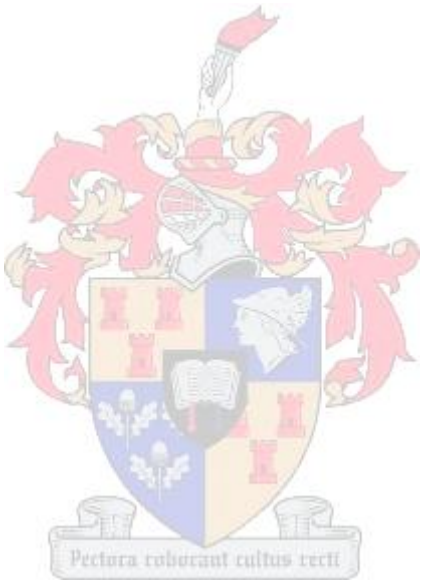
In Pursuit of God Numbers for the Puzzle *Wrapslide*

Ghiete van Zyl



Year project presented in partial fulfilment of the requirements for the degree of
Bachelor of (Industrial) Engineering
in the Faculty of Engineering at Stellenbosch University

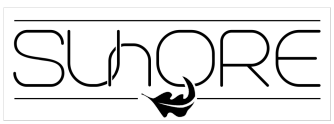
Supervisor: Prof JH van Vuuren
Co-supervisor: Dr AP Burger

November 2016

Declaration

By submitting this project electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Date: November 1, 2016



Abstract

The toroidal slide-puzzle *Wrapslide* is reminiscent of Rubik's celebrated cube and is available for (free) download on all smart devices. It consists of a 4×4, 6×6, or 8×8 grid of tiles in two, three or four different colours (depending on the player's desired level of difficulty). The objective of the game is to group tiles of the same colour into the four quadrants in as few moves (and little time) as possible, one colour per quadrant, by executing a series of finger sliding moves on the smart device. Each move involves horizontal toroidal translation of either the top or bottom half of the grid, or vertical toroidal translation of either the left or right half of the grid.

An enumeration approach is followed in this project to determine some of the the so-called God numbers for *Wrapslide*. The God number of the puzzle is the minimum number of moves required to solve any of its scrambled states. It took thirty six years and a massive parallel computing effort by Google to determine the God number for Rubik's cube as twenty. This was done by showing that some scrambled states of Rubik's cube require a minimum of twenty moves to solve, but that there exists no scrambled state of the cube requiring twenty one or more moves to solve. The same type of analysis approach is followed in this project for *Wrapslide*. A heuristic solution procedure is also designed (and implemented on a computer) for solving a *Wrapslide* mixed state in an optimal or hopefully near-optimal number of moves. This heuristic may contribute to future estimations of the values of or conjectures on upper bounds for as yet unknown God numbers.

Uittreksel

Die torus-gebaseerde spel *Wrapslide* herinner sterk aan Rubik se gevierde kubus en is (kosteloos) aflaaibaar op enige slimfoon of tablet. Dit bestaan uit 'n 4×4, 6×6 of 8×8 skikking van teëls in twee, drie of vier verskillende kleure (afhangend van die moeilikheidsgraad deur die speler vereis). Die doel van die spel is om teëls van dieselfde kleur in die kleinste aantal skuiwe (en so gou as moontlik) in die vier kwadrante van die skikking saam te groepeer, een kleur per kwadrant, deur 'n reeks vingergly-aksies op die slimfoon of tablet uit te voer. Elke skuif behels horisontale torus-gebaseerde translasie van die boonste of onderste helfte van die skikking, of vertikale torus-gebaseerde translasie van die linker of regter helfte van die skikking.

'n Enumerasiebenadering word in hierdie projek gevolg om sommige van die sogenaamde God-getalle vir *Wrapslide* te bepaal. Die God-getal van die spel is die kleinste aantal skuiwe waarmee *enige* van die geskommelde toestande daarvan opgelos kan word. Dit het ses en dertig jaar en 'n massiewe parallelle berekeningspoging deur Google vereis om die God-getal vir Rubik se kubus as twintig vas te pen. Hierdie waarde is bepaal deur aan te toon dat daar sommige geskommelde toestande van Rubik se kubus bestaan wat minstens twintig skuiwe vereis om op te los, maar dat daar geen geskommelde toestand bestaan waarvoor die oplossing een en twintig of meer skuiwe vereis nie. Dieselfde soort analitiese benadering word in hierdie projek vir *Wrapslide* gevolg. 'n Heuristiese oplossingsprosedure word ook ontwerp (en op 'n rekenaar geïmplementeer) waarvolgens enige geskommelde *Wrapslide* toestand in 'n optimale of hopelik byna optimale aantal skuiwe opgelos kan word. Hierdie heuristiek mag in die toekoms bydra tot afskattings van die waardes van of die formulering van vermoedens oor bogrense vir huidig onbekende God-getalle.

ECSA Exit Level Outcomes Reference

Outcome	Reference	
	Section	**Page**
1. Problem solving: Demonstrate competence to identify, assess, formulate and solve convergent and divergent engineering problems creatively and innovatively.	*All*	*All*
5. Engineering methods, skills and tools, including information technology: Demonstrate competence to use appropriate engineering methods, skills and tools, including those based on information technology.	*3,4 & 5*	*19–51*
6. Professional and technical communication: Demonstrate competence to communicate effectively, both orally and in writing, with engineering audiences and the community at large.	*All*	*All*
9. Independent learning ability: Demonstrate competence to engage in independent learning through well developed learning skills.	*2,3,4,5 & 6*	*9–56*
10. Engineering professionalism: Demonstrate critical awareness of the need to act professionally and ethically and to exercise judgement and take responsibility within own limits of competence.	*All*	*All*

Acknowledgements

The author wishes to acknowledge the following people and institutions for their various contributions towards the completion of this work:

- My supervisor, Prof JH van Vuuren, to whom I owe my deepest gratitude for his endless support, patience and immense effort. I could not have asked for a better supervisor and mentor.
- Dr AP Burger, for his generosity with his time and his valuable insights.
- My family, for their sympathetic ear, endless support and unwavering belief in me.
- Finally, my friends, for four years of memories I will cherish forever.

Table of Contents

Abstract **iii**

Uittreksel **v**

ECSA Exit Level Outcomes Reference **vii**

Acknowledgements **ix**

List of Reserved Symbols **xiii**

List of Acronyms **xv**

List of Figures **xvii**

List of Tables **xix**

1 Introduction **1**

1.1 Background 1

1.2 Problem Statement 4

1.3 Project Objectives 4

1.4 Research Methodology 6

1.5 Scope 7

1.6 Timeline 8

1.7 Project Organisation 8

2 Literature Review **9**

2.1 Symmetry, Isometries and Equivalence Classes 9

2.2 Tilings of the Plane and the Torus 12

2.3 Tree Search Protocols 12

2.4 Prior Investigations Related to the Puzzle Wrapslide 16

2.5 Chapter Summary 17

3 State Enumeration Trees of the Puzzle *Wrapslide* 19

3.1 Encoding of *Wrapslide* States 19

3.2 The Equivalence Class of a Given *Wrapslide* State 22

3.3 Generation of a Branch and Bound State Enumeration Tree 25

3.4 Selection of Class Representatives 28

3.5 God Numbers for Selected *Wrapslide* Grids 31

3.6 Chapter Summary 33

4 A Heuristic Solution Procedure for the Puzzle *Wrapslide* 35

4.1 Assignment of Objective Function Values to *Wrapslide* States 35

4.2 Proposed Heuristic for Solving *Wrapslide* States 36

4.3 A Worked Example 41

4.4 Chapter Summary 46

5 Implementation of the Heuristic 47

5.1 User Interface Design 47

5.2 Results Obtained by the Heuristic 50

5.3 Chapter Summary 51

6 Summary and Conclusions 53

6.1 Project Summary 53

6.2 Suggestions for Future Work 54

6.3 What the Author has Learnt 55

6.4 How this Final Year Project may Benefit Society 56

References 57

A Project Timeline 59

List of Reserved Symbols

Symbol	Meaning
Θ_R	Symbol used to denote the *God number* for Rubik's cube
$\Theta_W(n, c)$	Symbol used to denote the *God number* for the puzzle *Wrapslide*, where n represents the dimensions of the *Wrapslide* grid (in number of cells) and c represents the number of colours contained within the given grid

List of Acronyms

B&B: Branch and Bound

BFS: Breadth First Search

DFS: Depth First Search

GB: Gigabyte

GHz: Gigahertz

GPU: Graphics Processing Unit

GUI: Graphical User Interface

IDE: Integrated Development Environment

RAM: Random Access Memory

List of Figures

1.1 Rubik and his cube 2
1.2 The inventor and two states of the puzzle *Wrapslide* 4
1.3 A succession of four moves away from the solved state 5
1.4 Two *Wrapslide* states used to establish *God number* bounds 6

2.1 An example of a reflection transformation 10
2.2 The six symmetries of a hexagon 10
2.3 Examples of identities, unique isometries and non-unique isometries 11
2.4 Subunits of the plane and the torus 12
2.5 Examples of tilings 13
2.6 An example of a rooted tree data structure generated by the B&B method 14
2.7 Partial search trees generated by the B&B method according to BFS traversal . 14
2.8 Partial search trees generated by the B&B method according to DFS traversal . 15

3.1 The process of encoding a *Wrapslide* state as a binary number 21
3.2 Four isometric *Wrapslide* states produced by clockwise rotation 22
3.3 Equivalent states produced by "cutting open" the torus differently 23
3.4 Sixteen isometric *Wrapslide* states produced by clockwise rotation and translation 24
3.5 Thirty two isometric states, produced by rotation, translation and reflection . . . 25
3.6 Thirty two states that are isometric 26
3.7 An example of a game tree for a mixed 4×4 four-colour *Wrapslide* state 26
3.8 A conceptual illustration of a state enumeration tree 27
3.9 The state enumeration tree for the 4×4 two-colour *Wrapslide* grid 29
3.10 Twenty four equivalent states produced by means of colour permutations 30
3.11 6×6 two- and four-colour states whose red cells are identically distributed . . . 33

4.1 An objective function value associated with a *Wrapslide* state 36
4.2 A pair of partial game trees generated by the proposed heuristic 37

4.3 A conceptual representation of the first component of the heuristic 39
4.4 The integration of the first and second components of the heuristic 40
4.5 Two move sequences that may be implemented to exchange two cells 42
4.6 Three components of a worked example demonstrating the heuristic 43
4.7 Conversion of the ordered list of class representatives to an actual move sequence 44

5.1 The input and output options presented to the user on entering the GUI 49
5.2 A blank grid and a specified mixed state for which the *Solve* button may be clicked 49
5.3 Two possible output windows 50

A.1 Project timeline in Gantt-chart form 59

List of Tables

1.1 The progression of bound improvement on the God number for Rubik's cube . . 3

3.1 The number of distinct states for various possible grid sizes and number of colours 20
3.2 Colour code assignment for a grid containing two, three and four colours 21
3.3 Binary to decimal conversion calculations: $(01001101)_2 = (77)_{10}$ 31
3.4 State equivalence classes in the various levels of the state enumeration trees . . . 32

4.1 The number of states for which the branching procedure is executed 38

5.1 The results obtained by the first heuristic implementation 52
5.2 The results obtained by the second heuristic implementation 52

CHAPTER 1

Introduction

Contents

1.1 Background 1
1.2 Problem Statement 4
1.3 Project Objectives 4
1.4 Research Methodology 6
1.5 Scope 7
1.6 Timeline 8
1.7 Project Organisation 8

1.1 Background

In 1974, Professor Erno Rubik (see Figure 1.1(a)) gave the world what is arguably the most successful and engrossing recreational puzzle in history — Rubik's cube, shown in Figure 1.1(b). Rubik's cube, however, turned out to be much more than a child's game. With its 43 quintillion possible configurations, the cube has fascinated mathematicians — perhaps far more than it ever entertained any child. One of the most common questions pestering mathematicians studying Rubik's cube was this: What is the smallest number of moves with which *any* of the cube's possible scrambled states can be solved?

The sheer volume of calculations required to answer this question was so overwhelming that it was said only the mind of a Deity could possibly find this value. Thus, the term *God number* was coined, denoted here by Θ_R. The search for the value Θ_R started almost immediately and by 1980 the lower bound $\Theta_R \geq 18$ had been established [24]. This lower bound follows from the fact that certain scrambled states of Rubik's cube require at least eighteen moves to solve. Shortly thereafter, in 1981, Morwen Thistlewaite proved that $\Theta_R \leq 52$. The development of more efficient solution procedures allowed for the continual improvement of upper bounds on Θ_R(shown in Table 1.1) until finally, in July 2010, thirty six long years after first posing the question, the value of Θ_R was found. Tomas Rokicki, Herbert Kociemba, Morley Davidson and John Dethridge [24] proved that all of the approximately 43 quintillion scrambled states can be solved in twenty moves or fewer, whereas certain scrambled states require at least twenty moves, yielding the value $\Theta_R = 20$.

They accomplished this admirable feat by employing several innovative devices. First, they partitioned the scrambled states of Rubik's cube into 2 217 093 120 mutually exclusive sets of

(a) *Prof Erno Rubik*

(b) *Rubik's cube*

FIGURE 1.1: *Rubik and his cube.*

19 508 428 800 states each [24]. Secondly, they reduced the number of sets that needed to be solved to 55 882 296 by eliminating symmetries and using set covering techniques. The third element of their approach was that they did not search for optimal solutions to each scrambled state. Instead, they looked only for solutions involving twenty moves or fewer. These solutions were found using a computer program they wrote to solve the states of a single set (containing 19 508 428 800 scrambled states) in about twenty seconds. Finally, using the strategy described above, as well as 35 CPU years donated by Google, they solved each of the approximately 43 quintillion possible states, thus ending the long search for the Θ_R.

Fortunately, mathematicians were not sentenced to an eternity of tedium now that Rubik's cube had been beaten. The age of the smart device brings with it a new opportunity for mathematicians to try their hand, or rather brain, at finding the *God number* for the "Rubik's cube" of the technological age — *Wrapslide* [3]. The puzzle *Wrapslide* was developed by Dr. Alewyn Burger, shown in Figure 1.2(a), and released in 2014. It can be played on any Apple or Android device and is available free of charge from the App Store [3] and Google Play [12].

To start the game, the player selects a grid size and a number of colours. The level of difficulty increases with an increase in grid size and/or the number of colours selected. The grid sizes available are 4×4, 6×6 and 8×8, and the number of colours can be selected as two, three or four. Once these selections have been made, *Wrapslide* scrambles the colours as shown in Figure 1.2(b), and a timer starts.

The player can now make the following moves: (S)he can slide the top or bottom half of the grid left or right and/or (s)he can slide the left and right half of the grid up or down. This is accomplished by sliding one's finger over the touch screen of a cellular telephone or tablet. The results of these moves can be anticipated as *Wrapslide* is a toroidal sliding block puzzle. This means that the top and bottom rows of each half are virtually "joined," and the left-most column of a half is similarly "joined" to its right-most counterpart. Therefore, if a player slides a half-row over the top-most boundary of the grid, it reappears at the bottom. Similarly, when the player slides a half-column off the right-hand side of the grid, it reappears on the left-hand side. This wrapping property of moves gave rise to the name of the puzzle. An example of four successive moves on a 4×4 grid with four colours is provided in Figure 1.3 to clarify the above explanation.

The objective of the puzzle is to move all the blocks of the same colour into a quadrant, as shown in Figures 1.2(c) and 1.3(a). It does not, however, matter which colour is grouped in which quadrant. Ultimately, a player should aim to achieve this unscrambled state in as few

TABLE 1.1: *The progression of bound improvement on the value of the God number for Rubik's cube [24].*

Date	Lower bound	Upper bound	Gap	Due to
July, 1981	18	52	34	Morwen Thistlethwaite
December, 1990	18	42	24	Hans Kloosterman
May, 1992	18	39	21	Michael Reid
May, 1992	18	37	19	Dik Winter
January, 1995	18	29	11	Michael Reid
January, 1995	20	29	9	Michael Reid
December, 2005	20	28	8	Silviu Radu
April, 2006	20	27	7	Silviu Radu
May, 2007	20	26	6	Dan Kunkle and Gene Cooperman
March, 2008	20	25	5	Tomas Rokicki
April, 2008	20	23	3	Tomas Rokicki and John Welborn
August, 2008	20	22	2	Tomas Rokicki and John Welborn
July, 2010	20	20	0	Tomas Rokicki, Herbert Kociemba, Morley Davidson, and John Dethridge

moves as possible.

God's number for *Wrapslide* on an $n \times n$ grid (where $n \in \{4, 6, 8\}$) with $c \in \{2, 3, 4\}$ colours is denoted by $\Theta_W(n, c)$ and is defined similarly as for Rubik's cube. It is the smallest number of moves with which *any* scrambled state can be solved, where a move is defined as a single finger slide over the touch screen over any distance. Burger himself demonstrated that any scrambled state of the 4×4 grid with four colours can be solved in at most twelve moves [6]. Since the scrambled state in Figure 1.4(b) requires at least twelve moves to solve, however, the value of one of the *God numbers* for *Wrapslide* is known, namely $\Theta_W(4, 4) = 12$.

Although the value of $\Theta_W(n, c)$ for $n \in \{6, 8\}$ and $c \in \{2, 3, 4\}$, as well as $\Theta_W(4, 2)$ and $\Theta_W(4, 3)$, are still unknown, some progress has been made in establishing the value of $\Theta_W(6, 4)$. A pair of Dutch students have, for example, proved that each of the possible

$$\frac{36!}{(9!9!9!9!)} = 21\,452\,752\,266\,265\,320\,000$$

6×6 grid states is reachable by a succession of moves from the unmixed state [10]. Building on this discovery, Burger [6] established the bounds

$$21 \leq \Theta_W(6, 4) \leq 31. \tag{1.1}$$

The lower bound was established through the discovery of the state in Figure 1.4(a), which cannot be solved in twenty moves or fewer. The upper bound was determined by showing that any colour can be grouped into a quadrant in twelve moves or fewer. The remaining sub-puzzle of arranging the other three colours can furthermore be completed in nineteen moves or fewer, even when only shifting quadrants downward and/or to the left. Summing these two values results in the upper bound of thirty one mentioned above. It is anticipated that the bounds in (1.1) will have to be improved several times in order to find the value of $\Theta_W(6, 4)$ (just as the

(a) *Dr. Alewyn Burger* (b) *Scrambled state* (c) *Solved state*

FIGURE 1.2: *The inventor and two states of the puzzle* Wrapslide. *(a) Dr. Alewyn Burger, (b) a scrambled state of the* 6×6 *instance with four colors and (c) the corresponding solved state.*

Rubik's cube bounds were adjusted thirteen times before the final value, $\Theta_R = 20$, was obtained, as illustrated in Table 1.1). The iterative improvement of these bounds will require an efficient approach to solving the numerous mixed states of the puzzle *Wrapslide*.

1.2 Problem Statement

The aim of this project is twofold: first, to design, implement and demonstrate an enumeration approach that allows for the implicit exploration of a search space consisting of all the distinct *Wrapslide* states for a grid of small dimensions and secondly to design, implement and demonstrate a computer-implemented heuristic solution procedure capable of solving scrambled states of the puzzle *Wrapslide* in an optimal or near-optimal number of moves. The demonstration of the latter includes the solution of a number of *Wrapslide* scrambled states as well as the recording and analysis of the statistics generated by the heuristic. This enumeration approach and heuristic have the potential to establish (or aid in the conjecturing of) good bounds on the values of the *God number* $\Theta_W(n, c)$ for the pairs $(n, c) = (4, 2), (4, 3)$ and $(6, 2)$.

1.3 Project Objectives

The following ten objectives are pursued in this project:

I To *conduct* a methodological study of the literature related to:

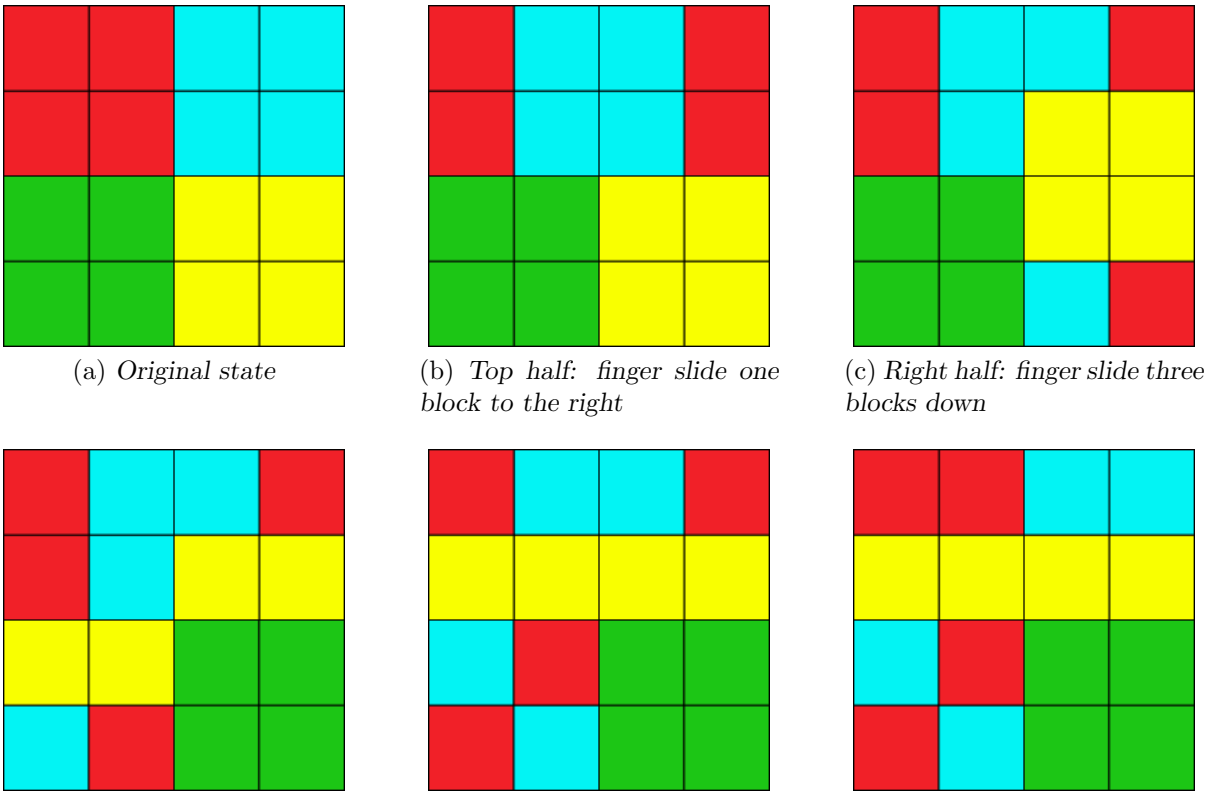
(a) symmetry, isometries and equivalence classes,

(b) the various symmetries generated by tilings in the plane and on a torus,

(c) the well-known *branch and bound* (B&B) method for solving combinatorial optimisation problems,

(d) the best-known tree traversal protocols commonly used in conjunction with the method of Objective I (c) and,

(e) computations that have been performed to determine the currently best-known God number bounds.



