Applications of Gröbner Basis Methods to Graph Colouring Problems

Maksim Mijović

00597196

Supervisor: Dr. Alexander M. Kasprzyk

This is my own work unless otherwise stated

Project submitted to Department of Mathematics, Imperial College London

in partial fulfillment of the requirements for the degree of

Master of Science (MSci)

June 2013

Abstract. Gröbner bases are a relatively new discovery in mathematics and give us a powerful tool for exploring the ideal membership problem. Through some careful algebraic formulation we can encode the information contained in a graph via a collection of polynomials. Using the tools of Gröbner basis theory we can begin to address problems from graph theory, particularly proper k-colourability of graphs from an interesting, alternative angle.

Popular number puzzles, such as Sudoku, may be expressed in graph theoretic language, so that we may now explore the solution of such puzzles via the use of Gröbner basis methods and see what successes and problems this may bring us.

This paper offers an introduction to the relevant theory from the fields of graph theory, group theory and ideal theory, as well as Gröbner bases, and then goes on to explore the application of Gröbner basis methods to Sudoku puzzles, using the SAGE computer algebra package, including critical evaluations of the methods used, results obtained and difficulties faced.

Acknowledgments

I would like to thank first and foremost my supervisor, Dr. Alexander M. Kasprzyk, for his guidance, patience and inspiration. With his help I have created a piece of work that I am proud of and that exceeds the expectations I had when I began it, in November 2012.

I would also like to thank Dr. Kasprzyk for accepting my project proposal in November, at a relatively late stage, as well as Prof. Paolo Cascini and the Imperial College Mathematics Department for allowing me to change to a new project and supervisor at this point in time.

Finally, I would like to thank Alastair J. Litterick and Prof. Martin W. Liebeck of Imperial College London, as well as Prof. Eamonn O'Brien of the University of Auckland, for their expertise with the MAGMA computational algebra software which helped bring together the material of section 4.5.

Contents

List of Figures

1 Introduction

In 1852, Francis Guthrie proposed the famous Four Colour Theorem; that any separation of the plane into contiguous regions (a *map*) needs no more than 4 colours so that any pair of adjacent regions are of different colours. It wasn't until 1976 that Appel and Haken were able to conclusively prove this theorem, after numerous failed attempts by mathematicians before them. The proof they gave was not, however, of the elegant kind we usually associate with the solution of a problem in mathematics. Rather, it was a laborious effort with considerable computer assistance; breaking the problem down into almost 1500 separate cases and attacking these one by one over more than 1000 hours REF.

In fact, this was the first major mathematical proof that was completed with computer assistance. Furthermore, it seemed to show that computational methods and graph colouring go hand in hand. Buchberger's work on Gröbner bases in the 1960s greatly advanced the field of Computational Algebra [5], and with the growth of this field came interesting results; notably the solution of the ideal membership problem using Gröbner bases. In the 1980s, new theory linking Graph Colouring and Ideal Theory started to appear in several places, the work of Lovász [13] among them. It was an intuitive step from here to use Gröbner bases and Computational Algebra to address these problems and this gave rise to a host of works on the theory of determining colourability using Gröbner Bases.

In this paper we will explore the theory behind this application of Gröbner Bases to Graph Colouring problems, including enumerating graph colourings and identifying unique colourings. Following this, we will examine some examples from the popular number game Sudoku, and how we can apply the theory to yield solutions in a variety of cases. Naturally, we will provide some groundwork in Graph Theory and Gröbner Bases for readers that are not as well acquainted with these fields, as well as examples to supplement the theory. A summary of successes enjoyed and difficulties experienced, as well as suggestions for further study, will round off the work.

2 Preliminaries

2.1 Graphs and Graph Colouring

The definitions in this section explain the basics of graph theory and follow [3] *unless otherwise stated.*

2.1.1 Graphs

A ***graph*** $G = (V, E)$ is an ordered pair of disjoint sets such that V is a set of points and E is a set of unordered pairs of points from V. We call $V = \{x_0, ..., x_n\}$ the set of ***vertices*** and $E = \{x_i x_j \,|\, i \in I,\, j \in J\}$ (for some indexing sets I and J) the set of ***edges*** of G. A ***subgraph*** of G is written $G' = (V', E')$ where $V' \subseteq V$ and $E' \subseteq E$ and for any vertex $e = x_i x_j \in E'$ we have $x_i, x_j \in V'$. We can say $G' \subseteq G$. Two vertices are ***adjacent*** if they are connected by an edge.

