On the Performance Analysis of Solving the Rubik's Cube Using Swarm Intelligence Algorithms

Dissertation Submitted in partial fulfilment of the requirements of the degree of

Master of Technology

in

Computer and Information Science

by

Jishnu Jeevan

43519006

Under the guidance of

Dr. Madhu S. Nair

DEPARTMENT OF COMPUTER SCIENCE

COCHIN UNIVERSITY OF SCIENCE AND TECHNOLOGY

April 2021

DEPARTMENT OF COMPUTER SCIENCE
COCHIN UNIVERSITY OF SCIENCE AND TECHNOLOGY
ERNAKULAM, KOCHI-682022

CERTIFICATE

This is to certify that the dissertation work entitled On the Performance Analysis of Solving the Rubik's Cube Using Swarm Intelligence Algorithms is a bonafide record of work carried out by Jishnu Jeevan (43519006) submitted to the Department of Computer Science in partial fulfilment of the requirements for the award of the degree of Master of Technology in Computer and Information Science at Cochin University of Science and Technology during the academic year 2021.

Supervisor,

Dr. Madhu S. Nair
Associate Professor
Dept. of Computer Science
CUSAT

Dr. Philip Samuel
Professor and Head
Dept. of Computer Science
CUSAT

DATE : 27 April 2021

DECLARATION

I declare that the work presented in this dissertation titled **On the Performance Analysis of Solving the Rubik's Cube Using Swarm Intelligence Algorithms** represents my ideas in my own words and where others' ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be cause for disciplinary action by the University and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Signature:

Name: Jishnu Jeevan
Reg No: 43519006

Date: 27 April 2021
Place: Ernakulam

ACKNOWLEDGEMENT

The successful completion of this project was only possible because of the help and information provided by others. It's my pleasure to convey gratitude to all of them. First and foremost, I would like to express my deep sense of gratitude to Dr. Madhu S. Nair for his support, suggestions, and motivation throughout the completion of the project and for all the facilities provided to complete this project work. I also express my sincere gratitude to all faculty members and non-teaching staff in the Computer Science department for the support and cooperation throughout the course of the action.

I am extending my gratitude to all my batch mates, friends, and family members who inspired me to do this project systematically. I would also like to thank my classmate and friend Ayshabi M. K. and also my professor from B.Tech Dr. Jeswin Roy D'couth for all the support and guidance they have given me, without which I could not have completed this project work.

I thank God Almighty for giving me the strength, knowledge, ability, and opportunity to undertake this work, without which it would not have been possible to complete this project work.

Jishnu Jeevan

ABSTRACT

Swarm intelligence algorithms have been one of the most popular types of algorithms used for solving optimization problems. They are nature-inspired algorithms that mimic the phenomenon occurring in nature to solve optimization problems. These natural phenomena are intelligent animal behavior used by animals for survival from hunting prey, migration, escaping predators, and reproduction. Some examples are ant colonies, flocking of birds, hunting patterns of hawks, herding behavior of animals, bacterial growth, fish schooling, and intelligent microbial organisms. The Rubik's cube is a 3D combinatorial puzzle having six faces covered by nine stickers, each of one of six solid colors: white, red, blue, orange, green, and yellow. The objective is to turn the scrambled cube, where each side will have more than one color, into a solved cube. The solved cube will have only one color on each side. In this study, four swarm intelligence algorithms, particle swarm optimization, greedy tree search algorithm, ant colony optimization, and discrete krill herd optimization, are used to investigate which algorithm out of the four can solve the Rubik's cube with the optimum number of moves in the optimum amount of time.

Keywords: Ant colony optimization, Combinatorial Optimization, Greedy Tree Search, Krill Herd Optimization, Particle Swarm Optimization, Rubik's Cube, Swarm Intelligence.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Swarm intelligence algorithms have become very popular in solving various problems ranging from routing in telecommunication networks [1], simulation of crowds [2], using nanobots to kill cancer tumors [3], optimization of functions, missing data prediction [4], feature selection [5], parameter setting of deep neural networks [6] etc.

An area where swarm intelligence algorithms have not seen much application is solving combinatorial optimization problems like solving the Rubik's cube. This study focuses on applying different swarm intelligence algorithms for solving the Rubik's cube in the optimum amount of time with the optimum number of moves. The algorithms used for the survey are particle swarm optimization (PSO), greedy tree search algorithm, ant colony optimization (ACO), and discrete krill herd optimization (DKHO). These algorithms have been optimized to solve the Rubik's cube in the optimum amount of time using an optimum number of moves.

1.1 Motivation

Swarm intelligence (SI) is the collective behavior of decentralized, self-organized systems that can be either natural or artificial.

Swarm intelligent systems consist of a population of agents that interact with each other and with their environment. The agents follow simple rules that tell them how to interact with the environment and with each other. These interactions can sometimes seem random when we observe the behavior of each agent individually. This random local behavior leads to the emergence of intelligent global behavior, unknown to the individual agents. Some examples are ant colonies, flocking of birds, hunting patterns of hawks,

herding behavior of animals, bacterial growth, fish schooling, and intelligent microbial organisms. Swarm intelligence algorithms are popular for solving many problems because they are cheap, robust, and simple.

Swarm intelligence algorithms are used in the optimization of problems in the continuous search domain and also in the discrete search domain. Finding the solution to the Rubik's cube is a problem in the discrete search domain. Solving the Rubik's cube is a challenge for humans and computers due to the large number of states that the cube can take, explained in Sect. 2.1. The main idea behind any swarm intelligence algorithm is to have a population of agents that move around the search space and try to find the solution to the given problem. The motivation behind this research is to understand if it is possible to have a population of agents that move around the large search space of the Rubik's cube and find a solution to it. Another motivation behind this research is that swarm intelligence algorithms have not been used in solving the Rubik's cube. Although they have been used for solving discrete search domain problems, they have not been applied for solving a combinatorial puzzle with a large search space like the Rubik's cube. Genetic algorithms have been used for solving the Rubik's cube [7, 8, 9] but these algorithms sometimes fail and won't completely solve the cube. Also, these algorithms can only solve the easier cases of the Rubik's cube where the cube is only a few moves away from being solved.

1.2 Problem Statement

The problem statement for this study is to understand if it is possible to design a swarm intelligence algorithm or optimize an existing swarm intelligence algorithm so that it can traverse the large search space of the Rubik's cube and return a solution for the Rubik's cube. Also, to analyze the algorithms to find out which type of swarm intelligence algorithm can solve the Rubik's cube with the optimum number of moves in the optimum amount of time.

The objectives of this study are as follows:

1. To optimize the particle swarm optimization algorithm to solve the Rubik's cube and do an analysis to evaluate its performance.

2. To optimization the IDA* tree search algorithm to form a greedy tree search algorithm to solve the Rubik's cube and do an analysis to evaluate its performance.

3. To optimize the ant colony optimization algorithm to solve the Rubik's cube and do an analysis to evaluate its performance.

4. To optimize the discrete krill herd optimization algorithm to solve the Rubik's cube and do an analysis to evaluate its performance.

5. Do a comparative study of the four algorithms to find out which algorithm can solve the Rubik's cube in the optimum amount of time with the optimum number of moves.

1.3 Major contribution of the Dissertation

This is the first-ever study conducted where swarm intelligence algorithms are used for solving the Rubik's cube. Particle swarm optimization algorithm, IDA* tree search algorithm, ant colony optimization algorithm, and discrete krill herd optimization algorithm has been modified to solve the Rubik's cube. These algorithms could solve the cube successfully for the majority of the cube configurations. Individual analysis and a comparative analysis of these algorithms have been performed to find out which algorithm can solve the cube with an optimum number of moves in the optimum amount of time. With the successful modification of the swarm intelligence algorithms to solve the Rubik's cube, my work has proved that swarm intelligence algorithms can be used for solving the Rubik's cube. This work can be extended to solve any problem that is similar to solving the Rubik's cube. These problems are discrete tree search problems where the objective is to reach from the current state to the goal state. Some examples are protein folding, decision tree-based learning, path finding for robots, and AI agents in video games.

A major challenge when it came to optimizing the existing swarm intelligence algorithms to solve a Rubik's cube was to implement a method such that all the agents can search the huge search space of the Rubik's cube without getting lost and moving further away from the solution. The major contribution of this dissertation is the implementation of a method that allows the agents to communicate with each other and regroup to a common point when a certain number of iterations is reached or when a certain condition is met, and begin the search again from the new location. This regrouping ensures that the agents don't get lost in the huge search space and move further away from the solution. The regrouping strategy of each algorithm will be explained in the following chapters.

1.4 Thesis Outline

The rest of the thesis is organized as follows. Chapter 2 provides information and history of the Rubik's cube and explains why it is such a challenging problem for humans and computers. The chapter also provides an overview of the swarm intelligence algorithms used in this study to solve the Rubik's cube from particle swarm optimization (PSO), greedy algorithm and IDA* algorithm, ant colony optimization algorithm (ACO), discrete krill herd optimization algorithm (DKHO). To understand how any swarm intelligence algorithm works the chapter also explains what a fitness function is and how it's used by swarm intelligence algorithm for solving optimization problems. The chapter also explains Kociemba's algorithm and how it acts as a fitness function when swarm intelligence algorithms are used to solve the Rubik's cube.

Chapter 3 provides a detailed explanation of how the PSO algorithm is optimized to solve the cube, how the IDA* algorithm is optimized to solve the Rubik's cube greedily, how the ACO algorithm is optimized to solve the Rubik's cube, and how the DKHO algorithm is optimized to solve the Rubik's cube [1]. The chapter also explains how the time complexity of each algorithm is derived.

Chapter 4 contains the experimental analysis and observations for each algorithm. An algorithm is accessed individually to evaluate its performance and there is also a comparative analysis where each algorithm competes with each other to find out which swarm intelligence algorithm can solve the Rubik's cube optimally. The chapter also explains why some algorithms perform better compared to others.

Chapter 5 provides the conclusion of this study and how further improvements can be made.

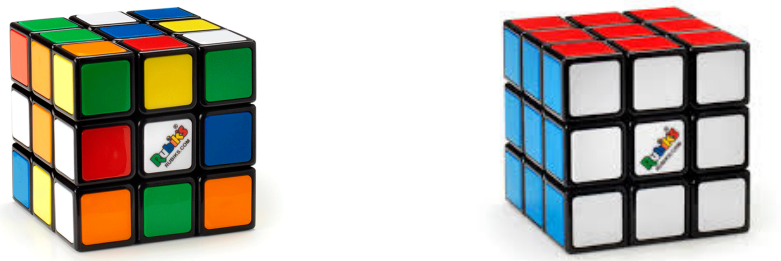
[1] All algorithms where implemented using Python. The code is available at: `https://github.com/JishnuJeevan/Swarm-Intelligence-Algorithms-for-Solving-Rubik-s-Cube`

Chapter 2

Literature Survey

2.1 Rubik's cube

The Rubik's cube is a 3D combination puzzle invented by Hungarian sculptor and professor of architecture Ernő Rubik in 1974. The Rubik's cube has six faces and each of the six faces is covered by nine stickers each of one of six solid colors: white, red, blue, orange, green, and yellow. The objective is to turn the scrambled cube, where each side will have more than one color, into a solved cube by turning the sides of the cube. The solved cube will have only one color on each side of the cube.



(a) Scrambled Cube (b) Solved Cube

Figure 2.1: The Rubik's cube

The Rubik's cube is considered to be one of the most popular and best-selling toys of all time. It is estimated that over 350 million cubes have been sold worldwide. This worldwide popularity of the cube has led to various speed solving competitions being held, where the objective is to solve the Rubik's cube and other variations of the puzzle as fast as possible. Official competitions are conducted all over the world by the World Cube

Association (WCA) [10].

The moves applied to the cube are denoted using Singmaster notation [11]. The notation is relative in nature which allows algorithms to be written in such a way that they can be applied regardless of which side is designated the top or how the colors are organized on a particular cube. F (Front): the side currently facing the solver, B (Back): the side opposite the front, U (Up): the side above or on top of the front side, D (Down): the side opposite the top, underneath the Cube, L (Left): the side directly to the left of the front, R (Right): the side to the right of the front side.