(a) *Original state*

(b) *Top half: finger slide one block to the right*

(c) *Right half: finger slide three blocks down*

(d) *Bottom half: finger slide two blocks to the left*

(e) *Left half: finger slide one block up*

(f) *Top half: finger slide one block to the right*

FIGURE 1.3: *A succession of four moves away from the solved state of a* 4×4 Wrapslide *grid with four colours.*

II To *establish* an enumeration framework for the solution space of *Wrapslide* instances which is capable of eliminating tiling symmetries.

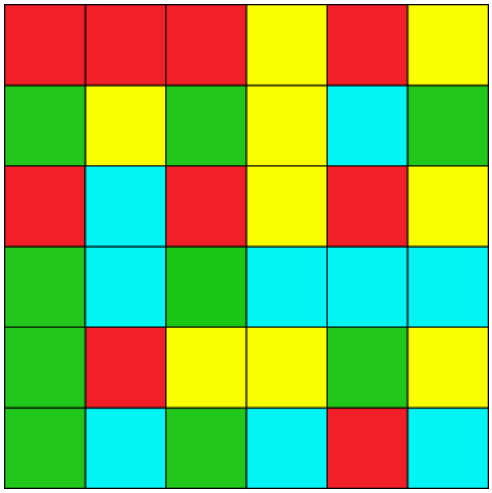
III To *employ* the enumeration framework of Objective II in order to establish bounds on the God numbers for small *Wrapslide* instances.

IV To *formulate* an objective function capable of quantifying the degree of colour heterogeneity in the four quadrants of a (mixed) *Wrapslide* state.

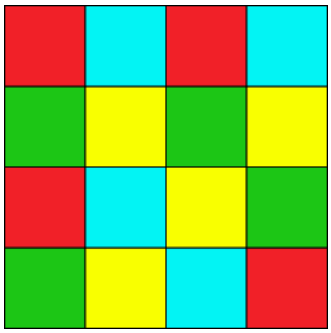
V To *design* a heuristic procedure for solving a *Wrapslide* scrambled state in an optimal or hopefully near-optimal number of moves.

VI To *implement* the enumeration framework of Objective II and the heuristic of Objective IV in a suitable software environment. The software implementation of the enumeration framework should be capable of generating all the distinct states of a small *Wrapslide* grid dimension when all tiling symmetries are eliminated. The software implementation of the heuristic should be capable of storing and analysing statistics generated by the heuristic solution procedure (such as the computation time and number of *Wrapslide* moves required to arrive at a solution).

VII To *demonstrate* the working of the implementation of Objective VI by applying it to a number of *Wrapslide* instances.



(a) *A scrambled state of the* 6×6 *grid with four colours used to establish the currently best lower bound on the* God number $\Theta_W(6,4)$



(b) *A scrambled state of the* 4×4 *grid with four colours requiring at least 12 moves to solve*

FIGURE 1.4: *Two states of the puzzle* Wrapslide. *(a) A scrambled state of the* 6×6 *grid with four colours used to establish the currently best lower bound on the* God number $\Theta_W(6,4)$ *and (b) a scrambled state of the* 4×4 *grid with four colours requiring at least 12 moves to solve.*

VIII To *evaluate* the efficiency and the degree of optimality of solutions obtained by the heuristic implementation of Objective V in the context of the puzzle instances of Objective VII.

IX To *develop* a *graphical user interface* (GUI) for the heuristic implementation of Objective VI which will allow users to input a *Wrapslide* mixed state and will return, as output, the computation time, an upper bound on the number of moves required to arrive at a solution and the actual move sequence generated by the heuristic implementation.

X To *identify* future courses of action which may contribute to the determination of God numbers for *Wrapslide*.

XI To *recommend* further areas of study related to the subject matter of this project.

1.4 Research Methodology

The execution of this project consists of six stages. The first phase comprises a methodological study of the literature related to the topics mentioned in Objective I of §1.3. The literature on symmetry, isometries and equivalence classes mentioned in Objective I (a) forms the initial foundation of the author's understanding of the mathematical characteristics of the puzzle *W*rapslide.

One of these important characteristics of *Wrapslide* is the existence of tiling symmetries which give rise to the existence of isometric states. Isometric states are states obtained by consideration of several transformations of a given scrambled state. These specific transformations (such as 90°, 180° and/or 270° rotation of the entire grid around a vertical axis perpendicular to the grid and through its centre point, or reflection about an axis within the grid through its centre point) result in states that appear different from the original state but are, in fact, equivalent to the original state for the purposes of determining the optimal solution sequence length from

the scrambled state to a solved state. The literature study therefore covers the exploration of the symmetries generated by tilings in the plane and on a torus, in pursuit of Objective I(b).

The next part of the literature review — a study of the well-known B&B method and the best-known tree traversal protocols commonly employed in conjunction with the B&B algorithm — provides a starting point for the establishment of an enumeration framework in fulfilment of Objective II. The literature study concludes with an overview of studies of the properties of *Wrapslide* that have already been established.

The second phase of the project — development of a *Wrapslide* state enumeration framework and heuristic solution procedure — is carried out in pursuit of Objectives II–IV of §1.3. It commences with the conceptualisation of a suitable method for representing *Wrapslide* states mathematically. This is followed by the development of techniques aimed at reducing the number of states that have to be examined before a solution attempt is made, by avoiding consideration of states that are isometric. These techniques reduce the number of calculations that need to be performed, consequently reducing the processing and time requirements of the heuristic solution procedure.

The products of the above-mentioned efforts are utilised to generate a state enumeration tree which allows for the implicit exploration of each distinct state of a given *Wrapslide* grid, in fulfilment of Objectives II–III. These products are also used in the design a suitable heuristic which may be used to solve a given *Wrapslide* mixed state (approximately). This phase also includes the development of software functions capable of simulating the execution of all the possible player moves during solution of a puzzle instance.

The elements of the second project phase are used in the third phase to implement the designed state enumeration framework and heuristic solution method in a software environment, in fulfilment of Objective V of §1.3. The storage and analysis of statistics generated by applying the solution procedure to small puzzle instances allows for the inspection and possible improvement of the heuristic employed. The third phase of the project also includes a detailed demonstration of the working of the heuristic *Wrapslide* solution procedure in the context of a number of puzzle instances.

The fourth phase of the project consists of the evaluation of the *Wrapslide* solution procedure developed. Examining the efficiency of the heuristic and the degree of optimality of the solutions thus obtained (in terms of the number of moves required to solve a given scrambled *Wrapslide* state) enables the author to determine the potential of the solution procedure with respect to its contribution to the improvement of the currently best-known bounds on God numbers for the puzzle. Finally this phase also includes the establishment of bounds on God numbers for small *Wrapslide* grid sizes, as described in Objective III of §1.3.

The penultimate phase consists of the development and implementation of a user-friendly GUI in fulfilment of Objective VIII.

The project concludes in the sixth stage which consists of a documentation of suggestions for further areas of study and recommendations in respect of future courses of action in pursuit of the improvement of God number bounds for *Wrapslide*, in fulfilment of Objectives IX and X of §1.3.

1.5 Scope

The focus in this project is on the development of a heuristic solution procedure for the puzzle *Wrapslide*. In order to reduce the complexity of the associated problem, only the 4×4 and

6×6 grids containing two, three and four colours are considered. Although this leaves the 8×8 grid unexamined, the complexity of such large Wrapslide instances places large instances beyond the scope of a fourth-year project. Furthermore, the determination of the exact minimum number of moves required to solve any given scrambled *Wrapslide* state falls outside the scope of this project. Instead, the primary project aim is to contribute a heuristic capable of quickly solving the puzzle in a satisfactory or, if possible, near-optimal number of moves. Finally, the determination of the exact values of all the God numbers for *Wrapslide* grids of the dimensions mentioned above is beyond the scope of this project, although the establishment of good bounds on these numbers for small puzzle instances is a secondary project aim.

1.6 Timeline

The timeline associated with the execution of the five project phases, discussed in §1.4, is represented in Gantt chart form in Appendix A.

1.7 Project Organisation

This report consists of five further chapters, excluding the current introductory chapter. Chapter 2 is a review of literature related to the topic of the project. The third chapter contains a description of how concepts explored in Chapter 2 may be applied to find equivalence classes of states of the puzzle *Wrapslide* and how branch and bound state enumeration trees may be generated to establish God numbers for small *Wrapslide* grids. Chapter 4 focusses on the development of a heuristic solution procedure for the puzzle *Wrapslide* which is capable of solving a mixed *Wrapslide* state in an optimal or near-optimal number of moves. The fifth chapter is devoted to the description of the development of a GUI for the software implementation of the heuristic described in Chapter 4. The final chapter, Chapter 6, contains a project summary, suggestions for future work and the author's reflections on the skills acquired during the completion of this project as well as the benefit it may hold for society.

CHAPTER 2

Literature Review

Contents

2.1 Symmetry, Isometries and Equivalence Classes 9
2.2 Tilings of the Plane and the Torus 12
2.3 Tree Search Protocols 12
2.4 Prior Investigations Related to the Puzzle Wrapslide 16
2.5 Chapter Summary 17

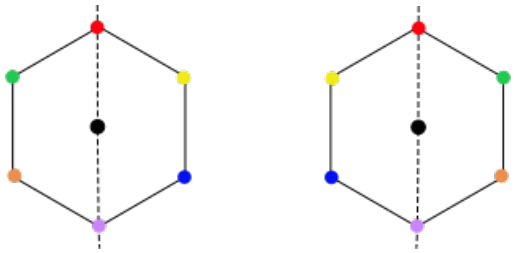
An important element in the development of a solution procedure for the puzzle *Wrapslide* is the identification and representation of "equivalent" states of a given *Wrapslide* instance. This chapter contains brief overviews of various basic concepts such as symmetry, isometries, the plane and the torus as topological objects, and tilings of the plane and the torus. It also includes a brief description of the well-known branch and bound method for solving optimisation problems and two of its commonly employed tree traversal protocols. The chapter closes with a description of academic research conducted in the context of the puzzle *Wrapslide* up to date.

2.1 Symmetry, Isometries and Equivalence Classes

This section opens with an overview of basic notions related to the concept of symmetry, as most readers will be more familiar with, and have a better intuitive understanding of, symmetry than of isometries. These notions are then used to aid in an explanation of the concepts of isometries and of equivalence classes, both of which are used extensively in the remainder of this project.

Before the concept of symmetry may be introduced, however, it is necessary to define the notion of a *transformation*. A *transformation* of an object may be described as a formula that defines the change(s) affected to a geometric shape, point or line in order to produce a specified new object. Athough many different types of transformations exist, the focus in this section falls on transformations that produce symmetries in the plane, where the word *plane* refers here to the Euclidian plane parametrised by ordered pairs of real numbers (x, y) and denoted by $\Re^2$ [13].

According to Stewart I & Golubitsky M [26], a symmetry may be described as a transformation of an object that leaves it *apparently* unchanged. The adjective 'apparently' is used to communicate that although the object resulting from an application of the transformation appears to be identical to the original object it has, in fact, moved. For instance, although the shape resulting from reflecting the hexagon in Figure 2.1(a) around a vertical axis through

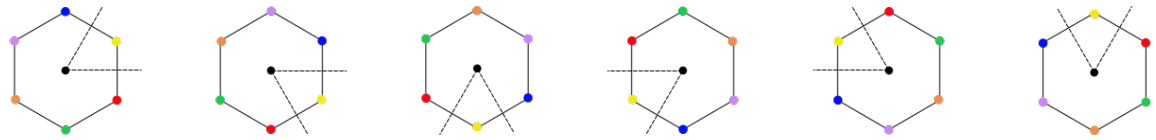


(a) (b)

FIGURE 2.1: *An example of a reflection transformation. (a) The original hexagon with a vertical line through its centre point and (b) the reflection of the hexagon in (a) around the vertical line thourgh its centre point.*

its centre point (as indicated by the dashed line) looks identical to the original, each point on the object is no longer in its original position. The change in positions of the coloured vertices on the hexagon corners from Figure 2.1(a) to Figure 2.1(b) illustrate the movement of certain points on the object that has taken place.

By applying this transformation, called a *reflection*, two symmetries of an object are obtained: the original object (occasionally called the *identity*) and the reflected object. Other transformations may produce more symmetries, depending on the properties of the object. For instance, rotation of the hexagon in Figure 2.1(a) about an axis through its centre point and perpendicular out of the page produces a symmetric object for each 60° *rotation* of the object around its centre point, yielding a total of six symmetries (including the original object). These symmetries are shown in Figure 2.2.



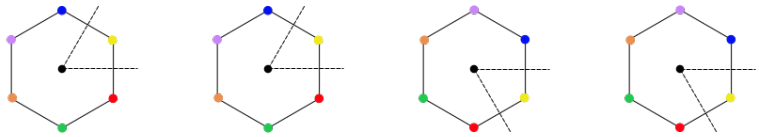
(a) *Original* (b) *60°* (c) *120°* (d) *180°* (e) *240°* (f) *300°*

FIGURE 2.2: *The six symmetries of a hexagon. The hexagon in (a) is the original shape and the others are the symmetries of rotation transformations applied through multiples of 60°.*

Another transformation used to produce symmetries of an object is *translation*. A translation transformation occurs when each point on a shape in the plane moves the same distance up or down and/or left or right. Thus if, for instance, the centre point of the hexagon in Figure 2.1(a) is moved five units to the right every point on the hexagon will move five units to the right if a translation is applied to the hexagon.

An important concept when considering the objects produced by the transformations mentioned above is that of a *unique* object. In this context, the descriptor *unique* is used to convey the fact that there is at least one point on the object that is not in the original position. To clarify this statement, consider Figure 2.3. It is evident that four 90° rotations, as shown in Figure 2.3(b), are equivalent to a 360° rotation and therefore produce the identity of the original object. Similarly, a 420° rotation, shown in Figure 2.3(d), produces the identity of the 60° rotation in Figure 2.3(c). Thus a 60° rotation of the hexagon produces a unique object, while a 420° rotation transformation does not.

The last type transformation that may be used to produce a symmetry of an object in the



(a) *Original* (b) $4 \times 90°$ (c) *60°* (d) *420°*

FIGURE 2.3: *Examples of identities, unique isometries and non-unique isometries. (a) An original hexagon (identity), (b) the original hexagon rotated* $4 \times 90° = 360°$ *resulting in the identity, (c) an isometry of the original hexagon obtained by a 60° rotation of the identity, and (d) a* $420°$ *rotation of the original hexagon or a* $360°$ *rotation of the isometry in (c), both yielding the same hexagon.*

plane is *glide reflection*. Glide reflections are produced by the combination of a reflection and a nonzero translation in the direction of the axis of reflection [9]. Glide reflections are not, however, relevant in the context of this project and are therefore not considered in more detail.

The first three types of transformations mentioned above are, however, extremely relevant in the context of this project, particularly in respect of isometries. An *isometry* may be described as a transformation h: $\Re^2 \mapsto \Re^2$ that preserves distance. Thus, if h is an isometry, $||h(\boldsymbol{v}) - h(\boldsymbol{w})|| = ||\boldsymbol{v} - \boldsymbol{w}||$ for all vectors $\boldsymbol{v}$ and $\boldsymbol{w}$ in $\Re^2$ [9], where $||\cdot||$ is a norm. Similar to the observations made for symmetry it should be noted that while all translations are isometries, the same cannot be said of reflections and rotations. Generally, only rotations about the origin and reflections about an axis through the origin are isometries. Furthermore, it has been shown that all isometries of the plane, excluding the identity, may be classified as one of the four types of transformations mentioned above [8]. Therefore, all the isometries of a plane can be found using one or a combination of these isometries.

The last important concept considered in this section is that of an *equivalence class*. To support a description of this concept, the notion of a *symmetry group* is first considered. According to STEWART I & GOLUBITSKY M [26], a *group* is a *closed* system of transformations that, if combined, produce another member of the same group. If each of the transformations in such a group produces a symmetry of a given object in the plane, this group is called the *symmetry group* of the object. If, for instance, the 60° and 120° rotation symmetries of the hexagon form part of a symmetry group, then that group must also include the 180° rotation symmetry since the combination of the 60°and 120° rotations result in the 180° rotation symmetry. In simpler terms, any combination of transformations in a symmetry group must produce the *apparently* unchanged object since, as STEWART I & GOLUBITSKY M [26] explain, if one leaves an object "unchanged" twice (or $n \in \{2, 3, \ldots, \infty\}$ times), it remains "unchanged."

Applying the same reasoning to isometries it is evident that the combination of two or more isometries produce yet another isometry. The number of *unique* objects that can be produced are, however, limited. These unique isometries may be used to form an *equivalence class* of objects. An equivalence class is a subset $\mathcal{S} = \{y \in X \mid y \sim x\}$ of a set X, where the *equivalence relation* on X is a relation denoted by $\sim$ and satisfies the following: $x \sim x$ for all x (the so-called *reflexive property*), $x \sim y$ if $y \sim x$ (the so-called *symmetric* property) and finally, $x \sim y$ and $y \sim z$ imply $x \sim z$ (the so-called *transitive* property) [25]. In other words, $\mathcal{S}$ may be described as the set of all elements that are equivalent to x. All the objects in such a class may be considered equal for the purposes of analysis and can be represented by a single object, called the *canonical form* or *class representative*. Thus, once a class representative has been identified, calculations performed in respect of the class representative may be used to predict the behaviour of all the other objects in its class. The considerable utility of this principle will become evident in the

succeeding chapters of this report.

2.2 Tilings of the Plane and the Torus

According to SCHWARTZ RE [25], the notion of a square torus is a compromise between the Euclidian plane considered in §2.1 and the unit sphere in $\Re^3$. This description is used to convey that, while the square torus is flat at every point on the plane (like the Euclidian plane), it is also bounded like the sphere. SCHWARTZ RE [25] clarifies this concept by alluding to a virtual two-dimensional being, call it a *bug*, that crawls across the square torus. If the bug were to crawl over the top-most edge of the torus, it would "reappear" at the bottom of the torus in the same horizontal position. Similarly, should the bug crawl off the left-hand side of the torus, it would "reappear" on the right-hand side in the same vertical position. The reason this would occur becomes evident when the unit cell in Figure 2.4(a) is embedded on a torus, as shown in Figure 2.4(b). The edges of the unit cell labelled, N, E, S and W, correspond to the "edges" indicated on the torus.

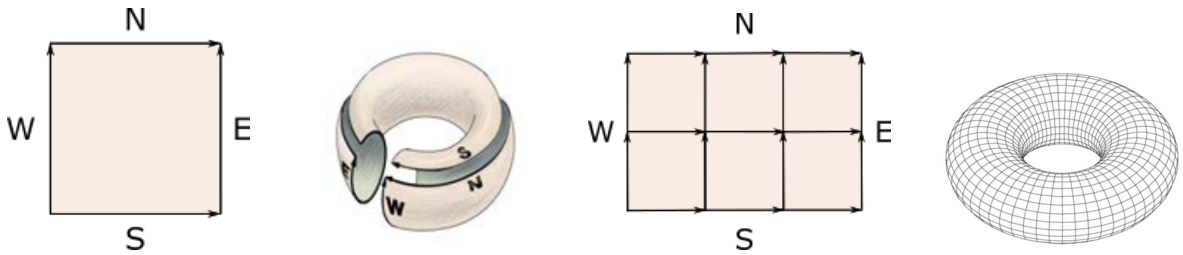


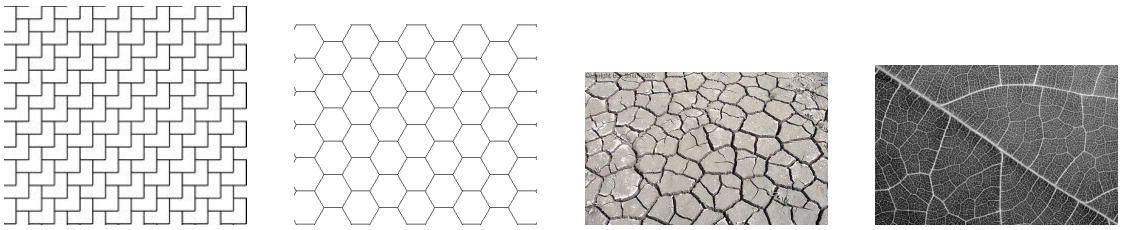
FIGURE 2.4: *Subunits of the plane and the torus. (a) A unit cell, (b) the unit cell embedded to a torus, (c) 2×3 unit cells embedded in the plane and (d) $m \times n$ unit cells embedded on the torus [20].*

Larger structures, containing m unit cells in the vertical direction and n unit cells in the horizontal direction, may similarly be created, as shown in Figure 2.4(c). Figure 2.4(d) demonstrates how such an $m \times n$ plane may be embedded on a torus.

The above decomposition of the torus into finite sized subsets may also be achieved by viewing the unit cells as *tesselations* or *tilings* on the surface of the torus, where a *tiling* of a surface is a *covering* of the surface by non-overlapping shapes with no spaces in-between [14]. Some examples of tilings are shown in Figure 2.5. The classification of features on the surface of a torus as tilings allows for two important realisations. First, all the subsets or *tiles* do not have to be identical, as is the case in Figures 2.5(c) and 2.5(d). Secondly, if the torus is composed of tiles, tiling symmetries may exist. Thus, the principles of symmetry, isometries and their equivalence classes, as discussed in the previous section, may be applied to find a set of tori "equivalent" to a given torus.

2.3 Tree Search Protocols

One of the most commonly used methods for solving combinatorial optimisation problems is the celebrated *branch and bound* (B&B) method [7]. This method, first proposed by LAND AH & DOIG AG [18] in 1952, can be applied to problems having a combination of discrete and continuous decision variables and guarantees the identification of an optimal solution as



(a) (b) (c) (d)

FIGURE 2.5: *Examples of tilings. (a) An example of a periodic tiling, (b) an example of a hexagonal tiling, (c) an example of a tiling occuring in nature formed by dried mud, and (d) a close-up of the veins on a leaf which form leaf subsections analogous to a tiling.*

output [7]. The general B&B method may be described by considering an objective function $f(\boldsymbol{x})$, of decision variables $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$, which must be minimised over a region of feasible solutions, $\mathcal{S}$. This region, called the *search space* or *feasible region*, contains all values of the decision variables which produce candidate solutions and is determined by some set of constraints defined for the optimisation problem.

The B&B method explores this search space by employing an iterative process to generate a data structure resembling a rooted tree, as shown in Figure 2.6. The root, or starting node, (at the top) of the tree represents the full set of candidates in the solution space. The first iteration of the solution process partitions this solution space into smaller convex[1] subspaces, called *nodes*, by employing the first of the two main B&B procedures, called *branching*. The branching procedure partitions a given set into two or more smaller, mutually exclusive subsets (whose union is the original set) [7]. The subsets are represented by new nodes in the search tree and an upper bound on the minimum of the objective function value over each is obtained. The order in which the nodes in the B&B tree are traversed (and consequently the order in which the minimum objective function value for each set is bounded from above) depends on the tree search protocol employed. The branching process is repeated for each node until one of two termination criteria are met, in which case a node is said to be *fathomed*. The first termination condition is met when no feasible solution exists for the objective function over the subset associated with the node in question. The second termination condition is determined by the second main process employed by the B&B algorithm, called *bounding* [7].