A ***path*** is a graph P of the form $V(P) = \{x_0, ..., x_n\}$, $E(P) = \{x_0x_1, x_1x_2, ..., x_{n-1}x_n\}$ and a graph is ***connected*** if, for any pair of vertices $(x, y) \,|\, x, y \in V$, there exists a path from x to y. A ***tree*** is a connected graph not containing and cycles, where a ***cycle*** is a path whose end-vertices coincide.

We also remark that the graphs that concern us in this project will not have multiple edges, that is to say $e_1 = x_i x_j \in E$ and $e_2 = x_i x_j \in E$ such that $e_1 \neq e_2$. Our graphs will also not contain loops; edges of the form $e_1 = x_i x_i \,|\, x_i \in V$.

The ***order*** of G is the number of vertices and its ***size*** is the number of edges. The size of a graph of order n (also called a *graph on n vertices*) must be between 0 and $\frac{n(n-1)}{2}$ (each of the n vertices can be joined to, at most, $n-1$ others, divided by 2 to avoid double-counting).

A graph of order n and size $\frac{n(n-1)}{2}$, i.e. where every pair of vertices is joined by an edge, is called a ***complete graph***.

We also note that a ***clique*** of a graph G is a subgraph, G', which is also complete.

2.1.2 Graph Colouring

Given a graph G, we wish to "colour" each vertex so that no two adjacent vertices have the same "colour" (in practice we often number the vertices rather than colouring them). The least number of colours needed to do so for a given graph is called the ***chromatic number*** $\chi(G)$. If we specify the number of colours we want to use in the colouring, it is called a ***k-colouring***. Thus a k-colouring of the vertices of G is a function $c : V(G) \longrightarrow \{1, 2, ..., k\}$ such that each $c^{-1}(j) = \{v \in V | c(v) = j\}$ is a set of vertices, no pair of which is adjacent.

The ***Graph polynomial***, f_G, of a graph $G = (V, E)$ is defined

$$f_G := \prod_{i<j,(x_i,x_j)\in E} (x_i - x_j)$$

where the $i < j$ condition ensures we don't count edges twice.

2.2 Graph colouring and planarity

Once again, the preliminary theory from this section, unless otherwise stated, is from [3].

In 1852 Francis Guthrie proposed what would become a very well known mathematical problem; the 4 colour theorem. The statement of the problem is that any planar map (a separation of the plane into contiguous regions) can be coloured with 4 colours such that no two adjacent regions share the same colour. If we model each contiguous region as a vertex and let an edge connect vertices that represent adjacent regions in the map, then the equivalent statement is that any planar graph is 4-colourable. A planar graph, naturally, is one that can be drawn in the plane such that no two edges intersect in a point other than a vertex.
After eluding prominent mathematicians and receiving false proofs; this problem was finally solved in the 1970s by Appel and Haken via the use of computers [7].

Evidently this gives a nice way for narrowing the number of colours needed for a colouring, however it is necessary to know whether or not the graph in question is planar. Sometimes this is very simple; one merely exhibits a planar configuration of the graph in question, for more complex graphs we need a better method. This, however, requires a few more definitions.

Two graphs, $G = (V_1, E_1)$ and $H = (V_2, E_2)$, are ***isomorphic*** if there exists an isomorphism $\phi : V_1 \tilde{\longrightarrow} V_2$ such that $(v_i, v_j) \in E_1 \iff (\phi(v_1), \phi(v_2)) \in E_2$. That is, the vertex set isomorphism preserves adjacency.
H is a ***subdivision of graph*** G (or H is a ***topological*** G ***graph***) if H is obtained from G by subdividing some of the edges of G; where subdividing an edge means replacing it with some number of paths that have at most their end vertices in common. An example of subdivision:

Finally, two graphs are ***homeomorphic*** if they have isomorphic subdivisions.

