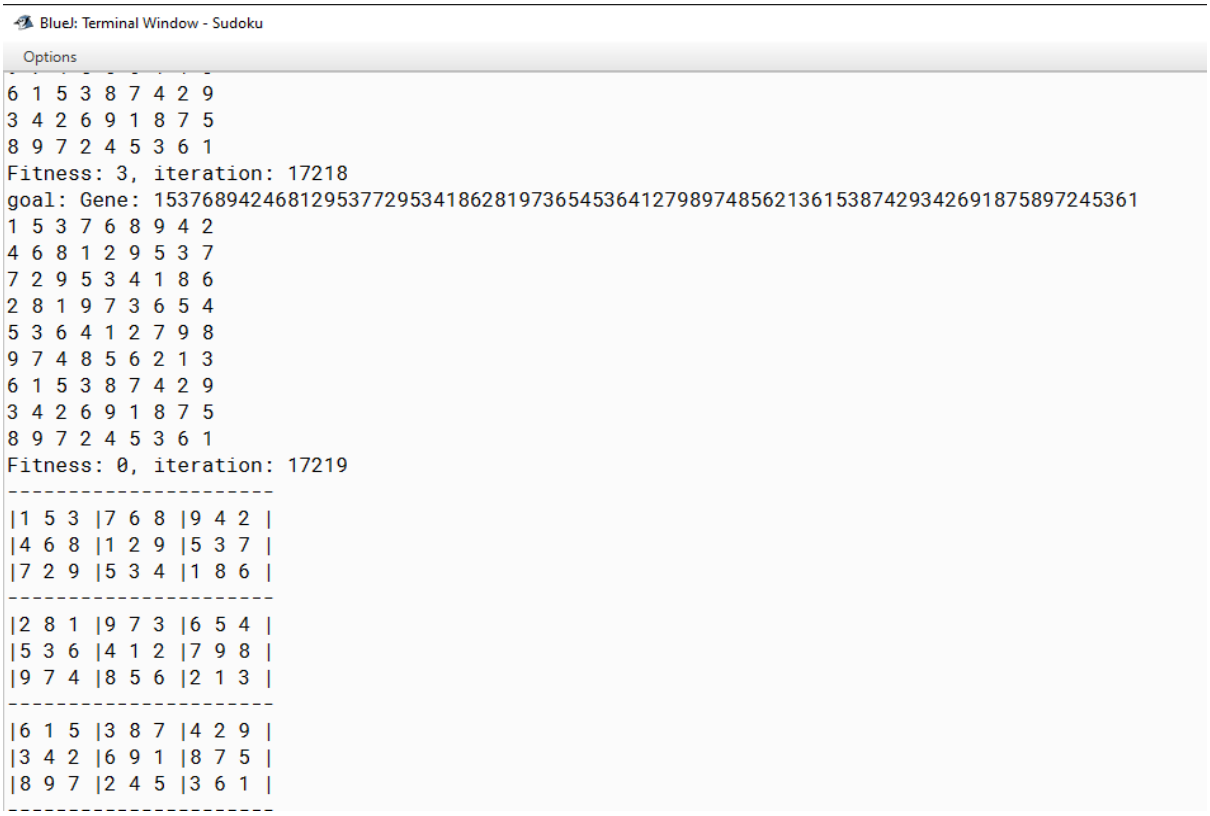

Figure 7(c): Output of Optimal solution 2

7.3 SUMMARY

A GA-based Sudoku generator is discussed in this project. The GA approach performs efficiently in relation to standard backtracking algorithms, at least for a 9×9 grid. By choosing large enough population sizes, and reasonable settings for other GA parameters, it is able to generate most Sudoku problems. In our project, we have found that while executing the code multiple times, a considerable amount of variations has been noticed in the following parameters i.e. the total number of iterations obtained after every end of the execution and execution time at which we obtain the desired optimal solution. With reference to the Figure (6) and (7), we observed the variations in the number of iterations, the fitness value and the execution time at which we obtained the optimal solution of our Sudoku problem.

Therefore, we can say that the execution time may vary after every execution of the program code. As we are familiar with the definition of Genetic Algorithm, changing the parameters according to fundamental Genetic Algorithm steps had a profound impact upon performance, but it was not obvious how to generalize the loop execution from one problem instance to the next.

However, the Sudoku generator often gets stuck in local minima and requires restarts in order to successfully find the problem solution. Reasons for poor GA performance with Sudoku are perplexing, since this appears to be an optimization problem that is well suited for evolutionary strategies. Nonetheless, Sudoku proved to be an entertaining (and challenging) way to introduce GAs in an Artificial Intelligence class.