The bounding procedure is used to allow for partial or selective enumeration of the solution space in order to reduce the number of computations, and consequently, the amount of time required to find an optimal solution. This is accomplished by comparing the bound on the minimum objective function value associated with the current node of the search tree with the best feasible solution found so far over all the previous nodes. This value is called the *incumbent upper bound*. If the objective function value of the current node is better (smaller) than the incumbent upper bound, the objective function value of this solution becomes the new incumbent upper bound. If, however, the current solution has an objective function value at least as large as the incumbent upper bound, then the node in the search tree associated with the solution is terminated, thus discarding the corresponding subset of solutions associated with the node. The process terminates when all the nodes have either been branched or fathomed, at which point the incumbent upper bound is taken as the objective function value of an optimal solution [21].

The computational time required to reach this solution is, however, intricately linked to the order in which the nodes are traversed or 'visited,' as dictated by the so-called *traversal protocol*

[1] A convex subset or subspace, $\mathcal{S} \subseteq \Re^n$, is a set for which a line segment between *any* two points in $\mathcal{S}$ lies completely in $\mathcal{S}$ [11].

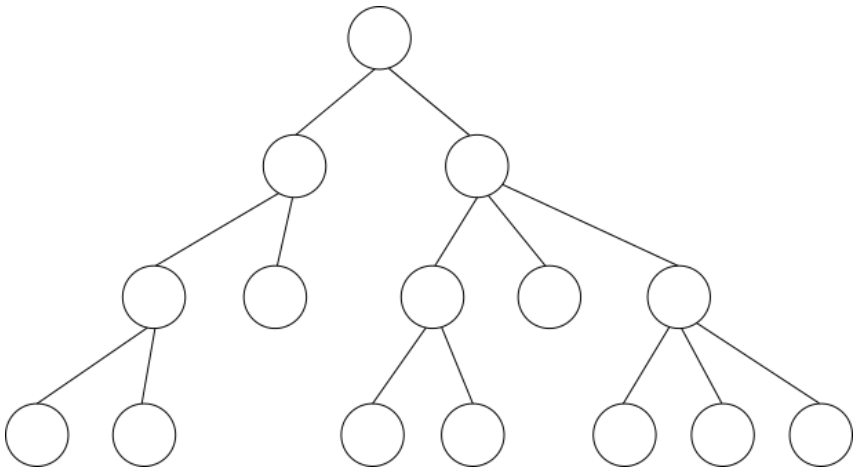
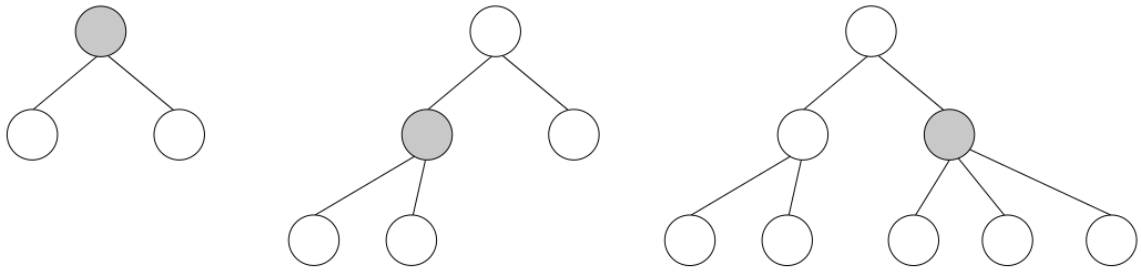


FIGURE 2.6: *An example of a rooted tree data structure produced by the application of the B&B method.*

selected for the problem. Several traversal protocols have been developed for such search trees, of which the *Breadth First Search* (BFS) and *Depth First Search* (DFS) protocols are arguably most commonly used. According to the BFS, the nodes of the search tree are explored in order of increasing distance from the starting node. It partitions the tree into layers, $L_0, L_1, L_2, \ldots$, such that the shortest path from the root to each node in layer L_i is i [15]. The BFS protocol dictates that each node in a level is visited, and either branched or fathomed, before any of the nodes in further levels are visited. Furthermore, the order in which nodes are visited within a level is determined by the objective function value calculated for each node. Nodes are visited in order of decreasing desirability. That is, for the minimisation problem being considered, nodes are visited in order of increasing objective function values. Figure 2.7 illustrates the first three iterations of the generation of the search tree in Figure 2.6 when the BFS protocol is employed. The node being considered during each successive iteration is coloured grey, where consideration of a node entails calculation of its objective function value and either termination or branching.



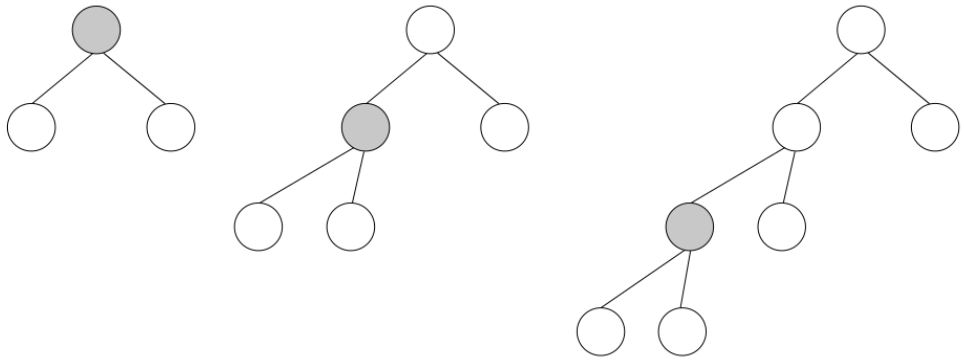
(a) *1st iteration* (b) *2nd iteration* (c) *3rd iteration*

FIGURE 2.7: *Partial search trees generated by the first three iterations of the B&B method employing the BFS traversal protocol to construct the search tree in Figure 2.6.*

It may intuitively be seen that the BFS protocol will always find the shortest path in the search tree from the root to the optimal solution which can, for instance, be used to compute the minimum number of moves necessary to solve a given puzzle. There are, however, two obvious disadvantages of the BFS protocol. First, an optimal solution far from the root will typically require a large amount of computation time. Secondly, since each node in the current level of the search tree must be stored in order to be able to generate the next level, implementation of the BFS protocol on a computer may require an excessive, possibly even infeasible, amount

of memory, depending on the size of the problem's solution space [16]. A number of alternative traversal protocols, including the DFS protocol, have been designed to alleviate this problem.

As the name suggests, the DFS protocol explores the maximum possible depth along a branch of the search tree before backtracking and exploring another path. Thus if, for instance, the branching procedure has just been applied to node a to produce two child nodes b and c, the next node to be considered will be node b (the left-most[2] child node is generally considered first). If node b is not fathomed, branching is applied to find its child nodes. Once again, the left-most of the child nodes of b, call it d, is considered first and if it too is not fathomed, branching is applied to d. This process is continued until the node being considered is fathomed or the maximum depth of the tree has been reached, at which point the search backtracks to the closest node that has not yet been considered, and the process is repeated. Figure 2.8 illustrates how the search tree in Figure 2.6 would be generated when employing the DFS protocol rather than the BFS protocol. Once again, the node being considered during each of the successive iterations is coloured grey.



(a) *1st iteration* (b) *2nd iteration* (c) *3rd iteration*

FIGURE 2.8: *Partial search trees generated by the first three iterations of the B&B method when employing the DFS traversal protocol to construct the search tree in Figure 2.6.*

An important characteristic of the DFS protocol is that implementation of this protocol on a computer requires the storage of a stack[3] containing only the nodes along the path from the root to the current node. It is obvious that the maximum required length of such a stack for a particular search tree is the depth of the tree. As the depth of a search tree is often much smaller than its maximum breadth (number of nodes in a level), the DFS protocol is often considered much more memory-efficient than the BFS protocol which, as previously mentioned, requires each node on a level to be stored in order to generate the nodes of the succeeding level. Another advantage of the DFS protocol is that it can find an optimal solution far from the root much faster than the BFS protocol, given that the optimal solution lies on the left-hand side of the search tree. Unfortunately, however, if the search tree is quite deep and the solution lies more towards the right-hand side of the search tree, the DFS protocol might take a considerable amount of time to find this solution, even if it lies extremely close to the root. Therefore, the selection of a suitable traversal protocol is crucial when solving large optimisation problems, especially if a limited computational budget is available for their solution.

[2] According to some pre-selected ordering applied to rank alternatives from left to right.

[3] A *stack* is a data structure commonly used in digital implementations of the B&B algorithm. It is helpful to consider a stack as a container from which objects are removed and into which objects are inserted according to the LIFO (Last-In-First-Out) principle. Elements may be added or removed from the top of the stack only. It should also be noted that a stack is a recursive data structure; it is either empty or it consists of a top element and the remaining elements which make up another stack [2].

2.4 Prior Investigations Related to the Puzzle Wrapslide

The number of prior investigations into the puzzle *Wrapslide* reflects the relative recency of its release, in September 2014, compared to, for instance, Rubik's cube. Not only are the number of investigations limited, all possible colour and grid size combinations have not even been considered. Investigations have rather tended to focus on the 4×4 and 6×6 four-colour grids. These investigations have, however, yielded valuable insights. The most relevant such insight in the context of this project, includes a proof that all the possible mixed states of the 6×6 four-colour grid are reachable from the unmixed state [10], the discovery of the God number $\Theta_W(4,4)$, and the establishment of upper and lower bounds on the God number $\Theta_W(6,4)$.

As mentioned in the previous chapter, the value of $\Theta_W(4,4)$ was determined by the creator of the puzzle [6]. He accomplished this feat by employing the B&B method discussed in §2.3. In order to generate the necessary search tree, the unmixed state was taken as the root (starting node) and the branching procedure was employed to generate every representative of the equivalence class of isometries of the 4×4 square torus state that could be reached from the current state (node) by executing a single move. Therefore, the level of a given state (equivalence class representative) in the search tree is the minimum number of moves required to reach the unmixed state from that state. Consequently, the minimum number of moves necessary to solve *any* possible state was determined by computing the distance from the state (equivalence class representative) furthest from the unmixed state in the search tree. It is evident that the state furthest from the unmixed state would lie in the last level of the search tree. The value of, $\Theta_W(4,4)$ is therefore equivalent to the depth of the search tree. By applying this reasoning, the value $\Theta_W(4,4) = 12$ was found [5].

This method could not be applied to the 6×6 four-colour grid as time constraints and the processing and storage capabilities of the equipment available limit the size of a search tree that can feasibly be generated. The method could, however, be used to establish bounds on the God number $\Theta_W(6,4)$. For instance, by applying a similar method to only one of the four colours in the 6×6 grid, it was established that any colour can be grouped into a quadrant in twelve moves or fewer. Similarly, BURGER AP [6] proved that the remaining three colours can be grouped into separate quadrants in nineteen moves or fewer, even when only moving quadrants downward and/or to the left. By summing these values he determined the currently best available upper bound of thirty one on the value of $\Theta_W(6,4)$. As mentioned in the previous chapter, a lower bound was established through the discovery of the state in Figure 1.4(a) which cannot be solved in twenty moves or fewer. Therefore, the currently best bounds are $21 \leq \Theta_W(6,4) \leq 31$.

Further research in the context of the 6×6 four-colour grid has also been conducted by Professor Bruce MacKenzie [19] who applied notions from group theory in an attempt to find the diameter[4] of the state graph[5] of the puzzle *Wrapslide* and to determine the likely depth distribution of the search tree for the 6×6 four-colour grid. These methods have, thus far, not contributed to the establishment of a better bound on the value of $\Theta_W(6,4)$, and so this line of research is not discussed in further detail.

[4]The *diameter* of a graph is the largest distance between two vertices of the graph.

[5]The *state graph* of the puzzle *Wrapslide* is a graph whose vertices correspond to class representatives of the game's states and in which two vertices are adjacent (joined by and edge) if the corresponding states can be reached from one another within a single game move (finger slide).

2.5 Chapter Summary

This chapter contains a review of the concepts necessary for the development of a solution procedure for the puzzle *Wrapslide*, as well as for the establishment of bounds on and values of its associated God numbers. It opened with descriptions of basic concepts related to the notions symmetry, isometries and equivalence classes in §2.1. This was followed by descriptions of the plane and the torus as topological objects, as well as tilings of the plane and the torus, in §2.2. The next section, §2.3, contained a description of the popular B&B method for solving combinatorial optimisation problems coupled with an explanation of the best-known tree traversal protocols employed in conjunction with the B&B algorithm. The chapter closed with an outline of prior academic investigations related to the puzzle *Wrapslide* in §2.5.

CHAPTER 3

State Enumeration Trees of the Puzzle *Wrapslide*

Contents

3.1 Encoding of *Wrapslide* States 19
3.2 The Equivalence Class of a Given *Wrapslide* State 22
3.3 Generation of a Branch and Bound State Enumeration Tree 25
3.4 Selection of Class Representatives 28
3.5 God Numbers for Selected *Wrapslide* Grids 31
3.6 Chapter Summary 33

The determination of God numbers for the various *Wrapslide* grids requires the exploration of several extensive search spaces. As in the approach followed to find the God number for Rubik's cube, methods may be developed to reduce the size of these search spaces. To this end, this chapter contains an exposition on the methods used to identify and represent equivalence classes of given *Wrapslide* states. The chapter opens in §3.1 with a description of the process followed to convert a *Wrapslide* state to a numerical representation. The focus in §3.2 is on the transformations and colour permutations that produce states belonging to the equivalence class of a given *Wrapslide* state. Next, the feasibility of generating B&B enumeration trees for *Wrapslide* states is explored. Section 3.3 opens with an explanation of how the B&B method may be applied to identify an optimal move sequence for solving a mixed *Wrapslide* state. Furthermore, §3.4 sees the implementation of the techniques introduced in previous sections to demonstrate how smaller B&B enumeration trees, yielding comparable results, may be generated. Detailed descriptions of the branching procedure and termination criteria are included. The penultimate section contains a description of a method whereby the values of the God numbers $\Theta_W(4,2), \Theta_W(4,3)$ and $\Theta_W(4,4)$ may be found. The chapter closes with a brief summary.

3.1 Encoding of *Wrapslide* States

Recall, from §1.1, that a *Wrapslide* God number is the smallest number of moves with which *any* scrambled state of the puzzle can be solved. It is evident that, in order to find such a God number by brute force, each of the possible mixed states of the puzzle must be considered. The number of possible mixed states for an $n \times n$ *Wrapslide* grid containing c colours is given by the

multinomial coefficient[1]

$$\binom{n^2}{\underbrace{(\frac{n}{2})^2,\ldots,(\frac{n}{2})^2}_{c-1\text{ entries}},(5-c)(\frac{n}{2})^2} = \frac{(n^2)!}{[(\frac{n}{2})^2!]^{c-1}[(\frac{n}{2})^2(5-c)]!}. \tag{3.1}$$

For example, as mentioned in §1.1, the 6×6 grid containing four colours has

$$\frac{36!}{9!9!9!9!} = 21\,452\,752\,266\,265\,320\,000$$

distinct states. The numbers of distinct *Wrapslide* states are tabulated in Table 3.1 according to (3.1) for the various possible grid sizes and numbers of colours. Exploring search spaces of the sizes shown in Table 3.1 by brute force would require a prohibitive computational budget and memory storage capacity. This problem may be alleviated partially by the implementation of the concepts of equivalent states and class representatives, as described in §2.1.

TABLE 3.1: *The number of distinct* Wrapslide *states for various possible grid sizes and number of colours.*

Grid size	Two colours	Three colours	Four colours
4×4	1.820×10^3	9.009×10^5	6.306×10^7
6×6	9.414×10^7	4.412×10^{14}	2.145×10^{19}
8×8	4.885×10^{14}	1.102×10^{27}	6.621×10^{35}

In order to perform the calculations that allow for the identification of such equivalent states and their associated class representatives, however, a means of representing or encoding a given *Wrapslide* state is required. The puzzle state encoding adopted in this project involves arbitrarily assigning each colour a numeric value from one to c, as shown in the first column of Table 3.2. A given state may therefore be represented by an $n \times n$ array containing values from the set $\{1,\ldots,c\}$, as illustrated in Figure 3.1(b) for the state shown in Figure 3.2(a). This state encoding convention facilitates efficient performance of a large portion of the required calculations and permits natural visualisation of a state, due to the similar dimensions of the array and its corresponding state, coupled with the simple mapping between the set of colours and the set $\{1,\ldots,c\}$.

Timeous identification of a class representative requires the construction of an equivalence class and an efficient means of comparing the states within this equivalence class. In order to accomplish this, each equivalence class state is converted to a natural number. This is achieved by assigning one or two bits to each numeric value in the game state encoding described above. With the aim of minimising memory requirements, only one bit (a 1 or 0) is assigned to a colour when the grid contains only two colours, whereas grids containing three or four colours require

[1]Let $n_1 + n_2 + \cdots + n_c = m$, and define the so-called multinomial coefficient

$$\binom{m}{n_1, n_2, \ldots, n_c} = \frac{m!}{n_1!n_2!\cdots n_c!}.$$

Then the $\binom{m}{n_1,n_2,\ldots,n_c}$ represents the number of distinct partitions of m distinct objects into c distinct groups of respective sizes $n_1, n_2, \ldots, n_c$. In the $n \times n$ *Wrapslide* puzzle with $c \in \{2,3,4\}$ colours, there are $n_i = (\frac{n}{2})^2$ squares of colour i for all $i \in \{1,2,\ldots,c-1\}$ and $n_c = (5-c)(\frac{n}{2})^2$ squares of colour c in any solved game state, for $n \in \{2,3,4\}$. Furthermore, $m = n^2$. The number of distinct game states is therefore the number of distinct partitions of the n^2 *Wrapslide* grid cells into c colour classes of respective sizes $n_1 = (\frac{n}{2})^2, \ldots, n_{c-1} = (\frac{n}{2})^2$ and $n_c = (5-c)(\frac{n}{2})^2$.

TABLE 3.2: *An example of colour code assignment for a grid containing two, three and four colours.*

Colour	Number	Two colour grid	Three colour grid	Four colour grid
Red	1	1	00	00
Green	2	0	01	01
Blue	3	—	10	10
Yellow	4	—	—	11

the assignment of two bits per colour. Examples of colour code assignments for the two-, three- and four-colour grids are shown in the last three columns of Table 3.2. These bits are concatenated into a bit string which is easily converted to a decimal value. Figure 3.1 shows an example of the implementation of this method. The bit string, '10110100000111101011101100010001', in Figure 3.1(e) is obtained by reading out lexicographically the game state encoding in Figure 3.1(d). When viewed as a binary number, the game state encoding therefore corresponds to the decimal representation 3 021 912 849. A 4×4 *Wrapslide* state containing four colours may, for instance, be represented by a 32-bit string, which corresponds to a natural number when viewed as a number in the binary system.

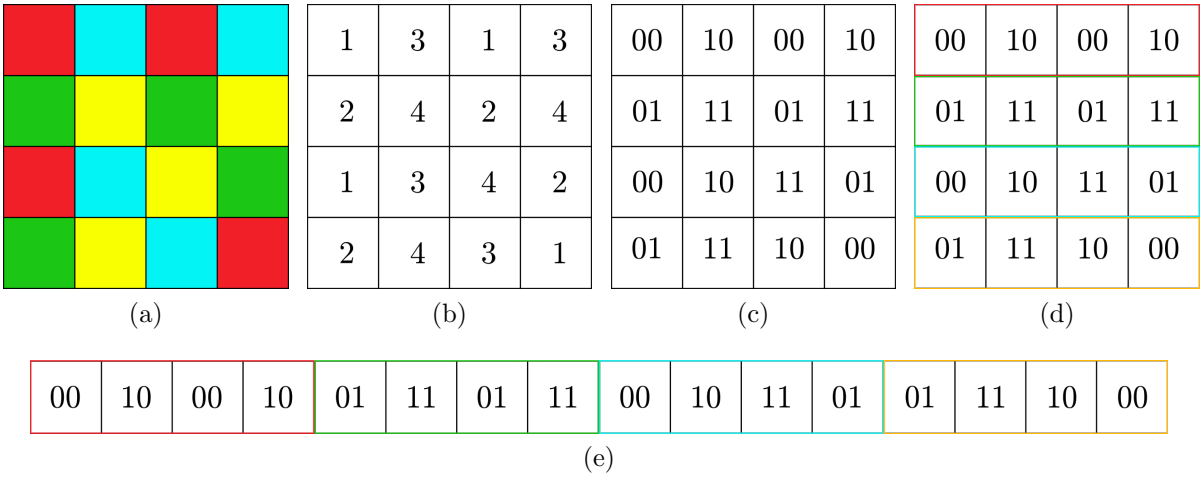


FIGURE 3.1: *The first steps of the method used to represent a graphical state as a binary number. (a) A mixed state, (b) its representative numeric array, (c) the corresponding binary array and (e) the concatenated bit string.*

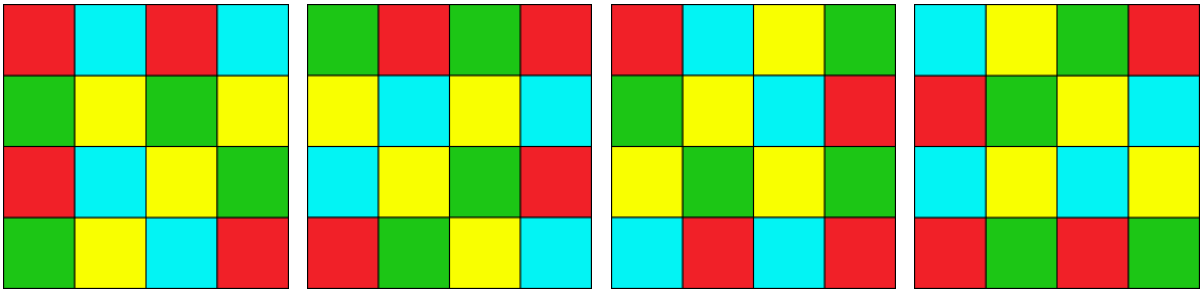
The length of the bit string required depends on the dimensions and number of colours contained in the grid. For instance, a 4×4 grid with two colours will only require a bit string of sixteen bits, as each colour can be represented by a single bit and the grid contains sixteen cells. A 6×6 grid containing four colours, however, requires a bit string of $2 \times (6 \times 6) = 72$ bits to represent a state (because each colour is assigned two bits and the grid contains $6 \times 6 = 36$ cells).

It should be clear that the conversion from a graphical to a decimal representation of a *Wrapslide* state, as described above, is reversible.

3.2 The Equivalence Class of a Given *Wrapslide* State

The method of *Wrapslide* game state encoding, as described in §3.1, efficiently facilitates construction of an equivalence class of states for any particular state. This is achieved by applying the isometries (distance-preserving transformations) discussed in §2.1 to the original state and exploring all colour combinations that result in states that are equivalent to the original state. This process is described here by referring to the four transformations of rotation, translation, reflection (described in §2.1) and colour permutation, in the order mentioned. The equivalence class construction process is elucidated throughout this discussion in the context of the game state in Figure 3.1(a), referred to as the *original state* for the purpose of discussion.

