DEGREE PROJECT, IN COMPUTER SCIENCE , FIRST LEVEL
STOCKHOLM, SWEDEN 2015

Algorithms for solving the Rubik's cube

A STUDY OF HOW TO SOLVE THE RUBIK'S CUBE USING TWO FAMOUS APPROACHES: THE THISTLEWAITE'S ALGORITHM AND THE IDA* ALGORITHM.

HARPREET KAUR

KTH ROYAL INSTITUTE OF TECHNOLOGY

CSC SCHOOL

Algorithms for solving the Rubik's cube

A study of how to solve the Rubik's cube using two popular approaches: the Thistlewaite's algorithm and the IDA* algorithm.

Harpreet Happy Kaur (hkau@kth.se)
DD143X Degree Project in Computer Science, first level
Supervisor: Arvind Kumar
Examiner : Örjan Ekeberg

CSC, KTH May 27, 2015

Abstract

There are different computational algorithms for solving the Rubik's cube, such as Thistlewaite's algorithm, Kociemba's algorithm and IDA* search algorithm. This thesis evaluates the efficiency of two algorithms by analyzing time, performance and how many moves are required to solve the Rubik's cube. The results show that the Thistlewaite's algorithm is less efficient than the IDA* algorithm based on time and performance. The paper attempts to answer which algorithm are more efficient for solving the Rubik's cube.It is important to mention that this report could not prove which algorithm is most efficient while solving the whole cube due to limited data, literature studies and authors are used as an argument to prove that the Korf's algorithm is more efficient.

Abstract

Det finns ett antal olika algoritmer för att lösa Rubik's kuben , såsom Thistlewaite algoritm , Kociemba algoritm och IDA* algoritm. Denna rapport utforskar effektiviteten av två optimala algoritmer genom att analysera tid , prestanda och hur många operationer som krävs för att lösa Rubik's kuben. Resultaten visar att Thistlewaites algoritm är mindre effektiv än IDA * algoritmen baserad på tid och prestanda. Denna rapport försöker besvara på frågan: vilken algoritm är mer anpassad för att lösa kuben. Det är viktigt att nämna att rapporten inte kunde bevisa vilken algoritm är mest effektiv för att lösa hela kuben, och därför användes literaturstudier och olika författare för att bevisa att Korf's algoritm är mer effektiv.

Contents

0.0.1 Terminology

1. **Facelet/Cubie:** The Rubik's cube is made of twenty subcubes or cubies that is attached to the cent

2. **Middle layer** Consists of one stage, and is the layer between L and R.

3. **Parity of edges and corners** The overall parity must be even according to the laws of the cube.

4. **Layer:** Contains one side of the cube.

5. **Tetrad:** is made up of four corners in an orbit.

6. **Cost bound:** cost of the current iteration.

7. **Permutations:** Ordering of the faces of the Rubik's cube.

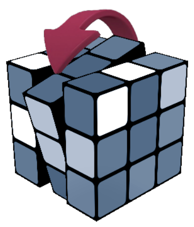


Figure 1: Middle layer [35]

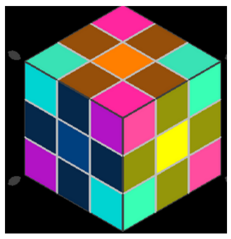


Figure 2: Pink shadows representing a tetrad [36]

Chapter 1

Introduction

The Rubik's cube is a 3-D combination puzzle invented in 1974 by Erno Rubik, a Hungarian professor. The motivation for developing the cube was that he wanted to create a model that explains the three dimensional geometry to his students. The Rubik's cube was conceived as a teaching tool, where Erno Rubik could demonstrate the structural design to his students. Also known as the "Magic Cube", the Rubik's cube has become a hit worldwide for nearly a generation. Individuals of all ages have spent hours, weeks or months trying to solve the cube. As a result, over 350 million Rubik's cubes have been sold worldwide, making it the world's most famous, bestselling and well-known puzzle to have a rich underlying logical, mathematical structure[1].

The standard version is a 3-D combination puzzle with six faces covered by nine stickers, each of the faces have one of six colors; white, red, blue, orange, green and yellow. The puzzle is scrambled by making a number of random moves, where any cube can be twisted 90,180 or 270 degrees. The task is to restore the cube to its goal state, where all the squares on each side of the cube are of the same color(fig b) To solve a scrambled Rubik's(fig a) cube, one needs an algorithm, which is a sequence of moves in order to come closer to a solution of the puzzle. Such sequences usually require more than fifty to hundred moves for a regular human being. However, it is believed that any cube can be solved in less than twenty moves.[4]

There are several computational approaches for solving the Rubik's cube.

(a) Scrambled cube

(b) Solved cube

However, the standard approaches for solving the Rubik's cube in this research are Thistlethwaite's algorithm and the IDA* algorithm.

1.1 Problem Definition

The most basic way a computer can represent a Rubik's cube is to use graph theoretical concepts, where the Rubik's cube can be represented as a graph. Algorithms such as depth first search and breath first search can be implemented to find a sequence of edges(cube twists) to solve the cube, by reaching into a desirable configuration. This is referred to as a brute force search. The name brute force search is commonly used in computer graph theory and is a technique that has been extensively applied for finding solutions to problems in for example Artificial intelligence. [9]

The problem with brute force search is performance. Algorithms such as depth first search or breadth first search are impractical on large problems.[17] The brute force search is only feasible for small problem instances and has some serious limitations, which are overcome by Thistewaite's algorithm and the IDA* algorithm. Therefore, this thesis will analyze the Thistlethwaite's algorithm instead of the depth first search(as chosen before). [24]

1.1.1 Artificial intelligence

Over the decades, the Artificial intelligence research community has made various contributions to mankind's knowledge. Finding a solution to a constraint search problem is thought to be a proof of intelligence and therefore, one might think that if a machine can solve the cube, it can be assumed to be intelligent. Questions such as, is it possible to accomplish a human like behavior in computers?, still serves as an unsolved mystery.

The use of games and puzzles in Artificial research dates back to its earliest days in 1950, where the idea of creating the illusion of human intelligence in a computer has been illustrated throughout the history of computer science. However, finding optimal solutions for the Rubik's cube still serves as a challenging problem today. [5] For researches all around the world, the following problems are challenges for AI research:

- The Rubik's cube contains about $4.3252x10^19$ states, making it impossible to store in the memory. This is equal to the number of states reachable from any given state.
- The minimum number of moves required to solve a Rubik's cube. [6]

It is not possible for any human being to physically count each permutation, and solving problems such as finding optimal solutions cannot be solved only by humans. Therefore, computers act as a utilitarian tool to solve problems such as Rubik's cube puzzle, since computers are good at numeric calculations, memorizing large sets of data and can search quickly. However, humans are really good at generalizing and using knowledge, something that computers are primitive in comparison. [5]

1.2 Problem statement

This thesis will explore two popular approaches for solving the Rubik's cube. Both methods are based on the idea to solve the Rubik's cube by using subproblems. The performance of an algorithm is measured by comparing the time and space complexity, if the algorithm is optimal etc.

- To compare the time, moves and performance efficiency of the Korf's algorithm(IDA*) and Thistlewaits algorithm.

1.3 Motivation

Variations of possible cube configurations have inspired many computer scientists as well as mathematicians to study the Rubik's cube, since the cube contains 43 quintillion states. Professors and scientists around the world have often used the Rubik's cube as a case for studying optimal solutions, creating new algorithms(step-by-step guide to solve a Rubik's cube) or permutations(ordering of the faces of the cube).

The reason for choosing the Rubik's cube as a platform is because it has a significant importance in the computer research field. It serves as an example of what the complex counting equations can be used for while studying the permutations. The Rubik' cube has also been a useful tool to test for the Artificial Intelligence to learn, develop new generic algorithms, and improve existing ones that can be applied to several other problems as well. [2]

1.4 Purpose

The first goal of this project is to investigate different approaches for solving the Rubik's cube to find out which uses the least amount of time, memory, and moves. The second goal of this project is to motivate other programmers to seek out, try to and analyze different search algorithms for solving the Rubik's cube.

Chapter 2

Background

This background consists of five different parts. The first part covers the Rubik's cube, followed by graph theory and popular approaches for solving the Rubik's cube, and thereafter an introduction of heuristic search. In the last section the complexity of the IDA* algorithm is covered.

2.1 The Rubik's cube

2.1.1 Rubik's mechanics

The standard version 3x3x3 consists of two distinct components: the core and the outer cubes. The core can be represented as a central cube with six attached octagonal, where each octagon is allowed to rotate in a 90, 180 or 270 degree in either direction. The outer cube is attached to the core and consists of three types: the slides, the edges and the corners. There are in total six side pieces; each side piece is attached to one of the octagons of the core and these pieces cannot be moved. Rubik's cube has twelve edge pieces and eight corners. Each edge piece has two visible faces and each corner piece has three visible faces. Any given edge piece or corner piece can be rotated in any edge position, however some permutation sets are not feasible. One side piece, four corners and four edges make up one face of a Rubik's cube, where each face is labeled with a solid color. Finally, these twenty six pieces form a Rubik's cube(fifty four colored stickers) that are able to rotate through all three dimensions. The aim is to

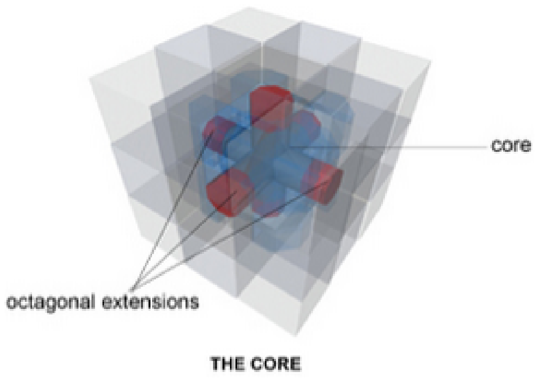
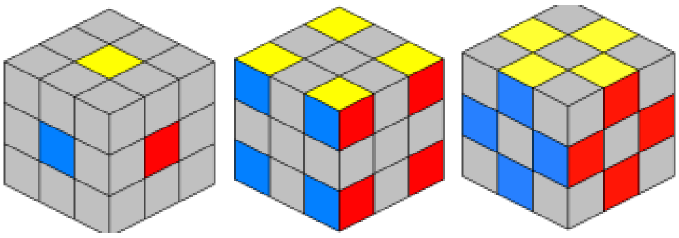