A letter by itself, for example, F, B, U, D, L, and R, indicates that the corresponding side of the cube should be turned 90 deg clockwise. A letter followed by a prime symbol (’), for example, F’, B’, U’, D’, L’, and R’, indicates that the corresponding side of the cube should be turned 90 deg anticlockwise. A letter followed by the number 2, for example, F2, B2, U2, D2, L2, and R2, indicates that the corresponding side of the cube should be turned 180 deg or two turns.

There are two ways to measure the length of the solution. One is called a quarter-turn metric (QTM) where 180 deg turns are counted as two moves, while in the half-turn metric (HTM) they are considered as one move. The half-turn metric is also called as face turn metric (FTM) or outer block turn metric (OBTM). For this study, the half-turn metric is used as it is the most popular notation and it is used as a standard in the cubing community, especially in competitions conducted by the WCA [12].

A Rubik’s cube has 43,252,003,274,489,856,000 $\approx$ 43 quintillion number of states. Mathematicians and computer scientists have been trying to find the minimum number of moves required to solve the cube. The minimum number of moves needed to solve any configuration of the cube is called God’s number. Many attempts have been made to find out God’s number, where researchers kept lowering the solution length for almost 30 years [13]. It was finally proved that all possible configurations of the Rubik’s cube, all 43 quintillion cases, can be solved in 20 moves or less in half-turn metric [14]. This was done by using the symmetry property of the cube where certain cases were eliminated because they were mirrors or symmetry of other cases. This reduced the possible states of the cube to two billion [15]. All two billion configurations of the cube were given to supercomputers at Google, which solved every one of those configurations in 20 moves or less [14].

2.2 Fitness Function

For any swarm intelligence algorithm to work we need a fitness function or an objective function. The fitness function is used to evaluate how good or how fit an agent is. Swarm intelligence algorithms are used for solving optimization problems where the goal is to minimize or maximize the given function. For a minimization problem (or a maximization problem), an agent is said to be fit if it is close to the minimum value (or maximum value) of the function compared to the other agents. In most cases, the function that needs to be optimized will act as the fitness function. Fitness functions are essential for swarm intelligence algorithms as they are used to guide the agents in finding the minimum point (or maximum point) of the function.

2.3 Kociemba's Algorithm

When swarm intelligence algorithms are used for optimizing continuous functions the continuous function will act as the fitness function. Finding a solution to the Rubik's cube is a discrete search domain problem, almost similar to a tree search problem. For this problem, the Manhattan distance heuristic is used as the fitness function [16]. Manhattan distance heuristic, speaking in the context of a combination puzzle like the Rubik's cube, is the distance, in terms of the number of moves, between the solved state of the cube and the current state.

The Manhattan distance can be computed using an algorithm that can solve the Rubik's cube. For this study, Kociemba's algorithm is used as the fitness function, for calculating the Manhattan distance [17, 18, 19]. The Kociemba's algorithm was used to prove that every state of the Rubik's cube can be solved in 20 moves or less [14]. Kociemba's algorithm is the most optimal cube solving algorithm which can solve any configuration of the cube in 20 moves or less in half-turn metric.

The states of the Rubik's cube can be grouped according to different properties. In Koceimba there are three groups G_0, G_1 and G_2. The group G_0 contains all the possible states of the cube that can be reached by applying only the moves $\langle$U, D, R, L, F, B$\rangle$ from the solved state. The group G_1 contains all the states that can be reached by applying only the moves $\langle$U, D, R2, L2, F2, B2$\rangle$ from the solved state, and the group G_2 contains only a single state which is the solved state. Kociemba's algorithm works in two phases:

1. The first phase is to take the cube from the scrambled state which is in group G_0 to any of the states in a group G_1.

2. The second phase is to solve the cube by taking the cube which is now in group G_1 to G_2 which is the solved state.

Algorithm 1 explains the working of Kociemba's algorithm [1] [2].

2.4 Particle Swarm Optimization (PSO)

In particle swarm optimization (PSO) [20] there is a population of candidate solutions, here called particles. These particles move around the search space of the given problem iteratively, trying to find the optimum solution to the given problem. The movement of these particles is determined by simple mathematical formulas describing their position, the current position of the particle in the search space, and velocity, how far the particle should move in the search space from the current position. The equations used to describe the position of the particle and its velocity is:

$$x_i(t+1) = x_i(t) + V_i(t+1) \tag{2.1}$$

$$V_i(t+1) = WV_i(t) + r_1C_1(P_i(t) - x_i(t)) + r_2C_2(G(t) - x_i(t)) \tag{2.2}$$

where $x_i(t)$ is the position of particle i in the search space at time t, $x_i(t+1)$ is the next position of the particle i in the search space at time $(t+1)$, $V_i(t)$ is the velocity of the particle i at time t and $V_i(t+1)$ is the velocity of particle i at time $(t+1)$. r_1 and r_2 are random numbers between 0 and 1, $WV_i(t)$ is the inertia component and W is the inertia coefficient which is a constant, $r_1C_1(P_i(t) - x_i(t))$ is the cognitive component and $r_2C_2(G(t) - x_i(t))$ is the social component where C_1 is the personal acceleration coefficient which is a constant and C_2 is the global acceleration coefficient which is also a constant. $P_i(t)$ is the personal best solution for the particle i till time t and $G(t)$ is the global best solution till time t.

2.5 Greedy algorithm and Iterative Deepening A* algorithm (IDA*)

A greedy algorithm is an algorithm that will try to solve an optimization problem by selecting the local optimum solution at each stage of the problem. These algorithms will sometimes find the global optimum solutions to the problem, but sometimes they can

[1] The pseudo code for Kociemba's algorithm is taken from: `https://www.jaapsch.net/puzzles/compcube.htm#kocal`

[2] Python implementation of Kociemba's algorithm can is available at: `https://github.com/hkociemba/RubiksCube-TwophaseSolver`

Algorithm 1 Kociemba's Algorithm

maxLength=9999

```
1: function KOCIEMBA(position p)
2:     for depth d from 0 to maxLength do
3:         Phase1search( p; d )
4:     end for
5: end function
1: function PHASE1SEARCH( position p; depth d )
2:     if d=0 then
3:         if subgoal reached and last move was a quarter turn of R, L, F, or B
   then
4:             Phase2start( p )
5:         end if
6:     else if d>0 then
7:         if prune1[p]≤d then
8:             for each available move m do
9:                 Phase1search( result of m applied to p; d-1 )
10:            end for
11:        end if
12:    end if
13: end function
1: function PHASE2START(position p)
2:     for depth d from 0 to maxLength - currentDepth do
3:         Phase2search( p; d )
4:     end for
5: end function
1: function PHASE2SEARCH( position p; depth d )
2:     if d=0 then
3:         if solved then
4:             Found a solution!
5:             maxLength = currentDepth-1
6:         end if
7:     else if d>0 then
8:         if prune2[p]≤d then
9:             for each available move m do
10:                Phase2search( result of m applied to p; d-1 )
11:            end for
12:        end if
13:    end if
14: end function
```

come up with solutions that are near to the global optimum or maybe even worse than

the global optimum. This is because selecting a local optimum solution at each stage may not guarantee that they lead to the global best solution. An example of a greedy algorithm is Dijkstra's algorithm, which is used for finding the shortest path between nodes in a graph [21].

The greedy tree search algorithm used for this study is a modification of the algorithm explained in the paper *Finding Optimal Solutions to the Rubik's Cube Using Pattern Databases* by [22]. In this paper, the subdivision method is used to solve the cube, where the first eight corners of the cube are solved followed by the six edges and then the other six edges. The optimal way to solve the eight corners of the cube in any configuration has been computed and the solution is stored in a database. The same has been done for the six edges of the cube and also for the other six edges of the cube. The pattern database stores intermediate solutions of the cube. When intermediate states are encountered while solving the cube the database is searched to see if there is a solution for that state. Branches that don't lead to the solved state are pruned off using the A* approach. This method is called the iterative deepening A* approach (IDA*), iteratively search each depth to see if the intermediate stage has been reached, where the branches that do not lead to a solved state are pruned off using the A* approach.

The A* algorithm is a heuristic search algorithm used for finding optimal paths in a graph. In the A* algorithm there is a heuristic function that is used to evaluate how good a path is. The algorithm works by assigning a threshold value to the heuristic cost function. Any path whose heuristic value is greater than the threshold value is not considered for the search. The threshold value prunes out the branches that are farther away from the goal state. During each iteration of the algorithm, the threshold value decreases when better paths are obtained. Algorithm 2 explains the working of the IDA* algorithm[3].

2.6 Ant Colony Optimization Algorithm (ACO)

Ant colony optimization (ACO) algorithm is a metaheuristic optimization algorithm which is a probabilistic technique used for solving problems that can be reduced to finding good paths through graphs [23, 24]. This algorithm is inspired by the behavior of biological ants. To find the food source ants will leave behind a pheromone trail from the location that they are starting from, to the location of the food source. The pheromone trail is

[3]The pseudo code for IDA* algorithm is taken from `https://www.jaapsch.net/puzzles/compcube.htm#ida`

Algorithm 2 IDA* Algorithm

```
1: function IDA( position p )
2:     for depth d from 0 to maxLength do
3:         Treesearch( p; d )
4:     end for
5: end function
1: function TREESEARCH( position p; depth d )
2:     if d=0 then
3:         if p is solved then
4:             Hooray!
5:         end if
6:     else if d>0 then
7:         if prune1[p]≤d and prune2[p]≤d then
8:             for each available move m do
9:                 Treesearch( result of m applied to p; d-1 )
10:            end for
11:        end if
12:    end if
13: end function
```

left behind so that the ants can communicate with each other on how far or near the food source is. If the food source is near then the ants will leave behind more amount of pheromones, else the pheromone level will be less.

The pheromone deposited by an ant k, on an edge (i, j) is denoted by:

$$\Delta\tau_{i,j}^{k} = \begin{cases} \frac{1}{L_k}, where\, k^{th}\, ant\, traversed\, edge\, (i,j) \\ 0, otherwise \end{cases} \tag{2.3}$$

This means that if an ant k travels from node i to node j of a graph then it will deposit some amount of pheromone represented by $\Delta\tau_{i,j}^{k}$, which is equal to $\frac{1}{L_k}$ where L_k is the length of the path, distance from node i to node j, found by the ant k. If the path length is more, then the pheromone level deposited will be less.

The total amount of pheromones deposited is the sum total of the pheromones deposited by all the ants that traversed the edge (i, j), represented by:

$$\tau_{i,j}^{k} = \Sigma_{k=1}^{m} \Delta\tau_{i,j}^{k} \tag{2.4}$$

where m represents the total number of ants that traversed the edge (i, j). To decrease

the amount of pheromone deposited after each iteration an evaporation constant p is introduced and Eq. 2.4 becomes:

$$\tau_{i,j}^{k} = (1-p)\tau_{i,j} + \Sigma_{k=1}^{m}\Delta\tau_{i,j}^{k} \tag{2.5}$$

The probability of selecting an edge is given by:

$$P_{i,j} = \frac{(\tau_{i,j})^{\alpha}(\eta_{i,j})^{\beta}}{\Sigma_{all\,edges}(\tau_{i,j})^{\alpha}(\eta_{i,j})^{\beta}} \tag{2.6}$$

where

$$\eta_{i,j} = \frac{1}{L_k} \tag{2.7}$$

For example if an ant wants to select an edge from one of the three edges a, b, and c then using the above equations (Eqs. 2.3 - 2.6) the probability of selecting the three edges are calculated and found to be as follows, $P = [0.76, 0.19, 0.05]$. Now the cumulative sum is calculated as $Cumulative\_Sum = [0.76 + 0.19 + 0.05, 0.19 + 0.05, 0.05] = [1, 0.24, 0.05]$. A random number r is generated and is between the range $[0, 1]$. If $0.24 \leq r \leq 1.00$ node a is selected, if $0.05 \leq r \leq 0.24$ node b is selected and if $0.00 \leq r \leq 0.05$ node c is selected. This method of selecting the edge is called as roulette wheel selection [25].

2.7 Discrete Krill Herd Optimization (DKHO)

Krill herd optimization (KHO) is a swarm intelligence algorithm that simulates the herding behavior of krill individuals to solve optimization problems [26].