The original state in Figure 3.1(a) may be successively rotated clockwise through 90°, to produce a total of four states, including the original, as shown in Figure 3.2. The resulting states are isometric to the original state and may therefore be considered equivalent to the original state. A practical motivation for the equivalence of these states may be obtained by visualising the puzzle on an electronic device such as a smartphone. Suppose the original state in Figure 3.2(a) is what the puzzle would look like when seen on the smartphone in its normal orientation. The second state, shown in Figure 3.2(b), shows the same puzzle state as viewed when the phone is rotated clockwise to lie on its right-hand side. Similarly, Figures 3.2(c) and 3.2(d) show the same puzzle state as viewed when the phone is upside-down and lying on its left-hand side, respectively.



(a) *Original* (b) *90° Rotation* (c) *180° Rotation* (d) *270° Rotation*

FIGURE 3.2: *Four* Wrapslide *states that are isometric to the state in Figure 3.1(a) produced by clockwise rotation of that state.*

To obtain the second set of equivalent states, the torus described in §2.2 is considered. As shown in §2.2, a grid of cells in the plane may be embedded on a torus. This process may be reversed by "cutting open" the torus. The same torus may be "cut open" differently, resulting in grids that look different when embedded in the plane but which, in fact, represent the same torus tiling. Combining this principle with the fact that a certain colour does not have to be grouped into any specific quadrant in order to reach a solved *Wrapslide* state, four equivalent states are obtained for each of the four equivalent states produced when cutting open the torus. This claim is illustrated in Figure 3.3. The green and red lines in Figures 3.3(e)–3.3(h) indicate the different paths along which the toroidal embedding of a *Wrapslide* state may be "cut open" to generate equivalent plane tilings. Figures 3.3(a)–3.3(d) show the states that result from cutting open the torus embedding corresponding to the original state in Figure 3.1(a) along the paths indicated in the figures directly below them. Cutting the toroidal embedding of the original state in Figure 3.1(a) along the outer central ring (indicated by the red line in Figure 3.3(f)) and along the vertical ring (indicated by the green line) directly opposite the original centre point (indicated by the black dot) to embed the tiling in the plane, for example, results in the

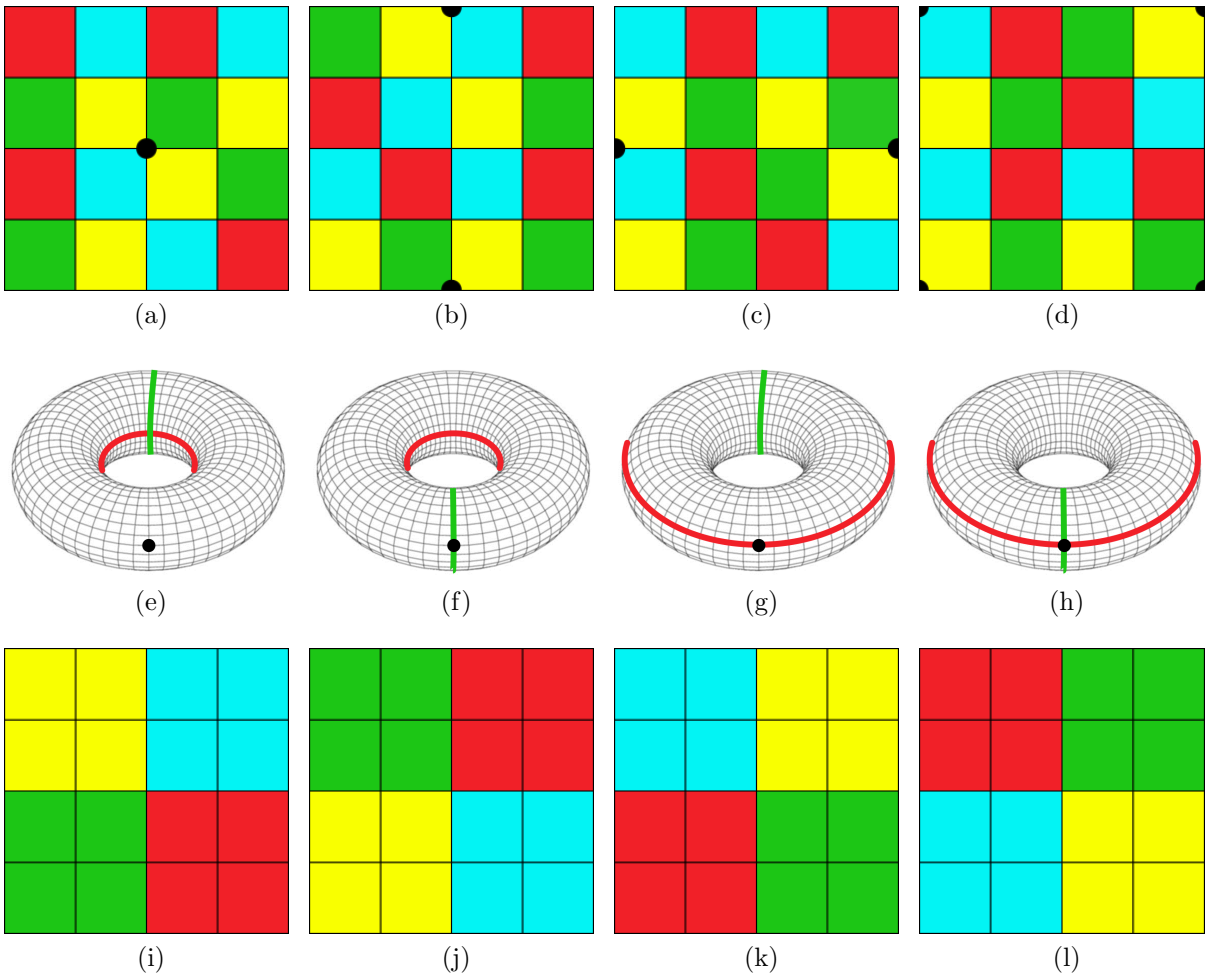


FIGURE 3.3: *Equivalent states produced from a toroidal tiling by "cutting open" the torus differently.*

state shown in Figure 3.3(b).

The third row of images in Figure 3.3 illustrate why the four cutting paths in Figures 3.3(e)–3.3(h) produce equivalent states for the puzzle *Wrapslide*. Each of the states in Figures 3.3(i)–3.3(l) have been produced by taking Figure 3.3(i) as the original state and cutting its torus as indicated in the figure directly above. Therefore, the same "transformation" that was applied to produce the corresponding figure in the first row, was applied to find each of these states. Consequently, the moves required to reach the state in Figure 3.3(j) from the state in Figure 3.3(i) are the same as the moves required to reach the state in Figure 3.3(b) from the state in Figure 3.3(a). It can be seen that, although the states look different, each colour is still grouped into a single quadrant. As the puzzle *Wrapslide* does not require the colours to be grouped into a specific quadrant to achieve a solved state, the states in Figures 3.3(i)–3.3(l) are all considered solved and consequently equivalent for the purposes of finding God numbers.

While the torus model illustrates why the states in Figures 3.3(a)–3.3(d) may be considered equivalent, identifying them by embedding the original state in Figure 3.1(a) on a torus, cutting it in four different ways and embedding each of these tilings in the plane again is a cumbersome process. Fortunately, the same results may be obtained by sliding the $\frac{n}{2}$ left-most cells upward, $\frac{n}{2}$ cells to the left and sliding the entire grid $\frac{n}{2}$ cells upward *and* $\frac{n}{2}$ cells to the left (with toroidal wrapping). The corresponding movement of the (partial) black dot in Figures 3.3(a)–3.3(d)

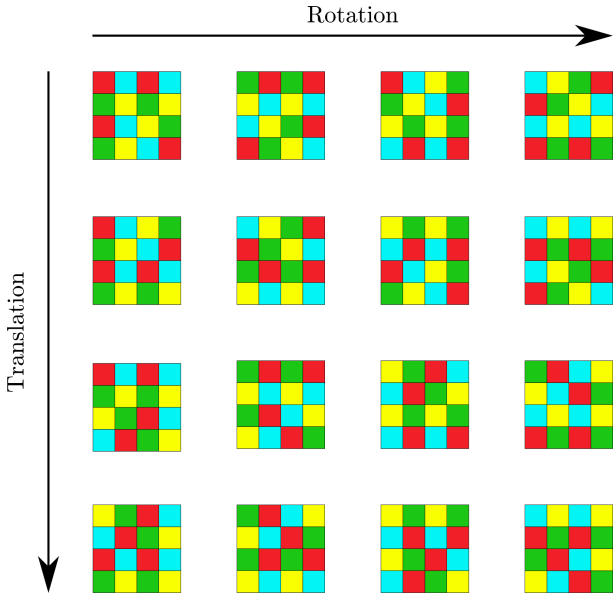


FIGURE 3.4: *Sixteen* Wrapslide *states that are isometric to the state in Figure 3.1(a) produced by successive* 90° *clockwise rotation and translation. The states shown in the first row are the states produced by clockwise rotation of the state in Figure 3.1(a). The states in the second row are produced by moving the right and left halves, of the corresponding state in the first row, two cells up. The states in the third row are obtained by moving the top and bottom halves of the corresponding first row state two cells to the right. The fourth row states are obtained by moving the right and left halves of the corresponding first row state two cells up* and *moving the top and bottom halves two cells to the right.*

illustrates these cell movements.

The transformations described above produce four isometric states for each of the four equivalent states produced by rotation in Figure 3.2. Thus, at this stage, a total of $4 \times 4 = 16$ states that are equivalent to the original state have been identified. These states are shown in Figure 3.4.

The last type of isometry used to identify members of an equivalence class is reflection. Each of the sixteen equivalent states obtained thus far may be reflected around a vertical axis through the grid centre within its plane to produce a total of thirty two equivalent states. Once again, a practical motivation for the equivalent states thus produced follows by visualising each state in Figure 3.4 on a smartphone. The reflected state shows what the grid would look like if it were possible to view the puzzle from the back of the smartphone. Reflecting each of the sixteen equivalent states in Figure 3.4 therefore gives rise to a total of $16 \times 2 = 32$ equivalent states, as shown in Figure 3.5.

These are the only *transformations* that result in equivalent states. Further equivalent states may, however, be obtained by considering alternative colour combinations. Since the specific quadrant into which a colour is grouped in the solved state is irrelevant (as long as each cell of that colour is in the same quadrant), swapping, for instance, all the red cells with all the blue cells will not affect the number of moves required to reach a solved state. Therefore, the states in which the positions of the blue and red cells are reversed may be considered equivalent. Permuting the colours, it follows that $4! = 24$ equivalent states arise for each of the thirty two equivalent states in Figure 3.4, giving rise to a total of 768 states that are all equivalent to the state in Figure 3.1(a). These $32 \times 24 = 768$ states form an equivalence class whose members may be considered the same for the purposes of computing God numbers.

The number of states in such an equivalence class will, of course, be smaller for smaller grids

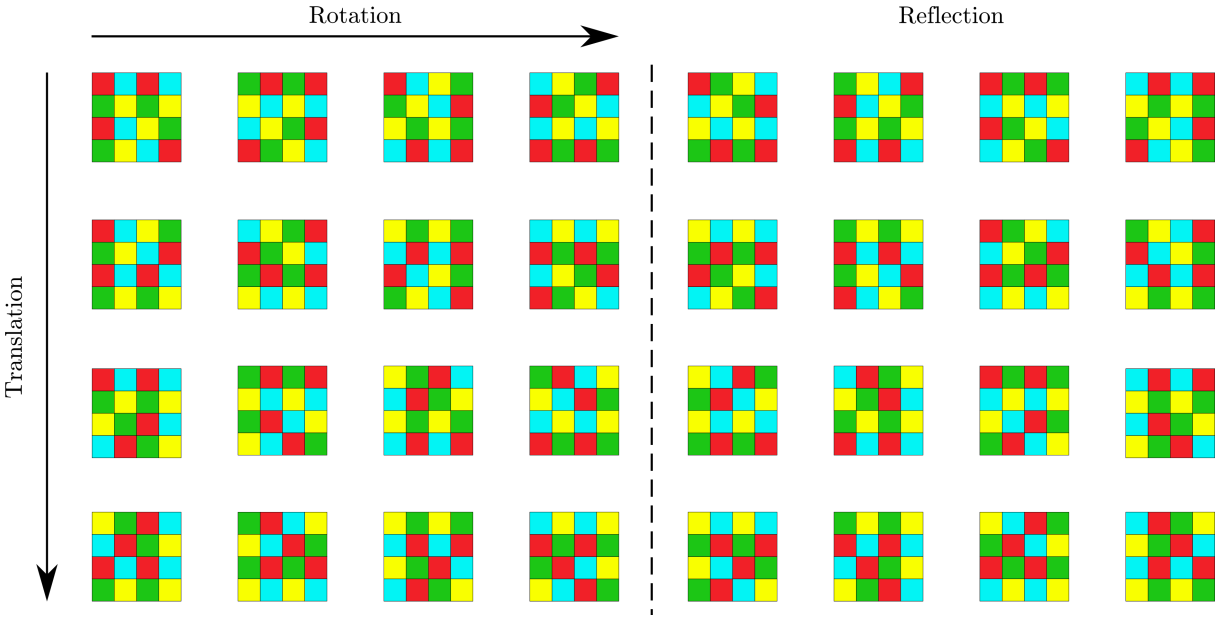


FIGURE 3.5: *Thirty two* Wrapslide *states that are isometric to the state in Figure 3.1(a), produced by rotation, translation and reflection. The states to the left of the dashed line are produced as described in the caption of Figure 3.4. The states to the right of the dashed line are obtained by reflecting the states on the right about an axis represented by the dashed line (within the page). Thus the right-most state in any row of the figure is the reflection of the left-most state in that same row about a vertical axis within the page through its centre point.*

or for grids containing fewer colours. For instance, each of the states of a 4×4 three-colour grid will have at most $32 \times 2! = 64$ states in its equivalence class. Figure 3.6 illustrates an example of an instance where the transformations that are employed to produce an equivalence class generate identical states. For this instance permuting the yellow and green cells will also produce identical states. Therefore, the number 64 in this instance is, in fact, an upper bound on the number of states in an equivalence class rather than the exact number.

3.3 Generation of a Branch and Bound State Enumeration Tree

It has been established that, in order to find the value of $\Theta_W(n, c)$, the optimal move sequence for every possible mixed state of an $n \times n$ c-colour *Wrapslide* grid must be considered (either implicitly or explicitly). This can be done by constructing a so-called *game tree* for every possible state of a given *Wrapslide* grid.

A game tree [1] is an example of a B&B search tree where the root of the search tree represents the mixed state to be solved, the branching procedure represents the moves that may be executed and the search terminates when the solved state is encountered. Either the DFS or the BFS protocol may be used for the construction of a game tree, although the DFS protocol does not facilitate the elimination of duplicate nodes as effectively as the BFS protocol, may become stuck along a path that does not contain the solved state and hence fail to terminate and, most importantly, is not guaranteed to yield an optimal move sequence (a shortest move sequence to the solved state). The BFS protocol, on the other hand, will always find the shortest path from the root of the search tree to the optimal solution (see §2.3). Therefore, in the context of a *Wrapslide* game tree, the BFS protocol will find a shortest move sequence between the given mixed state and the desired solved state. Figure 3.7 contains an excerpt of the game tree for

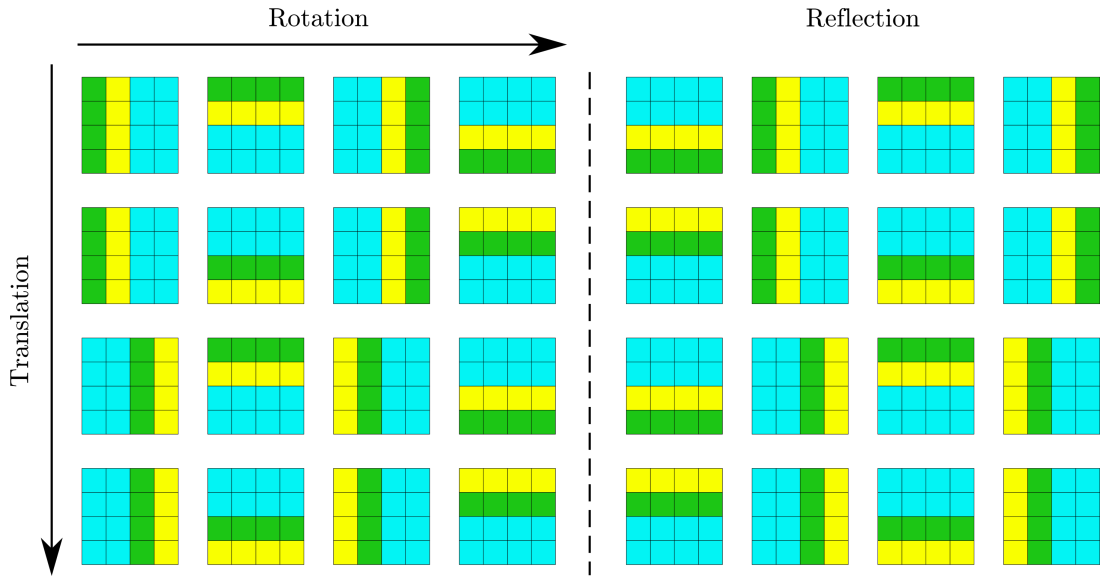


FIGURE 3.6: *Thirty two* Wrapslide *states that are isometric to the* 4×4 *three-colour state in the top left corner, produced by rotation, translation and reflection. The states were generated according to the method described for Figure 3.5. Note that there are only eight distinct states in the figure and hence any state in the figure is a member of an equivalence class of cardinality eight.*

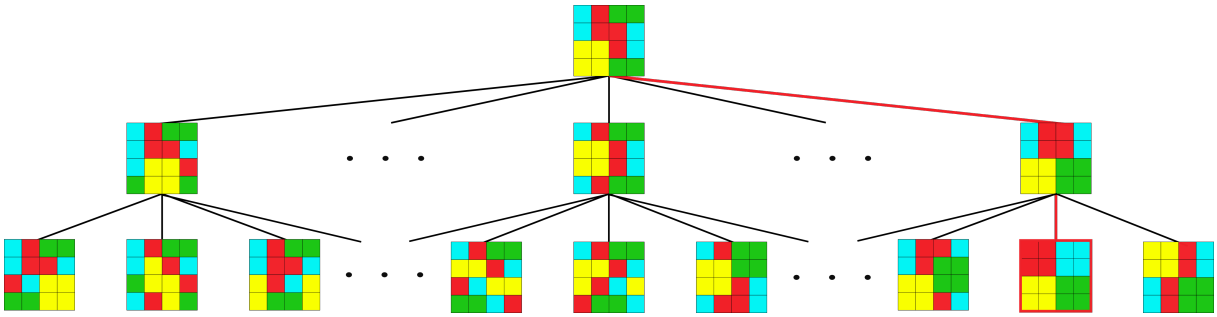


FIGURE 3.7: *An example of a game tree for a mixed* 4×4 *four-colour* Wrapslide *state.*

a 4×4 four-colour mixed state with an optimal move sequence (consisting of two moves) for reaching the solved state indicated in red.

It is clear that the breadth of the search tree will increase exponentially as the lengths of optimal move sequences (the depth of the search tree) increases. More specifically, when no equivalent states are eliminated from the game tree, a branching factor[2] of $(n-1)4$ can be expected. The rate of expansion of the breadth of the game tree may, however, be reduced by employing the techniques described in the previous sections. These techniques are implemented to design a branching procedure which eliminates any equivalent states from the search tree. The branching procedure accomplishes this by performing two steps every time it is executed.

The first step involves generating all the *states* (as opposed to class representatives) that can be reached from the current node by executing a single move (these states are referred to as the *child states* of the node). The number of child states produced for a node depend on the size of the grid. Since a player can choose to move the top, bottom, left or right half of an $n \times n$ grid 1 to $n-1$ cells up, down, left or right (moves involving more than $n-1$ cells will produce duplicate states), $4(n-1)$ not necessarily distinct child states are obtained for the current node

[2]The *branching factor* refers to the number of child nodes generated for a given node [1].

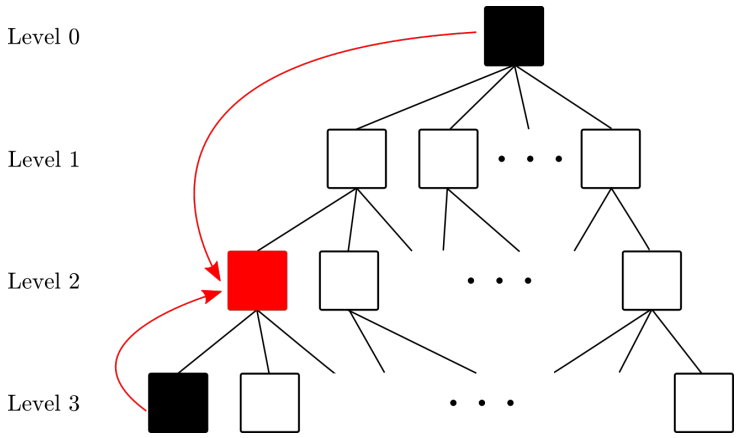


FIGURE 3.8: *A conceptual illustration of a state enumeration tree which shows why it is impossible for a class representative to reappear more than two levels below its first occurrence, if the state enumeration tree is generated correctly. Each block denotes a class representative, the black blocks denote the same class representative and the red block denotes a class representative one move away from the class representative denoted by the black blocks. Since the class representative denoted by the red block is only one move away from the class representative denoted by both black blocks and the BFS protocol always produces the shortest path between two nodes (when duplicates are discarded) the class representative denoted by the red block should already appear in Level 1.*

by executing single moves.

These child states are not necessarily class representatives, nor do they necessarily belong to different equivalence classes, which results in an unnecessarily large game tree. To remedy this, the second step of the branching procedure involves converting each child state to its class representative and removing any duplicates from the current node's list of child class representatives. In effect, siblings are compared for equivalent states during this second step of the branching procedure. These steps are repeated when the next node in the level is visited as dictated by the BFS protocol. If, however, the most recent node to be branched upon is the last node of the current level, an additional test is performed before proceeding to class representatives in the next level. This test involves comparing all the class representatives in the next level (the children of the current level's nodes) to each other, the nodes of the current level and the nodes of the previous level. Once again, any duplicates are discarded. The remaining children make up the next level of nodes. The same results are thus obtained as would have been found if the child nodes were to be compared to all the higher level nodes in the search tree, but as a significant decrease in computation and storage requirements. Figure 3.8 illustrates why it is impossible for a class representative to reappear more than two levels below its first occurrence, *if* the tree is generated correctly. The black blocks in the figure represent repeating class representatives. Since the black blocks in Level 0 and 3 are the same class representative, the red block in Level 2 must be one move away from both black blocks. Furthermore, the BFS protocol produces the shortest path between any two nodes when duplicates are discarded. Thus, the class representative denoted by the red block should already have appeared in Level 1, a contradiction. Consequently, it is not necessary to test for duplicate class representatives more than two levels higher than the level currently being considered.

If no new nodes originate from a current level node, the node is fathomed. The branching procedure is repeated until there are no more unfathomed nodes or the class representative of the solved state is reached.