Figure 2.1: A Rubik's Cube


(a) Center pieces (b) Corner pieces (c) Edges pieces

restore the cube to its goal state, where all the faces on each side of the cube are of the same color. Every solver of the Cube uses an algorithm, which is a sequence of moves in order to come closer to a solution of the puzzle.[2]

There are 8! permutations of the eight corners, each of which can be arranged in three orientations. This means that there are 3^8 possibilities for each permutation of the corner pieces. There are 12! permutations of the twelve edges ,where each piece has two possible orientation. [3] So each permutation of the edge pieces has ?2^12 arrangements, and altogether total number of states is $8!x3^8x12!x2^12$. However, these permutations are split into twelve classes that makes it possible to only transform between states in the same class. So altogether there are $8! * 3^8 * 12! * 2^12(4, 3x10^19)$ states reachable from one state. [4]

2.1.2 Definitions

To refer to the Rubik's cube, one could use the notation by David Singmaster to name the six faces of the Rubik's cube as following right(R), left (L), up(U), down (D), front(F) and back(B). Each face can be rotated in two different directions; the first one is the 90-degree clockwise and the second one is the counter-clockwise turn, which is the opposite direction. [3]

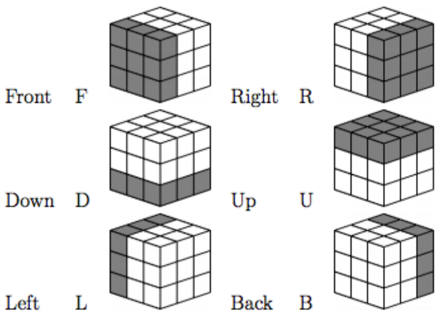
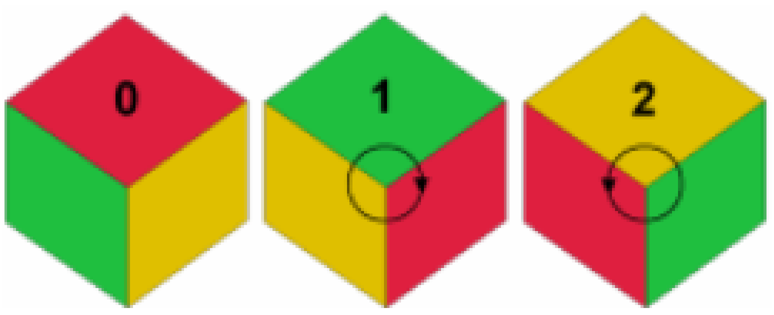


Figure 2.2: Notation [3]

2.1.3 Orientation

An edge cubie can have a good or a bad orientation. An edge is considered to be a good orientation if the piece can be brought back to its original position in the right orientation, without twisting the UP or Down faces of the Rubik's cube. The Up and Down



[10]

(a) Fig 2:3 Corner orientation resulting in three orientation: 0(corrected oriented), 1(clockwise) and 2(counterclockwise).

turns of the Rubik's cube will flip the edge orientation of the Upper or Down cubies, where the L or R turn will preserve the corner orientation of the Left and Right pieces. Since edges only have two possible orientations, it can be represented as a 1(good orientation) or 0(bad orientation) respectively.

For the corner cubies of the Rubik's cube, things are more complicated since there are three possible orientations. If a corner cubie is corrected oriented,this will be denoted by 0 and when a corner cubie is twisted clockwise or counterclockwise, the orientation will be denoted by 1 or 2 respectively. Note that there are also another way to measure the orientation of each cubie. If a corner cubie have a L or R facelet that belongs to either the L or R face, this can be denoted by 0. Otherwise, the orientation will be determined as figure. (Fig2.3)

Furthermore, this definition will be more apparent when the Thistlethwaite's algorithm is explained in further sections. [10]

2.2 Graph theory to solve the Rubik's cube

Graph theory is widely explored, implemented and used to study various applications in the field of Computer Science. Graph theoretical concepts are able to effectively represent and solve complex problems such as the Rubik's cube. The major role of this theory in computer science is the development of graph algorithms, that can be used to solve several problems that are represented in the form of graphs. A Rubik's cube can be represented as a graph; where the vertices are a set of all possible configurations and the edges represents the connection between configurations, that are one twist away from each other. It is known to find the shortest path between two or more vertices in the graph.Lastly, this theory is also effectively to use in problems such as finding solutions for solving the Rubik's Cube. [9]

Breadth first search and depth first search are very easy to program either iteratively or recursively, and can be applied to find a sequence of edges (cube twists) that will solve the cube, by reaching into a desirable configuration.[13]

2.2.1 Depth first search algorithm

Depth first search is an algorithm for searching the tree or graph data structures. It is a technique that has been widely applied for finding solutions to problems in Artificial Intelligence. [10] This method has also been widely acknowledged as a powerful technique for solving various graph problems as well as for treading mazes, but whose properties have not yet been widely analyzed.[11] Assume one is given a graph. Starting from one vertex of G, the algorithm traverses along the edges, from vertex to vertex until some depth cutoff is reached. When it reaches as far as possible(run of edges), then it backtracks to the next most recently expanded node and begins a new exploration from this point. Lastly, the algorithm will traverse along all the edges of G, each exactly one time.

The only path that will be stored in order to execute the algorithm will be the path between the initial node and the current node. Since, the depth first search algorithm only stores the path between the initial node and the current path, it is bound to search all the paths in the graph to the cutoff depth. In order to examine the time complexity of this algorithm, one needs to define a new parameter e that stands for edge branching factor. The time complexity is $O(b^d)$. Note that this factor stands for the average number of different operators applicable to a given state. [12] The space complexity of this algorithm is $O(bm)$, where b stands for the branching factor and m stands for the maximum depth of the tree. [14]

2.2.2 Benefits and limitations of the depth first search

A depth first search is an algorithm that has minimal space requirements and is space efficient. One can implement the depth first search algorithm with a last-in first-out stack or implement as a recursive implementation. The memory requirements of this algorithm are linear in the uttermost depth of the search, since the algorithm only needs to store those states currently in the search stack. Another advantage of this algorithm is that if it founds a solution without exploring the whole tree, then the time and space it takes will be lower. However, a depth first search algorithm has several drawbacks. [16]

Given a configuration of the cube, there are a finite number of movements. If one wants to find the solution to a Rubik's cube, one needs to consider that the entire state space for this problem is enormous, which may cause the algorithm to choose an edge leading to a subgraph that does not contain a solution state and is enormous to expand fully. This results in a solution that will never be returned by this algorithm, since the code might rotate the same plane alternatingly back and forth making it impossible for the program to make any further progresses. [24] To overcome the blindness of the depth first search algorithm, the constraints of pruning, limited depth search are added. [14]. Move pruning is an example of pruning that uses a low - overhead technique for reducing the size of a depth first search tree, making it possible to find solutions as well as performing better. [15] By limiting how many edges/depths the algorithm will search until it backtracks, it will always give a solution if there is one to be found at the given depth. This is known as depth limited search.

Some other major disadvantage of depth first approaches is that in a graph with multiple paths to the same state, any depth first search may generate far more nodes than there are states, since it cannot detect duplicate nodes. [17] As a result, the entire number of generating nodes may be greater than the number of nodes generated by other algorithms.[16]

2.2.3 Breadth first search

Breadth-first search is an algorithm for traversing and searching a tree or a graph. It starts at the tree root and explores the neighbour nodes first, and then moves to the next state until a goal state is reached. The idea behind this algorithm is to search one level of the search tree at a time, by using a queue (First in first out). The first solution path found by breadth-first search will be the shortest path, since it always explores and expands all the nodes at a given depth before expanding any nodes at a greater depth.

The time complexity of this algorithm is proportional to the number of nodes generated as the depth first search algorithm,$O(b^d)$. The space complexity is also$O(b^d)$. [14]

2.2.4 Limitations and benefits of using the breadth first search algorithm

An advantage of using the breadth first search over the depth first search algorithm is that it always finds the shortest path to a goal. Although this algorithm does not have the subtle properties of the depth first search (minimum requirements etc), it still provides advantages such as[17]:

- A BFS is a complete search.
- One can avoid infinite loops, if they check for repeated states in their algorithm, this algorithm will never get trapped.
- The tree has no loops.
- If there exist a solution, the BFS algorithm will always find it.
- It will always find the shortest solution even though there is more than one solution.[14]

The major disadvantage of this algorithm is the memory that requires to store all the nodes, compared to the depth first search algorithm that has minimal space requirements. This results in exponential space complexity, which may exhaust the available memory on the computer in a matter of minutes. Therefore this algorithm is impractical on large problems.[17]

2.3 Popular approaches

There are several computational approaches for solving the Rubik's cube, such as Thistlethwaite, Kociemba's algorithm and IDA* algorithm. These advanced algorithms are the most commonly used in computer science for solving the Rubik's cube. These are based on group theory concepts and advanced concept such as traversal.[8]

2.3.1 Thitslewaite's algorithm

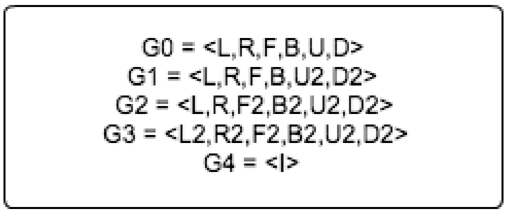
Thistlethwaite's algorithm was invented in 1981 by Morwen Thistletwaite, a British professor of mathematics. This algorithm is based on a mathematical theory, called group theory and on extensive computer searches, which studies the algebraic structures known as groups. The concept of a group is central to algebra theory and is a powerful formal method for analyzing abstract and physical systems, and has a high importance in solving the Rubik's cube related to mathematics problems. [10]

Definition " A group G consists of a set of objects and a binary operator, *, on those objects satisfying the following four conditions

- The operation * is closed, so for any group elements h and g in G, h $*$ g is also in G.
- The operation * is associative, so for any elements f, g, and h, (f $*$ g) $*$ h = f $*$ (g $*$ h).
- There is an identity element e $\in$ G such that e $*$ g = g $*$ e = g.
- Every element in G has an inverse g -1 relative to the operation * such that g $*$ g -1 = g -1 $*$ g = e. " [8].