Figure 2.1: Example of subdivision

This gives us enough technical terminology to expose a theorem on the planarity of graphs by Kazimierz Kuratowski [3]:

Theorem 1. *Kuratowski's theorem*

A graph, G, is planar if and only if it does not contain a subgraph that is homeomorphic to K_5 or $K_{3,3}$

K_5 is the complete graph on 5 vertices, and $K_{3,3}$ is shown below

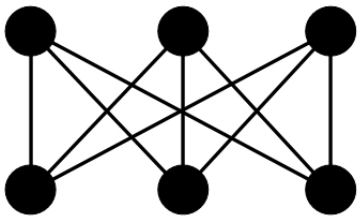


Figure 2.2: $K_{3,3}$, the utility graph

The corollary of this theorem that most interests us is as follows: if we can take K_5 or $K_{3,3}$ and subdivide them so that we get a subgraph of some graph G, then G is not planar. Later on in this project we will consider Sudoku puzzles, these can be modeled as graphs (c.f. section 4.3) where vertices represent cells of the Sudoku and edges represent cells that are in the same row, column or sub-grid (see section 4.1), and we will ask whether these graphs are k-colourable for a given k. Clearly Kuratowski could save us some time; if these graphs are planar we can say they are certainly 4-colourable by the 4 colour theorem. So let us examine whether the graphs of the Sudoku and Shidoku (4×4 Sudoku) are planar.

The graph of the Sudoku has a subgraph of the form:

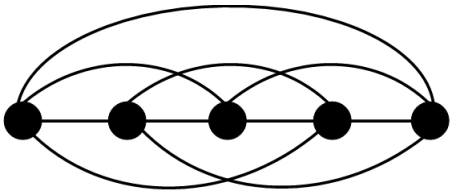


Figure 2.3: Subgraph of a Sudoku isomorphic to K_5

This can easily be seen by considering any 5 elements in the same row, for instance; none can have the same colour as another, so pairwise they are all adjacent. This leaves us with a subset of 5 vertices, each joined to all the others. By definition this is the complete graph on 5 vertices; K_5 (so it is isomorphic to K_5). Hence, by Kuratowski's theorem and since G itself is a subdivision of$G$, the graph of a Sudoku is not planar.

What about the graph of a Shidoku? Consider the graph K_5 and its subdivision shown below (*left* and *right*, respectively):

Figure 2.4: Subdivision of K_5

If we compare this subdivision to the graph of a Shidoku in 4.3 it's clear that the subdivision is a subgraph of the graph of the Shidoku (with vertex set, say, $V = \{0, 1, 4, 5, 8, 9, 13\}$). By Kuratowski, therefore, we conclude that the Shidoku graph is not planar, so that we cannot use the 4 colour theorem to make our work easier in this case either.

In the case of non-planar graphs (graphs which, when drawn in the plane, have at least two edges intersecting in a point that is not a vertex), results linking "planarity" and colouring exist as well, but for a slightly different notion of "planarity".

An ***embedding*** of a graph G on a surface (2-manifold) Σ, is a map from the vertices of G to points on Σ that takes the edges of G to arcs in Σ (homeomorphic, in the topological sense, to $[0,1]$) such that no two arcs on Σ intersect except in the images of the vertices of G [18]. We can think of this as a graph being "planar" on the surface Σ (even though it's not planar in the conventional sense).
As an example, consider K_5. This graph is not planar; we can trivially show this by Kuratowski's theorem (although proving this fact makes up part of the proof for Kuratowski's theorem). If we draw K_5 on a torus, however, it becomes "planar". That is to say, K_5 can be embedded in a torus.

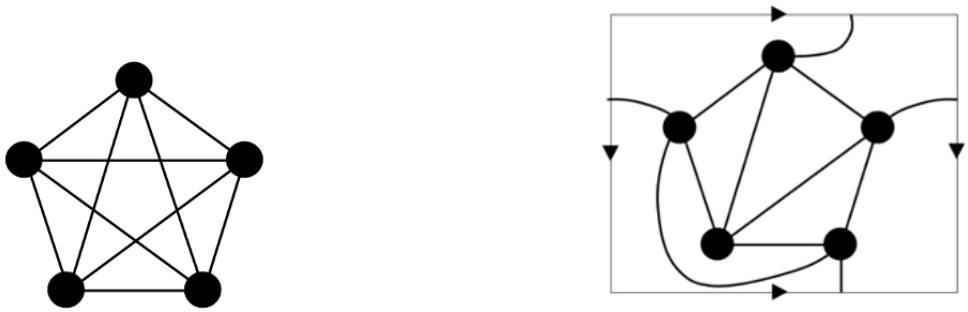


Figure 2.5: K_5 in the plane and on a torus

Recall that the ***genus*** of a surface is the maximum number of cuts along non-intersecting, closed, simple curves on the surface, that can be made such that the surface remains connected. For convenience this can be though of as the number of "holes" the surface has. The torus, therefore, has genus 1. The following result of Heawood [3] links the genus of a surface with the maximum chromatic number of a graph embedded in it.