To solve the Rubik's cube a variation of KHO called discrete krill herd optimization algorithm (DKHO) is used that solves problems in the discrete search domain [27]. The original DKHO proposed is used for optimizing graph-based network routes which works as follows:

1. Initialize road graph $G = (V, E)$ and road parameter matrix $p = \{p_1, p_2, ..., p_k\}$ for k parameters. Here $k = 2$.

2. Initialize krill matrix consisting of ID, chromosome for path traversed, number of nodes, sum of parameter vector, illumination factor or fitness.

3. Initialize normalized illumination factor (L^k) of each node depending on number

of krill present there and $0 \leq L^k \leq 1$. The illumination factor of a krill can be calculate in two ways:

$$L_1^k = MAX(l_1, l_2, ..., l_n) \tag{2.8}$$

$$L_2^k = \frac{\Sigma(l_1, l_2, ..., l_n)}{n} \tag{2.9}$$

where k is the number of nodes or paths, n is the total number of krills on the k^{th} node, $(l_1, l_2, ..., l_n)$ is the fitness of the krill on the k^{th} node. L_2^k is used to find the average illumination factor and L_1^k is used for finding the maximum fitness for maximization problem (or minimum fitness for minimization problem).

4. Path selection stage.

   (a) If L^k of adjacent node $\geq Threshold$

      i. Calculate probability $P_{i,j}$ for n number of paths according to illumination factor L_1^k or L_2^k.

      ii. Perform photostatic swarming movement. This movement is based on the phenomenon that when a krill eats more food it produces light from an organ called photophores organ through an enzyme-catalyzed chemical reaction. This causes the other krills to move to the location where the light intensity is more.

   (b) If L^k of adjacent node $< Threshold$

      i. Calculate probability $P_{i,j}$ for n number of paths according to the estimation factor $f_j()$

$$f_j(p_1, p_2, ..., p_n) = \begin{cases} MIN\forall[\psi(p_1), \psi(p_2), ..., \psi(p_n)] \\ MIN[w_1\psi(p_1) + w_2\psi(p_2) + ... + w_n\psi(p_n)] \end{cases} \tag{2.10}$$

$$P_{i,j} = \frac{f_j()}{\Sigma f_k()} \tag{2.11}$$

      ii. Perform concentration enhancement movement. This movement of the krill is based on the phenomenon that krills feed on an organism called phytoplankton. The krills have sense organs that detect the presence of phytoplankton. The krills tend to move to the region where there is a high concentration of phytoplankton so that they can consume a large amount of phytoplankton. The phytoplankton produces a chemical secretion and

the krills detect this secretion to estimate their density. To decide whether or not the krill should investigate a concentration is based on the dimension vector of parameters $\{p_1, p_2, ..., p_n\}$ and weight $\{w_1, w_2, ..., w_n\}$ represented as $f_j()$.

(c) If $P_{i,j}$ is the same for more than one path then perform random movement where the krills select a random path.

where $0 \leq Threshold \leq 1$ can be a margin for decision making. The $Threshold$ value can be constant or a random value that is adaptive.

5. If the fitness of a krill is poor, then perform predator's prey operation where the krills with poor fitness value are killed off or eliminated from the swarm.

6. If the population of the krills is less than the original population perform reproduction.

   (a) Choose the best swarm at the node.

   (b) Randomly choose a krill with a common gene point.

   (c) Perform reproduction between the two krills. Reproduction is done by taking the common gene point of the krill and swapping the tail of the chromosome of one krill with the tail of the chromosome of the other krill. The chromosome here is the paths traversed by the krills.

   (d) If the swarm size is less than a $Threshold$ value then perform non sac spawning where the off springs are produced at the current node.

   (e) If the swarm size is greater than a $Threshold$ value then perform sac spawning where the off springs are produced after a certain number of nodes have been traversed.

   where the $Threshold$ value can be a constant, a random value that is adaptive, or it can be a function of the swarm size and the paths available.

7. If the fitness value of the krill is low or if looping occurred, where the krill traverses the same path over and over again casing a cyclic loop

   (a) Perform predator's rush operation where the krills move backward instead of forwards. This is inspired by the fact that krills camouflage themselves to hide from predators and performs an escape reaction called lobstering which is a high speed backward movement through the water.

   (b) If the fitness value of the krill is low, preform predator's prey operation, killing or elimination of the unfit krill from the swarm.

   (c) Else preform lobstering, backwards movement of the krill.

8. If the illumination factor of the krill is very low perform moulting which is similar to a mutation in the genetic algorithm. The krill changes a random gene point of its chromosome (the path traversed) with another random gene point.

9. If the krill reached the destination calculate the fitness value of the krill.

10. Update the best path matrix with the global best.

11. Provide the best path matrix to the vehicle.

Chapter 3

Proposed System

3.1 PSO for Solving the Rubik's Cube

Finding the solution to the Rubik's cube is a discrete search domain problem, so the PSO algorithm has to be optimized in order for it to solve the cube.

Let $\mathbb{M}_{HTM}$ be the set of all moves that can be applied to the cube in half-turn metric, $\mathbb{M}_{HTM} = \{$R, R', R2, L, L', L2, F, F', F2, B, B', B2, U, U', U2, D, D', D2$\}$.

Let $\mathbb{S} = \{S_1, S_2 ... S_n\}, n \approx 4 \times 10^{18}$ be the set of possible states that the cube can take. Let $S_i(t)$ be the state reached by the particle i at time t. Let $S_i(t+1)$ be the state reached by the particle i at time $(t+1)$. Let $\mathbb{V}$ be the set of moves that are applied to the state $S_i(t)$ to reach the state $S_i(t+1)$ where $\mathbb{V} \subset \mathbb{M}_{HTM}$ and $\mathbb{V}$ can have duplicate elements. Let S_{P_i} be the state which gives the personal best fitness score for the particle i. Let S_G be the state which gives the global best fitness score.

Calculate personal acceleration P_A and the global acceleration G_A.

$$P_A = Kociemba(S_{P_i}) – Kociemba(S_i(t)) \tag{3.1}$$

$$G_A = Kociemba(S_G) – Kociemba(S_i(t)) \tag{3.2}$$

where $Kociemba(S_i(t))$ is the fitness value of the state S_i at time t.

The new state $S_i(t+1)$ is given by:

$$S_i(t+1) = \begin{cases} S_{P_i} + \mathbb{V}, & \text{if}(P_A < 0 \, and \, G_A \geq 0) \text{ or} \\ & \text{if}(P_A < 0 \, and \, G_A < 0 \, and \, P_A < G_A), \text{where } |\mathbb{V}| = |P_A| \\ S_G + \mathbb{V}, & \text{if}(G_A < 0 \, and \, P_A \geq 0) \text{ or} \\ & \text{if}(P_A < 0 \, and \, G_A < 0 \, and \, G_A \leq P_A), \text{where } |\mathbb{V}| = |G_A| \\ S_i(t) + \mathbb{V}, & \text{if}(P_A > 0 \, and \, G_A > 0), \text{where } |\mathbb{V}| = Min(P_A, G_A) \\ S_i(t) + \mathbb{V}, & \text{if}(P_A > 0 \, and \, G_A = 0), \text{where } |\mathbb{V}| = P_A \\ S_i(t) + \mathbb{V}, & \text{if}(G_A > 0 \, and \, P_A = 0), \text{where } |\mathbb{V}| = G_A \\ S_i(t) + \mathbb{V}, & \text{if}(P_A = 0 \, and \, G_A = 0), \text{where } |\mathbb{V}| = 1 \end{cases} \tag{3.3}$$

where $S_{P_i} + \mathbb{V}$ means, go to the state that gives the personal best fitness score for the particle i, and apply the set of moves in $\mathbb{V}$, where $|\mathbb{V}|$ denotes that the number of moves to be applied to the cube is equal to the absolute value of P_A. $S_G + \mathbb{V}$ means to go to the state that gives the global best fitness score and apply the set of moves in $\mathbb{V}$, where $|\mathbb{V}|$ denotes that the number moves to be applied to the cube is equal to the absolute value of G_A. $S_i(t) + \mathbb{V}$ means from the current state $S_i(t)$ apply a set of moves in $\mathbb{V}$ where the numbers of moves to be applied to the cube are determined using any of the last four conditions in Eq. 3.3.

After each iteration, if the current state of the particle i gives a fitness value which is better (less) than the personal best S_{P_i} or the global best S_G then the personal best or the global best are updated accordingly.

In Eq. 3.3 the set of moves in $\mathbb{V}$ are kept as minimum as possible. The reason is that the moves applied to the cube are selected randomly. If a large number of random moves are applied to the cube, especially when the particle is near to the solved state, then there is a chance that the particle might overshoot the solution due to the random nature of selecting the moves.

3.1.1 Time Complexity Analysis of PSO

The PSO algorithm will run for N iterations and in each iteration, there are P particles that will use Eq. 3.3 to find out the number of random moves to be applied to the cube. Finding out the number of random moves to be applied to the cube and applying those moves takes a constant time of $O(1)$ for every particle P. The total time complexity is

$O(N \times P \times 1) = O(NP)$.

3.2 Greedy Tree Search Algorithm

The greedy tree search algorithm used for this study applies moves that reduce the fitness at each depth. From any state of the Rubik's cube, eighteen moves can be applied to it. These eighteen moves can take the cube from the current state to eighteen new states. If any of these states have a fitness value less than the previous state, then the move that gives the minimum fitness is applied to the cube. This move selection method decreases the fitness value at each depth.

If the next eighteen moves do not lead to a state that has a fitness value less than the previous state, then the last applied move is undone and a move that gives the next minimum fitness value is applied to the cube. This stage is called backtracking.

Cycles occur when a previously reached state is encountered again and the move to be applied from this state is the same move that was applied when this state was previously encountered. Since a greedy approach is used when selecting the next move to be applied to the cube, the algorithm can apply the same move that it previously applied when it first encountered that state, causing the algorithm to go through a cyclic loop and never reaching the solved state. Cycles can be prevented by checking if the next move to be applied to the cube will make it reach a state that has been encountered before. If this state has been encountered before then this move will not be applied to the cube and the next move is applied. Algorithm 3 provides an overview of the steps used in the greedy tree search algorithm.

3.2.1 Time Complexity Analysis of Greedy Tree Search

The greedy algorithm will run for N iterations and in each iteration, the algorithm will check if a move has a fitness value less than the current fitness value and it will apply that moves or it will do backtracking if it cannot find a move whose fitness is less than current fitness.

The moves that can be applied to the cube are fixed, here it is eighteen moves. The algorithms need to check only the available moves and see which one gives a fitness value less than the current fitness value. So this will take a time of $O(18)$.

Algorithm 3 Greedy tree search algorithm

Input: Scrambled cube
Output: Solved cube with solution

1: $iteration = 0$
2: $reachedStates = \{\}$
3: $solution = \{\}$
4: $S_{current} \leftarrow S_{scrambled}$
5: **while** $iteration \leq LIMIT$ **and** $IsNotSolved(S_{current})$ **do**
6: $\mathbb{M}_{HTM} = \{$R, R', R2, L, L', L2, U, U', U2, D, D' D2, F, F', F2, B, B' B2$\}$
7: $currentFitness \leftarrow Kociemba(S_{current})$
8: $newMove \leftarrow FindMinimumMove(S_{current}, \mathbb{M}_{HTM})$ // Find the move which has minimum fitness from all the possible moves
9: $S_{new} \leftarrow S_{current} + newMove$ // Apply the move
10: $newFitness \leftarrow Kociemba(S_{new})$ // Find fitness
11: **if** $S_{new} \in reachedStates$ **then**
12: $\mathbb{M}_{HTM} \leftarrow \mathbb{M}_{HTM} - newMove$ // Remove the move applied from the set of all moves
13: **go to** 8
14: **else if** $newFitness > currentFitness$ **then**
15: $oldMove \leftarrow ReturnLastElement(solution)$
16: $solution \leftarrow solution - oldMove$ // Remove last move applied from solution
17: $S_{current} \leftarrow S_{current} - oldMove$ // Undo the move
18: $\mathbb{M}_{HTM} \leftarrow \mathbb{M}_{HTM} - oldMove$
19: **go to** 8
20: **else**
21: $S_{current} \leftarrow S_{new}$
22: $reachedStates \leftarrow reachedStates \cup S_{new}$
23: $solution \leftarrow solution \cup newMove$
24: **end if**
25: $iteration \leftarrow iteration + 1$
26: **end while**

If the algorithm cannot find a move whose fitness is less than the current fitness then it will go back to the previous depth and undo the move that was applied and apply the move which gives the next minimum fitness value. The maximum depth the algorithm can go back to is equal to the number of iterations the algorithm has done. In the worst case, the algorithm will reach the last iteration of N and realize that no move will give a fitness less than the current fitness. So it will go to the previous depth and try again. In the worst-case scenario, the algorithm will go back to the very first depth and try again. So in the worst case, the backtracking loop will be executed a maximum of N times. So the time complexity for backtracking is $O(N)$. So the total time complexity is

$O(N \times (18 + N)) = O(N \times N) = O(N^2)$.