In general, the Thistlethwaite's algorithms divides the problem into four independent subproblems by using the following four nested groups Gi:

The functional principle of this algorithm is to start with a cube in Gi, and then move some target cubies to their expected positions by using only moves from nested groups Gi. And repeat this until the Rubik's cube is entirely solved, meaning that it arrives in G4. In general, every stage of the



G0 = <L,R,F,B,U,D>
G1 = <L,R,F,B,U2,D2>
G2 = <L,R,F2,B2,U2,D2>
G3 = <L2,R2,F2,B2,U2,D2>
G4 = <I>

(a) Nested Group Gi

Thislewaits algorithm is based on a table that shows a transition sequence for each element in the current coset space $Gi{+}1\ Gi$ to the next one $I = I{+}1$.

Definition "Given a Group G and a subgroup $H < G$, a coset of H is the set of $Hg = hg : h \in H$; thus, $H < G$ partitions G into cosets. The set of all cosets is written $h\rangle G$."[8]

The coset spaces describe the reduced form of the Rubik's cube, which directly results in the following; the total number of reachable states is reduced by using moves from some subgroups. In other words, this means that the number of permitted moves decreases each time the program moves to a new group/stage. Further explanation is described down below where exact orders for each group are calculated. [8]

Getting from group G0 into group G1

GO represents all the states in the Rubik's cube. This is the number of states reachable from any given state.

Group	Positions
GO	$4.33 * 10^19$

In the first stage, all the edge-orientations are fixed. By definition(section 2.1.3), an edge is correctly oriented if the piece can be brought back to its original position without making use of U and D turns. The edges in the Up or Down-face will flip the edge orientation on every Up and Down turns.

Getting from group G1 into group G2

The first coset space $G1\angle G0$ contains all the cube states. In this state, all edge-orientations are fixed.This stage has a factor of 2^11 that corresponds to the fact that the edges are fixed in this stage. [10]

Group	Positions	Factor
G1	$2.11 * 10^16$	$2^11 = 2048$

What has been achieved in the previous stage is the correct orientation of edge pieces. This results that quarter turns of both F and B are now prohibited. The process of transferring the cube from G1 to G2 can be divided in two parts. The first part is to put the middle edges in the middle layer. Second, the corners are correctly oriented.

Getting from group G2 into group G3

In the second coset space $G2\angle G1$, all the edges are fixed.The 90 degrees turns of UP and Down faces are prohibited in this stage. The second stage has a factor that corresponds to the fact that all the corners are correctly oriented and puts the middle edges in the middle layer.

Group	Positions	Factor
G2	$1.95 * 10^10$	$1,082,565$

In this stage, the corners are fixed into their natural orbits and quarter turns of both F and B faces are not allowed anymore. This makes sure that the corner orientation and the middle edges remain fixed.[7] The third stage puts the corner into their G3- orbits and places all the edge pieces in their correct position, and also fixes the permutation of the corner and edges. G3 - orbits means that the set of positions that are reached by the corner cubies by only using moves from G3.

Getting form group G3 into group G4

In the third coset space $G3\angle G2$, using moves from G3 can solve the cube.

Group	Positions	Factor
G3	$6.63 * 10^5$	29400

In the final stage, the cube can be solved by using only double moves. This makes sure that the edges and the corners stays in their slices. In this stage, the remaining edges and corners will be restored to the correct position, until the cube is entirely solved.

Final stage

The final stage G4 represents the solved state. This means that there are possible states in group G3 that needs to be fixed, until it can transfer to the solved state. In other words, this means that the permutations of each edge slice and edge corner are solved in this stage.

Group	Positions
G4	1

Finally, the Thitslewaites algorithm is relatively easy to implement. This algorithm has been used worldwide, especially in Tomas Rokicki's cube programming contest. The main task of this contest is to write a short program that solves the Rubik's cube. By using the Thitslewaites algorithm combined with computer searches, the program can quickly give very short solutions. However, these solutions might not be optimal and may require more moves than other efficient algorithms, such as the kociemba's algorithm. [10]

2.3.2 Kociemba's algorithm

Kociemba's algorithm is an improved version of the Thistlethwaite's algorithm that works by dividing the problem into only two subproblems. This method applies an advanced implementation of IDA* search to be able to calculate, remove symmetries from the search tree, and solve the cube close to the shortest number of possibilities.[8]

Applying these solutions combined with computer searches might not always give optimal or minimal solutions. In 1997, Richard Korf developed an algorithm which he had optimally solved a random cube. [4]

2.4 Heuristic search

In Artificial intelligence heuristics searches have a major part in the search strategies, since it helps to reduce the number of alternatives from an exponential of a polynomial number. Most of the previous works have been focused on algorithms such as A*, which is a famous Artificial intelligence search technique. This algorithm uses heuristic knowledge to find the shortest path for search problems. However, there are several limitations of this algorithm. [18]

The main disadvantage of this algorithm is that it assumes that there is only one path from the start to the goal state, even though problem spaces may contain multiple paths to each state. The second disadvantage of applying this algorithm is that in order to determine the accuracy of the heuristic search, one needs to determine the cost of the optimal solution for that state. This may require large memory use. In order to solve larger problems that require larger memory usage, one needs to use an admissible search. [19]

2.4.1 IDA* algorithm

The Korf's algorithm was invented by Richard Korf in 1997 and is a graph path search algorithm that finds the optimal solution for solving the Rubik's cube. Richard Korf described the method in his paper as following: *"IDA* is a depth first search algorithm, that looks for increasingly longer solutions in a series of iterations, using a lower-bound heuristic to prune branches* *once a lower bound on their length exceeds the current iterations bound"* . [4] It is a variant of depth first iterative deepening (DFID), which uses depth first searches to find the destination. A normal depth first search ends when it has traversed along all nodes, and there are no more nodes left to expand. But a depth first iterative deepening algorithm stops when a depth limit has been reached, and terminates when the goal is encountered.

This algorithm improves upon depth first iterative deepening search algorithm, by using a heuristic search h(n). It finds the shortest path between a start node and a goal node by using a heuristic function $f(n) = g(n)+H(n)$ applied to each node n of the search space, where g(n) stands for the cost of reaching node n from the initial state, and h(n) stands for the cost of reaching a goal from node n.

This function will stop searching along the current path, when its total cost $g(n) + h(n) > cutoff$. Initially, the value of the cutoff is the heuristic estimate of the cost between a start state to a goal state. Note that the path between the start state to a current state needs to be a part of a depth first search in order to reach the goal state etc. [20]

If the heuristic function is admissible, in other words, if h(n) never overestimates the actual cost from node n to a goal, then the IDA* search algorithm guarantees to find an optimal solution for solving the Rubik's cube. [21]

2.4.2 Manhattan distance

Korf used a variant of Manhattan distance as a heuristic function, where distance is equal to the minimum number of moves to correct position and orientation. For each cube compute the minimum number of moves required to correct position and then sum, these values over all copies, and divide it by eight. Since the algorithm has to be admissible, one needs to divide the sum by eight since every twist moves four corner and four edge cubes.

However, the basic Manhattan distance gives a low value, and therefore the search may be excessively computation intensive if the basic Manhattan distance is utilized directly. Therefore, a modified admissible heuristic search is used. [4]

The modified heuristic search can be evaluated as following;

$$h(n) = max\{h_{corners}(n), h_{e1}(n), h_{e2}(n)\}$$

The $h_{corners}(n)$ calculates the minimum number of moves to fix all the corners in the correct position. $h_{e1}(n)$ calculates the minimum number of moves to fix half of the edge cubes, and $h_{e2}(n)$ fixes the rest of the edges. [4]

2.4.3 An example of IDA* search

Nodes A to M is marked with its f-value, and Nodes A and K are a set of the start and goal nodes respectively. In this figure, only the generated nodes are shown on the picture, whose f-vales are less or equal to the f-value of node K. In the first iteration, the cost bound(cutoff) is the f-value of the start node A, which is 1. In this iteration, nodes A and C are selected for expansion.

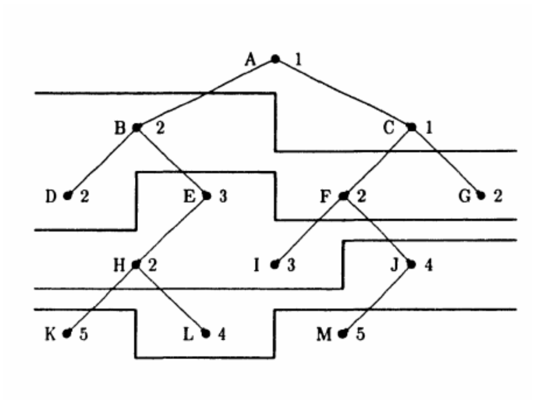


Figure 2.3: An example of IDA* search [31]

In the next iteration, the cost bound(cutoff) is set to be the smallest f-cost of any node that exceeded the cost bound of the initial iteration. Therefore, cost bound is equivalent to two. In this iteration, nodes B, D, F, G, A, C are selected for expansion. On the final iteration, the algorithm terminates once a goal node is reached. In this case, only nodes A, B, D, E, H and K are selected for expansion. [31]

2.4.4 Benefits and limitations of the IDA* algorithm

To find an optimal solution, one needs to use an admissible search algorithm. Algorithms such as A* are impractical on large problems, since it checks for the previously generated states.