Theorem 2. *The number of colours needed to colour a graph drawn on an orientable surface of genus* $\gamma \geq 1$ *is at most*

$$H(\gamma) = \left\lfloor \frac{7 + \sqrt{1 + 48\gamma}}{2} \right\rfloor$$

An easy corollary of this result is that any graph embedded on a torus can be coloured properly with 7 colours; this is easily seen by substituting the genus of a torus, $\gamma = 1$, into Heawood's formula.

What this allows us to do is set an upper bound for the chromatic number of an embedded graph, which then gives us an upper bound to the chromatic number of such a graph in the plane (although it may or may not be planar). The contrapositive of Heawood's theorem shows us that the graph of a

Sudoku is certainly not embeddable in a torus, since its chromatic number is, by definition, 9.
It is very hard to spot an embedding in a surface of the graph of a Shidoku, with 16 vertices, let alone that of a Sudoku with 81 vertices; hence this approach does not lend itself so well to our purposes, though it is interesting nonetheless. This failure motivates the continuation of the search for answers in another direction; that of computational algebra, the theory of which may be found in section 2.4.

2.3 Group theory

Group theory forms a small part of this project, but since we will later rely on certain concepts from the field, it is appropriate that the reader be prepared with a few relevant definitions, which broadly follow [11].

A ***Group***, G, is a set of elements $\{g_1, ..., g_n\}$ together with a binary operation, $\circ$, which is a map $\circ : G \times G \longrightarrow G$ via $(a, b) \mapsto a \circ b$. The group must satisfy the following laws:

1. $\forall a, b \in G$, $a \circ b \in G$; closure under $\circ$
2. $\forall a, b, c \in G$, $(a \circ b) \circ c = a \circ (b \circ c)$; associativity under $\circ$
3. $\exists e \in G \,|\, \forall g \in G$, $g \circ e = e \circ g = g$; existence of an identity under $\circ$
4. $\forall g \in G$, $\exists h \in G \,|\, g \circ h = h \circ g = e$; existence of an inverse of each element under $\circ$

An example of a group that is relevant to us is the ***permutation group on*** n ***letters***, S_n. The elements of this group are tuples (and combinations of tuples) of total length up to n, of the form $(a\,b\,c)$ which describe how elements from a set of size n are mapped onto themselves (permuted). In this case $(a\,b\,c)$ means $a \mapsto b$, $b \mapsto c$, $c \mapsto a$.
The binary operation is combination of tuples (permutations), which is done by considering how each tuple maps a given element, from right to left. For example; $(1\,2)\,(2\,3)$ is the same as $(2\,3\,1)$. This can be seen by considering $(1\,2)$ as $(1\,2)\,(3)$ (as an element of S_3), and $(2\,3)$ as $(2\,3)\,(1)$; the element 2 is mapped to 3 by the right-most tuple, and then 3 is mapped to 3 by the next tuple to the left, hence overall 2 is mapped to 3, and so on.

Given a group $G = (\{g_1, ..., g_n\}, \circ)$, we naturally get the concept of a ***subgroup*** H of G, written $H \leq G$, as group $H = (\{g_{I_1}, ..., g_{I_k}\}, \circ)$ where $k \leq n$ and I is some indexing set, so that

$\{g_{I_1}, ..., g_{I_k}\} \subseteq \{g_1, ..., g_n\}$. The binary operation $\circ$ is the same for G and H.

A group can be said to ***act*** on more than one set. What this means is that we can meaningfully apply an element of the group to more than one set of objects. An easy example of this is to think of the elements of S_3 acting on the vertices of a triangle which we label $1, 2, 3$. This is equivalent to rotations, reflections and combinations thereof, of the triangle. Notice, however, that S_3 also acts on the set of edges of the triangle. For example, $(1\,2)$ swaps vertices 1 and 2, but it also swaps edges $(1, 3)$ and $(1, 2)$.