3.3 ACO for Solving the Rubik's Cube

The ACO is suitable for solving the Rubik's cube as it is used for solving graph-based problems. A few optimizations has to be made so that the ACO algorithm can properly solve the cube.

The pheromone deposited by an ant k, on an edge (i, j) is given by Eq. 2.3. This means that if an ant k travels from node i to node j then it will deposit some amount of pheromone represented by $\Delta\tau_{i,j}^k$ which is equal to $\frac{1}{L_k}$, where L_k is the length of the path between node i and node j found by the ant k. In the context of a Rubik's cube, the length of the path is the distance from the current state of the cube to the solved state. So if an ant k applies a move R to the cube from state S then it will go to the state $S+R$. The state $S + R$ will have a fitness value, which is the number of moves needed to solve the cube from the state $S + R$ given by Kociemba's algorithm, which is between 1 and 20. So for a Rubik's Cube, i represents the current state of the cube S and j represents the next state of the cube which can be reached by applying any of the possible moves in $\mathbb{M}_{HTM}$ = {R, R', R2, L, L', L2, F, F', F2, B, B', B2, U, U', U2, D, D', D2} to the cube from the state S. Let $\mathbb{M}$ be a singleton set with only a single element and this element can be any one of the possible eighteen moves that can be applied to the cube. So the Eq. 2.3 becomes:

$$\Delta\tau_{S,(S+\mathbb{M})}^k = \begin{cases} \frac{1}{Kociemba(S+\mathbb{M})} & , where\, k^{th}\, ant\, traversed\, the\, edge\, (S,\, S+\mathbb{M}) \\ 0 & , otherwise \end{cases} \tag{3.4}$$

Now the total amount of pheromones deposited is the sum total of the pheromones deposited by all the ants that traversed the edge between the state S and $(S + \mathbb{M})$, represented by

$$\tau_{S,(S+\mathbb{M})}^k = \Sigma_{k=1}^m \Delta\tau_{S,(S+\mathbb{M})}^k \tag{3.5}$$

where m is the total number of ants that traversed the edge between state S and state $S + \mathbb{M}$. The probability of selecting an edge is given by the following equation:

$$P_{S,(S+\mathbb{M})} = \frac{(\tau_{S,(S+\mathbb{M})})^\alpha (\eta_{S,(S+\mathbb{M})})^\beta}{\Sigma_{all\, edges} (\tau_{S,(S+\mathbb{M})})^\alpha (\eta_{S,(S+\mathbb{M})})^\beta} \tag{3.6}$$

where

$$\eta_{S,(S+\mathbb{M})} = \frac{1}{Kociemba(S+\mathbb{M})} \tag{3.7}$$

The roulette wheel technique is used for selecting the edge after the probability of each edge is computed. In the original ACO algorithm proposed by Dorigo [23], the value of α and β are set to 1 and 5, respectively. Through experimental analysis, it was found that the values of α and β of 1 and 5 did not affect the performance of the ACO algorithm when it comes to solving the Rubik's cube. Since the values of α and β do not affect the performance, they are set to 1.

Assumptions

Some assumptions are made before the algorithm begins its execution. If there are n ants that need to search the Rubik's cube search tree, then each ant will start traversing the depth of the tree one after the other. So the ants will follow the pheromone trail left by the previous ants. The ants select an edge based on the probability which is determined by the pheromone values left by the ants that traversed the tree before the current ant. For the first ant that is about to traverse the tree, no ant has traversed the tree before it and the pheromone trail is 0. Equation 3.6 now becomes:

$$\begin{aligned} P_{S,(S+\mathbb{M})} &= \frac{(\tau_{S,(S+\mathbb{M})})^{\alpha}(\eta_{S,(S+\mathbb{M})})^{\beta}}{\Sigma_{all\ edges}(\tau_{S,(S+\mathbb{M})})^{\alpha}(\eta_{S,(S+\mathbb{M})})^{\beta}} \\ P_{S,(S+\mathbb{M})} &= \frac{0 \times (\eta_{S,(S+\mathbb{M})})^{\beta}}{\Sigma_{all\ edges}(0 \times (\eta_{S,(S+\mathbb{M})})^{\beta}} \\ &= 0/0 \\ &= Undefined \end{aligned}$$

So for paths that have not been traversed the probability becomes undefined. To prevent this we assume that an ant has traversed the entire tree of possibilities for the Rubik's cube and has deposited a pheromone value of 0.000001 at every branch of the tree. This value is small but not 0. This ensures that the probability value does not become undefined. This is done for the proper implementation of the algorithm.

Pheromone dictionary

In ACO there is a matrix called as the pheromone matrix. This matrix is similar to the adjacency matrix of a graph where the elements in the matrix represent the distance between the nodes of the graph. In the pheromone matrix, the elements of the graph represent the amount of pheromone between the nodes of the graph. For solving the Rubik's

cube a pheromone dictionary is used instead of a pheromone matrix. The pheromone values for an edge will be stored in the pheromone dictionary.

It will be a dictionary containing the key, which is a state of the Rubik's cube, and the value will be a list of key-value pairs that show the pheromone value at each edge from the current node. As more states are reached by the ants through traversal, those states will be added to the pheromone dictionary.

Depth traversed and regrouping

Since the ants will try to find the solution to the Rubik's cube by searching the depths of the tree one depth after another, there should be some way to ensure that they do not get lost in the huge search space. For this a regrouping strategy is implemented.

All ants will regroup to a common node if any of the two following conditions are satisfied.

1. An ant has found a state whose fitness value is less than the global minimum fitness. Then all the ants will regroup to this new state that gave the new global minimum fitness value.

2. All ants have searched till a certain depth, which is equal to the global minimum value, and if no ant has found a state whose fitness is less than the global minimum then they will regroup from the starting state and try again.

When the ants regroup back to the starting state when a new global minimum value could not be found, the pheromone value of all the edges of the states that were found after the starting state will be set back to 0.000001. All the new state entries in the pheromone dictionary which was found after the starting state will be removed. This is done so that the ants can try again and won't get lost in the search space due to pheromones left behind during the previous iteration.

3.3.1 Time Complexity Analysis of ACO

The ACO algorithm will run for N iterations and each iteration will have P ants which will use Eq. 3.6 to calculate the probability for each of the moves that can be applied to the cube. After the probability for each move is calculated, a random number is generated and the move is selected based on roulette wheel selection. The number of moves that can be applied to the cube will always remain a constant, eighteen moves. If calculating the

probability for one move takes a constant time of $O(1)$ then calculating the probability for eighteen moves is $O(18)$. The time required to generate a random number is $O(1)$ and if a linear search is used to find out which move has a probability closest to the random number then the time complexity for this search is $O(18)$. So the time complexity is $O(N \times P \times (18 + 1 + 18)) = O(N \times P \times 18) = O(NP)$.

3.4 DKHO for Solving the Rubik's Cube

The DKHO algorithm for solving the Rubik's cube has four stages.

Move selection

In the move selection stage, the krills must decide which move they must apply to the cube, from the current state to reach the next state. The krills select the move based on two methods. One is the greedy move selection method and the other is the probability move selection method.

In the greedy move selection method, the krill will select the move that gives a fitness value that is less than the fitness of the current state. This move selection is the same as the one used by the greedy tree search algorithm explained in Sect. 3.2

In probability move selection, each krill will assign probability values for each move and, the roulette wheel technique is used for selecting the move. This is similar to the move selection method used by the ACO algorithm (Sect. 3.3). Since there is no concept of pheromone trail like in the ACO algorithm, the probability is calculated as follows

$$P_{S,(S+\mathbb{M})} = \frac{(\eta_{S,(S+\mathbb{M})})}{\Sigma_{all\ edges}(\eta_{S,(S+\mathbb{M})})} \tag{3.8}$$

where

$$\eta_{S,(S+\mathbb{M})} = \frac{1}{Kociemba(S+\mathbb{M})}$$

Selecting a move by computing its probability is done when the greedy method cannot find a move whose fitness value is less than the fitness value of the current state.

Killing of unfit krills

This stage occurs after all the krills have selected a move to apply to the cube. All the unfit krills are killed in this stage. A krill is said to be unfit if its fitness value is greater

than a threshold value. The threshold value is calculated as follows:

$$threshold = \frac{globalMinimum}{2} + globalMinimum \tag{3.9}$$

where the $globalMinimum$ is the global minimum fitness value of the current iteration. The threshold value is 50% greater than the global minimum value. If the range of values that are said to be fit is too small then there is a chance that many of the krills will be killed off, and only one krill will survive. This can cause problems in the reproduction stage as there won't be enough krill to do reproduction to bring the population back to normal. If the range of values that are said to be fit is too high then no krill will get killed, and this can cause many of the krills to wander around in the search space without finding the solution, which can slow down the time needed to find the solved state.

Reproduction

After the killing stage, the remaining krills will do reproduction till the population is back to normal. For the remaining krills to do reproduction, they need to have some sort of genetic information that they can pass onto their offsprings. In the context of the Rubik's cube, the genetic information is the move sequence that the krill has applied to the cube. The children are produced by cutting the move sequence of the parents at a random point and swapping the tail of both the move sequences. The offsprings will have the new move sequence. This kind of reproduction can cause the offsprings to take paths that were not explored by their parents, which can lead the children to have fitness value better than that of the parents or the global minimum.

If krill A has a move sequence of [R, L, U, B, D] and if krill B has a move sequence of [R', F', D2, B2, R2] and if both the move sequence are cut at index 3 then after reproduction krills A and B will produce offspring AB_1 with the move sequence [R, L, U, B2, R2] and offspring AB_2 with the move sequence [R', F', D2, B, D].

Mutation

Mutation happens after the reproduction stage and mutation is done to the offsprings that are not fit. This means that their fitness value is greater than the threshold value in Eq. 3.9. When two krills reproduce to give offsprings the fitness of these offsprings may be the same as that of the parents or sometimes even better than that of the parents. In certain cases the offsprings may be unfit. The mutation is done to these unfit krills to improve their fitness values.

Mutation is done by taking the move sequence of the unfit krill and replacing the moves at random points with a single random move at each point.

3.4.1 Time Complexity Analysis of DKHO

The KHO algorithm will run for N iterations and in each iteration, there will be four stages: move selection stage, killing of unfit krill stage, reproduction stage, and mutation stage.

In the first stage, which is the move selection stage, P krills need to select a move to be applied to the cube. They can do so using the greedy move selection method or using the probability move selection method. Since the number of moves that can be applied to the cube is constant, eighteen moves, the time complexity will also be a constant. So whichever move selection method is used by the krill, the time complexity will be $O(18)$. So for the first stage the time complexity is $O(P \times 18) = O(P)$.

The second stage is the killing stage. In this stage, the fitness of every krill is checked to see if it is less than the threshold value and all the unfit krills are killed off. This stage will have a maximum time complexity of $O(P)$ as the fitness of every krill needs to be checked.

The third stage is the reproduction stage, where reproduction is done between two krills to produce two new krills. The reproduction is done by cutting the gene of the krill at a random point, here is the move sequence applied to the cube, and swapping the tails of the move sequence to produce two new krills. Since the gene is cut at a random point, the reproduction between two krills takes a constant time. So the reproduction stage will have a time complexity of $O(P)$. This is because the reproduction is done till the population is back to the original population count, which is P.