IDA* search algorithm uses the depth first algorithm to be able to take advantage of the low space requirements, since the depth first search is linear with tree depth instead of the number of vertices. The algorithm reduces its memory requirements by performing depth first searches bounded by an f-cost value, which limits the length of the expanded path.

The combination of large savings, good performance and automated analysis makes the IDA* search algorithm a useful tool for solving the Rubik's cube, since it is based on iterative deepening that uses less memory than A* algorithm. [22]

Though IDA* has several advantages and is a useful tool for solving the Rubik's cube it also has drawbacks. One of the disadvantages is that this algorithm can generate the same nodes repeatedly like depth first search algorithm, since it retains no path information between iterations. The algorithm will only re-expand a path if it is necessary, and therefore this might be a costly operation, otherwise it won't. [24] Since it does not keep any previously explored path, the time complexity of this algorithm is quite high. Hence, the drawbacks of this algorithm while solving the Rubik's cube can be covered by numerous advantages of this algorithm. [24]

2.5 Complexity

There are 43 quintillion different states that can be reached from any given configuration according to the Saganesque slogan. Therefore the complexity of the cube problem is complex and huge. Another important question is the least number of steps needed to solve a Rubik's cube. This number is known as "Gods number". The god's number has been shown to be equally low as twenty moves. Every position of Rubik's cube can be solved in twenty six moves in the quarter turn metric or in twenty moves or less in half-turn metric. [4]

2.5.1 Time and Space complexity of IDA* algorithm

The time complexity of the IDA* algorithm depends on the quality of the heuristic function, which can determine if the complexity of the algorithm is exponential or linear. The complexity of the algorithm will be exponential, if the heuristic returns zero for every state since this algorithm becomes a brute force search. On the other hand, if the heuristic returns the exact cost to reach a goal, the complexity will be linear.

One needs to consider following that the heuristic search depends on the branching factor of problem space, solution depth of problem instance and the heuristic evaluation function. However, the dominant factor in the running time of IDA* algorithm is the number of node generations that depend on the cost of an optimal solutions, and the number of nodes in the heuristic function, or in the brute force tree. [23]

The time complexity of IDA* is analyzed by Richard Korf under the assumption that the heuristic cost estimate h(n) is consistent. Initially, the heuristic function h(n) is consistent if it is equal or less than $k(n, n') + h(n')$, where n stands for node n and n' stands for any neighbor. The value of k(n,n') is the cost of the edge from node n to any neighbor n'. For example, the heuristic function illustrated in the figure 2.6 is inconsistent.

One can also define this as following: a heuristic function h(n) is consistent if it is equal or less than $k(n, m) + h(m)$, where k(n,m) is the value of the cost of an optimal path from n to m. However, it is difficult to determine for a real heuristics, since obtaining optimal solutions are extremely difficult and sometimes impossible.

The running time of IDA* is proportional to number of node expansions, only if a node can be expanded and its children evaluated in a constant time. For any consistent function, an admissible search algorithm must continue to expand all nodes whose total cost, $f(n) = g(n) + h(n)$ is less than the

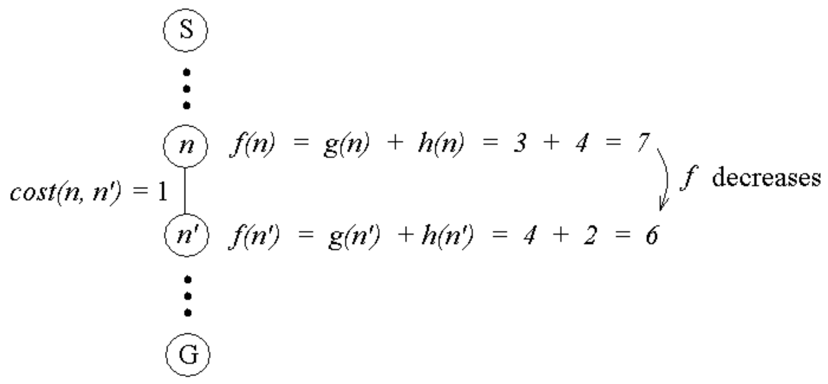

S
n
n'
G
cost(n, n') = 1
f(n) = g(n) + h(n) = 3 + 4 = 7
f(n') = g(n') + h(n') = 4 + 2 = 6
f decreases

Figure 2.4: An inconsistent heuristic function [32]

cost of the optimal solution. Note that $f(n) < c$ is a sufficient condition to expand node n etc.

The cost of the optimal solution will be equal to c in the final iteration of the IDA* search algorithm, where the worst case will expand all nodes n if its total cost $f(n) = g(n) + h(n)$ is less or equal to c. [19]

Lastly, the space complexity of the IDA* algorithm is linear instead of exponential, since the algorithm is performing a depth first search that only requires linear space. Furthermore, the space complexity of IDA* is asymptotically optimal. [7]

Chapter 3

Method

The following section describes the methods used in this research.The first part of this chapter describes the choice of language and the algorithms that will be used for evaluation.The rest of the chapter will describe how to represent a Rubik's cube and the testing environment.

3.1 Programming Language

The programming language of choice is Java, which is a widely used object-oriented, class-based programming language. The main reason for using Java is because it provides object-oriented features,garbage collection(automatic memory management) and extensive functionality. [26]

The downside of using Java compared to languages such as C or C++ is that the performance may be slower and the compiler may take more time than a good C++ compiler. However, the performance downside is not as performance critical as for example games, since slow performance can make a game unplayable. Therefore, using algorithms to find solutions for solving the cube is more acceptable to have delays, slower speed and performance than games. Despite knowing this, the main reason for using Java is because the language is simple to use and handles memory allocation, and has other benefits.[27]

3.2 The Thistletwaite's algorithm

The Thistlewaite's algorithm has four phases as mentioned in the background section. This is illustrated in Figure 3.1.

1. The edges are correctly oriented. This is easy and can be done by considering if an edge piece is bad or good. In this phase, move edge pieces of Up or Down face by avoiding quarter turns of Up or Down, and then cure them by performing these quarter turns. For further explanation, please read section 2.1.3.

2. This phase can be divided into two parts. First, put the middle edges in the middle layer. Second, orientation of the corners are correctly oriented. The first part is easy and can be done in the same way as the previous phase. For example, the edge pieces FU, FD, BU, BD are brought into their slices. Note that each corner piece has a L-facet or R- facet, and therefore in this stage each of these facets will lie on either the L or the R face.

3. This is the trickiest stage. The corners are placed in their respectively tetrads, the edges into their appropriate slices and the parity of the edge and corner permutation is even. To be able to place the corners into their orbits, one needs to calculate which coset of form G3aB of the permutation of corners lies in, where a and b is 1.

4. Solves the cube by fixing both the corner and edge permutation. In this stage, only 180 turns are used.

The Breadth first search

The search that is used in this algorithm is the breadth first search, which is shown in figure 3.2. The search algorithm will continue as long as there are enough moves to reach a solution. To be able to overcome the drawbacks of the exponential growth that requires for example too much memory, a table is created for each stage. This table represents how many more moves are required and how far the program is from completing the sub-problems.

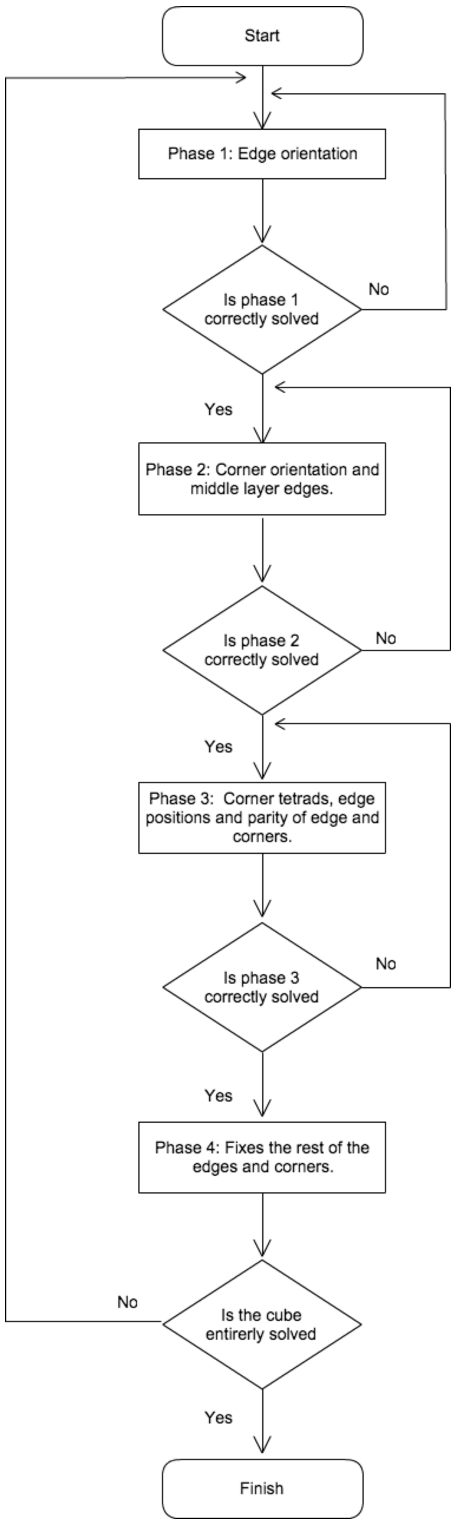

Start
Phase 1: Edge orientation
Is phase 1 correctly solved
No
Yes
Phase 2: Corner orientation and middle layer edges.
Is phase 2 correctly solved
No
Yes
Phase 3: Corner tetrads, edge positions and parity of edge and corners.
Is phase 3 correctly solved
No
Yes
Phase 4: Fixes the rest of the edges and corners.
Is the cube entirerly solved
No
Yes
Finish