A method of solving any mixed state within a minimum number of moves has thus been es-

tablished — a solution of the root state of the tree, involving the smallest possible number of moves, may be traced out as the unique path from the root of the game tree to the class representative of the solved state on the largest numbered level of the tree. This procedure alone is, however, not enough to find the value of the God number $\Theta_W(n, c)$. As mentioned before, minimum numbers of moves to the solved state from *every* mixed state must be known in order to establish the value of $\Theta_W(n, c)$. It is, however, not necessary to construct a game tree for every possible mixed state. Rather, game trees need only be generated for each state equivalence class (taking the class representative as the root of the search tree), because the minimum number of moves required to solve a mixed state is the same for each member of its equivalence class. Nevertheless, constructing a game tree for each equivalence class of the grid is at present computationally infeasible. For instance, the 6×6 four-colour grid has

$$\frac{36!}{(9!)^4(32)4!} = 27\,933\,271\,180\,033\,000$$

equivalence classes. Generating such a large number of game trees, some of which may already be quite extensive, is not realistic given the storage and processing limits of the computing technology currently available.

In order to circumvent this problem, two modifications are made to the game tree construction process so as to generate an entire state enumeration tree. First, the solved state is instead selected as the root of the search tree. Secondly, the branching procedure no longer terminates when a certain state (the solved state in the context of the game tree) is reached. Rather, the recursion stops when there are no more unfathomed nodes. Since each mixed state may be reached from the solved state [10], the process will only terminate once the search tree contains each class representative of the grid. Furthermore, as was shown for game trees, the depth of a node in the state enumeration tree is the minimum number of moves required to solve any state in the equivalence class represented by that node. The mixed state(s) with the largest number of moves in their optimal solution sequence will, therefore, be represented by a node in the largest numbered level of the state enumeration tree, called the *height* of the tree. It follows that $\Theta_W(n, c)$ equals the height of the state enumeration tree. An example of a state enumeration tree generated by this method is shown in Figure 3.9.

3.4 Selection of Class Representatives

The process of generating a *Wrapslide* state enumeration tree discussed in the previous section utilises the notion of a class representative, as described in §2.1. Section 2.1 does not, however, contain an explanation of how the class representative is chosen. This choice is, of course, arbitrary. The establishment of a convention by which a class representative is chosen may nonetheless facilitate an efficient computer implementation of the methods described. The following convention was adopted for class representative selection in this project: the class representative of a given equivalence class is the state, within that equivalence class, with the smallest corresponding decimal value.

Finding and comparing the decimal value of each state in an equivalence class may, however, become very time-consuming when the equivalence class is sizeable. This concern is addressed by exploiting a certain characteristic of the method used to find the decimal value of a state.

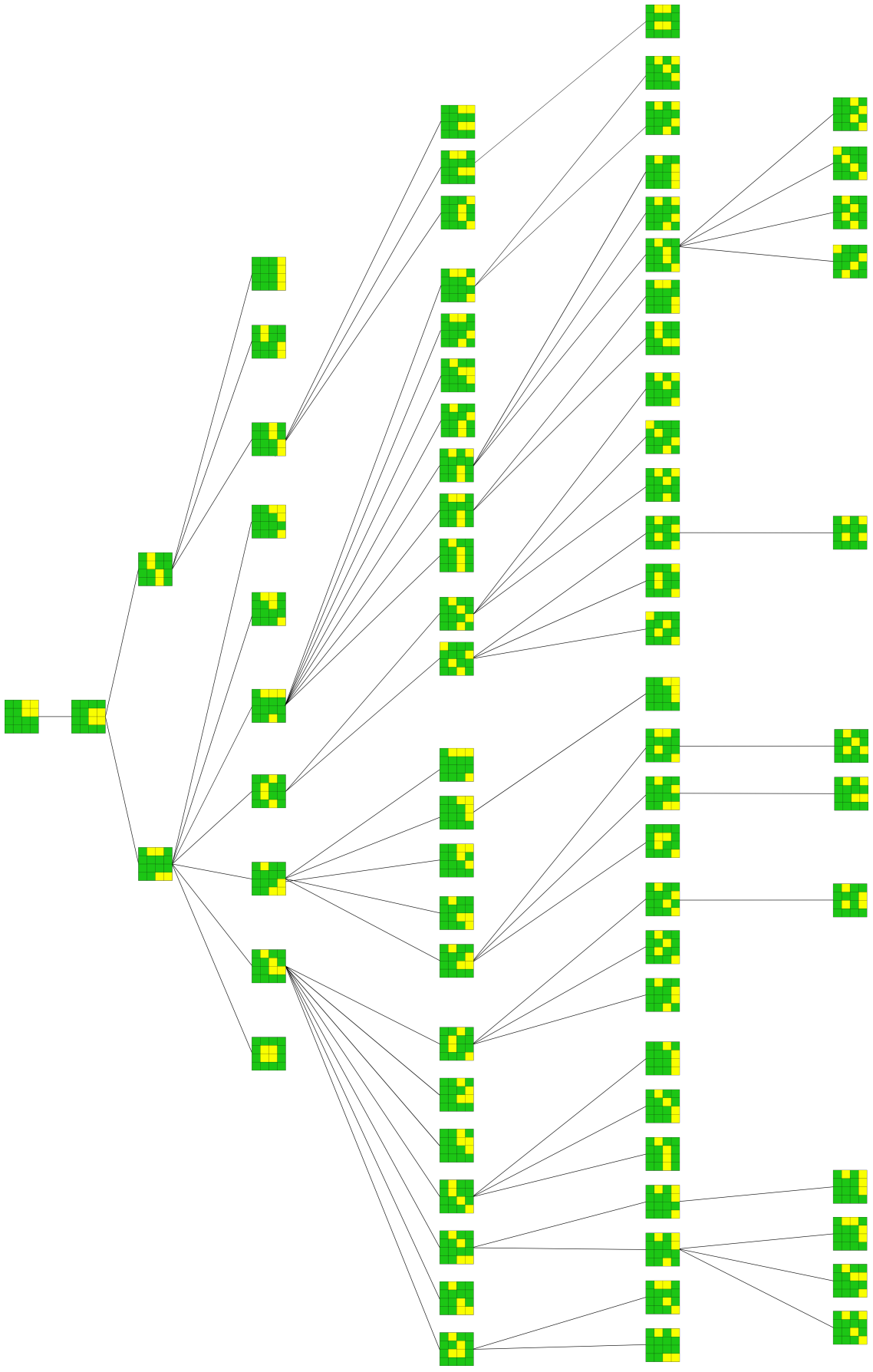


FIGURE 3.9: *The state enumeration tree for the* 4×4 *two-colour* Wrapslide *grid. The height of this tree is six, yielding the God number* $\Theta_W(4,2) = 6$.

Consider the array

$$\boldsymbol{A} = \begin{bmatrix} 1 & 3 & 1 & 3 \\ 2 & 4 & 2 & 4 \\ 1 & 3 & 4 & 2 \\ 2 & 4 & 3 & 1 \end{bmatrix}$$

which represents the game state in Figure 3.1(a) according to the colour enumeration convention of Table 3.2. A total of $4! = 24$ states equivalent to that in Figure 3.1(a) may be obtained by means of colour permutations. These twenty four equivalent states are shown in Figure 3.10. It is evident that switching, for instance, all the blue and green cells or all the red and blue cells will result in equivalent states. These specific permutations result in the states shown in Figures 3.10(b) and 3.10(j), respectively.

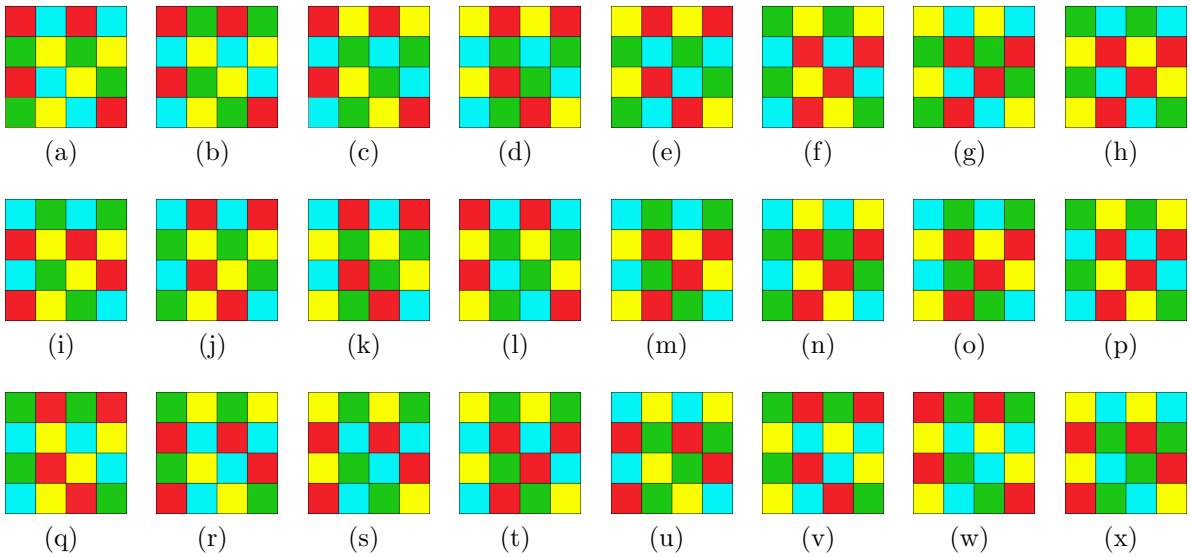


FIGURE 3.10: *Twenty four equivalent states of a* 4×4 *two-colour* Wrapslide *scrambled state produced by considering all the possible colour permutations for the given state.*

It follows that the array obtained by switching the positions of, for instance, all the twos and threes or all the ones and twos in $\boldsymbol{A}$ represents another array also representing an equivalent state to that in Figure 3.10(a). The decimal value associated with each of these arrays will, however, differ. In order to find the smallest decimal value, the numbers in the array must be permuted so that they appear in the array in ascending order when the array is read lexicographically. For instance, in $\boldsymbol{A}$ the number 1 appears first, therefore no changes are made. The next number to appear, however, is 3 although a 2 has not yet appeared. Thus, all the twos and threes may therefore be swapped in $\boldsymbol{A}$ to produce the equivalent array representation

$$\boldsymbol{B} = \begin{bmatrix} 1 & 2 & 1 & 2 \\ 3 & 4 & 3 & 4 \\ 1 & 2 & 4 & 3 \\ 3 & 4 & 2 & 1 \end{bmatrix}$$

of a state equivalent to that represented by $\boldsymbol{A}$.

To understand why this array will yield the smallest decimal value among all the decimal value representations of states in the equivalence class of the state in Figure 3.10(b), consider the following k-bit binary number to decimal value conversion method. First, the values of 2^x are calculated for $x \in \{0, 1, \ldots k-1\}$. For instance, to convert an 8-bit binary number to a decimal

value, Table 3.3 is constructed. Next, each bit in the binary number is visited, starting at the last digit and working backwards. If the bit is a 1, a value of 2^x is recorded for the x-th bit thus considered. Otherwise, for a 0-digit a value of zero is recorded. The decimal value representation is obtained by summing the recorded values together. Thus, the later a 1 appears in the binary number, the less it increases the value of the decimal number so that the smallest decimal number will correspond to the binary number in which the first 1 appears the latest (when comparing binary numbers of the same length). Referring to Table 3.2, it may be seen that the bit assignment convention adopted will produce a binary number with the first 1 occurring at the latest possible digit when the numbers (form one to four) appear in ascending order in the array representation of a *Wrapslide* state. Employing this method, the class representative decimal value can be identified by finding the minimum of thirty two decimal values, as opposed to considering the full set of 768 decimal values of the equivalence class. It is now possible to determine whether or not two states are equivalent by comparing the decimal values of their class representatives. If the class representatives have the same decimal value representation, the states belong to a single equivalence class; otherwise, they do not.

TABLE 3.3: *Binary to decimal conversion calculations:* $(01001101)_2 = (77)_{10}$.

Binary number	0	1	0	0	1	1	0	1
2^x	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
	128	64	32	16	8	4	2	1
Recorded value	0	64	0	0	8	4	0	1

3.5 God Numbers for Selected *Wrapslide* Grids

The methods described in the previous sections were implemented by the author in Python 3.5 [22] in order to construct state enumeration trees for the 4×4 two-, three- and four-colour *Wrapslide* grids. The numbers of class representatives on each enumeration tree level, the total numbers of class representatives present in each enumeration tree, and the time and memory required to construct each enumeration tree are shown in Table 3.4. The computation times shown in Table 3.4 were obtained on a personal computer with an Intel i7 5820k core, running at 4GHz with 8GB RAM and Dual R9 390 GPUs within a Windows 7 operating system.

It follows from the second column of Table 3.4 that there are twelve 4×4 two-colour *Wrapslide* state equivalence classes which require at least six moves for their solution, but that there are no equivalence classes which necessarily require at least seven moves for their resolution. The following novel result has therefore been established.

Theorem 3.1 *The God number for the* 4×4 *two-colour* Wrapslide *puzzle is* $\Theta_W(4, 2) = 6$.

The following novel result similarly follows from the third column of Table 3.4.

Theorem 3.2 *The God number for the* 4×4 *three-colour* Wrapslide *puzzle is* $\Theta_W(4, 3) = 10$.

As was already mentioned in §2.4, the God number $\Theta_W(4, 4)$ was established in 2014 by BURGER AP [6]. His result may be confirmed from the fourth column of Table 3.4 as follows.

Theorem 3.3 (BURGER AP [6]) *The God number for the* 4×4 *four-colour* Wrapslide *puzzle is* $\Theta_W(4, 4) = 12$.

TABLE 3.4: *The numbers of state equivalence classes in the various levels of the state enumeration trees generated for the* 4×4 *two-, three-, four-colour and* 6×6 *two-colour* Wrapslide *grids. Computation times are represented in the format hours:minutes:seconds.*

Level	4×4 Two-colour	4×4 Three-colour	4×4 Four-colour	6×6 Two-colour
0	1	1	1	1
1	1	3	1	1
2	2	7	3	5
3	10	28	9	47
4	24	107	34	356
5	28	445	126	2 489
6	12	1 684	523	17 484
7	—	5 160	2 261	109 672
8	—	6 260	9 389	525 914
9	—	825	30 472	≥ 1
10	—	3	37 256	?
11	—	—	4 085	?
12	—	—	14	?
13	—	—	—	?
Total state equivalence classes	78	14 523	84 174	$\geq$ 655 968
Computation time	00:00:11.66	01:09:35.61	07:41:31.78	62:07:25.69
RAM requirement	3.46GB	4.02GB	5.32GB	7.09GB

Although the state enumeration tree for the 6×6 two-colour *Wrapslide* puzzle is too large to confirm the God number $\Theta_W(6, 2)$ in this project, the following bound follows from the last column of Table 3.1.

Theorem 3.4 *The God number for the* 6×6 *two-colour* Wrapslide *puzzle is* $\Theta_W(6, 2) \geq 9$.

As mentioned in §2.4, BURGER AP proved that any colour in a 6×6 four-colour grid can be grouped into a quadrant in twelve moves or fewer [6]. This was done in order to help establish an upper bound on the God number for the 6×6 four-colour *Wrapslide* grid. The minimum number of moves required to group the cells of any colour into a single quadrant (for a 6×6 four-colour grid) may also be used to deduce the God number of the 6×6 two-colour *Wrapslide* puzzle.

To elucidate the above statement, consider Figure 3.11. The red cells in the states shown in Figures 3.11(a) and 3.11(c) have the same configuration. Therefore, the same minimum number of moves are required to group the red cells of the states shown in Figures 3.11(a) and 3.11(c) into single quadrants (resulting in Figures 3.11(b) and 3.11(d), respectively). Since the state in Figure 3.11(c) contains only two colours, grouping all the red cells in this state into a single quadrant results in a solved state, as shown in Figure 3.11(d). From the result obtained by BURGER AP that any colour in a 6×6 four-colour grid can be grouped into a single quadrant in twelve moves or fewer and the fact that the red cells in Figure 3.11(a) cannot be grouped into a single quadrant in fewer than twelve moves [6] it may be deduced that solution of the state in Figure 3.11(c) requires at least twelve moves. It follows that the God number of the 6×6 two-colour *Wrapslide* puzzle is $\Theta_W(6, 2) = 12$. This result could not be confirmed in this

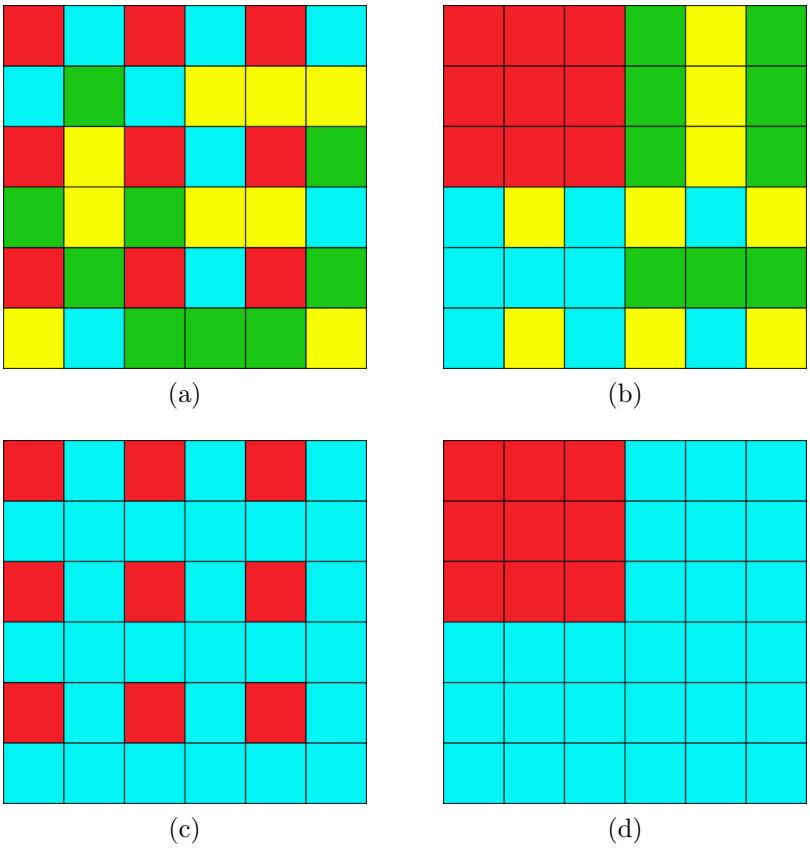


FIGURE 3.11: *Examples of* 6×6 *two- and four-colour* Wrapslide *states whose red cells are identically distributed.*

project due to the limited computer memory available to the author. The value $\Theta_W(6,2) = 12$, however, satisfies the lower bound established in Theorem 3.4.

3.6 Chapter Summary

This chapter was devoted to a description and application of techniques required to generate state enumeration trees containing all *Wrapslide* class representatives. Section 3.1 contained a discussion on a method for representing a *Wrapslide* state as a decimal value. This was followed in §3.2 by an illustration of the techniques used to generate the equivalence class of a given *Wrapslide* state. The focus in §3.3 shifted to the actual construction of *Wrapslide* state enumeration trees that allowed the author to establish the God numbers for two *Wrapslide* grids and confirm that for another grid. The section also included an overview of the notion of a game tree as well as comprehensive descriptions of the branching and termination criteria employed in the generation of such trees. Section 3.4 was devoted to a description of the conventions and methods used to identify class representatives, and the last section, §3.5, saw the establishment of the values of three *Wrapslide* God numbers upon application of the techniques described in the chapter.

CHAPTER 4

A Heuristic Solution Procedure for the Puzzle *Wrapslide*

Contents

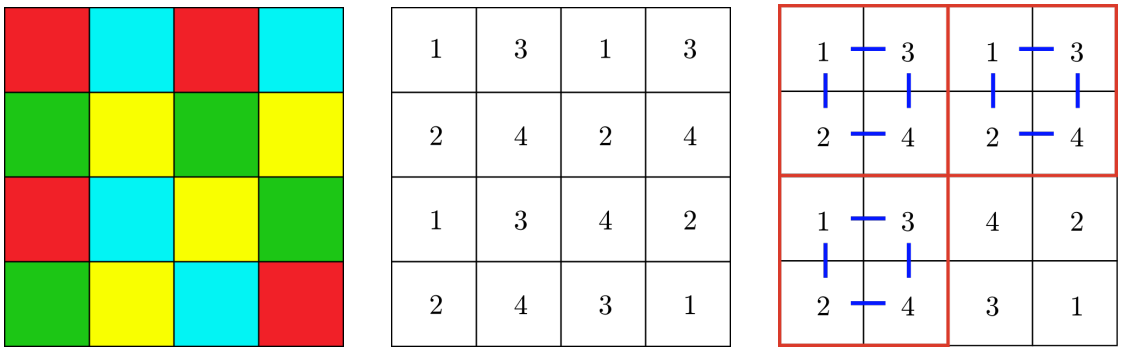
4.1 Assignment of Objective Function Values to *Wrapslide* States 35
4.2 Proposed Heuristic for Solving *Wrapslide* States 36
4.3 A Worked Example 41
4.4 Chapter Summary 46

This chapter contains a description of the methodology developed to solve a *Wrapslide* mixed state in an optimal or near-optimal number of moves. It opens in §4.1 with an explanation of the approach used to associate an objective function value with a *Wrapslide* state which measures the degree of colour heterogeneity in each of the state quadrants. A heuristic solution procedure is next put forward in §4.2. The functioning of the heuristic solution procedure is partitioned into two components: a game tree component and a state enumeration tree component. The branching and termination procedures implemented in the game tree component of the heuristic and the technique used to find the moves contained in an optimal solution sequence provided by the state enumeration tree component are described in some detail. The focus of the section finally shifts to the potential contribution of such a heuristic in respect of the establishment of unknown God number values for *Wrapslide* grids. The chapter finally closes with a brief summary in §4.3.

4.1 Assignment of Objective Function Values to *Wrapslide* States

The development of a heuristic solution procedure for solving a mixed *Wrapslide* state requires a means of measuring the degree of colour heterogeneity in each of the state's four quadrants. This may be accomplished by the assignment of an objective function value (described in §2.3) indicative of the degree to which a given mixed state resembles a solved state.

In this project, the objective function value associated with a state is determined by considering the numerical representation of the state, as shown in Figure 4.1(b). The objective function value of the state represented by the numerical representation is the sum of the absolute differences between adjacent cells in the various quadrants. This sum is taken over the top two quadrants (which are henceforth referred to as quadrants one and two) and the lower-left quadrant (henceforth referred to as quadrant three). For instance, the objective function value of the



(a) (b) (c)

FIGURE 4.1: *An objective function value associated with a* Wrapslide *state.*

state in Figure 4.1(a) may be obtained by summing together the absolute differences between the numeric values joined by blue lines in Figure 4.1(c) for each of the first three quadrants (whose boundaries are indicated in red). This yields a value of $|1-3|+|1-2|+|2-4|+|3-4|=6$, $|1-3|+|1-2|+|2-4|+|3-4|=6$ and $|1-3|+|1-2|+|2-4|+|3-4|=6$ for the first, second and third quadrant, respectively. The objective function value associated with the state in Figure 4.1(a) is therefore $6+6+6=18$.