The final stage is the mutation stage where unfit krills undergo mutation by replacing random points of the move sequence with a random move. Since random points of the move sequence are replaced with a single random move, the time complexity is constant. So the mutation stage will have a time complexity of $O(P)$ as every krill is checked to see if they are unfit and needs to undergo mutation.

So the overall time complexity is $O(N \times (P + P + P + P)) = O(NP)$.

Chapter 4

Experimental Analysis and Results

This chapter provided the experimental analysis and results obtained for the algorithms. Sections 4.1 to 4.4 tells about the individual analysis of the algorithms while Sect. 4.5 provides a comparative analysis of all the algorithms. In Sect. 4.5 each algorithm is made to compete against each other to find out which algorithm can solve the Rubik's cube optimally. Section 4.6 discusses the time complexity of each algorithm and explains why some algorithm performs better than other. More details about each experiment are provided in the following sections.

4.1 Experimental Analysis and Results for PSO

Table 4.1 shows the observations obtained for PSO. The algorithm was run for 10,000 iterations with 50 particles.

The table consists of eight columns. The column called *Length* shows the scramble length given to the algorithm which is the number of moves used to scramble the cube. The second column, *%* shows the success rate of finding the solution out of five attempts. For an attempt to be successful at least one particle has to find the solution. The next column *Actual* shows the solution length estimated by Kociemba's algorithm. The column *Obtained* shows the solution length found by the PSO algorithm. It shows the average of all the solution lengths obtained by all the particles. The column *First* shows the first iteration at which the solution was found by a particle. The column *Last* shows the iteration at which the last particle found the solution. The column *Average* shows the average number of iteration it took between each particle to find the solution. The last column *Total* shows out of the fifty particles how many were able to reach the solved state.

Table 4.1: Observations for particle swarm optimization algorithm

Length	%	Actual	Obtained	First	Last	Average	Total
5	5/5	9	27.4	125	3591	69.32	50
		5	18.82	35	7198	143.26	50
		5	11.36	24	1781	35.14	50
		8	18.44	74	2889	56.3	50
		11	14.18	649	3033	47.68	50
10	5/5	14	28.24	48	2456	48.16	50
		15	30.14	113	2004	37.28	50
		15	37.22	69	2858	55.78	50
		10	25.2	69	4203	82.68	50
		16	48.1363	8396	9642	28.31	44
15	5/5	20	45.48	363	2481	42.36	50
		18	49.28	150	318	3.36	50
		20	56.52	187	389	4.04	50
		17	46.38	65	2544	49.58	50
		15	40.40	103	1965	37.24	50
20	5/5	20	50.46	119	1735	32.32	50
		20	51.14	100	196	1.92	50
		20	63.78	2413	6639	84.52	50
		20	59.26	135	2330	43.9	50
		20	50.93	2358	9666	166.09	44
25	5/5	19	50.22	85	1146	21.22	50
		19	49.72	94	1539	28.9	50
		20	63.16	2267	2444	3.54	50
		20	51.5	251	2451	44.0	50
		20	71.46	3069	3881	16.24	50
30	5/5	19	49.2	1142	1216	1.48	50
		20	59.18	245	361	2.32	50
		18	68.32	262	1834	31.44	50
		19	54.32	153	2026	37.46	50
		20	52.12	46	265	4.38	50

Figures 4.1 and 4.2 shows the *Iteration vs Fitness* graph obtained for PSO. The Y-axis shows the fitness score and the X-axis shows the number of iterations which is on a semi-log scale.

Figure 4.1 is for the best case where the average iteration it takes for the next particle to find the solved state is minimum (1.48). If the average iteration between the particles to find the solved state is minimum, then the PSO algorithm will finish its execution much faster.

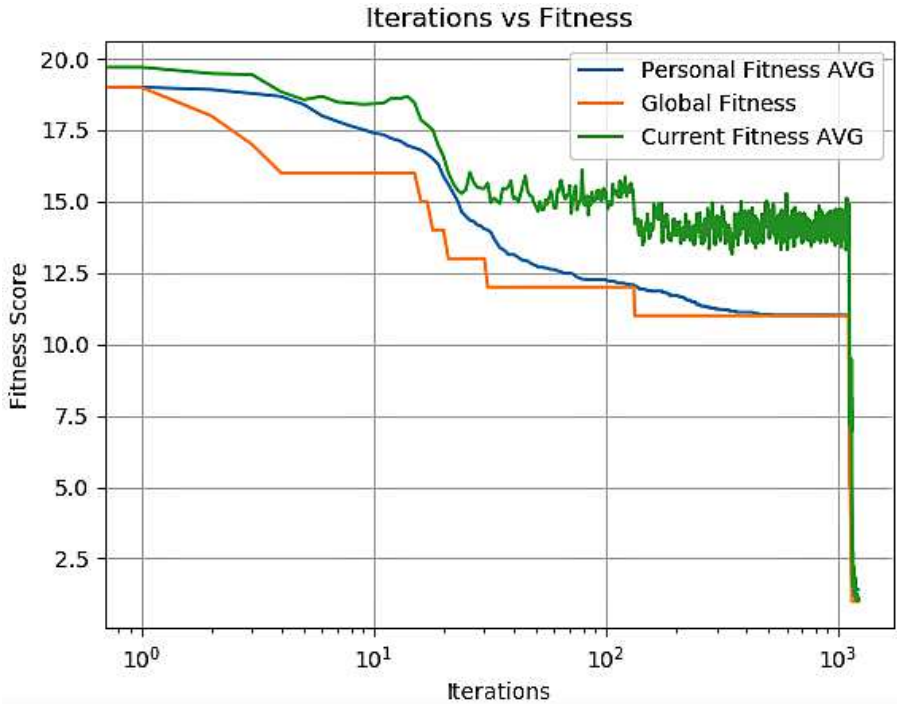


Figure 4.1: Iteration vs Fitness graph of PSO, for a scramble length of 30 moves on its first attempt. The graph obtained is for the best case.

Figure 4.2 is for the worst case where the average iteration it takes for the next particle to find the solved state is maximum (166.09). If the average iteration between the particles to find the solved state is large, then the PSO algorithm will take a longer time to finish its execution.

4.2 Experimental Analysis and Results for Greedy Tree Search Algorithm

Table 4.2 shows the observations obtained for the greedy tree search. The algorithm ran for 10,000 iterations. The first column shows the length of the scramble that was applied to the cube. The second column shows out of five attempts how many attempts were successful. The third column shows the length of the solution predicted by Kociemba's algorithm. The fourth column shows the length of the solution given by the greedy tree search. The fifth column *MinFit* shows the minimum fitness value reached by the program. If the cube is solved then the minimum fitness found by the program is one. The next column *MaxDeep* tells us the maximum depth that the algorithm went to find the solution. The last column tells us the total iterations it took to solve the cube.

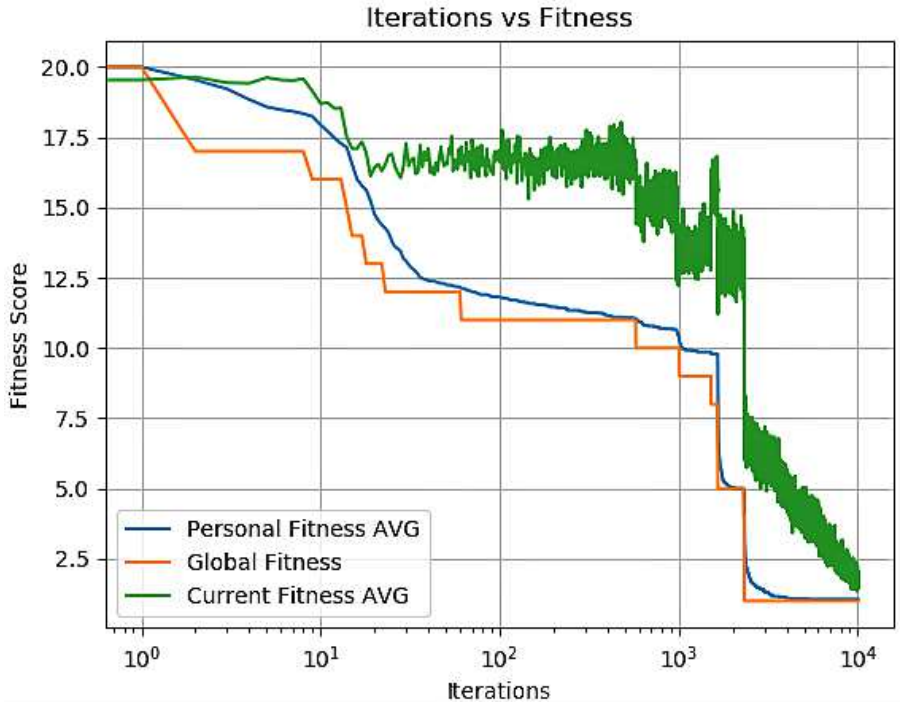


Figure 4.2: Iteration vs Fitness graph of PSO, for a scramble length of 20 moves on its fifth attempt. The graph obtained is for the worst case.

In the column *Obtained* a value is of the form NA(19). This means that the solution was not found and when the program stopped its execution after 10,000 iterations it gave a solution of length 19 which won't completely solve the cube. In the column *MinFit* for the scramble length of 20 in its third row, the cube is not solved and the minimum fitness is 7. This means that the cube is 7 moves away from the solved state when the program is terminated. So NA(19) means that the actual solution to solve the cube was not found but instead we obtained a solution of 19 moves that will give a state which is 7 moves away from the solved state.

Figures 4.3, 4.4, and 4.5 shows how the fitness value of the cube changes with every iteration and also the change in the global minimum value with every iteration.

Figure 4.3 is for the best case where the global minimum value decreases with each iteration along with the current fitness.

Figure 4.4, is for the average case. In the average case, the solution to the cube is found, but the algorithm did a few backtracking along the way. This is indicated by the spikes in the red plot which indicates the current fitness during each iteration.

Table 4.2: Observations for greedy tree search algorithm

Length	%	Actual	Obtained	MinFit	MaxDeep	Iterations
5	5/5	5	5	1	7	16
		5	6	1	6	6
		10	6	1	6	6
		11	5	1	5	5
		5	5	1	5	5
10	5/5	17	18	1	18	18
		13	13	1	13	13
		19	20	1	20	21
		15	12	1	12	12
		13	12	1	12	12
15	5/5	20	22	1	22	44
		19	21	1	21	21
		20	21	1	21	23
		20	29	1	29	48
		20	23	1	24	425
20	4/5	20	20	1	20	20
		20	44	1	44	1496
		20	NA(19)	7	27	10000
		20	22	1	22	24
		19	23	1	23	23
25	4/5	20	NA(22)	10	29	10000
		20	24	1	24	27
		20	23	1	23	26
		19	23	1	23	23
		20	25	1	25	30
30	3/5	20	23	1	23	23
		20	25	1	25	2932
		20	28	1	28	38
		20	NA(25)	12	33	10000
		20	NA(29)	9	36	10000

Figure 4.5, is for the worst cases where the solution to the cube could not be found and the minimum fitness reached is greater than 1.

4.3 Experimental Analysis and Results for ACO

For the experimental analysis of PSO (Sect. 4.1), the number of particles is 50 and the number of iterations is 10000. This is to ensure that the cube gets solved by covering a large number of states with a large number of particles and also to ensure that the particles have time to cover a large number of states with a large number of iterations.