Figure 3.1: Thistlewaite's algorithm

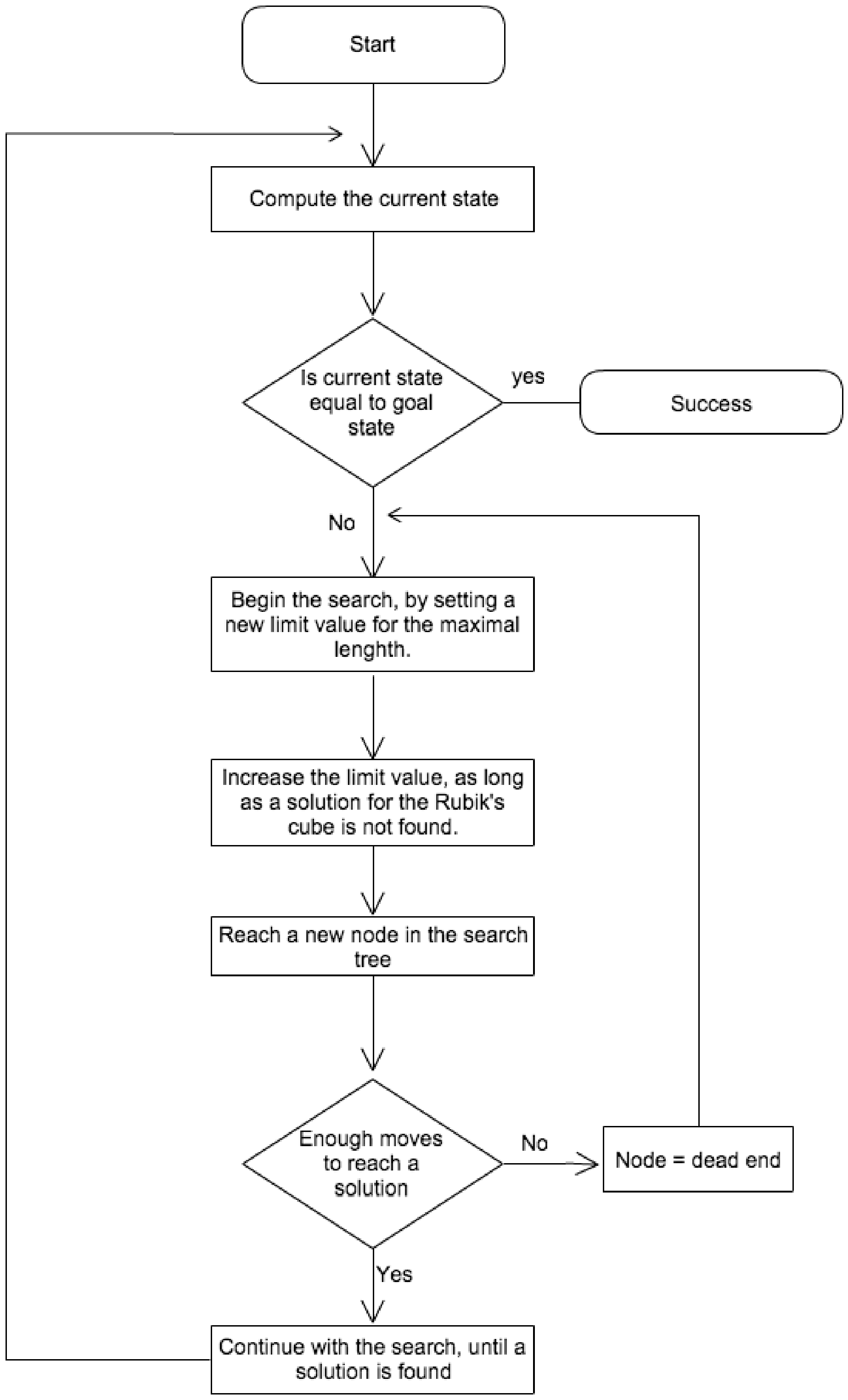

Start
Compute the current state
Is current state equal to goal state
yes
Success
No
Begin the search, by setting a new limit value for the maximal lenghth.
Increase the limit value, as long as a solution for the Rubik's cube is not found.
Reach a new node in the search tree
Enough moves to reach a solution
No
Node = dead end
Yes
Continue with the search, until a solution is found

Figure 3.2: Breadth first search

3.3 The IDA* algorithm design

3.3.1 Pseudocode

Algorithm 1 The IDA* algorithm

```
1: procedure IDAStar* ()
2:     cutoff ← f(s) = h(n)
3:     while goal node is not found or no new nodes exist do
4: DFS search to explore nodes with f-values within cutoff
5:         if goal not found  then
6:             extend cutoff to next unexpanded value if there exists one
```

[33]

3.3.2 Flowchart

The IDA* algorithm can be described as following:

1. At each iteration, perform a depth-first search that keeps track of the cost evaluation $f = g + h$ of each node generated.

2. If a node is generated whose total cost $(g + h)$ exceeds the given thresholds, its path is cut off.

3.This threshold starts at the cost of the initial state. At each iteration, the cost threshold increases in each iteration. The threshold that will be used for the next iteration will be the minimum cost of all values that exceeded the current threshold.

4.The algorithm terminates when the total cost does not exceed the current threshold.[28]

The algorithm is illustrated in Figure 3.3.

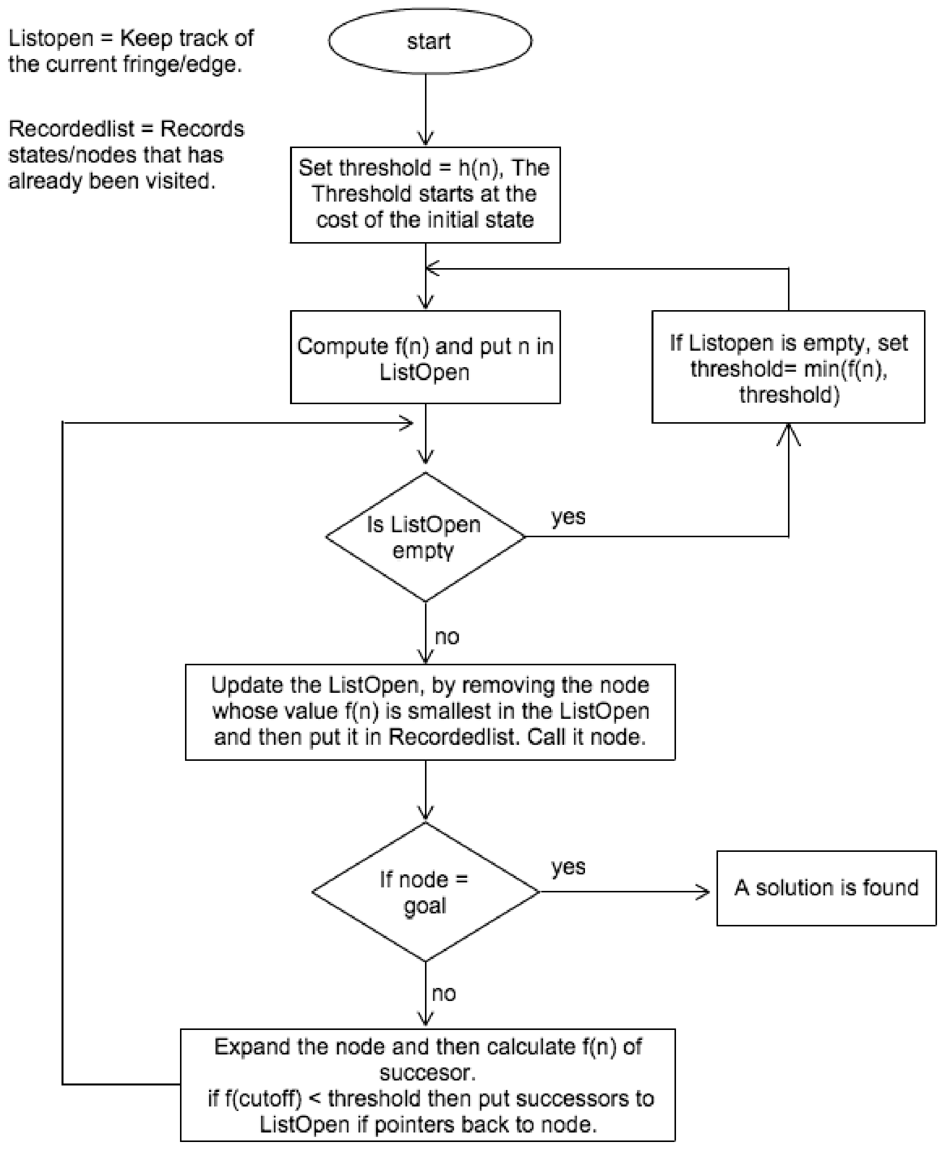

Listopen = Keep track of the current fringe/edge.
Recordedlist = Records states/nodes that has already been visited.
start
Set threshold = h(n), The Threshold starts at the cost of the initial state
Compute f(n) and put n in ListOpen
If Listopen is empty, set threshold= min(f(n), threshold)
Is ListOpen empty
yes
no
Update the ListOpen, by removing the node whose value f(n) is smallest in the ListOpen and then put it in Recordedlist. Call it node.
If node = goal
yes
A solution is found
no
Expand the node and then calculate f(n) of succesor.
if f(cutoff) < threshold then put successors to ListOpen if pointers back to node.

Figure 3.3: The IDA* algorithm

3.3.3 Heuristic search

1. The Rubik's cube is only restricted to the corners. The heuristic search computes the minimum number of moves to fix all the corners.