Later on in this project we will be interested in a particular subgroup of S_n called the ***automorphism group of a graph***, $Aut(G)$. Given a graph $G = (V, E)$, $V = \{v_1, ..., v_n\}$, $Aut(G)$ is the set (we've yet to show it's a group) of permutations of V which preserve the edge-vertex connectivity of the graph. That is to say, two vertices are connected by an edge in the image of the graph acted on by $g \in Aut(G)$ if and only if they are connected by an edge in the original graph.
The identity permutation, $()$, is an obvious element of $Aut(G)$, this is also one of the conditions of $Aut(G)$ being a group. As a further example, consider the graph below:

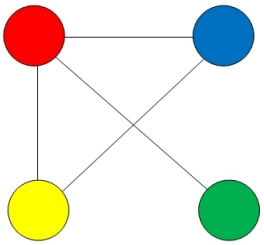


Figure 2.6: Graph isomorphism example

In this case elements of $Aut(G)$ are elements of S_4 acting on the set $\{yellow, red, blue, green\}$. Clearly $yellow \mapsto red$, $red \mapsto yellow$, $green \mapsto green$, $blue \mapsto blue$ is not an element of the automorphism group because swapping yellow and red means we now have an edge connecting the yellow and green vertices, which was not the case in the original graph. On the other hand, $yellow \mapsto blue$, $blue \mapsto yellow$, $red \mapsto red$, $green \mapsto green$ is an element of the automorphism group; it's easy to see that after swapping the colours yellow and blue, only the vertices that were connected by an edge before the permutation are connected by an edge now, and no vertices that were connected have ceased to

be connected.

Clearly, if a given permutation is in $Aut(G)$, then its inverse must also be in $Aut(G)$, since if the edge set is unchanged by permuting in one direction, it must be unchanged by applying the inverse permutation.

Associativity under combination of permutations comes from the associativity under combination which holds true in the parent group, S_n (where n is the number of vertices of G). The proof of this is trivial and can easily be seen by writing out two sets of n "letters" and drawing arrows between them to represent a permutation, then comparing the effect of $s_1 \circ (s_2 \circ s_3)$ and $(s_1 \circ s_2) \circ s_3$.

Finally, closure of $Aut(G)$ can easily be seen by considering the action of a permutation in $Aut(G)$ on the edge set, E, of G, rather than the vertex set, V. It's clear that $g \in Aut(G)$ if and only if the action of g on the edge set does not change the elements of the edge set. Clearly, then, if g_1, $g_2 \in Aut(G)$, the action of both on the edge set does not change the elements of E, and hence the action of $g_1 \circ g_2$ on E does not change the elements of E, hence $g_1 \circ g_2 \in Aut(G)$ and we have closure.

2.4 Ideals and Gröbner Bases

2.4.1 Ideals and Varieties

In this work, we concern our selves with the ideals of a commutative multivariate polynomial ring. Our later aims require us to know how to determine whether a given polynomial resides in a given ideal; we will exhibit some preliminary results and concepts which come from [10], following [5], and then demonstrate how they are used to answer this question, which is called the Ideal Membership Problem.

We remind the reader that an ***ideal*** $I \subset R$ is a subset of a ring that is

- closed under addition
- closed under multiplication by elements in the ring
- contains the 0 element of the ring

In the polynomial ring $k[x_1, ..., x_n]$, given have polynomials $f_1, ..., f_m \in k[x_1, ..., x_n]$ with $I = (f_1, ..., f_m) = \{h_1 f_1 + ... + h_m f_m | h_i \in k[x_1, ..., x_n]\}$ we say that I is ***finitely generated*** and that $f_1, ..., f_m$ is a ***generating set*** for I. The Gröbner basis, which we'll see later, is a special case of this. An ideal that is generated by a single element of the ring is called ***principal***.

Given a field, k, $k^n := \{(a_1, ..., a_n) \,|a_i \in k\}$ is the ***n-dimensional affine space***. If $f \in k[x_1, ..., x_n]$, then we define the ***affine variety*** corresponding to f as

$$\mathbb{V}(f) := \{(a_1, ..., a_n) \,|f(a_1, ..., a_n) = 0\} \subseteq k^n$$

We can further generalise this notion to a collection of polynomials $f_1, ..., f_m \in k[x_1, ..., x_n]$ by

$$\mathbb{V}(f_1, ..., f_m) = \{(a_1, ..., a_n) \,|f_i(a_1, ..., a_n) = 0 \,\forall i \in [1, m]\} \subseteq k^n$$

A natural extension of this concept is the notation; $I = (f_1, ..., f_m)$, $\mathbb{V}(I) = \mathbb{V}(f_1, ..., f_m)$.