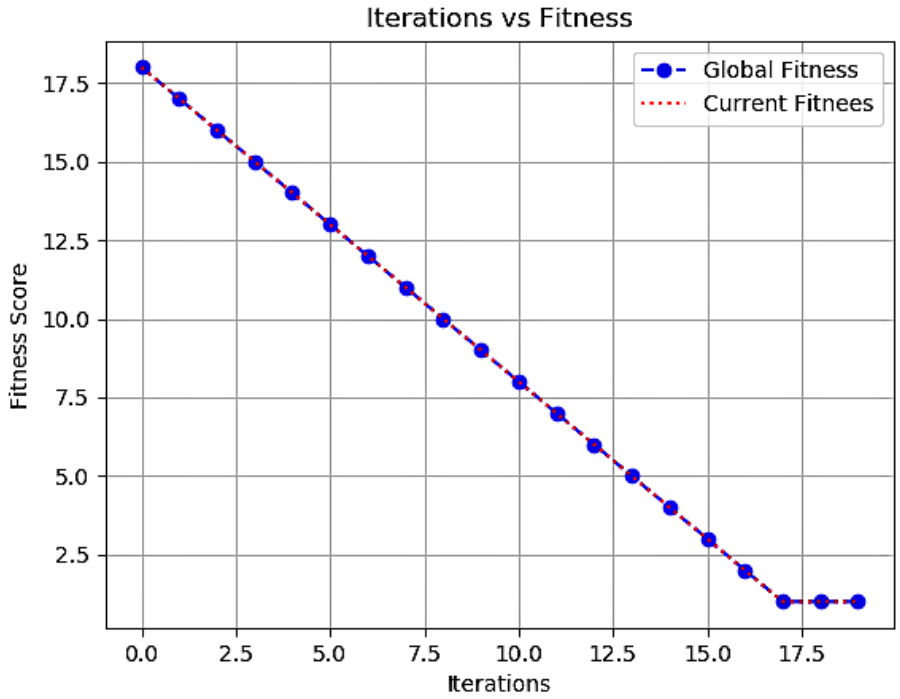


Figure 4.3: Iteration vs Fitness graph of greedy tree search, for a scramble length of 20 moves on its first attempt. The graph obtained is for the best case.

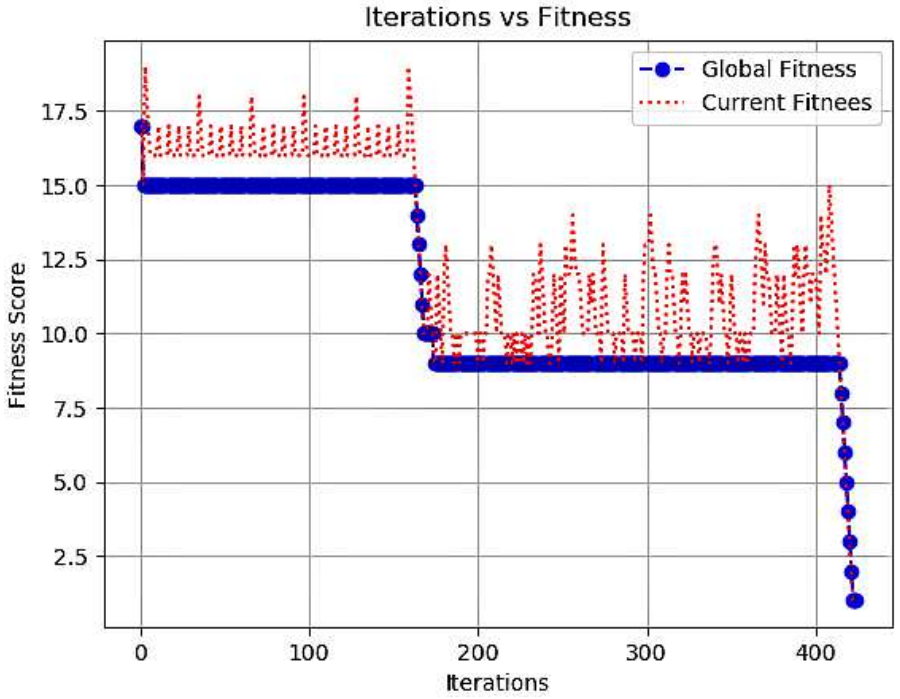


Figure 4.4: Iteration vs Fitness graph of greedy tree search, for a scramble length of 15 moves on its fifth attempt. The graph obtained is for the average case.

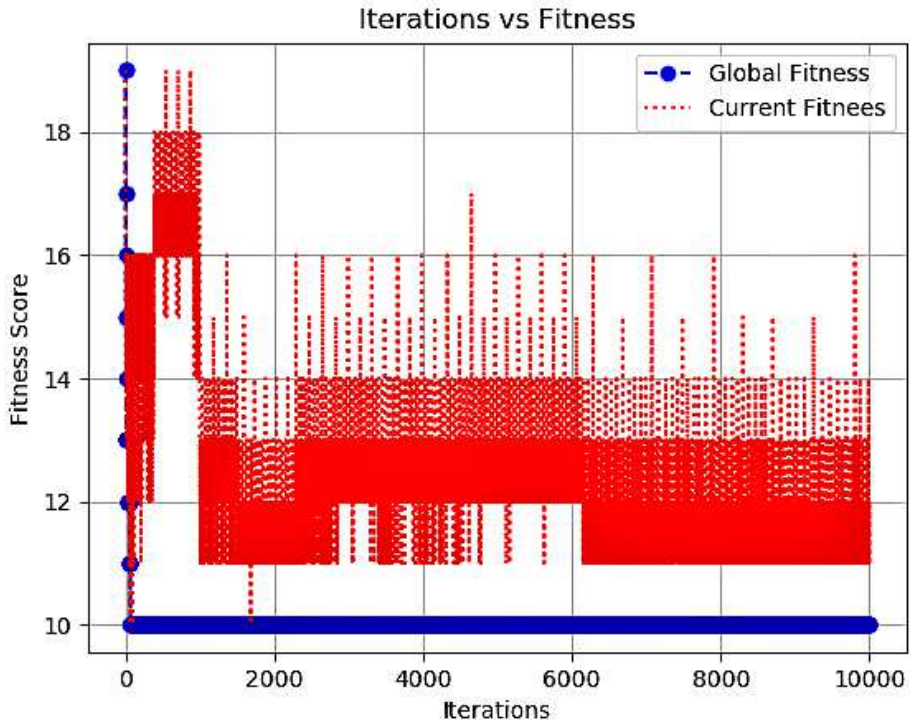


Figure 4.5: Iteration vs Fitness graph of greedy tree search, for a scramble length of 25 moves on its first attempt. The graph obtained is for the worst case.

The ACO takes more time to go through one iteration with 50 ants compared to one iteration with 50 particles in PSO. So the number of ants is reduced to 25 and the number of iterations is reduced to 100.

The columns of Table 4.3 are the same as that of Table 4.1. For the scramble length of 10 moves on its 5^{th} attempt, the ACO algorithm could not solve the cube within 100 iterations. The analysis for the scrambles of lengths 20, 25, and 30 moves using ACO was not done. For these scrambles, the ACO algorithm took around 15 hours to complete 25 iterations. From this, it became clear that the ACO algorithm would take days to complete the execution for scrambles of length greater than 20 moves. Therefore the analysis for these scrambles was not done.

The first type of graph shows how one of the ants finds the next minimum fitness in the current iteration and all the ants will regroup to the state that gave the minimum fitness in the next iteration. This will continue for all the iterations. So the fitness will decrease after each iteration. Figure 4.6 shows how after each iteration the fitness value decreases.

Table 4.3: Observations for ant colony optimization algorithm

Length	%	Actual	Obtained	First	Last	Average	Total
5	5/5	13	5	4	5	0.04	25
		12	5	4	5	0.04	25
		5	5	4	5	0.04	25
		10	5	4	5	0.04	25
		5	5	4	5	0.04	25
10	4/5	16	10	9	10	0.04	25
		20	18	17	18	0.04	25
		19	10	9	10	0.04	25
		18	10	9	10	0.04	25
		16	NA	NA	NA	NA	NA
15	5/5	18	18	17	18	0.04	25
		14	14	13	14	0.04	25
		17	15	14	15	0.04	25
		12	12	11	12	0.04	25
		15	15	14	15	0.04	25

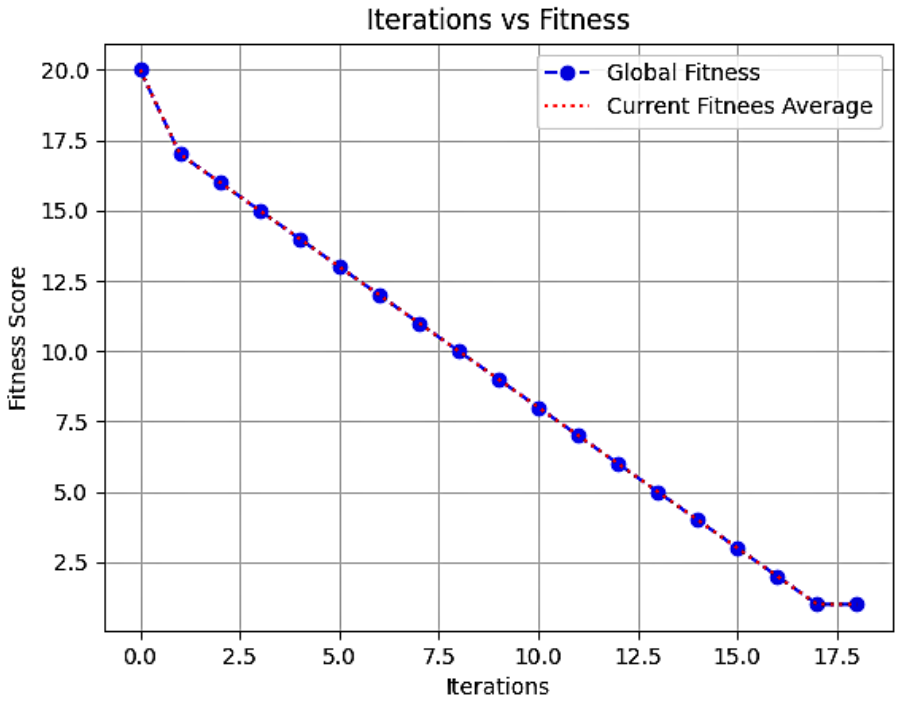


Figure 4.6: Iteration vs Fitness graph of ACO, for a scramble length of 10 moves on its second attempt. The graph obtained is for the best case.

The second type of graph shows how the ants regroup to the point where the minimum fitness was found if the new minimum fitness could not be found after a certain number of iterations. Figure 4.7 shows how the ants regroup to the state that gave the global minimum fitness after 11 iterations, as it was the first global minimum, and then regroup

after 10 iterations, as it was the next global minimum.

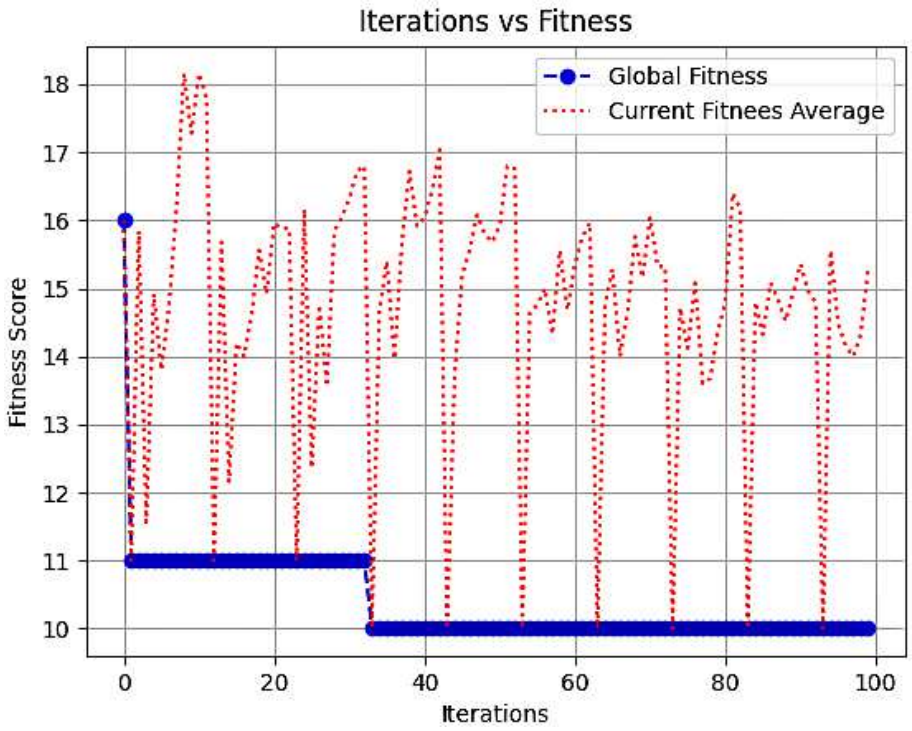


Figure 4.7: Iteration vs Fitness graph of ACO, for a scramble length of 10 moves on its fifth attempt. The graph obtained is for the worst case.