2. The Rubik's cube is only restricted to six edges. The heuristic search computes the minimum number of moves to fix the first set of edges.

3. Lastly, the heuristic search computes the minimum number of moves to fix the rest of the edges.

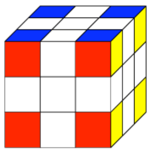


Figure 3.4: Corners

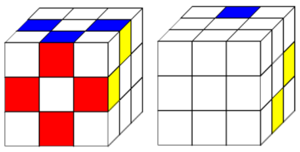


Figure 3.5: The edge set 1 and 2

3.4 Representing the moves

There are three different ways to represent the moves, which can be further divided into two sub moves for each movement. For this purpose, a move is considered to be a 90, 180 or 270 degree rotation relative to the rest of the cube. A rotation for example can be represented as following;

- A single letter L : Rotate Left face 90 degrees clockwise.

- A Letter followed by atmosphere L' : Rotate the left face 90 degrees counter clockwise.

- A letter with the number 2 after L2: Rotate the left face 180 degrees.

For example LU'D2 can be describe as following steps:

1. Rotate left face 90 degrees clockwise.

2. Rotate upper face(top face) counterclockwise

3. Rotate the Down(bottom) face

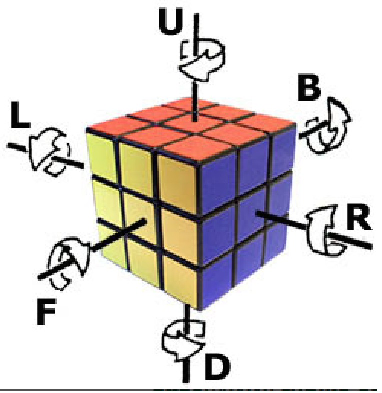
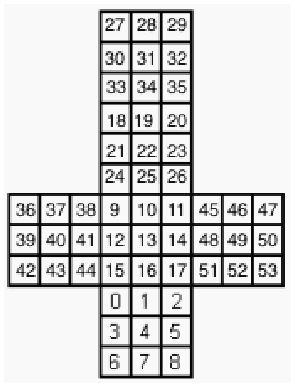


Figure 3.6: Represents the moves [34]

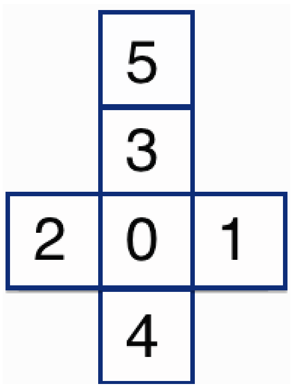
3.5 The Rubik's cube

The Rubik's cube is seen as a collection of six faces, which is made of nine square with different colors. The colors are seen as integer values from zero to five, as listed below:



- 0 : Up face (Top face)
- 5 : Down face(Bottom face)
- 3 : Back face
- 2 : Left face
- 1 : Right face
- 4 : Front face

The Rubik's is represented as following: An array of integers of 54 elements.



- Nr 4 Front face: rubikcube[0] - rubikcube[8]
- Nr 0 Up face : rubikcube[9] - rubikcube[17]
- Nr 3 Back face : rubikcube[18]- rubikcube[26]
- Nr 5 Down face :rubikcube[27] - rubikscube[35]
- Nr 2 Left face : rubikscube[36]- rubikscube[44]
- Nr 1 Right face : rubikscube[45] - rubikscube[53]

3.6 Testing environment

3.6.1 Hardware

Type	Name
Operating system	OS X Version 10.9.3
Memory	8 GB 1600 MHz DDR3
Graphics	Intel HD Graphics 4000
Storage	750 GB SATA Disk
Java	version 7

3.6.2 Software

Operating system of choice is OS X 10.9.3. All tests are executed from the bash shell Terminal version 2.4(326). The motivation for using this operating system is because it is an Unix-Based OS with a great user interface.

3.6.3 Measuring Time

Measuring time in Java is easy and may be evaluated in many different ways; wall clock time or CPU time. The time spent executing the code was measured in wall clock time by using the Java class Timer. The time was measured more than once, where the best results were considered as the average time from all these elements.

The default timer has different precisions depending on the platform. Since OS X/UNIX is used, the wall clock time may be easily affected by other processes being run on the same machine. Therefore, to minimize interference from other programs, one should terminate and shut down as many external programs as possible while running these tests.[29]

3.6.4 Measuring Performance

There are various ways to measure the performance of an algorithm, where the following categories matters in this research: completeness, optimal, time complexity and space complexity. An algorithm that is complete will always guarantee to find a solution if one. The time complexity of an algorithm quantifies the amount of time it is taken for the algorithm to complete as an arbitrary number/length of the string. The space complexity of an algorithm quantifies the amount of space taken by the algorithm in order to successfully complete the algorithm.

Even though time may be a significant factor, one needs to also consider that search may operate under certain memory constraints that make conservation of space take priority.Therefore time, completeness, time complexity and space will also determine which of these algorithm is most efficient while solving the Rubik's cube. [30]

Chapter 4

Results

This section will provide the results from the IDA* algorithm and Thistlet-waite's algorithm. These results are displayed separately in a tabular and graphical form. At last a comparison between both of these algorithms are presented.

4.1 The IDA* algorithm

Computation of the heuristic function is time consuming. It can take about twenty to forty minutes to compute the heuristic search on an average lap-top, since the application uses a large amount of memory. The table needs to be stored in RAM for the IDA* algorithm.

Table 5.1 gives an overview of the average runtime for each depth based on a movement, nodes/positions per the depth, the memory requirements and the cost to travel from node n to a goal.

Table 5.2 represents the total memory requirements and the number of moves the algorithm requires. Figure 4.1 shows the average runtime(s) for each depth.

In this research paper, the Korf's algorithm could not solve the Rubik' cube entirely. Therefore results from the cubecontest is presented as an example of how fast the Rubik's cube can be solved by using the IDA* algo-rithm. Antony Boucher solved the cube with four successive IDA* searches in 22 milliseconds per solutions, averaged 29.49 moves. However, according to the author Richard Korf's, the Rubik's cube can be solved in less than twenty moves. For further explanation, see the background section.

Table 5.1: Total runtime, Nodes/position, Path cost and Memory used for each depth

Depth	Average time(s)	Nodes/Positions	Path cost	Memory Used(MB)
1	0.001342	18	1.0	1
2	0.001729	243	2.0	2
3	0.002168	3,240	3.0	2
4	0.009177	43,254	4.0	3
5	0.044215	577,698 911,250	5.0	5
6	0.05448	7,706,988	6.0	9
7	0.11577	102,876,480	7.0	35
8	1.417755	1,373,243,544	7.0	39
9	14.261387	18,330,699,168	9.0	55
10	39.73267	244,686,773,808	10.0	149

Table 5.2: Total runtime for depths between 1- 10, memory requirements and total movements

Depth	Total memory used	Total movements
1-10	291 MB	10

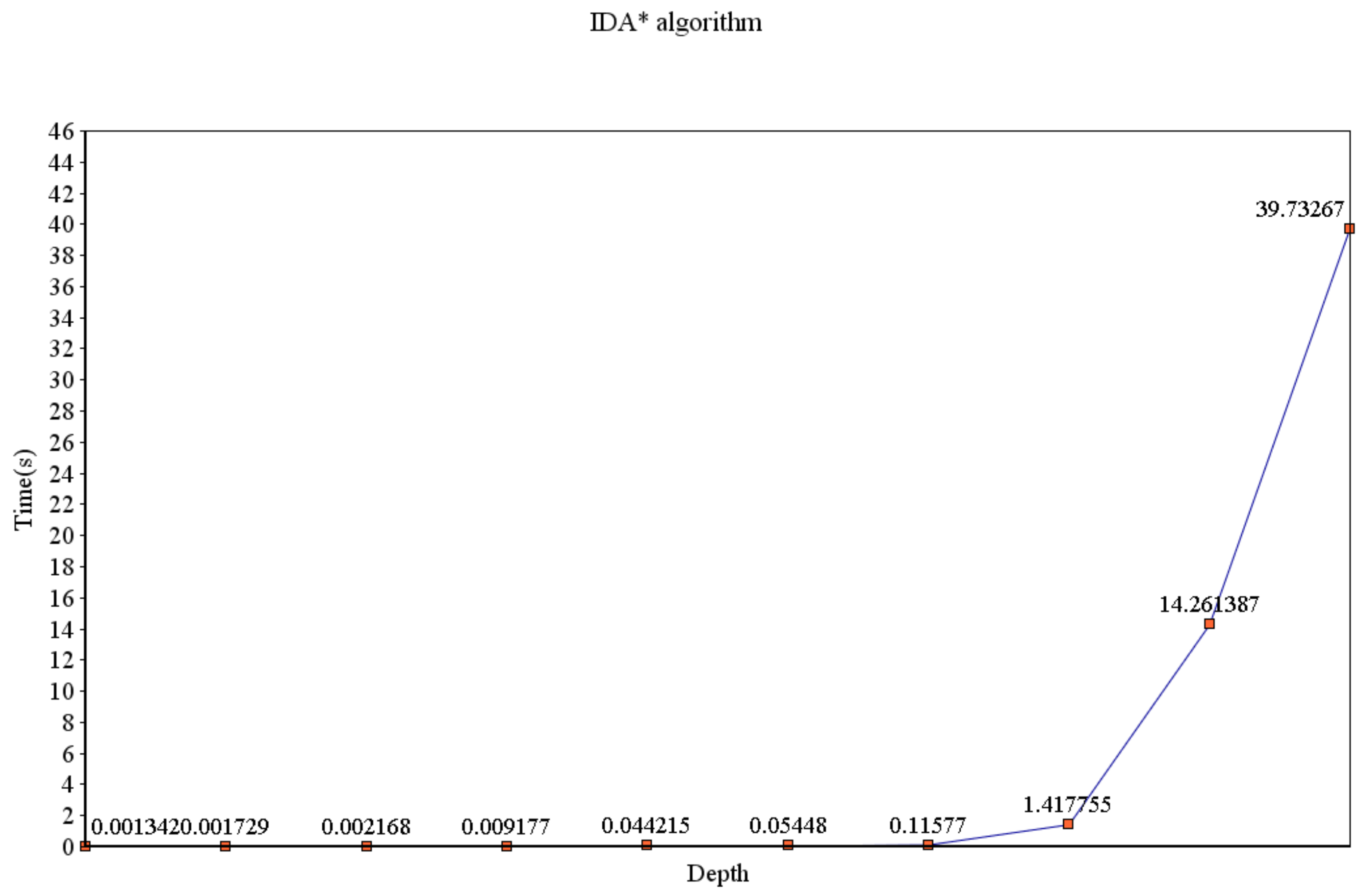


Figure 4.1: Total runtime for depth 1 - 10.