The ***ideal generated by*** V, where V is an affine variety (strictly speaking $\mathbb{V}(I)$), is defined as $\mathbb{I}(V) = \{f \in k[x_1, ..., x_n] \,|f(a_1, ..., a_n) = 0 \,\forall (a_1, ..., a_n) \in V\}$. It's clear to see that $\mathbb{I}(V)$ is an ideal by considering factors of the polynomials in this ideal. $I \subseteq \mathbb{I}(V)$, though the equality generally does not hold.

2.4.2 Monomial Orderings

In multivariate polynomial rings it becomes necessary for us to decide how to order our variables. To this end we define the following notions:
Given a monomial $x_1^{\alpha_1}...x_n^{\alpha_n} \in k[x_1, ..., x_n]$ its ***total degree*** is $\alpha_1 + ... + \alpha_n$. Furthermore, if we write $x^\alpha := x_1^{\alpha_1}...x_n^{\alpha_n}$ where $\alpha = (\alpha_1, ..., \alpha_n) \in \mathbb{Z}^n_{\geq 0}$ then we can write $|\alpha| = |(\alpha_1, ..., \alpha_n)| := \alpha_1 + ... + \alpha_n$ for the total degree of x^α. In this way we have a one to one correspondence of monomials in $k[x_1, ..., x_n]$ with points in $\mathbb{Z}^n_{\geq 0}$.
A ***total order*** on $\mathbb{Z}^n_{\geq 0}$ is a binary relation $>$ satisfying:

1. $>$ is transitive: $\forall \alpha, \beta \in \mathbb{Z}^n_{\geq 0}, \alpha > \beta$ and $\beta > \gamma$ implies $\alpha > \gamma$
2. $>$ is trichotomous: $\forall \alpha, \beta \in \mathbb{Z}^n_{\geq 0}$ precisely on of the following holds:
   (a) $\alpha > \beta$
   (b) $\alpha = \beta$
   (c) $\alpha < \beta$

A ***monomial ordering*** on $k[x_1, ..., x_n]$ is a total order $>$ on $\mathbb{Z}^n_{\geq 0}$ satisfying:

1. $>$ *respects addition*: If $\alpha, \beta, \gamma \in \mathbb{Z}^n_{\geq 0}$ are such that $\alpha > \beta$, then $\alpha + \gamma > \beta + \gamma$.
2. $>$ is *well-ordered*: Every non-empty subset of $\mathbb{Z}^n_{\geq 0}$ has a smallest element under $>$. We write $x^\alpha > x^\beta$ if and only if $\alpha > \beta$.

A typical monomial ordering is ***lexicographic ordering***; $>_{lex}$. If $\alpha, \beta \in \mathbb{Z}^n_{\geq 0}$ we write $\alpha >_{lex} \beta$ if the left-most non-zero entry of $\alpha - \beta$ is positive. That is to say, $\alpha >_{lex} \beta$ if, at the first entry where α and β are not equal, we have $a_i > \beta_i$. For this reason $>_{lex}$ is sometimes known as *dictionary ordering*.
An alternative monomial ordering, that is widely used in the computational setting, is ***degree reversed lexicographic ordering*** (also known as *degrevlex* or *grevlex*; $>_{grevlex}$). If $\alpha, \beta \in \mathbb{Z}^n_{\geq 0}$ we write $\alpha >_{grevlex} \beta$ if $|\alpha| > |\beta|$ or if $|\alpha| = |\beta|$ and the right-most non-zero entry of $\alpha - \beta$ is negative.
Later on, in section 5.2, we will look at how these two orders compare with other monomial orders when implementing Gröbner basis methods to solve Sudoku puzzles.

Example 3. In the ring $k[x, y, z]$ with $x > y > z$ and using $>_{lex}$ we have

$$x^2y^9z >_{lex} xy^9z^2$$

Furthermore

$$x^2y^9z >_{lex} x^2y^8z >_{lex} x^2y^8$$

Now that we have a way to order our variables, we have the necessary machinery for the following section.