8. FUTURE SCOPE

The successful applications of Soft Computing (SC) suggest that SC will have increasingly greater impact in the coming years. Soft Computing is already playing an important role both in science and engineering. In many ways, Soft Computing represents a significant paradigm shift (breakthrough) in the aim of computing, a shift that reflects the fact that the human mind, unlike state-of-the-art computers, possesses a remarkable ability to store and process information, which is pervasively imprecise, uncertain, and lacking categorically. Soft Computing can be extended to include computing not only from human thinking aspects (mind and brain) but also from bio-informatics aspects. In other words, cognitive and reactive distributed artificial intelligence will be developed and applied to large-scale and complex industrial systems. It is expected that evolutionary computation techniques will be applied to the development of more advanced intelligent systems.

Though our approach has very good results for easy and challenging Sudoku puzzles, it is not so ideal for difficult and challenging Sudoku puzzles. This awaits our deeper study in this algorithm, so as to make it more suitable to generate puzzles of varying difficulty levels.

Future work would also include, using genetic algorithm to come up with one possible solution to generate a Sudoku grid based on varying difficulty levels. The goal with such an exercise is to maintain the simplicity of our projected path and consuming times for finding optimum solution. Due to the randomness, in every step within the proposed algorithm, we have carefully tackled the repeated running loops of the program, as it can be disturbed anytime by several times of execution.

As we know that we are dealing with randomness, the number of iterations at which we get our desired solution is not fixed and as a result, the execution time varies for each and every execution of the program code. Therefore, it also includes to find one possible way to reduce the execution time to some extent even if the program is executed multiple times.

9. REFERENCES

[1] M.Koppen. “Applied Soft Computing: The Official Journal of the World Federation on Soft Computing.”pp. 1382- 1389, 2007.

[2] Aamulehti. Sudoku online via WWW: http://www.aamulehti.fi/sudoku/ (cited 10.10.2019)

[3]B.Felgenhauer and F.Jarvis, “Enumerating possible Sudoku puzzles,” http://www.afjarvis.staff.shef.ac.uk/maths/felgenhauer_jarvis_sudoku1.pdf, June 2005.

[4] L. Aaronson, “Sudoku science: a popular puzzle helps researchers dig into deep math,” IEEE Spectrum, pp. 16-17, Feb 2006.

[5] David Martin, Erica Cross and Matt Alexander. Using Backtracking to Come up with Sudoku Puzzles.www:http://www.csharpcorner.com/UploadFile/mgold/Sudoku0923005003323 AM/Sudoku.aspx?ArticleID-fba36449-ccf3-444f-a435-a812535c45e5 (cited 12.08.2019).

[6]T. K. Moon and J. H. Gunther, “Multiple constraint satisfaction by belief propagation: an example using Sudoku,” Proceedings of IEEE Mountain Workshop on Adaptive and Learning Systems, pp 122-126, 2006.

[7]Team #2306, “Generating Sudoku as an inverse problem” https://sites.math.washington.edu/~morrow/mcm/team2306

[8]T. Mantere and J. Koljonen, “Solving, rating and generating Sudoku puzzles with GA,” Proceedings of IEEE Congress on Evolutionary Computation, pp. 1382- 1389, 2007.

[9] J. –L. Lin, “An analysis of genetic algorithm behavior for combinatorial optimization problems,” Ph.D. dissertation, University of Oklahoma, Norman, OK, USA, 1993.

[10]A. Tuson and P. Ross, “Adapting operator settings in genetic algorithms,” Evolutionary Computation, vol. 6, pp. 161-184, 1998.

[11]Mathworks.com, “Genetic algorithm and direct search toolbox”, http://www.mathworks.com

[12] J. –W. Dang, Y. –P. Wang, and S. –X. Zhao, “Study on a novel genetic algorithm for the combinatorial optimization problem,” Proceedings of International Conference on Control, Automation and Systems, pp. 2538-2541, 2007.

[13] A. Moraglio, J. Togelius, and S. Lucas, “Product geometric crossover for the Sudoku puzzle,” Proceedings of IEEE Congress on Evolutionary Computation, pp. 470-476, 2006.

[14] Deep, K. and Thakur, M. (2007a). A new crossover operator for real coded genetic algorithms. Applied Mathematics and Computation, 188(1), pp: 895-911.

[15] Wei-Meng Lee, Programming Sudoku. New York, NY: Springer-Verlag New York, Inc., pp. 47-170, 2006.