4.2 The thistletwaite's algorithm

Figure 5.3 shows the total runtime for the Thistlewaite's algorithm, the number of moves the algorithm requires and the average time per movement. Figure 5.4 shows the total movement each subgroup requires while solving the Rubik's cube.

Table 5.3: Total runtime for the thistletwaite's algorithm.

Test	Total time taken(s)	Moves	Time per movement(s)
1	343.56	52.37	6.56
2	363.36	52.52	6.91
3	387.98	52.77	7.35
4	315.66	52.21	6.04

Table 5.4: Total movements for each subgroup

Subgroup	Moves
G0	7
G1	13
G2	15
G3	17

4.3 A comparison of the IDA* algorithm and the Thistletwaite's algorithm

Figure 4.2 shows the total runtime per movement for both of these algorithms, based on the best time. This figure gives an overview of which algorithm is faster based on the smallest tree(depth up to 10).

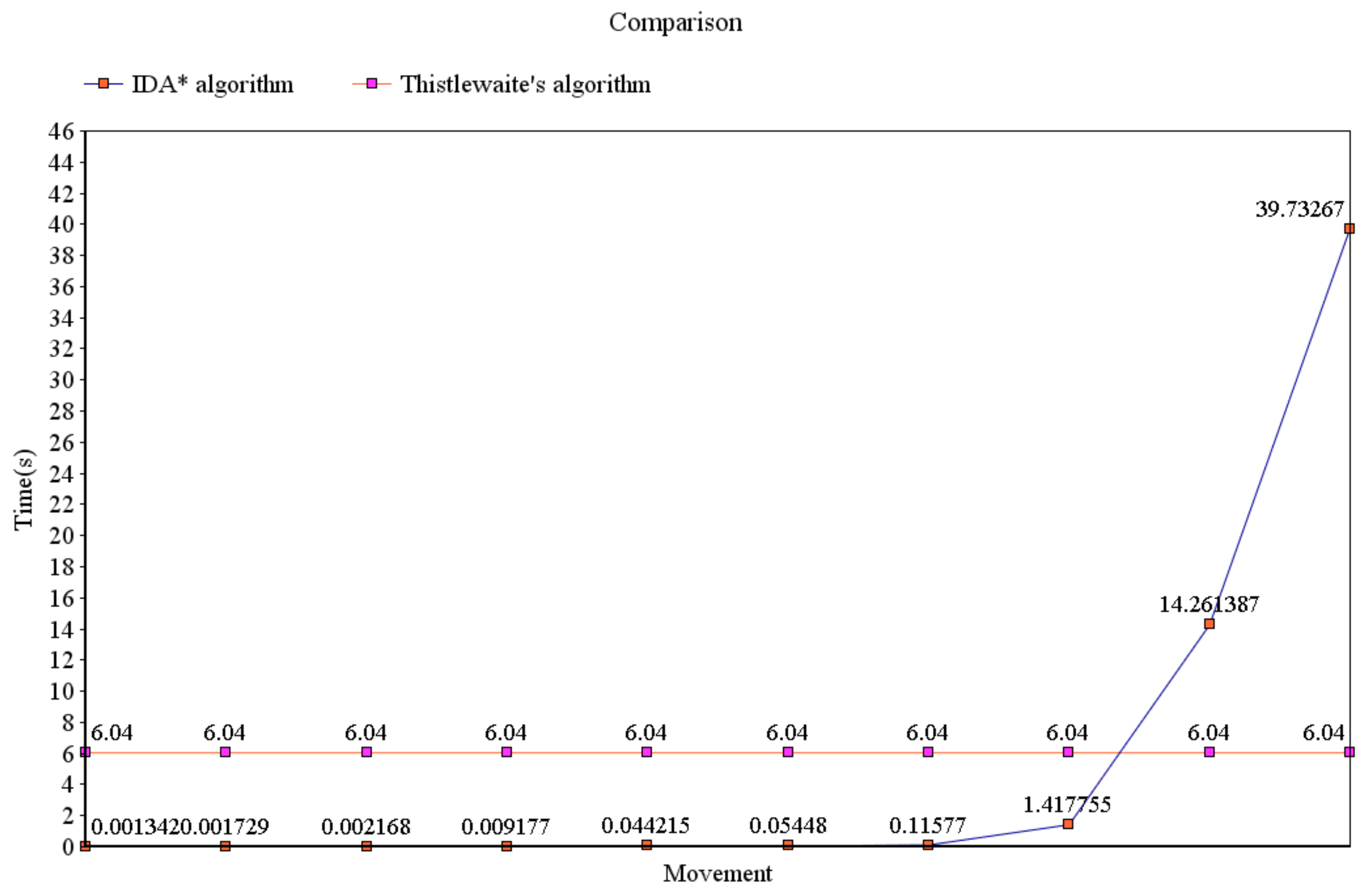


Figure 4.2: Comparison of both algorithms.

Chapter 5

Discussion

In this section the analysis of the results, recommendation and discussion about the limitations and further studies will be presented. The limitations are discussed based on restrictions on time and data of the IDA* algorithm.

5.1 Comparison

The results for both of the algorithms in figure 4.2 show that the IDA* algorithm is considerably faster than the thistletwaites algorithm based on time per movement. Thistlewaite's algorithm requires more than 52 moves and if it takes approximately 6.04 seconds per movement, then it will take 315 seconds to solve the Rubik's cube. However, when it comes to depth further than 9 then the Thistlewaite's algorithm seems to be faster, but one needs to consider that the Korf's algorithm can solve the Rubik's cube in less than twenty moves which makes this algorithm faster anyway. The IDA* algorithm gives an improvement of over five seconds before depth 9.

Furthermore, this raises the question of how much faster the IDA* algorithm is than the Thistlewaite's algorithm. In this case, only results were found by the IDA* algorithm up to depth 10 and using the IDA* algorithm gives an improvement of over seconds as seen on the table.(Less than the 9th movement). However,further study with more focus on the IDA* algorithm is therefore suggested. Under the circumstances,the answer to this question theoretically could be following; the speed of the algorithm depends on the function of the accuracy of the heuristic function. In this case, a better heuristic search will be the maximum of the sum of the corner cubes and edge cubes divided by four, rather than using the same variant as the Korf used in his research.

The reason why IDA* is a useful tool is because it is based on the heuristic evaluation function, and is therefore able to find solutions by examining a much smaller number of nodes than the Thistlewaite's algorithm would be. As a consequence, the IDA* runs much faster than the Thistlewaite's algorithm and requires fewer moves than the thistlewaite's algorithm.

Additionally, one needs to consider that the thistlewaite's algorithm in this research paper used the breadth first search, which is an exponential time algorithm. Since the IDA* requires only linear space: O(d) if the heuristic returns the exact cost to reach a goal, this makes the IDA* algorithm optimal in terms of solution cost, time and space over the Thitslewaite's algorithm using the breadth first search. The combination of large savings and good performance makes the IDA* search algorithm a useful tool for solving the Rubik's cube, since it is based on iterative deepening that uses less memory than A* algorithm. However, this algorithm is difficult and complicated to implement compared to the Thistlewaite's algorithm. Even though, the Thistlewaite's algorithm requires more moves than the IDA* algorithm and may require more time, the Thistlewaite's algorithm is easier to implement. Both algorithms are complete, since they always guarantee to find a solution if one exists.

5.2 Recommendation

In consideration of the difficulty of the Korf's algorithm, one could implement the Thistlewaite's algorithm with the IDA* algorithm to get a better result. The Thistlewaite's algorithm using breadth first search takes more time to complete the search, due to the weakness of its exponential time and space usage.Linear space algorithms are often more efficient to use while solving larger problems.Therefore, the Korf's algorithm has minimal space requirements and is space efficient, which makes the Korf's algorithm more efficient that the Thistlewaite's algorithm using the breadth first search. However, these limitations of the Thistlewaite's algorithm using breadth first search can be overcome by using the IDA* algorithm, to be able to take advantage of the low space requirements of the depth first search. There exist also algorithms that are more efficient than the Thistlewaite's algorithm such as the Kociemba's algorithm, which solve the cube in approximately half of the moves. However, the Thistlewaite's algorithm does not guarantee to find optimal solutions or minimal solutions, as the IDA* algorithm. This algorithm always guarantees to find optimal solutions if the heuristic function is admissible.

Another interesting finding from the experiments conducted is how inefficient brute force search algorithms are compared to the IDA* star algorithm. A possible explanation for the brute force search such as the depth first search is that the algorithm is impractical when dealing with larger problems such as solving the Rubik's cube, and therefore might not ensure that any solutions will be found. Therefore computational approaches for solving the Rubik's cube, such as the Thistletwaie's algorithm, Kociemba's algorithm or the IDA* search are more appropriate when dealing with larger problems.