2.4.3 Multivariate Division Algorithm (General Polynomial Division Algorithm)

Given a polynomial $f = \sum a_\alpha x^\alpha \in k[x_1, ..., x_n]$ and a monomial order $>$ we define the following:

1. The ***multidegree*** of f is $multideg(f) := \max\{\alpha | a_\alpha \neq 0\}$, where the maximum is taken with respect to $>$.
2. The ***leading coefficient*** of f is $LC(f) := a_{multideg(f)}$.
3. The ***leading monomial*** of f is $LM(f) := x^{multideg(f)}$.
4. The ***leading term*** of f is $LT(f) := LC(f)LM(f) = a_{multideg(f)}x^{multideg(f)}$.

Theorem 4. *Multivariate Division Algorithm*

Let $>$ be a monomial order on $k[x_1, ..., x_n]$ and let $f_1, ..., f_s \in k[x_1, ..., x_n]$. Then every $f \in k[x_1, ..., x_n]$ can be written in the form:

$$f = a_1 f_1 + ... + a_s f_s + r$$

where $a_i, r \in k[x_1, ..., x_n]$ and either $r = 0$ or none of the monomials in r are divisible by any of the $LT(f_1), ..., LT(f_s)$. If $a_i f_i \neq 0$ then $multideg(f) \geq multideg(a_i f_i)$. r is called the remainder of f on division by $f_1, ..., f_s$.

The proof of the statement also describes the algorithm for the division process which generates the form

$$f = a_1 f_1 + ... + a_s f_s + r \tag{2.1}$$

The algorithm is given as follows:

Input: $f_1, ..., f_s, f$

Output: $a_1, ..., a_s, r$

$a_1 := 0; ...; a_s := 0; r := 0$

$p := f$

WHILE $p \neq 0$ DO

    $i := 1$

    divisionoccurred:=false

    WHILE $i \leq s$ AND divisionoccurred=false DO

        IF $LF(f_i)$ divides $LT(p)$ THEN

            $a_i := a_i + \frac{LT(p)}{LT(f_i)}$

            $p := p - \left(\frac{LT(p)}{LT(f_i)}\right) f_i$

            divisionoccurred:=true

        ELSE

            $i := i + 1$

    IF divisionoccurred=false THEN

        $r := r + LT(p)$

        $p := p - LT(p)$

What is happening is that we are testing, while $p \neq 0$, whether the leading term of any f_i divides that

of p. If it does then we proceed in the way that the univariate polynomial division algorithm does and modify p to be the remainder in this division. If it doesn't (for all i) then we shift this leading term of p to the remainder column and move on to testing the next monomial of f (that is, the LT of the new p).

We will show that the statement

$$f = a_1 f_1 + ... + a_s f_s + p + r \tag{2.2}$$

is inductively true at all times.

It's clearly true initially; when all $a_i = r = 0$ it yields $f = p$. Let us assume it's true at some point in the algorithm. We have two choices, then next step is either a step where division occurs, or a step where it doesn't and we add $LT(p)$ to the remainder r.

In the first case the process stops and we have (for a given i) $a_i := a_i + \frac{LT(p)}{LT(f_i)}$ and $p := p - \left(\frac{LT(p)}{LT(f_i)}\right) f_i$ and so 2.2 becomes

$$\begin{aligned} f &= a_1 f_1 + ... + \left(a_i + \frac{LT(p)}{LT(f_i)}\right) f_i + ... + a_s f_s + p - \left(\frac{LT(p)}{LT(f_i)}\right) f_i + r \\ &= a_1 f_1 + ... + a_s f_s + p + r \end{aligned}$$

that is, it is unchanged.