4.4 Experimental Analysis and Results for DKHO

The DKHO algorithm for solving the Rubik's cube takes more time to go through one iteration with 50 krills compared to 50 particles in PSO. So the number of krills is reduced to 25 and the number of iterations is reduced to 100.

The columns for Table 4.4 is the same as that of Table 4.1 and Table 4.3. The last column *Total* shows the total number of krills that were able to reach the solved state. For some attempts, the total is 1. This is because only a single krill survived the killing stage and since no other krills survived it couldn't do reproduction to bring the population back to normal.

The first kind of graph shown in Fig. 4.8, is when each krill finds the next minimum fitness after each iteration. The fitness of every krill will decrease after each iteration and the cube gets solved after a certain number of iterations. These kinds of graphs are obtained for scrambles less than 10 moves long and the type of edge selection method

Table 4.4: Observations for discrete krill herd optimization algorithm

Length	%	Actual	Obtained	First	Last	Average	Total
5	5/5	10	7	6	6	0	25
		12	8	7	7	0	1
		11	7	6	6	0	25
		12	7	6	6	0	1
		12	12	11	11	0	25
10	5/5	13	13	12	12	0	25
		15	11	10	10	0	25
		10	10	9	9	0	25
		19	18	17	17	0	25
		14	11	10	10	0	25
15	5/5	20	23	22	45	0.92	25
		19	16	15	15	0	25
		20	20	19	19	0	25
		20	21	20	20	0	1
		20	23	22	22	0	1
20	5/5	20	22	21	42	0.84	25
		20	21	20	47	1.05	25
		20	20	19	19	0	25
		20	19	18	18	0	25
		20	20	19	19	0	25
25	5/5	20	21	20	20	0	25
		20	28	27	27	0	25
		20	34	33	33	0	25
		20	20	19	19	0	25
		19	19	18	18	0	25
30	5/5	20	23	22	33	0.44	25
		19	33	32	32	0	25
		19	25	24	24	0	25
		20	22	21	21	0	25
		20	21	20	66	1.84	25

used by the krill is the greedy approach. Since the greedy approach is used each krill will have the same fitness value.

In the next kind of graph, the krills spread out to find the solved state. It happens for scrambles of length 15 moves and above. It happens when the greedy method does not work because the fitness value at the next depth will be the same or will be greater than the current fitness. So the krills will use the probability method to select a move. This can cause the krills to select different moves causing them to follow different paths rather than the same path, shown in Fig. 4.9.

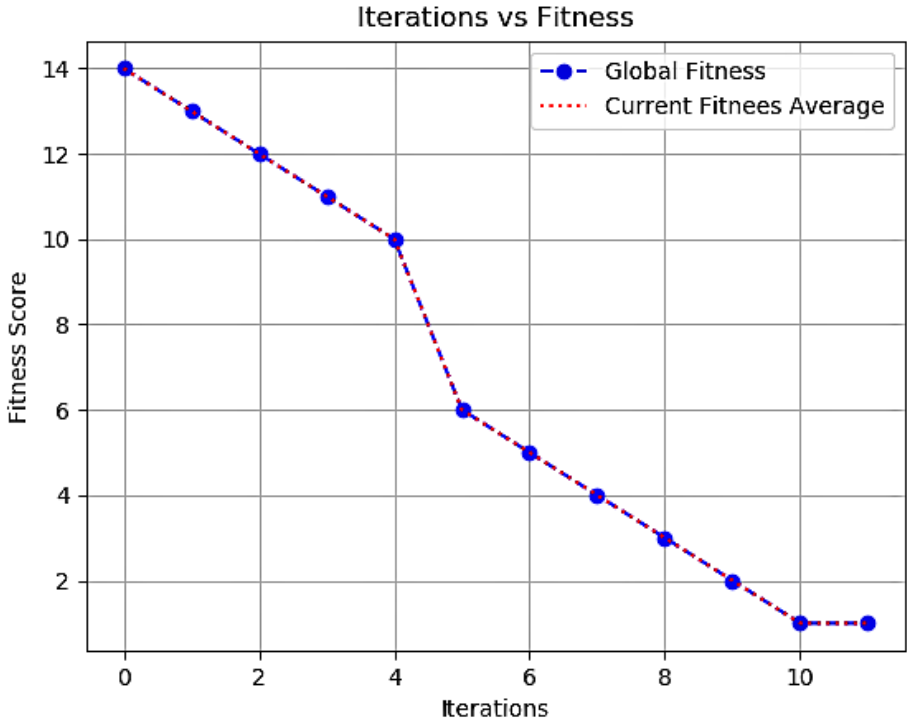


Figure 4.8: Iteration vs Fitness graph of DKHO, for a scramble length of 10 moves on its fifth attempt. The graph obtained is for the best case.

4.5 Comparative Analysis of the Algorithms

4.5.1 Experimental setup

To find out which of the above-described algorithms will give the optimum solution in the optimum amount of time, each algorithm is given scrambles of three difficulty levels. One is an easy scramble which is five moves long, the next is a medium scramble which is fifteen moves long, and the last one is a hard scramble which is twenty-five moves long. Each algorithm will be given three different scrambles for each difficulty level. The scrambles used are shown in Table 4.5. These scrambles have been generated randomly [1].

Table 4.6 has the observations for all the algorithms for the scrambles mentioned in Table 4.5. Table 4.6 contains ten columns. The first column shows the length of the scramble. The next shows the algorithms used. The next column shows the attempts that were successful out of the three tries. The column *Actual* shows the actual solution

[1]Python Code for generating random Rubik's cube scrambles is taken from: `https://github.com/BenGotts/Python-Rubiks-Cube-Scrambler`

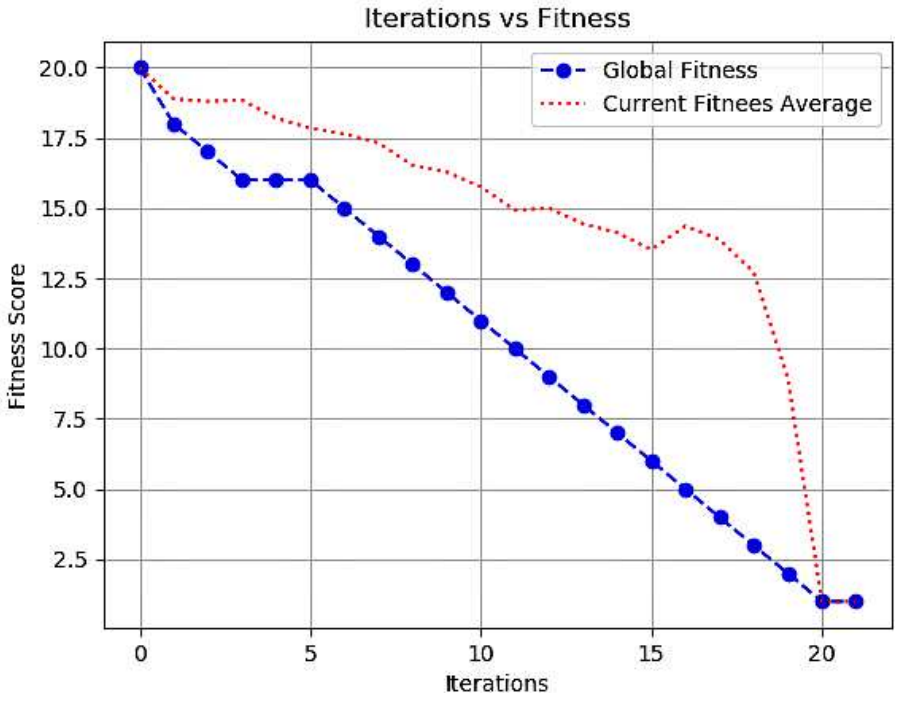


Figure 4.9: Iteration vs Fitness graph of DKHO, for a scramble length of 15 moves on its fourth attempt. The graph obtained is for the worst case.

Table 4.5: Scrambles used for comparative analysis of the algorithms

Length	Scramble
5	F2 L D2 B2 F B U' R2 B F' R2 B2 D F R2
15	U D2 F' U2 R' B2 L D' F2 U2 B2 R' F2 U L' D B L U B L2 U2 F' R' L' D2 R2 F U2 L' B2 L2 U B F2 D' U' R2 D B' U F' R' B' L
25	L2 D2 U' L2 F' R' U' F' R2 L U2 R2 F' U' R2 F2 D B U' D2 B' L F U' R' F R L D B' F' D L R D U2 L D' R F U' D' L F' D U R D L' F2 R U2 L' D B' R F L' B2 D2 F' L2 B' U L2 B R D' U' R2 B2 D L2 F U

length predicted by the Kociemba's algorithm. The column *Obtained* shows the solution length given by the corresponding algorithm. The column *MiniFit* shows the minimum fitness reached by the corresponding algorithm. If the cube is solved, then the minimum fitness is 1 otherwise it is greater than 1. The next column *First Time (s)*, shows the amount of time, in seconds, it took an agent to find the first solution. The next column *Total Time (s)*, shows the amount of time, in seconds, it took the algorithm to complete execution. The column *Avg. Time (s)* shows the average time it takes, in seconds, in between the agents to find the solution. The column *Total Agents* shows the total number of agents that were able to reach the solved state.

Six algorithms were used to do the comparison study. $PSO_{25/100}$, PSO with 25 particles and 100 iterations, $PSO_{50/10000}$, PSO with 50 particles and 10,000 iterations, $Greedy_{100}$, greedy algorithm with only 100 iterations, $Greedy_{10000}$, greedy algorithm with 10000 iterations, $ACO_{25/100}$, ACO with 25 particles and 100 iterations, $KHO_{25/100}$, KHO with 25 particles and 100 iterations. The parameters of PSO and greedy were changed to see how they would perform with a fewer number of iterations and particles.

For the column *Actual* in Table 4.6, which shows the fitness value estimated by Kociemba's algorithm, different fitness values are obtained for the same scramble. For scramble length of 5 in the first attempt for both the PSO's the actual value is 8 and for ACO and KHO it is also 8. For $Greedy_{100}$ it is 13, and for $Greedy_{10000}$ it is 12. This is because Kociemba sometimes gives two different fitness values for the same cube state. This is due to the design of Kociemba's algorithm. The algorithm is designed to solve the cube in less than 20 moves, but it is not designed to pick out the most optimum solution. As long as the solution length is less than 20 moves Kociemba's algorithm will return it. If a cube state can be solved in 13 moves and also in 8 moves and if Kociemba's algorithm found the 13 move solution first it will return that solution as it is less than 20 moves. This won't matter much for the analysis.

In Table 4.6 for $ACO_{25/100}$ for scramble length of 15 moves the first row is blank. This is because the ACO algorithm took around 14 hours to reach the 25 iteration. Due to a large amount of time it would take the ACO to complete its execution, the analysis was stopped after 25 iterations.

In Table 4.6 for scramble length of 25 moves $ACO_{25/100}$ was not included in the comparative analysis as ACO takes days to finish its execution when the scramble length is greater than 20 moves. (Explained in Sect. 4.3)

4.5.2 Observations

$PSO_{25/100}$ is suitable for solving scrambles that are less than five moves long but fails when the scramble length is more than five moves. To solve scrambles of greater length it needs more particles and iterations. $Greedy_{100}$ can solve easy, moderate, and difficult level scrambles, but it fails on occasion for difficult level scrambles.