5.3 Difficulty

The Rubik's cube search space has a branching factor of 18, and even though the program can expand 10 million positions per second, it will take approximately $4.3 * 10^12$ seconds to reach all possible positions. The current program is in a position to check all moves at a depth of ten within a reasonable time.However, regardless of the fact that the IDA* algorithm is capable of solving larger problems and is very beneficial while solving the Rubik's cube, it is difficult to make this work on an average computer due to stack overflow or out-of-memory errors. One reason might be that there exists an error in updating the f-limit cutoff or in the use of recursion in the program. The other reason might be that it does not work on an average computer. Furthermore, this can be solved by using the Thistlewaite's with the IDA* search algorithm.

5.4 Source of error

The time was measured more than once, where the best results is seen as the average time from all these elements to minimize the error. Even though the time was measured more than once, the wall clock time may have been affected by other processes being run on the same machine. Even though time is an important factor, one needs to also consider that search may operate under certain memory constraints that make conservation of space take priority, as mentioned before

5.5 Limitations of the IDA* algorithm

Computation of the heuristic function as seen in the result section is time consuming. It can take about twenty to forty minutes to compute the heuristic search on an average laptop, since the application uses a large amount of memory. The IDA* algorithm is very speedy for cubes up to 10 moves as seen on the table 5.1, which gives a good overview of the average runtime. Beyond that the heuristic function seems to break down, and may take hours to solve if the algorithm returns any solution for depths further than 10.

5.6 Further Studies

Future research in this field could investigate which algorithm is more efficient and give more valid results that can prove which algorithm is better. Currently limited results restrict the quality of the IDA* algorithm.

Chapter 6

Conclusions

Exponential algorithms such as A*, breadth first search are impractical on larger problems. These algorithms have some limitations, which are overcome by an algorithm called IDA* search. The IDA* algorithm is a useful tool for solving the Rubik's cube, but is difficult to implement. On the other hand, the Thitslewaite's algorithm with breadth first search is easier to implement than the IDA* algorithm,and works correctly without complication. Therefore, one needs to consider that both of these algorithms have advantages and disadvantages.

The results show that the Korf's algorithm is more efficient while solving the Rubik's cube based on the smallest tree. In order to achieve more accurate and valid result there is necessary to have further studies. Since the Korf's algorithm could not solve the Rubik's cube entirely, the assertion built on the different literature studies, or authors are used as an argument to prove that the Korf's algorithm is more efficient theoretically. Furthermore, tthe Korf's algorithm is more efficient based on performance measuring(Completeness, time complexity and space complexity).

Finally, this research paper attempts to answer which algorithm is more appropriate for solving the Rubik's cube up to 10th movement. In this case, the IDA* algorithm is more efficient but more difficult to implement, and therefore it is recommended to use the Thitslewaite's algorithm with IDA* search algorithm.

Chapter 7

Bibliography

[1] Julian Fleron, Philip Hotchkiss, Volker Ecke, Christine von Renesse, 2010. Rubik's cube. [pdf] Available at:http://www.westfield.ma.edu/ecke/110-games/Rubik-web.pdf [Accessed 1 April 2015].

[2] Illumin, 2010. A simple complexity. [Online] Available at:https://illumin.usc.edu/113/a-simple-complexity/ [Accessed 25 April 2015].

[3] SP.268, 2009. The Mathematics of the Rubik's Cube [PDF] Available at:http://web.mit.edu/sp.268/ www/rubik.pdf [Accessed 25 April 2015].

[4] Ricard Korf, 1997. Korf's Optimal Solution to Rubik's Cube Puzzle and Pattern Databases. [PDF] Available at:www-compsci.swan.ac.uk/csphil/CS335/appendixB.pdf [Accessed 15 May 2015].

[5] Haym Hirsh, 1999. Sliding-tile puzzles and Rubik's Cube in AI research. [PDF] Available at:https://skatgame.net/mburo/ps/IEEE.pdf [Accessed 25 April 2015].

[6] Aitopics, 2015. Rubik's cube. [Online] Available at:http://aitopics.org/topic/rubiks-cube [Accessed 25 April 2015].

[7] Morwen B. Thistlethwaite, 1981. Thistlethwaite's 52-move algorithm. [online] Available at:http://www.jaapsch.net/puzzles/thistle.htm. [Accessed 24 May 2015].

[8] Nail El-Sourani, Sascha Hauke, and Markus Borschbach. An Evolutionary Approach for Solving the Rubik's Cube Incorporating Exact Methods. [PDF] Available at: ¡http://www.genetic-programming.org/hc2010/7-Borschbach/Borschbach-Evo-App-2010-Paper.pdf¿ [Accessed 25 April 2015].

[9] S.G.Shirinivas, S.Vetrivel Dr. N.M.Elango. Applications of graph theory in computer science an overview. [PDF] Available at: ¡http://www.cs.xu.edu/csci390/12s/IJEST10-02-09-124.pdf¿ [Accessed 25 April 2015].

[10] Joren Heit, 2011. Building and Solving Rubik's cube in Mathworks Matlab.[PDF] [Accessed 15 May 2015].

[11] Shimon Even, 1979. Graphs Algorithms, 2nd edition. [Accessed 25 April 2015].

[12] Richard Korf. Depth-First Iterative-Deepening: An Optimal Admissible Tree Search. [PDF] [Accessed 25 April 2015].

[13] McGill Computer School.Lecture29Handouts. [Online] [Accessed 25 April 2015].

[14]S. Russell and P. Norvig, 1995. Artificial Intelligence: A Modern Approach. Prentice Hall, Upper Saddle River, N.J.

[15]Neil Burch, Robert C. Holte. Automatic Move Pruning in General Single-Player Games. [pdf]. Available at: ¡http://webdocs.cs.ualberta.ca/holte/Publications/SoCS11MovePruning.pdf¿ [Accessed 25 April 2015].

[16] Larry A. Taylor and Richard E. Korf. Pruning Duplicate Nodes in Depth First Search. [pdf] Available at: ¡https://www.aaai.org/Papers/AAAI/1993/AAAI93-113.pdf¿ [Accessed 25 April 2015].

[17] Richard Korf, 2008. Linear-Time Disk-Based Implicit Graph Search [pdf] Available at: http://www.cs.helsinki.fi/u/bmmalone/heuristic-search-fall-2013/Korf2008.pdf¿[Accessed 25 April 2015].

[18] Sven Koenig Maxim Likhachev Yaxin Liu David Furcy. Incremental Heuristic Search in Artificial Intelligence. [pdf] Available at: ¡http://idm-lab.org/bib/abstracts/papers/aimag04a.pdf¿[Accessed 25 April 2015].

[19] Richard Korf, 2001. Time complexity of iterative-deepening-A. [pdf] Available at: ¡http://www.sciencedirect.com/science/ article/pii/S0004370201000947¿[Accessed 25 April 2015].

[20] John F. Dillenburg and Peter C. Nelson. Improving the Efficiency of Depth-First Search by Cycle Elimination. [pdf] Available at: ¡http://tigger.uic.edu/ dillenbu/papers/cycle2.pdf¿ [Accessed 25 April 2015].

[21] Richard Korf. Depth-First Iterative-Deepening: An Optimal Admissible Tree Search. [PDF] [Accessed 25 April 2015].

[22]Neil Burch, Robert C. Holte. Automatic Move Pruning in General Single-Player Games. [pdf]. Available at: ¡http://webdocs.cs.ualberta.ca/holte/Publications/SoCS11MovePruning.pdf¿ [Accessed 25 April 2015].

[23]Richard E. Korf, Michael Reid, 1998. Complexity Analysis of Admissible Heuristic Search. [pdf]. Available at: ¡https://www.aaai.org/Papers/AAAI/1998/AAAI98-043.pdf¿[Accessed 25 April 2015].

[24]Mikhail Simin – Arjang Fahim, 2011. Efficient memory-bounded search methods[online] Available at: ¡http://www.cse.sc.edu/ mgv/csce580f11 /gradPres/580f11SiminFahimIDAStar.pdf. [Accessed 25 April 2015]

[25] Setiawan, E.I, Santoso, J. ; Maryati, I, 2013. Locating nearest public facility using IDA$*$ algorithm. [pdf][Accessed 25 April 2015].

[27] The saylor Foundation. Advantages and Disadvantages of Object-Oriented Programming. [pdf] Available at: ¡http://www.saylor.org/site/wp-content/uploads/2013/02/CS101-2.1.2-AdvantagesDisadvantagesOfOOP -FINAL.pdf¿. [Accessed 25 April 2015].

[28]Ram Meshulam, 2004. Artificial Intelligence Lesson 3. [online] Available at: ¡http://u.cs.biu.ac.il/ haimga/Teaching/AI/lessons/lesson3.pdf. [Accessed 25 April 2015]

[29]Peter Sestoft, 2014. Microbenchmarks in Java and C. [pdf]Available at: ¡https://www.itu.dk/people/sestoft/papers/benchmarking.pdf¿. [Accessed 25 April 2015]

[30] AvM. Tim Jones. Artificial Intelligence A systems approach.

[31] Brain Patrik, 1991. An analysis of iterative-Deepening- A*. [PDF][Accessed 20 May 2015].

[32] Heuristic Search, 2015. [online]. Available at: http://science.slc.edu/jmarshall/courses/2005/fall/cs151/lectures/heuristic-search/. [Accessed 25 May 2015].

[33] Deepak Khemani, 2014. McGraw-Hill Education.

[34] Rodion Gorkovenko, 2013 - 2014. Rubik's cube. [online] [Accessed 25 May 2015].

[35] Ruwix, 2015. Advanced Rubik's cube notation. [online] Available at: http://ruwix.com/the-rubiks-cube/notation/advanced/ [Accessed 26 may 2015].

[36] Puzzling beta, 2015. What is the meaning of a "tetrad twist" in Thistlethwaite's algorithm. [online]. Available at: http://puzzling.
stackexchange.com/questions/5402/what-is-the-meaning-of-a-tetrad-twist-in-thistlethwaites-algorithm. [Accessed 26 may 2015].