States with small objective function values are expected to require fewer moves, on average, in order to reach a solved state than states with large objective function values. This intuitive expectation is based on the fact that an objective function value of zero would require the sum of the absolute differences between all the numeric values in a quadrant to be zero, for each of the first three quadrants. This will be the case when, for three of the four numeric values, all the cells with a given numeric value (and consequently all the cells of a certain colour) are grouped together in a quadrant. Only three quadrants are considered in the computation of the objective function value associated with a game state since, if three of the quadrants' sums of absolute differences equals zero, then the fourth quadrant cannot contain cells with differing numeric values (and so a solved state has been reached).

4.2 Proposed Heuristic for Solving *Wrapslide* States

The heuristic developed to solve a mixed state of the puzzle *Wrapslide* in an optimal or hopefully near-optimal number of moves consists of two components. The first component employs the B&B algorithm (described in §2.3) and the BFS tree traversal protocol (also described in §2.3) to generate pairs of game trees which, as mentioned in Chapter 3, take a user-specified mixed state as the root of each game tree. An example of such a game tree pair is shown in Figure 4.2, where the red block denotes the mixed state to be solved. The tree generation procedure used is similar to the generation procedure described for game trees in Chapter 3. There are, however, some key differences in respect of the branching and termination processes. First, while the branching procedure once again generates child states that may each be reached by executing a single move on the parent state, each level of the search tree is now generated by executing *either* vertical (that is to say sliding the left or right half of the puzzle up or down) *or* horizontal moves (sliding the top or bottom half of the puzzle to the right or left). The type of move (vertical or horizontal) according to which states are generated alternates between the levels of each tree in the game tree pair. Thus, if the states in the current level were generated by executing horizontal moves on states in the previous level, then the states in the next level are generated by

executing vertical moves on states in the current level. The first level of the branching procedure in the first game tree generates states that may be reached from the root state by executing horizontal moves (as indicated by the directions of the arrows in Figure 4.2(a)), while the first level of the branching procedure for the second game tree generates states that may be reached from the root state by executing vertical moves (as indicated by the directions of the arrows in Figure 4.2(b)).

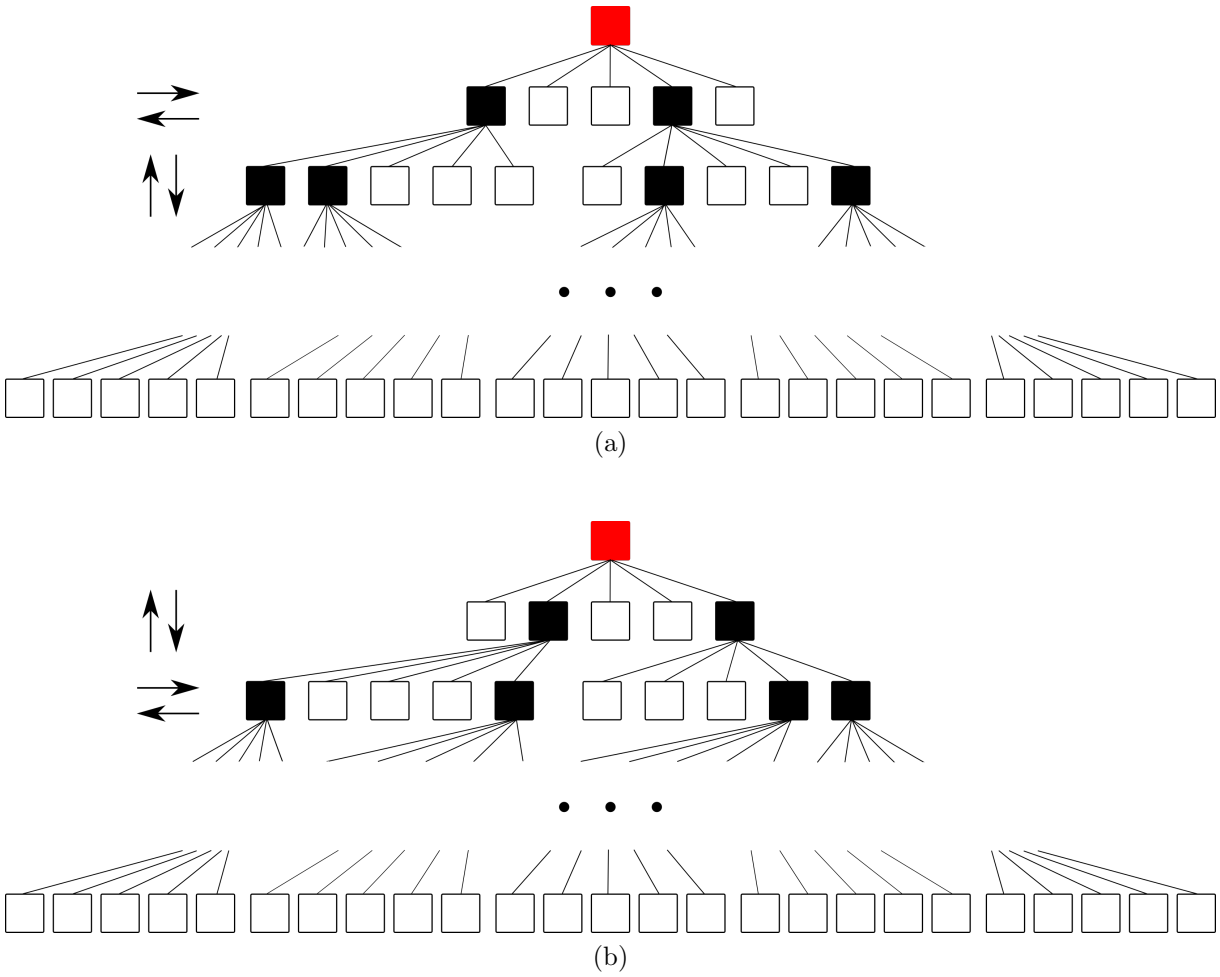


FIGURE 4.2: *A conceptual representation of a pair of partial game trees generated by alternately executing either vertical or horizontal moves for the m_i best states in the level i, where $m_1 = 2$ and $m_2 = 4$. The mixed state to be solved is denoted by the red block, the best states in each level are denoted by black blocks and the types of move used to generate the states in levels 1 and 2 of each game tree are indicated by the directions of the arrows. (a) The states in the odd numbered levels (the root lies in level 0) are generated by horizontal moves, while the states in the even numbered levels are generated by vertical moves. (b) Conversely, the states in the odd numbered levels are generated by vertical moves and the states in the even numbered levels are generated by horizontal moves.*

The branching procedure also differs in another regard. Instead of executing the branching procedure for *each* state in a level, as is done for game trees, the branching procedure is only applied to the m_i best states in the i-th level, where the descriptor *best* refers to the states associated with the smallest objective function values. For instance, in Figure 4.2 the branching procedure is applied to the $m_1 = 2$ and $m_2 = 4$ best states (represented by the black blocks) in levels one and two, respectively. This approach is followed to reduce the computation time of the heuristic. The values of m_i adopted in this project for 6×6 game states containing two, three or four colours are shown in Table 4.1. These values were obtained empirically by weighing the

improved solution quality associated with larger values of m_i against the accompanying increase in computation time. For instance, using the m_i-values in Table 4.1, it was found that the final solution quality decreased imperceptibly (when compared to the application of the branching procedure to *every* state of a level), while the computation time decreased significantly from fifteen minutes to approximately 0.8 seconds for a six-level game tree of one of the 6×6 grids.

TABLE 4.1: *The number of states, m_i (where i denotes the level), for which the branching procedure must be executed in different levels of the partial game trees generated for the heuristic solution procedure in the cases of* 6×6 *two-, three- and four-colour* Wrapslide *grids.*

m_i	Two colours	Three colours	Four colours
m_1	3	3	3
m_2	8	8	8
m_3	13	13	13
m_4	16	16	16
m_5	—	25	25
m_6	—	30	30
m_7	—	—	40
m_8	—	—	55

The tree generation process terminates when the game tree reaches a predetermined maximum depth or when the solved state, represented by a green block in Figures 4.3–4.4, is found. Recommended maximum allowable depths of the game trees were determined by considering the average rate of convergence to a minimum objective function value and the increase in computation time for larger game trees. Maximum allowable depths of four, six and eight were established empirically as suitable for the 6×6 two-, three- and four-colour *Wrapslide* grids, respectively.

If the tree generation process terminates because the search trees have reached the specified maximum depth, the k states with the smallest objective function values, represented by purple blocks in Figures 4.3–4.4, are chosen as the roots of k new pairs of game trees. The selection of k states, as opposed to the single state with the smallest objective function value, somewhat helps to prevent the heuristic from becoming trapped in local optima. The process is repeated until a state with an objective function value of zero (*i.e.* a solved state) is found. The values of k adopted in this project for the 6×6 two-, three- and four-colour *Wrapslide* grids are ten, fifteen and twenty, respectively.

Figure 4.3 illustrates how this technique may yield an optimal or near-optimal move sequence according to which a given game state may be solved. The red block in the figure once again represents the original mixed state, taken as the root of the first pair of search trees, while the purple block represents one of the k best states, selected from the first pair of search trees, once they had reached the maximum allowable depth (note that, not all of the k best states will necessarily lie in the last levels of the search trees, as in this example). The process terminates when the solved state, represented by the green block, is reached. The solution sequence, therefore, consists of the moves represented by the bold red branches. The first component of the heuristic solution procedure is therefore, in fact, analogous to the construction of a partial game tree. This partial game tree is generated by selecting and integrating several incomplete game trees generated during the execution of the solution procedure.

Despite the selection of k best states for further inspection, as opposed to only the best state, there remains a considerable probability that the heuristic might become trapped at states with small objective function values, from which the solved state cannot be reached within a number

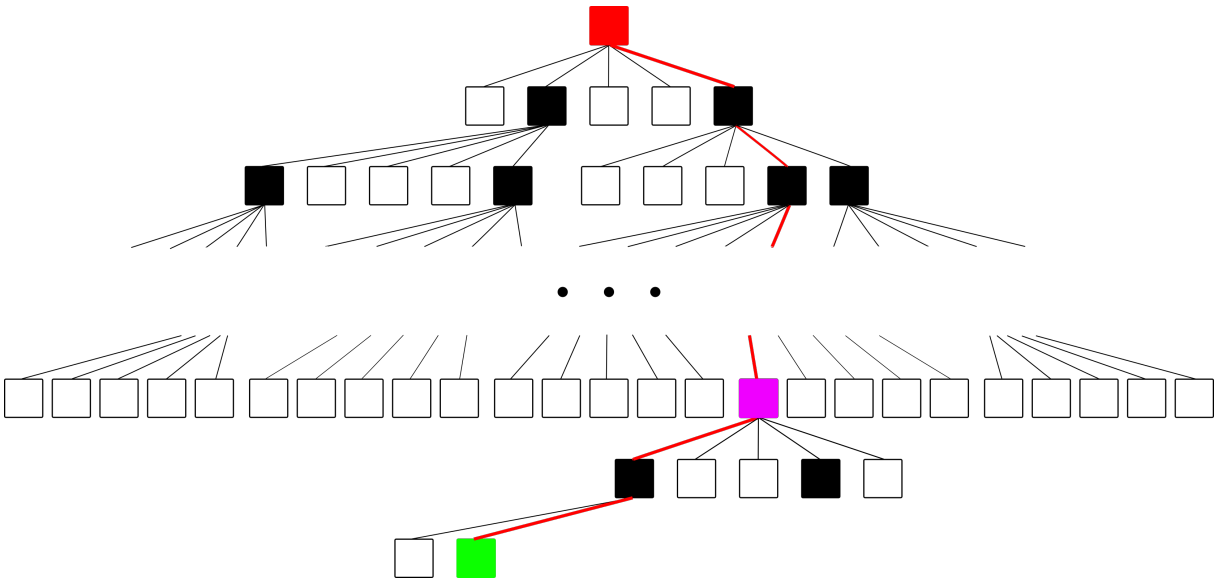


FIGURE 4.3: *A conceptual representation of the execution of the first component of the heuristic solution procedure, showing how a solution path (indicated by the red branches) may be obtained.*

of moves fewer than or equal to the specified maximum depth of a single search tree. In such cases, the solved state will never be reached and the heuristic will enter an infinite loop. This problem is remedied by the second component of the heuristic which makes use of incomplete state enumeration trees rather than game trees. While the state enumeration trees of the 6×6 and 8×8 grids are too large to generate in full (due to a limited memory capacity), the methods described in Chapter 3 may be utilised to generate the first few levels of these state enumeration trees. Since these state enumeration trees will look the same each time they are generated, the incomplete state enumeration tree of a given grid need only be generated once and stored. The relevant stored incomplete state enumeration tree may then be consulted whenever the infinite loop situation described above is encountered. Once the objective function values of the k best states have not decreased for a specified number of iterations of the partial game tree generation procedure described above, these k best states are converted to their class representatives and compared to the class representatives contained in the stored state enumeration tree. If one or more of the class representatives are present in both the list of class representatives of the k best states generated by the first component of the heuristic and the relevant partial state enumeration tree of the second component, the length of a heuristic move sequence may be determined according to which the original state can be solved.

To elucidate the above description, consider Figure 4.4. Once again, the red block in the figure represents the mixed state to be solved and the purple block denotes one of the k best states selected to be the root of a new pair of partial game trees. The green and grey blocks in the lower levels of the figure denote an upturned version of the first levels of the relevant partial state enumeration tree. Therefore, optimal move sequences from each of the states in these lower levels to a solved state, are known. The blue block represents the class representative of one of the k best states generated by the first component of the heuristic which is also contained in the stored state enumeration tree. Since a move sequence from the mixed state to the state represented by the blue block is known and an optimal move sequence from the solved state to the state represented by the blue block is known, a move sequence from the mixed state to the solved state is also known. This path, shown in red, provides an upper bound on the number of moves required to solve the original mixed state.

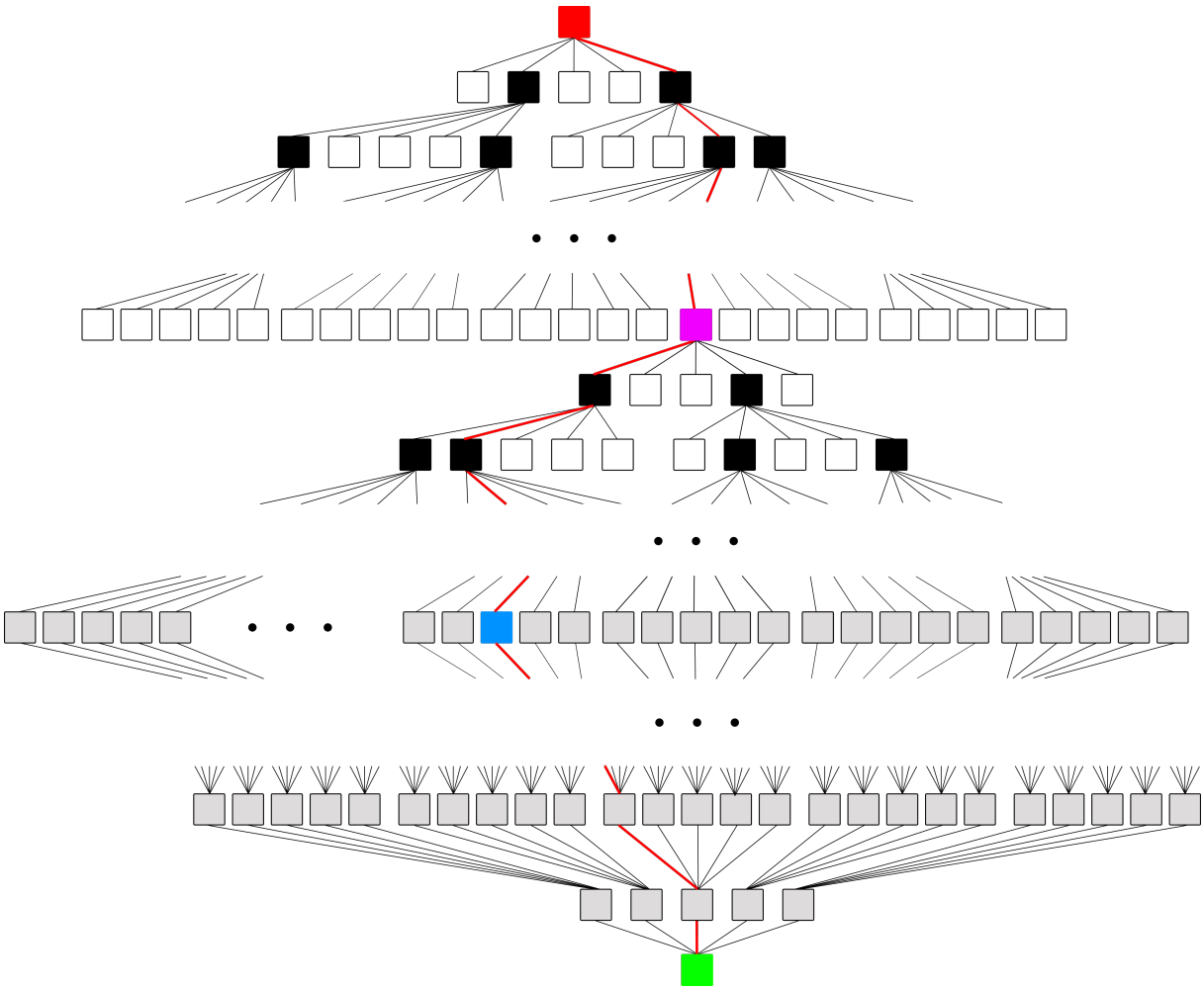


FIGURE 4.4: *A conceptual illustration of the integration of the output generated by the first component of the heuristic (denoted by the red, black and white blocks) and a stored partial state enumeration tree (denoted by the green, grey and blue blocks) yielding an upper bound on the number of moves required to solve the given (red) mixed state (denoted by the bold red branches).*

The path generated by the first component of the heuristic consists of moves that may be identified simply by comparing two states joined by a single branch of the tree and finding the move which will generate the second state when applied to the first (the moves that need be considered are either horizontal or vertical, depending on the levels in which the states lie). This is not, however, the case when considering the section of the path contained in the (incomplete) state enumeration tree. Recall that the states belonging to a certain equivalence class are considered equal for the purposes of finding God numbers, because their optimal solution sequences are of equal length. Their solution move sequences do not, however, contain identical moves. Consequently, the state enumeration trees do not explicitly specify the actual moves that must be executed in order to reach a solved state. This is evident in the state enumeration tree in Figure 3.9, where, for instance, no single move can be executed to reach the first state in the third level from its parent state in the second level. Nevertheless, as was mentioned in §3.2, a solution move sequence involving the smallest number of moves may be found by considering the strategy obtained when tracing the unique path from the root of the state enumeration tree to the current state. This path is, essentially, an ordered list of the class representatives of the states that lie along an exact solution sequence. Thus, the exact moves in the solution sequence may be found by means of the following method. Given the ordered list

of class representatives, start at the mixed state and sequentially consider two adjacent states, call them A (the current state) and B (the succeeding state), at a time. Find all the states belonging to B's equivalence class and generate all the child states that result when executing a single move from each of B's equivalent states. Store the move associated with each child state. Suitable moves may now be determined by searching for duplicates of A in the list of child states and storing the relevant move. Finally, execute this move on A to find the new orientation of B. This process must be repeated, each time taking the previous B as the new A and obtaining the new B from the list of class representatives, until the solved state is reached. Combining this move sequence with the move sequence obtained as output from the first component of the heuristic, a complete move sequence may be generated.

Despite this additional computational burden, consulting the stored (incomplete) state enumeration tree is significantly faster than generating a larger number of game tree generation iterations. Also, the state enumeration tree method always gives the smallest number of moves required to reach the solved state from the current state, while the game tree generation component of the heuristic only produces an upper bound on the minimum number of moves required to solve the given mixed state. Therefore, consulting the state enumeration tree will always yield an equal or superior quality final solution. To exploit this fact, when a 6×6 two-colour state with an objective function value of two is encountered, the heuristic automatically refers to the incomplete state enumeration tree of the 6×6 two-colour *Wrapslide* grid. This is the case because an objective function value of two is only possible when all the colours are grouped into quadrants with only two cells in the wrong quadrant, one of which must be the fourth (bottom-right) quadrant. This state represents a local minimum in respect of the objective function in §4.1. In addition, Burger AP [5] proved that any two cells of the 6×6 grids can be swapped in eight moves or fewer. Two examples of such eight-move sequences that may be used to exchange two cells are shown in Figure 4.5. If the stored state enumeration tree contains (at least) the first eight levels of the complete enumeration tree, every state associated with an objective function value of two will belong to one of the equivalence classes contained in the stored tree and hence an optimal move sequence from the state representing a local minimum described above to the solved state can be found (from the state enumeration tree).

The heuristic may also be used to determine an upper bound on the length of an optimal move sequence without specifying the exact moves contained in this move sequence. The heuristic is capable of providing this result in a fraction of the time required to find the precise moves involved. Since the value of a God number is determined by considering the lengths of the optimal move sequences, not the actual move sequences themselves, this output may be used profitably in pursuit of establishing *Wrapslide* God numbers. For instance, for an unknown God number, the heuristic may be employed to solve a large number of random *Wrapslide* instances so as to determine the distribution of the corresponding optimal move sequence lengths. This distribution may then be compared to the optimal move sequence length distributions of *Wrapslide* grids whose God numbers have already been determined in order to gauge the likely value of or conjecture an upper bound on the unknown God number value. Furthermore, such a distribution may possibly be used to determine the suitability of proposed methods for finding the God numbers of larger *Wrapslide* grids.

4.3 A Worked Example

In order to clarify the working of the heuristic described in the previous section, this section contains a comprehensive worked example of the steps followed to determine the moves associated with the sequence of states provided as output by the two components of the heuristic.