From Table 4.6 it can be observed that out of the six algorithms, $Greedy_{10000}$ is the

Table 4.6: Observations obtained from the comparative analysis of PSO, greedy tree search, ACO, and DKHO

Length	Algorithm	%	Actual	Obtained	MinFit	First Time (s)	Total Time (s)	Avg. Time (s)	Total Agents
5	$PSO_{25/100}$	2/3	8	NA	4	NA	153.50	NA	0
			5	13.11	1	116.44	127.27	1.203	9
			10	10.33	1	12.046	43.352	5.217	6
	$PSO_{50/10000}$	3/3	8	19.46	1	5919.28	7140.46	24.423	50
			5	13.06	1	377.987	429.542	1.031	50
			10	9.46	1	19.18 3	1013.27	19.881	50
	$Greedy_{100}$	3/3	13	6	1	14.498	14.498	0	1
			12	7	1	5.343	5.343	0	1
			10	6	1	2.887	2.887	0	1
	$Greedy_{10000}$	3/3	12	6	1	24.477	24.477	0	1
			12	7	1	5.161	5.161	0	1
			10	6	1	2.702	2.702	0	1
	$ACO_{25/100}$	3/3	8	8	1	4809.67	4890.09	3.216	25
			5	7	1	2216.48	2531.17	12.627	25
			10	5	1	511.729	549.209	1.499	25
	$KHO_{25/100}$	3/3	8	6	1	681.647	681.647	0	1
			5	5	1	56.113	67.769	0.466	25
			10	6	1	31.683	38.29	0.264	25
15	$PSO_{25/100}$	0/3	19	NA	16	NA	835.264	NA	0
			20	NA	9	NA	182.838	NA	0
			19	NA	14	NA	279.011	NA	0
	$PSO_{50/10000}$	3/3	19	61.2	1	100.932	708.394	12.149	50
			20	62.1	1	32184.6	32236.6	1.039	50
			19	62.24	1	12939.7	13776.2	16.728	50
	$Greedy_{100}$	3/3	19	21	1	52.61	52.61	0	1
			20	21	1	17.091	17.091	0	1
			19	22	1	19.509	19.509	0	1
	$Greedy_{10000}$	3/3	19	21	1	55.214	55.214	0	1
			20	21	1	13.694	13.694	0	1
			19	22	1	25.241	25.241	0	1
	$ACO_{25/100}$	2/3	-	-	-	-	-	-	-
			20	20	1	2851.4	2912.6	2.448	25
			19	31	1	18029.8	18091.5	2.466	25
	$KHO_{25/100}$	3/3	19	21	1	5815.15	5821.34	0.247	25
			20	20	1	744.811	759.916	0.604	25
			19	21	1	3643.75	4966.35	52.904	25
25	$PSO_{25/100}$	0/3	20	NA	17	NA	911.462	NA	0
			19	NA	17	NA	1501.029	NA	0
			20	NA	7	NA	249.412	NA	0
	$PSO_{50/10000}$	3/3	20	59.46	1	43118.9	48128.8	100.19	50
			19	69.2	1	48838.2	55075.4	124.74	50
			20	51.3	1	289.917	12207.8	238.35	50
	$Greedy_{100}$	2/3	20	20	1	17.40	17.40	0	1
			19	NA	9	NA	116.04	NA	0
			20	20	1	39.341	39.341	0	1
	$Greedy_{10000}$	3/3	20	20	1	19.333	19.333	0	1
			19	23	1	2822.34	2822.34	0	1
			10	20	1	41.276	41.276	0	1
	$KHO_{25/100}$	3/3	20	30	1	11264.4	11270.7	0.25	25
			19	31	1	6101.96	6110.65	0.347	25
			20	21	1	19622.6	21465.6	73.718	25

algorithm that can solve the Rubik's cube in the optimum amount of time and gives a solution of optimum length, almost equal to or even better than the one predicted by Kociemba. Though the number of iterations is set to 10,000 the algorithm manages to find the solution within a few iterations. A large number of iterations ensures that it can

look through all the branches of the cube tree.

The second best algorithm after $Greedy_{10000}$ is the DKHO algorithm. The length of the solution provided by the DKHO is equal to or near to the one predicted by Kociemba's algorithm. The amount of time it takes is 10 minutes for the best case and at least 8 hours for the worst case.

The next best algorithm after DKHO is $PSO_{50/10000}$. The reason why $PSO_{50/10000}$ is considered better than ACO is that it solves the cube in a reasonable amount of time compared to the ACO algorithm even though the solution length given by $PSO_{50/10000}$ is three to four times larger than the solution length estimated by Kociemba's algorithm. The time required by $PSO_{50/10000}$ to solve the cube is 2 hours for the best case and at least 12 hours for the worst case.

The ACO algorithm, compared to other algorithms is only suitable for solving the Rubik's cube when the scramble length is less than 10 moves. For scrambles greater than 10 moves it will take a large amount of time for it to find the solution. For the scramble length of 25 moves the ACO algorithm took around 16 hours to complete 25 iterations, and the fitness it attained was around 15 to 17. Due to this reason, the ACO is not effective when it comes to solving the cube when the scramble length is greater than 10 moves. When ACO solves the cube for small scrambles, the solution length provided by it is equal to or near to the one predicted by Kociemba's algorithm. We can't reduce the number of iterations hoping that it will improve the solve time. The ACO searches for the solved state one depth per iteration, and since it takes a minimum of 20 moves to solve the cube, the number of iterations must be greater than 20. Even if the iterations are kept around 20, it is not guaranteed that ACO will find the solution within 20 iterations, so the iterations have to be greater than 20 which can increase the solve time. Therefore ACO is only suitable for easy scrambles of length 10 moves or less.

4.6 Discussions Regarding Time Complexity

Table 4.7 shows the time complexity of the algorithms used for this study. The time complexity of Kociemba's algorithm is assumed to be a constant. It makes deriving the time complexity of the algorithms much simpler.

From Table 4.7 it can be observed that the time complexities of PSO, ACO, and DKHO all depend upon the number of iterations, N, and the total number of agents,

P. For greedy tree search, the time complexity depends only on the total number of iterations, N. For greedy tree search in the best case when no backtracking is done its run time will be $O(N)$. For other algorithms, their run time is $O(NP)$. From this, it can be concluded that the greedy algorithm will outperform all the other algorithms since its run time does not depend upon the total number of agents, P.

Table 4.7: Time complexity of the swarm intelligence algorithms

Algorithms	Time Complexity
Particle Swarm Optimization	$O(NP)$
Greedy Tree Search Algorithm	$O(N^2)$
Ant Colony Optimization	$O(NP)$
Discrete Krill Herd Optimization	$O(NP)$

Chapter 5

Conclusion

This study was conducted to determine which swarm intelligence algorithms can solve the Rubik's cube with an optimum number of moves in the optimum amount of time. The algorithms used for this survey are particle swarm optimization, a variation of IDA* tree search algorithm called greedy tree search, ant colony optimization, and discrete krill herd optimization. The algorithms were optimized to solve the Rubik's cube. The time complexity of all the algorithms was analyzed, followed by individual analysis of each algorithm.

To find out which algorithm can solve the Rubik's cube optimally, each algorithm was given an easy scramble, a moderately difficult scramble, and a difficult scramble, and for each difficulty level, the algorithms had three attempts to solve the cube. The time taken to solve the cube for each difficulty level was recorded along with the number of moves taken by each of the algorithms. The number of iterations and the number of particles for PSO and the greedy tree search were reduced to see how they would perform.

From the experimental analysis, it was observed that the greedy tree search algorithm solves the Rubik's cube in the optimum amount of time with an optimum number of moves followed by the discrete krill herd optimization algorithm, particle swarm optimization, and ant colony optimization. Even though the PSO gives a much longer solution than ACO, it does so in an optimum amount of time.

This study uses an objective function that has only a single objective, which is the number of moves needed to reach the solved state from the current state. The results could be improved if a multi-objective optimization function were to be used as the fitness function, where the objective is to not only minimize the number of moves needed to solve the cube but also the time. This work mainly focuses on finding the minimum number of

moves needed to solve the cube. While doing the experimental analysis, it became clear that the time required to solve the cube could also be improved.

Bibliography

[1] I. Kassabalidis, M. A. El-Sharkawi, R. J. Marks, P. Arabshahi, and A. A. Gray, "Swarm intelligence for routing in communication networks," in *GLOBECOM'01. IEEE Global Telecommunications Conference (Cat. No.01CH37270)*, vol. 6, pp. 3613–3617, 2001.

[2] Ying-yin Lin and Ying-ping Chen, "Crowd control with swarm intelligence," in *2007 IEEE Congress on Evolutionary Computation*, pp. 3321–3328, 2007.

[3] M. A. Lewis and G. A. Bekey, "The behavioral self-organization of nanorobots using local rules.," in *IROS*, pp. 1333–1338, 1992.

[4] C. Leke and T. Marwala, "Missing data estimation in high-dimensional datasets: A swarm intelligence-deep neural network approach," in *International Conference on Swarm Intelligence*, pp. 259–270, Springer, 2016.

[5] B. Xue, M. Zhang, W. N. Browne, and X. Yao, "A survey on evolutionary computation approaches to feature selection," *IEEE Transactions on Evolutionary Computation*, vol. 20, no. 4, pp. 606–626, 2016.

[6] G. Vrbančič, I. Fister Jr, and V. Podgorelec, "Swarm intelligence approaches for parameter setting of deep learning neural network: Case study on phishing websites classification," in *Proceedings of the 8th International Conference on Web Intelligence, Mining and Semantics*, pp. 1–8, 2018.

[7] S. Saeidi, "Solving the rubik's cube using simulated annealing and genetic algorithm," *International Journal of Education and Management Engineering*, vol. 8, no. 1, pp. 1–12, 2018.

[8] N. El-Sourani and M. Borschbach, "Design and comparison of two evolutionary approaches for solving the rubik's cube," in *International Conference on Parallel Problem Solving from Nature*, pp. 442–451, Springer, 2010.

[9] N. El-Sourani, S. Hauke, and M. Borschbach, "An evolutionary approach for solving the rubik's cube incorporating exact methods. applications of evolutionary computations," *LNCS*, vol. 6024, pp. 80–90.

[10] WCA, "World cube assocaiaiton," 2003. Accessed March 2021.

[11] D. Singmaster, *Notes on Rubik's magic cube*. Enslow Pub Inc, 1981.

[12] WCA, "Wca regulations," 2003. Accessed March 2021.

[13] R. V. Grol, "The quest for god's number," November 2010. Accessed March 2021.

[14] T. Rokicki, H. Kociemba, M. Davidson, and J. Dethridge, "The diameter of the rubik's cube group is twenty," *SIAM Review*, vol. 56, no. 4, pp. 645–670, 2014.

[15] T. Rokicki, H. Kociemba, M. Davidson, and J. Dethridge, "God's number is 20," 2010. Accessed March 2021.

[16] R. E. Korf, "Depth-first iterative-deepening: An optimal admissible tree search," *Artificial intelligence*, vol. 27, no. 1, pp. 97–109, 1985.

[17] J. Scherphuis, "Kociemba's algorithm," 2015. Accessed March 2021.

[18] H. Kociemba, "Cube explorer 5.12 htm and qtm," 2014. Accessed March 2021.

[19] H. Kociemba, "Cube explorer 5.01," 2014. Accessed March 2021.

[20] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proceedings of ICNN'95 - International Conference on Neural Networks*, vol. 4, pp. 1942–1948, 1995.

[21] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[22] R. E. Korf, "Finding optimal solutions to rubik's cube using pattern databases," in *AAAI/IAAI*, pp. 700–705, 1997.

[23] M. Dorigo, "Ant colony optimization—new optimization techniques in engineering," *Springer-Verlag Berlin Heidelberg by Onwubolu, GC, and BV Babu*, pp. 101–117, 1991.

[24] M. Dorigo, M. Birattari, and T. Stutzle, "Ant colony optimization," *IEEE Computational Intelligence Magazine*, vol. 1, no. 4, pp. 28–39, 2006.

[25] A. Lipowski and D. Lipowska, "Roulette-wheel selection via stochastic acceptance," *Physica A: Statistical Mechanics and its Applications*, vol. 391, no. 6, pp. 2193–2196, 2012.

[26] A. H. Gandomi and A. H. Alavi, "Krill herd: A new bio-inspired optimization algorithm," *Communications in Nonlinear Science and Numerical Simulation*, vol. 17, no. 12, pp. 4831–4845, 2012.

[27] C. Sur and A. Shukla, "Discrete krill herd algorithm–a bio-inspired meta-heuristics for graph based network route optimization," in *International Conference on Distributed Computing and Internet Technology*, pp. 152–163, Springer, 2014.