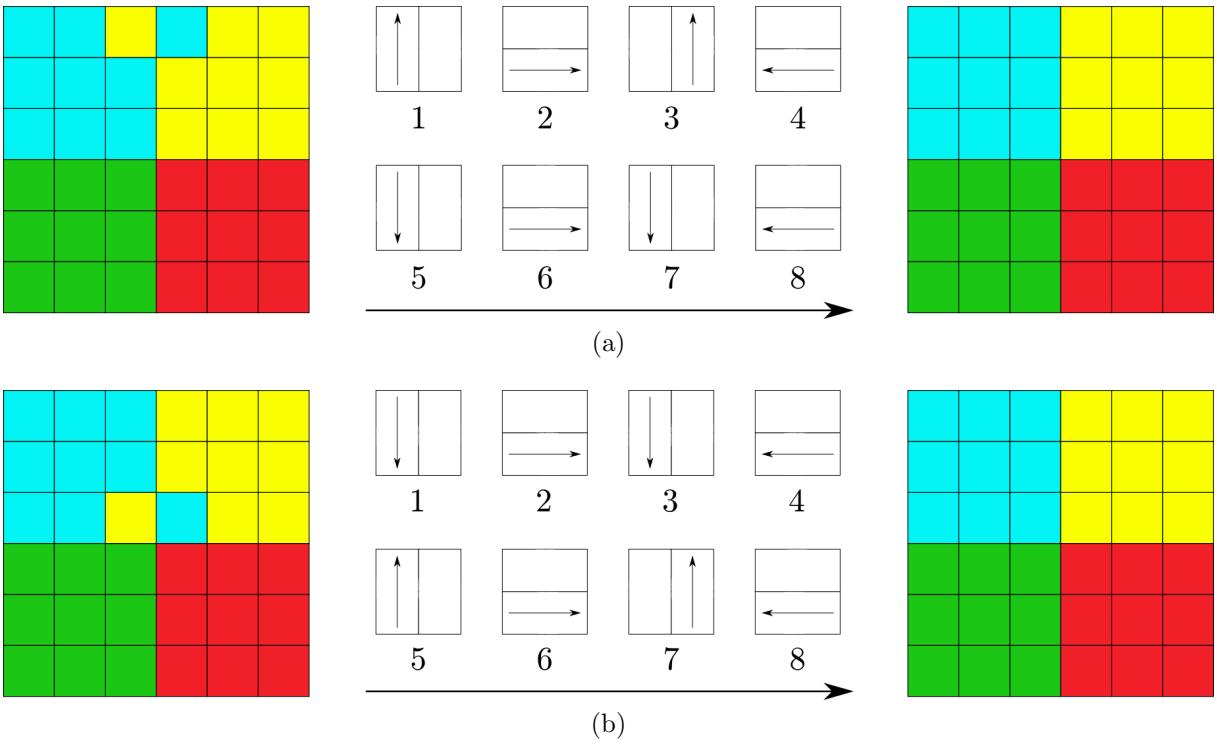


FIGURE 4.5: *Two move sequences that may be implemented to exchange two cells, without changing the groupings of the remaining cells.*

The techniques described in §4.2 are applied to the top-most state in Figure 4.6(a).

The first component of the heuristic generates partial game trees whose integration yield the first section of a solution. This solution section is shown in Figure 4.6(a). The last state in Figure 4.6(a) has an associated objective function value of 3. Three remains the best objective function value when an additional set of game tree pairs is generated, and so the heuristic may be assumed to be trapped at a local optimum. Consequently, the second component of the heuristic is initiated. As mentioned in §4.2, the second component of the heuristic compares the class representatives of the best states generated by the first component of the heuristic to the class representatives in the relevant partial state enumeration tree. In order to compare these states, the $k = 10$ best states generated by the first component of the heuristic must be converted to their class representatives. Figure 4.6(b) shows the last state of the first component of the solution (generated by the first component of the heuristic), outlined in red in Figures 4.6(a)–4.6(b) and henceforth referred to as the *coupling state*. The class representative of the coupling state is outlined in green in Figures 4.6(b)–4.6(c). The heuristic searches for this class representative in the partial state enumeration tree. Once this class representative has been found, the path from that class representative (in the state enumeration tree) to the class representative of the solved states is recorded. The path thus recorded for this example is shown in Figure 4.6(c).

For the states shown in Figure 4.6(a), the exact moves required to reach each state from its parent state are stored while the partial game trees are generated (this may be done since memory requirements do not have to be minimised for the heuristic as for the state enumeration trees). Therefore, no additional computations are required and the first two moves of the move sequence have been identified. They are *bottom half: 3 cells left* and *right half: 5 cells up*. As mentioned in §4.2, however, this is not the case for the path shown in Figure 4.6(c), obtained from the state enumeration tree. The conversion of the solution sequence extracted from the

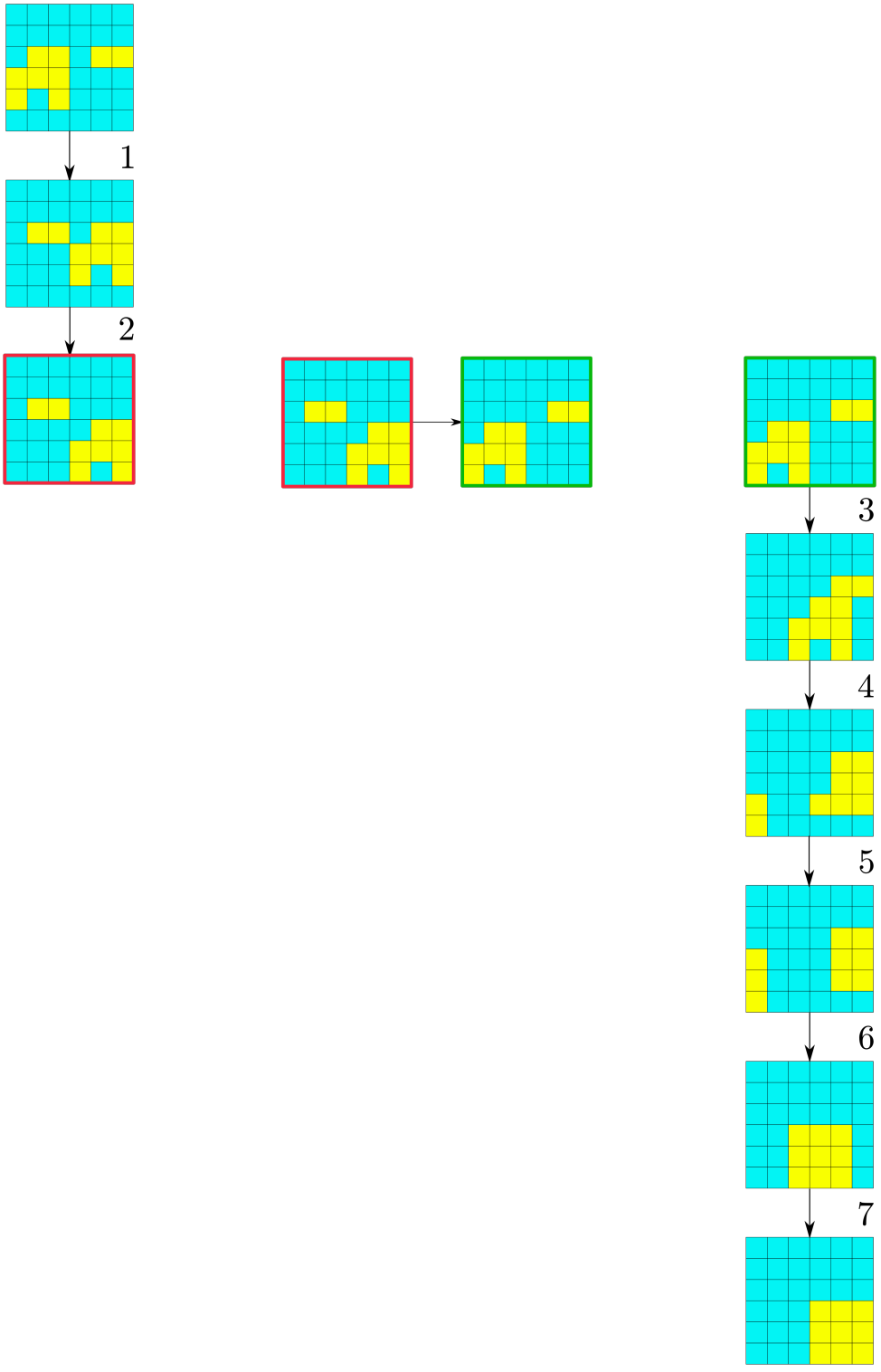


FIGURE 4.6: *(a) A move sequence generated by the first component of the heuristic, (b) the conversion of the coupling state to its class representative and (c) the corresponding list of ordered states obtained from the stored (incomplete) state enumeration tree. The result is a guarantee that the original state can be solved in seven moves or fewer.*

FIGURE 4.7: *Conversion of the ordered list of class representatives (obtained from the state enumeration tree) to generate an actual move sequence of states.*

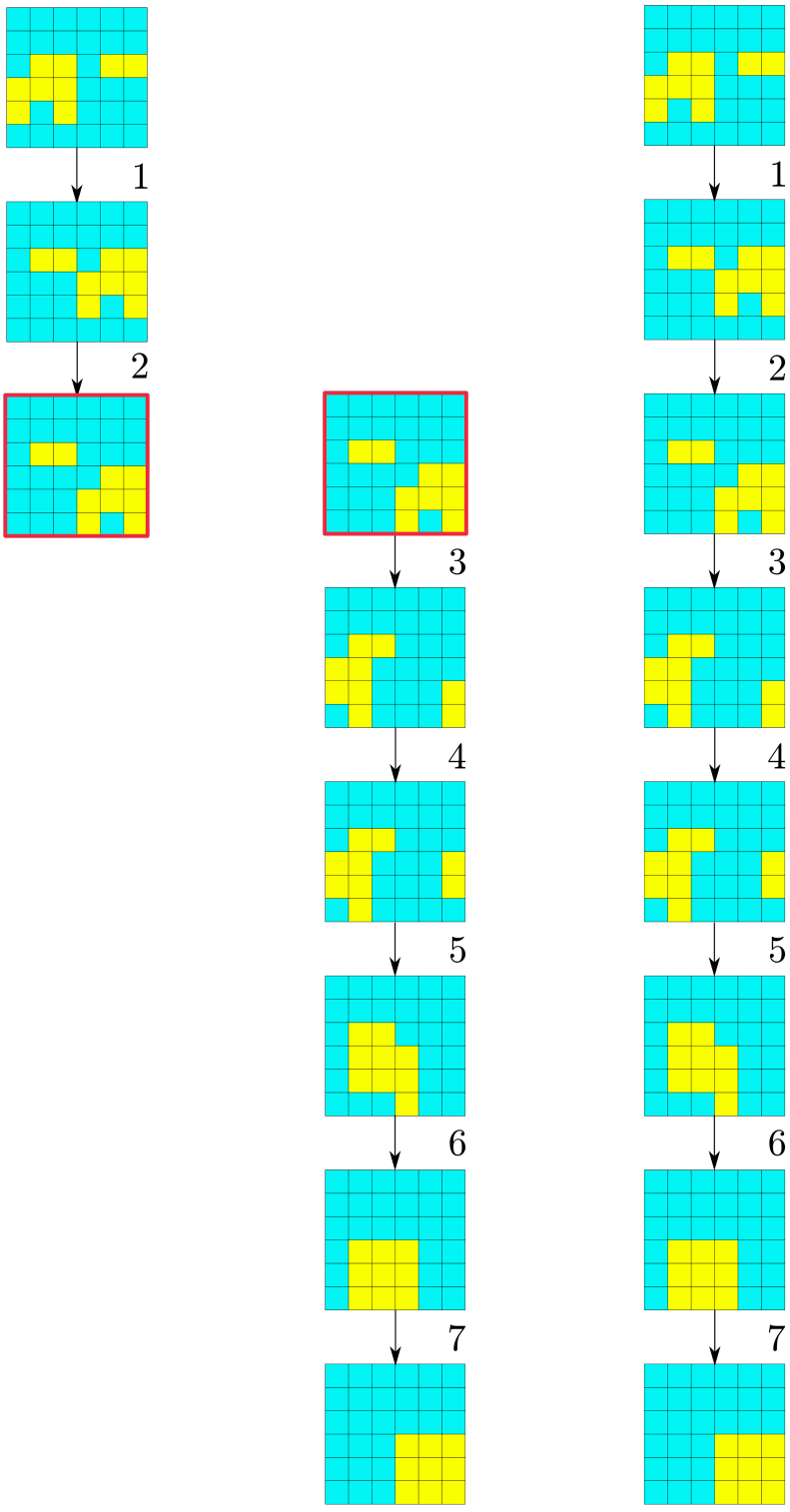


FIGURE 4.8: *Combination of the move sequences produced by (a) the first component and (b) the second component of the heuristic so as to obtain (c) a complete move sequence.*

state enumeration tree path (shown in Figure 4.6(c) and henceforth referred to as the ordered list of class representatives) to the states in the second part of the actual move sequence is illustrated in Figure 4.7.

The first column of states in Figure 4.7 contains the ordered list of class representatives and the second column of states contains the first pair of states (from the ordered list of class representatives) to be considered. Recall, from §4.2, that the two states taken as input for each iteration of the method used to convert the list of ordered class representatives to the states of an actual move sequence are referred to as states A and B, respectively. In the third column of Figure 4.7 it is shown that the class representative of the coupling state must be converted to the coupling state (once again outlined in red), before attempting to find the state in the equivalence class represented by the second class representative (in the ordered list of class representatives) which may be reached by executing a single move. The coupling state is therefore the first state A while the second class representative in the list of ordered class representatives is the first state B. The states in the fourth column of Figure 4.7 show the coupling state and the state that belongs to state B's equivalence class, which is reachable from state A by a single move. It may be seen, from the fifth column of states, that the equivalence class state obtained in the step illustrated in the fourth column is taken as state A during the next iteration of the process. This process is repeated until each class representative has been converted to the relevant state so as to obtain the actual move sequence of states shown in the last column of the figure. The moves associated with the sequence of states in the last column of Figure 4.7 are *bottom half: 4 cells left*, *left half: 1 cell up*, *bottom half: 4 cells left*, *right half: 5 cells up* and *bottom half: 4 cells left*.

As illustrated in Figure 4.8, the complete move sequence may therefore be found by combining the list of states obtained from the first component of the heuristic (shown in Figure 4.8(a)) and the move sequence produced by the second component of the heuristic (shown in Figure 4.8(b)) to obtain the move sequence shown in Figure 4.8(c). The complete list of moves is as follows: *bottom half: 3 cells left*, *left half: 5 cells up*, *bottom half: 4 cells left*, *left half: 1 cell up*, *bottom half: 4 cells left*, *right half: 5 cells up* and *bottom half: 4 cells left*.

4.4 Chapter Summary

This chapter was devoted to a description of a heuristic solution procedure capable of solving a *Wrapslide* mixed state in an optimal or hopefully near-optimal number of moves. The initial focus in §4.1 was on the assignment of an objective function value to a given *Wrapslide* mixed state. Thereafter, the focus shifted in §4.2 to a description of the heuristic solution procedure developed to solve mixed *Wrapslide* states. Section 4.2 included detailed discussions on the two main components of the heuristic, namely a game tree component and a state enumeration tree component. These discussions elucidated the branching and termination criteria of the game tree component, the determination of optimal move sequences from the state enumeration tree component and the integration of the two components into a single heuristic. A comprehensive worked example was presented in §4.3 in order to demonstrate the working of the procedure.

CHAPTER 5

Implementation of the Heuristic

Contents

5.1 User Interface Design 47
5.2 Results Obtained by the Heuristic 50
5.3 Chapter Summary 51

This chapter is devoted to a description of a computer implementation of the heuristic developed in Chapter 4. It opens in §5.1 with a description of the programming environment used to implement the heuristic. The focus of the section then shifts to a description of the *graphical user interface* (GUI) developed to enable users unfamiliar with computer programming environments to utilise the heuristic. The required user inputs, the output options available to the user and the formats of the GUI windows are explained. Thereafter, the results obtained by implementing the heuristic for a number of randomly generated *Wrapslide* states are described in §5.2. These results include average computation times required and average move sequence lengths uncovered by the heuristic. The chapter closes in §5.3 with a concise summary.

5.1 User Interface Design

The heuristic described in Chapter 4 was implemented by the author in Python 3.5 [22] using the PyCharm *integrated development environment* (IDE). Two different implementations of the heuristic were established. Both take a numeric array representing a user-specified mixed state to be solved as input and produce the computation time and the length of the solution move sequence found by the heuristic as output. There are, however, two key differences between the implementations. The first important difference is the additional output element of the second implementation, namely the exact moves contained in the solution sequence. The second notable difference is a consequence of this additional output element — an increased computation time for the second implementation.

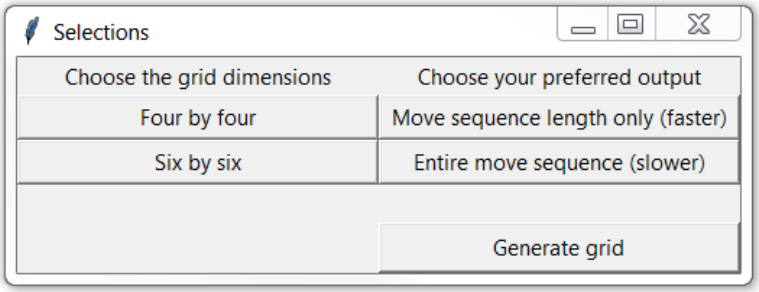
In order to obtain the length of and actual moves within a solution sequence for a given mixed *Wrapslide* state, a numeric array representing that state is required by both the Python implementations of the heuristic(s) as described above. This requirement might make the heuristic inaccessible to users unfamiliar with Python, or programming environments as a whole. In order to remove this obstacle, a GUI was therefore developed, using the Python *tkinter* package [23], to facilitate a more intuitive method of specifying the mixed state to be solved.

On opening the GUI, the user is presented with the window shown in Figure 5.1(a). The user may now select the dimensions of the grid to be solved as well as the desired output. The grid dimension selection is made by clicking on one of the buttons labelled *Four by four* or *Six by six*. The output selection is made by clicking the button labelled *Move sequence length only (faster)* or the button labelled *Entire move sequence (slower)*. Once a selection has been made, the corresponding button will appear sunken. For instance, clicking the *Six by six* and *Complete move sequence (slower)* buttons produces the window shown in Figure 5.1(b).

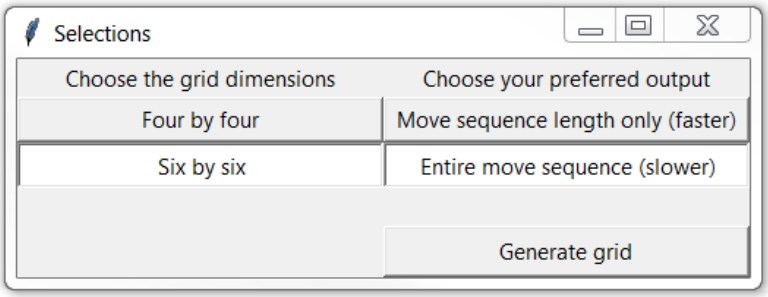
The user must make a selection in terms of both the dimensions of the grid to be solved and the desired output before clicking on the *Generate grid* button. This action will produce a blank grid (of the selected dimensions) and a bar of coloured cells, as shown in Figure 5.2(a). The user may now colour the cells of the blank grid so that it represents the mixed state to be solved. First, the colour to be inserted into the grid is selected. This is done simply by clicking on the desired colour in the colour bar. The user may then click on any of the cells in the blank grid in order to assign the selected colour to that cell. The colour of the white cell will change as soon as it is clicked upon. The same colour will be assigned to each cell clicked on until a new colour is chosen from the colour bar.

Once each cell has been assigned a colour, such as in the grid shown in Figure 5.2(b), the *Solve* button may be clicked. Clicking the *Solve* button will prompt the implementation to input the current configuration of the grid and convert it to the required numeric array format. Depending on the user's selection between the *Move sequence length only (faster)* and *Entire move sequence (slower)* buttons, the input array will be taken as input for the execution of the desired heuristic implementation. If the button labelled *Move sequence length only (faster)* was clicked, the heuristic described in Chapter 4 is executed to find the length of an optimal or near-optimal move sequence for the mixed state specified by the user. As indicated on the button label, this implementation outputs only the length of the move sequence found and executes much faster than the second implementation. The output thus produced for the mixed state in Figure 5.2(b) is shown in Figure 5.3(a). If, however, the user clicked the button labelled *Entire move sequence (slower)*, the second implementation is executed. While slower, this option will produce the output shown in Figure 5.3(b) for the mixed state in Figure 5.2(b), specifying the computation time, the length of the move sequence found by the heuristic and the exact moves contained in this move sequence.

The solution sequence moves are denoted by a string of letter-and-integer pairs, as shown in Figure 5.3(b). The letter in each pair may be either a T, B, R or L, indicating movement of the top (T), bottom (B), right (R) or left (L) half of the grid. The integer, whose value may range from one to three for the 4×4 grid or from one to five for the 6×6 grid, indicates the number of cells over which the half is moved. In this project, the convention adopted is that all moves are either upward or to the left. That is to say, Ts and Bs represent leftward moves of the top and bottom halves, respectively, while Rs and Ls denote upward moves of the right and left halves, respectively. For instance, the string 'B3 R5 T4' represents the move sequence where the bottom half is moved three cells to the left, the right half is moved five cells up and the top half is moved four cells to the left. Moves shifting cells downward or to the right need not be included in any move sequence since the toroidal nature of the puzzle dictates that the states reached by moving a half down or to the left may also be obtained by moving that half up or to the right. For instance, the state obtained by moving the top half of any 6×6 grid two cells to the right may also be obtained by moving the top half of that same grid $6 - 2 = 4$ cells to the left. Furthermore, whether the half is moved two cells or twenty it is still considered a single move, thus executing only left and/or downward moves will not increase the length of the solution sequence obtained.

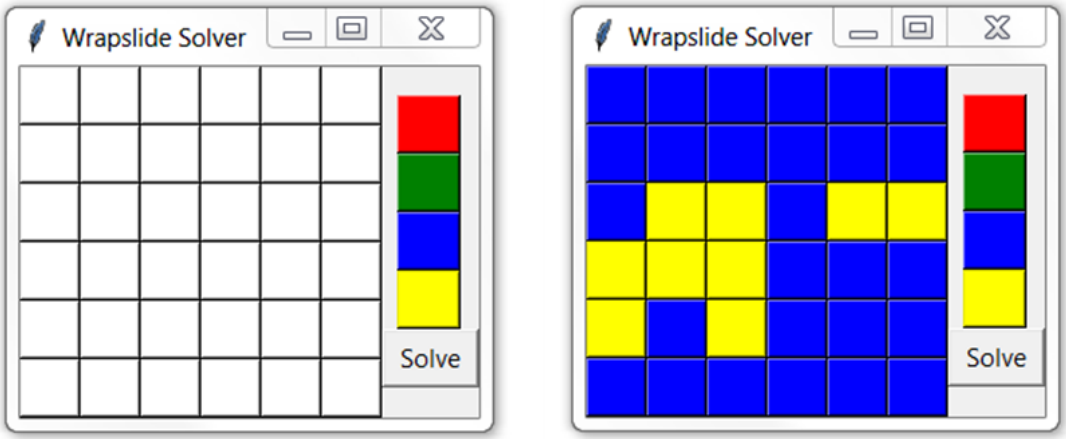


(a)



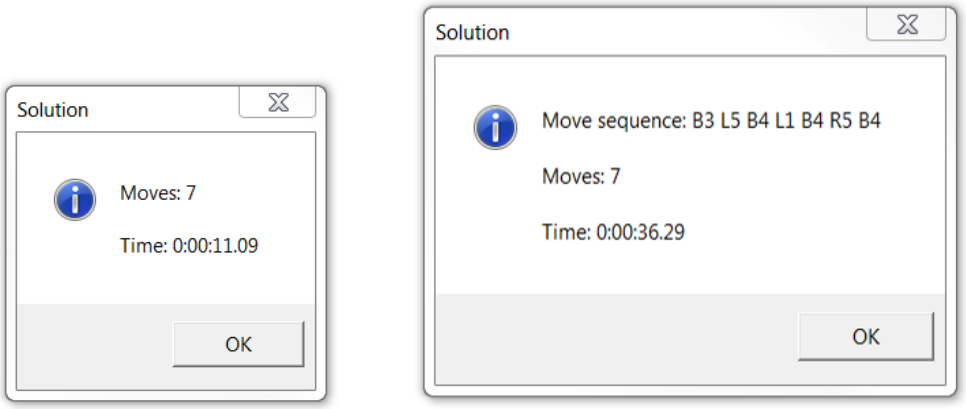
(b)

FIGURE 5.1: *The input and output options presented to the user upon launching the GUI.*



(a) (b)

FIGURE 5.2: *(a) A blank grid of the dimensions selected by the user and (b) an example of a specified mixed state for which the* Solve *button may be clicked.*



(a) (b)

FIGURE 5.3: *The output window generated for the mixed state in Figure 5.2(b) when (a) the* Entire move sequence (slower) *button or (b) the* Move sequence length only (faster) *button is clicked to indicate the preferred output.*

5.2 Results Obtained by the Heuristic

Both implementations of the heuristic were executed for (the same) hundred randomly generated mixed states of the 4×4 two-, three- and four-colour grids and for the 6×6 two-, three- and four-colour grids. The results thus obtained are shown in Tables 5.1 and 5.2.

Since the complete state enumeration trees have been generated for each of the 4×4 grids, the average optimal move sequence length for each grid may be calculated. The averages of the optimal move sequence lengths (obtained from the state enumeration trees) are $4.45, 7.34$ and 9.35 for the 4×4 two-, three- and four-colour grids, respectively. Comparing these average optimal move sequence lengths to the average move sequence lengths shown in the second column of Tables 5.1 and 5.2, one may obtain an indication of the degree of optimality of results returned by the heuristic. For instance, from the third rows of Tables 5.1 and 5.2 it may be seen that the average move sequence length obtained by the heuristic for a hundred random instances of the 4×4 two-colour grid is 4.16. This is smaller than the average optimal move sequence length of 4.45 calculated utilising the state enumeration trees. Since the heuristic was only applied to a hundred randomly generated states, an average move sequence length that is smaller than the average optimal move sequence length is possible (as opposed to if the heuristic were to be applied to every possible state, in which case the average move sequence length obtained by the heuristic would necessarily have been larger than or equal to the average optimal move sequence length). The small difference between the average move sequence length obtained by the heuristic and the optimal move sequence length calculated from the state enumeration tree for the 4×4 two-colour grid indicates a high degree of optimality of solutions obtained by the heuristic for the 4×4 two-colour grid. Similarly, the average move sequence length of 7.58 (in the fourth rows of the tables) obtained for the 4×4 three-colour grid is very close to the optimal move sequence length of 7.34 determined from the state enumeration tree of the 4×4 three-colour grid, once again indicating a high degree of optimality. The degree of optimality decreases slightly for the 4×4 four-colour grid for which an average move sequence length of 10.82 was obtained by the heuristic implementation, as shown in the fifth rows of Tables 5.1 and 5.2, compared to an average optimal move sequence length of 9.35.

Nevertheless, the average move sequence lengths obtained for the 4×4 grids are all smaller than

the corresponding God numbers. This is not, however, the case for the 6×6 two-colour grid. Although the complete state enumeration tree for the 6×6 two-colour grid was not generated, meaning that the actual average optimal move sequence for the 6×6 two-colour grid could not be calculated, it may still be deduced that the degree of optimality of solutions obtained by the heuristic is worse than for the 4×4 grids since an average move sequence length of 12.59 was obtained by the heuristic, while the God number of the 6×6 two-colour grid is known to have a value of twelve. Therefore, a decrease in the degree of optimality is observed not only with an increase in the number of colours, but also in the size of the mixed state.

Although the God numbers of the 6×6 three-colour and four-colour grids are unknown it is therefore reasonable to assume that the degree of optimality will continue to decrease as the number of colours increase. Therefore, it is expected that, not only are the average optimal move sequence lengths for the 6×6 three- and four-colour grids smaller than the average move sequence lengths obtained by the heuristic, but so also are the God numbers. The following conjecture is therefore put forward.

Conjecture 5.1 *The God number for the* 6×6 *three-colour grid satisfies* $\Theta_W(6, 3) \leq 21$.

Similarly, from the fifth columns of Tables 5.1 and 5.2, the following conjecture is put forward.

Conjecture 5.2 *The God number for the* 6×6 *three-colour grid satisfies* $\Theta_W(6, 4) \leq 28$.

The average move sequence length values in Tables 5.1 and 5.2 are identical since both implementations of the heuristic were executed for the same hundred states randomly generated for each colour and grid size combination. When comparing the computation times recorded in the last columns of Tables 5.1 and 5.2, it may be seen, as expected, that the implementation which does not provide an actual move sequence executes significantly faster. It may be seen that the difference between the average computation times obtained for the first and second implementation of the heuristic increase linearly, relative to the number of states in the solution sequence obtained (approximately 2.018 seconds per state).

Furthermore, the average computation time increases for larger grid size and more colours. The increase in computation time is not, however, a linear function of the increase in size of the associated search spaces. For instance, the search space of the 6×6 three-colour grid is $\frac{4.412 \times 10^{14}}{9.414 \times 10^{7}} = 4.687 \times 10^{6}$ times larger than the search space for the 6×6 two-colour grid. The corresponding computation time in Tables 5.1 and 5.2 only increase by factors of $\frac{175.33}{69.12} = 2.54$ and $\frac{232.96}{115.53} = 2.016$, respectively. The relatively small increase in computation time (when compared to the increase in the size of the search space) may be attributed to the unique branching procedure and termination criteria of the heuristic, as described in Chapter 4. This tailored branching procedure and termination criteria is, however, most likely also the cause of the decreasing degree of optimality observed. This trade-off is considered acceptable due to the large increase in computation time that is expected to result for a slight increase in optimality, as mentioned in §4.2.

5.3 Chapter Summary

This chapter was devoted to a description of implementations by the author of the heuristic solution procedure described in Chapter 4. This description opened in §5.1, with a high-level description of the two Python implementations of the heuristic aimed at generating an upper

TABLE 5.1: *The average move sequence length achieved and average computation time expended by executing the first heuristic implementation described in §5.1 for 100 randomly generated mixed states of the* 4×4 *two-, three- and four-colour grids and* 6×6 *two-, three- and four-colour grids. Computation times are represented in the format minutes : seconds.*

	Average move sequence length		Average computation time	
	4×4	6×6	4×4	6×6
Two colours	4.16	12.59	00 : 00.80	00 : 18.79
Three colours	7.58	21.06	00 : 01.42	01 : 09.12
Four colours	10.82	27.82	00 : 02.34	02 : 55.33

TABLE 5.2: *The average move sequence length achieved and average computation time expended by executing the second heuristic implementation described in §5.1 for 100 randomly generated mixed states of the* 4×4 *two-, three- and four-colour grids and* 6×6 *two-, three- and four-colour grids. Computation times are represented in the format minutes : seconds.*

	Average move sequence length		Average computation time	
	4×4	6×6	4×4	6×6
Two colours	4.16	12.59	00 : 08.38	00 : 43.65
Three colours	7.58	21.06	00 : 16.69	01 : 55.53
Four colours	10.82	27.82	00 : 24.24	03 : 52.96

bound on the length of an optimal move sequence for a given *Wrapslide* mixed state. The focus shifted in §5.2 to a description of the working of the GUI designed to make the heuristic more accessible to non-technically inclined users. Thereafter, the results obtained by applying these two implementations to a hundred randomly generated mixed states were presented in §5.2.

CHAPTER 6

Summary and Conclusions

Contents

6.1 Project Summary 53
6.2 Suggestions for Future Work 54
6.3 What the Author has Learnt 55
6.4 How this Final Year Project may Benefit Society 56

6.1 Project Summary

The report comprises six chapters. The introductory chapter was devoted to the project background, a problem statement, the project scope delimitation, the research objectives pursued and the research methodology employed in the project.

In fulfilment of Objective I in §1.3, Chapter 2 contained a thorough review of the concepts necessary for the development of a solution procedure for the puzzle *Wrapslide* as well as for the establishment of bounds on and values of its associated God numbers. The initial focus of the chapter was on the notions of symmetry, isometries and equivalence classes. Thereafter, the focus shifted to descriptions of the plane and the torus as topological objects, as well as tilings of these surfaces. The review continued with an explanation of the celebrated B&B algorithm coupled with a description of the best-known tree traversal protocols employed in conjunction with this method. The review closed with an overview of prior academic investigations related to the puzzle *Wrapslide*.

Chapter 3 was devoted to the description and development of techniques required to generate state enumeration trees containing all *Wrapslide* class representatives. The chapter opened with an explanation of the methodology employed to encode a *Wrapslide* state and represent it as a decimal value, in fulfilment of Objective II. The next section contained a description of how this encoding methodology was implemented in combination with the notions described in the literature review in order to generate equivalence classes for *Wrapslide* states. Thereafter, an account followed of the construction of state enumeration trees (in fulfilment of Objective III), which enabled the author to establish the God numbers for two *Wrapslide* grids and confirm that for another. This account included an outline of the notion of a game tree as well as detailed descriptions of the branching and termination criteria employed in the generation of such trees. The penultimate section was devoted to a description of the convention followed to identify class

representatives and the last section saw the establishment of the values of three *Wrapslide* God numbers.

The objective in Chapter 4 was the design of a heuristic solution procedure capable of solving a *Wrapslide* mixed state in an optimal or hopefully near-optimal number of moves. The initial discussion centred around the association of an objective function value with a given *Wrapslide* mixed state as a means of measuring the degree of colour heterogeneity in each of a state's four quadrants. Thereafter, a heuristic solution procedure was put forward in fulfilment of Objective IV. For clarity of explanation the heuristic was partitioned into two components: a game tree component and a state enumeration tree component. Both components' branching and termination criteria were described in some detail. A comprehensive worked example was also included to elucidate the working of the heuristic. The focus of the chapter finally shifted to the potential contribution of such a heuristic in respect of the establishment of unknown God number values for *Wrapslide* grids.

Two implementations by the author of the heuristic designed in Chapter 4 within a software environment were described in the penultimate chapter, in fulfilment of Objective V. The chapter contained a description of the GUI developed, which aims to make the heuristic accessible to a larger, non-technical audience. Thereafter, some statistics gathered by applying these implementations to a number of random *Wrapslide* states were presented in the second section of Chapter 5, in fulfilment of Objectives VI and VII.

6.2 Suggestions for Future Work

Although two novel God numbers were established during the completion of this final year project, the values of the God numbers of several *Wrapslide* grids remain unknown. Several future courses of action and areas of further study that may contribute to the establishment of bounds on or the actual values of these unknown God numbers have been identified.

The most easily implementable suggestion is to execute the existing implementations of the heuristic of Chapter 4 on better hardware. For instance, a computer memory capacity of 16GB RAM (as opposed to the 8GB RAM available to the author) would likely allow for the confirmation of the God number of the 6×6 two-colour grid. If a researcher has access to even more powerful computational resources (with larger storage capacities and improved processing capabilities), the implementation for the generation of state enumeration trees may be used as-is to determine the values of $\Theta_W(6,3)$, $\Theta_W(8,2)$ and hopefully even $\Theta_W(6,4)$. It is therefore evident that the acquisition of better computational resources is a vital step in the pursuit of *Wrapslide* God numbers. After all, despite years dedicated to the development of techniques for reducing the size of the search space, the determination of the value of the God number for Rubik's cube required 35 CPU years (donated by Google).

Future researchers may also aim to increase the speed of the heuristic of Chapter 4 by, for instance, increasing the speed at which a state enumeration tree is searched for a particular class representative. One method commonly employed for similar purposes is the bisection, or binary search, method which finds an item in an ordered list by repeatedly partitioning the section currently being considered into two halves and selecting the half which might contain the sought-after item for further consideration. The execution time of the heuristic implementations might also be sped up by alterations to the source code. Alternatively, researchers may consider increasing the speed at which the current implementation executes by increasing the processing speed of the computer it is executed on. This may be done by disabling CPU parking (specifically for Windows 7 operating systems) or over-clocking the computer on which the implementations

are run. These efforts will enable the heuristic to solve a much larger number of randomly generated states. If a larger number of randomly generated states is considered, the average move sequence length produced by the heuristics will be a better representation of the actual average move sequence length of the population (in other words all the distinct states of a *Wrapslide* grid). Increased accuracy of the average move sequence length should, in turn, translate to better conjectures on the values of as yet unknown God numbers.

Future research efforts may also focus on the identification, study and implementation of suitable algorithms for the solution of *Wrapslide* mixed states, such as the the IDA* algorithm employed to find optimal solutions of random instances of Rubik's cube [17].

Finally, there remains room for improvement in respect of the GUI. The GUI currently does not contain any protocols preventing users from, for instance, clicking the *Solve* button before the selecting both a grid size and preferred output or entering an impossible grid configuration (for instance, a user may currently colour a 6×6 grid to contain 11 cells of one colour and 25 of another colour, which would be impossible to solve). Should an invalid input event occur, the GUI should stop executing. For this purpose, safety nets aimed at preventing erroneous user inputs should be incorporated into the GUI.

6.3 What the Author has Learnt

The author's academic knowledge horizon was considerably widened by exposure to the notions of symmetry, isometries, equivalence classes and tilings of the plane and the torus, to name but a few. The author also gained a deeper understanding of the well-known B&B method and the best-known tree traversal protocols used in conjunction with the B&B algorithm. Furthermore, completion of the project has led to a considerable increase in the author's programming skills and the development of a working knowledge of the Python software environment.

The most valuable skill acquired by the author during the course of this project was the ability to think innovatively. The problem considered in this project is relatively novel and was, at its initial presentation to the author, completely unknown (to the author). Unlike some well-known combinatorial optimisation problems, like the *travelling salesman problem* (TSP) or the various types of facility location problems, the problem considered in this project has not been documented and reported on extensively, which required the development of completely new approaches to solve some of the problems encountered during this project. Many of the methods employed in this project may seem overly simple and obvious in hindsight, but their conception required imaginative thinking. Two small examples include the encoding method used to represent a *Wrapslide* state as a decimal value and the technique used to associate an objective function value with a mixed *Wrapslide* state.

Other problem-solving approaches adopted in this project were inspired by the literature. For instance, the reduction of the number of states that needed to be included in the state enumeration tree through the implementation of the notion of equivalence classes, was inspired by the methodology employed in establishment of the God number for Rubik's cube. The method used to generate equivalence classes for Rubik's cube could not, however, be applied directly to the puzzle *Wrapslide*. The author therefore needed to integrate theoretical knowledge related to the puzzle *Wrapslide* and the ideas employed in the establishment of the God number for Rubik's cube in order to produce original techniques capable of producing similar results.

Completion of the project also resulted in improved reasoning skills borne out of the need to, for instance, motivate why a class representative cannot reappear more than two levels below

its first occurrence in a *Wrapslide* state enumeration tree.

The author also learned how to conduct a research project over a sustained period of time, which demanded the development of effective time management skills and self-discipline. Finally, the author's communications skills improved considerably during the course of this project. This may be attributed to the need to communicate complicated principles and results clearly and concisely. The author's writing skills improved due to the writing of this report while her oral presentation skills were developed in preparation for the presentation of some of the work contained in this report at the 45th Annual Conference of the Operations Research Society of South Africa.

6.4 How this Final Year Project may Benefit Society

The puzzle *Wrapslide* may be viewed as a combinatorial optimisation problem. If it is, for instance, considered as an instance of the well-known assignment problem, the methodology developed to generate the state enumeration tree may be used to generate all possible assignment combinations. Once the list of possible assignment configurations has been generated, it may be stored and consulted when necessary. Alternatively, a *Wrapslide* state may be considered as a discretised solution space for a facility location problem, where the different colour cells represent different types of facilities. Once again, the methods employed in the project may be implemented to generate the complete set of possible facility distributions.

The primary contribution made in this project is, however, the theoretical support and hopefully stimulation of continued research. The work contained in this project could only be completed due to a foundation that was already laid by the efforts of BURGER AP [6] and the two Dutch students who proved that all possible mixed states of the puzzle *Wrapslide* may be reached from a solved state [10]. Similarly, the work conducted in this project may support the work of some future researcher. Ultimately, such a series of academic contributions may lead to the development of methods or areas of study that have widespread practical applications. For instance, Lattice theory was put forth as a purely theoretical concept in 1940 by Birkhoff [4] and was subsequently studied for many years before it was ever used in programme analysis and verification as it is so widely applied today.

References

[1] Adamchik VS, 2009, *Game trees*, [Online], [Cited June 2016], Available from `https://www.cs.cmu.edu/~adamchik/15-121/lectures/Game%20Trees/Game%20Trees.html`.

[2] Adamchik VS, 2009, *Stacks and queues*, Lecture Notes, Carnegie Mellon University, Pittsburgh (PA).

[3] Apple Inc., 2014, *WrapSlide on the App Store — iTunes — Apple*, [Online], [Cited March 2016], Available from `https://itunes.apple.com/za/app/wrapslide/id795712935?mt=8`.

[4] Birkhoff G, 1940, *Lattice theory*, American Mathematical Society, Providence (RI).

[5] Burger AP, 2015, *The WrapSlide Puzzle*, Presentation Slides, Stellenbosch University, South Africa.

[6] Burger AP, 2016, *Wrapslide* Inventor, Contactable at `alewynburger@gmail.com`.

[7] Clausen J, 1999, *Branch and bound algorithms — principles and examples*, Unpublished Report, Department of Computer Science, University of Copenhagen, Copenhagen.

[8] Cohen MM & Pilgrim KM, 2004, *Groups and geometry: A bridge to the math major*, Igarss 2014, **(1)**, pp. 1–5.

[9] Conrad K, *Plane isometries and the complex numbers*, Unpublished Report, Department of Mathematics, University of Connecticut, Connecticut(CT).

[10] Geelhoed D & Meester L, 2013, *De WrapSlide-puzzel algebraïsch bekeken*, Unpublished Report, Leiden University, Leiden.

[11] Gordon GJ, 2012, *3.1 Convex sets, Lecture 3*, Lecture Notes, Carnegie Mellon University, Pittsburgh (PA).

[12] Google Inc., 2014, *WrapSlide — Android Apps on Google Play*, [Online], [Cited March 2016], Available from `https://play.google.com/store/apps/details?id=com.wrapslide.android.wrapslide`.

[13] Johnson DL, 2001, *Symmetries*, Springer, London.

[14] Kell B, 2010, *21-110: Symmetry and tilings*, [Online], [Cited May 2016], Available from `http://math.cmu.edu/~bkell/21110-2010s/symmetry-tilings.html`.

[15] Kingsford C, 2013, *Graph traversals*, Lecture Slides, Carnegie Mellon University, Pittsburgh (PA).

[16] Korf RE, 1991, *Depth-first vs best-first search*, pp. 434–440.

[17] Korf RE, 1997, *Finding optimal solutions to Rubik's cube using pattern databases*, American Association for Artificial Intelligence, **97**, pp. 700–705.

[18] Land AH & Doig AG, 1960, *An automatic method of solving discrete programming problems*, Econometrica: Journal of the Econometric Society, **28**, pp. 497–520.

[19] MacKenzie B, 2015, *Wrapslide SM histogram*, [Online], [Cited May 2016], Available from `http://cubezzz.dyndns.org/drupal/?q=node%2Fview%2F540`.

[20] Nelson A, Newman H & Shipley M, *17 plane symmetry groups*, [Online], [Cited April 2016], Available from `https://caicedoteaching.files.wordpress.com/2012/05/nelson-newman-shipley.pdf`.

[21] Nguyen TH, 2003, *Lecture 10: Branch and bound methods*, Lecture Notes, Stanford University, Standford (CA).

[22] Python Software Foundation, 2015, *Python v3.5*, [Online], [Cited March 2016], Available from `http://www.python.org`.

[23] Python Software Foundation, 2015, *tkinter*, [Online], [Cited August 2016], Available from `http://www.python.org`.

[24] Rokicki T, Alto P, Kociemba H, Davidson M & Dethridge J, *God's number is 20*, [Online], [Cited April 2016], Available from `http://www.cube20.org/`.

[25] Schwartz RE, 2011, *Mostly surfaces*, American Mathematical Society, Providence (RI).

[26] Stewart I & Golubitsky M, 1992, Blackwell Publishers, Oxford.

APPENDIX A

Project Timeline

The expected timeline is given in Figure A.1 in Gantt-chart form.

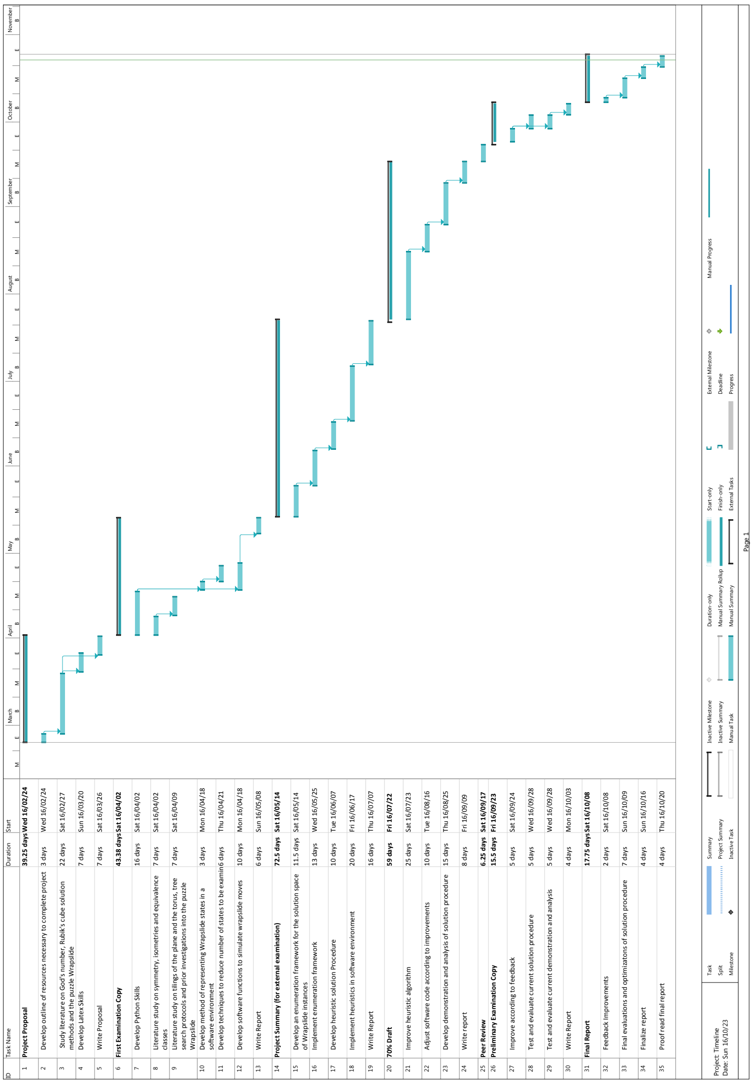


FIGURE A.1: *Project timeline in Gantt-chart form.*