If no division occurs, we move $LT(p)$ to the remainder column and change $r := r + LT(p)$ and $p := p - LT(p)$. Now 2.2 becomes

$$\begin{aligned} f &= a_1 f_1 + ... + a_s f_s + p - LT(p) + r + LT(p) \\ &= a_1 f_1 + ... + a_s f_s + p + r \end{aligned}$$

Again unchanged. So this form holds true at each stage. The WHILE loop continues until $p = 0$. At this point the algorithm terminates and we arrive at $f = a_1 f_1 + ... + a_s f_s + r$ which is exactly the form we want. We now need to show that the algorithm actually terminates. This is simple however because at each stage $multideg(p)$ is decreasing (until it becomes 0 when the algorithm terminates). Recall that the Leading Term is the one which has the monomial of highest multidegree under the given monomial order. If we're in a step where division occurs, then since we redefine $p' := p - \left(\frac{LT(p)}{LT(f_i)}\right) f_i$ and $LT\left(\left(\frac{LT(p)}{LT(f_i)}\right) f_i\right) = LT(f_i)\left(\frac{LT(p)}{LT(f_i)}\right) = LT(p)$ by definition of Leading Term, above. Thus the

LTs of p and $\left(\frac{LT(p)}{LT(f_i)}\right) f_i$ cancel and $LT(p')$ must have a lower multidegree. If no division occurs then we redefine $p := p - LT(p)$ so clearly the multidegree decreases. Multidegree is a non-negative quantity and we saw earlier that monomial orderings are well-ordered, so this positive, decreasing sequence of multidegrees must terminate, meaning that the algorithm terminates also.

Finally, we also asserted that $\forall a_i f_i \neq 0$, $multideg(f) \geq multideg(a_i f_i)$. Recall that, by construction the monomials of a_i have the form $\frac{LT(p)}{LT(f_i)}$ for some i. Thus $multideg(a_i f_i) = deg\left(LT(a_i f_i)\right) = deg\left(LT\left(\frac{LT(p)}{LT(f_i)} f_i\right)\right) = deg\left(LT\left(p\right)\right) = multideg\left(p\right)$. We just showed that $multideg\left(p\right)$ decreases at each step and hence, since $p := f$ initially, we must have $multideg\left(f\right) \geq multideg\left(a_i f_i\right)$ for all $a_i f_i \neq 0$.

□

2.4.4 Gröbner Bases and the Ideal Membership Problem

It should be noted that the coefficients and remainder of the division algorithm outlined above are not unique, we will define Gröbner bases now, which solve this problem. Combining them with the division algorithm will give us a method to address the ideal membership problem.

Given an ideal $I \subset k\left[x_1, ..., x_n\right]$ and a monomial order. A finite subset $\{g_1, ..., g_s\} \subset I$ with the property $\left(LT\left(g_1\right), ...LT\left(g_s\right)\right) = \left(LT\left(I\right)\right) := \left(\{LT\left(f\right) \mid f \in I\}\right)$ is called a ***Gröbner basis***.

Proposition 5. *Let $G = \{g_1, ..., g_s\}$ be a Gröbner basis for a non-zero ideal $I \subset k\left[x_1, ..., x_n\right]$. Let $f \in k\left[x_1, ..., x_n\right]$. Then there exists a unique $r \in k\left[x_1, ..., x_n\right]$ such that:*

1. *No term of r is divisible by one of $LT\left(g_1\right), ..., LT\left(g_s\right)$*
2. *There exists a $g \in I$ such that $f = g + r$*

In particular, r is the remainder on division of f by G no matter what order we list the elements of G when running the division algorithm.

Proof. By the division algorithm we immediately have the existence of $f = a_1 g_1 + ... + a_s g_s + r$ with r (by construction) satisfying condition 1. of the proposition. For condition 2. we merely set $g = a_1 g_1 + ... + a_s g_s$. For uniqueness, assume $f = g + r = g' + r'$, then $r - r' = g' - g \in I$. If $r \neq r'$ then $LT\left(r - r'\right) \in \left(LT\left(g_1\right), ...LT\left(g_s\right)\right)$ by the definition of a Gröbner basis, hence $LT\left(r - r'\right)$ is divisible by $LT\left(g_i\right)$ for some i (as $LT\left(r - r'\right)$ is a single term). By definition, no single term of r or r' was divisible by any $LT\left(g_i\right)$. Notice that $LT(r) = LT(r')$ otherwise (w.l.o.g.) $LT\left(r - r'\right) = LT(r)$ and hence $LT\left(r - r'\right)$ is not divisible by any $LT\left(g_i\right)$, contradicting the earlier statement. Let $a_\alpha x^\alpha$ and

$a_{\alpha'}x^{\alpha'}$ be the first terms where r and r' first differ (recall they are not equal by definition). Then we have two cases:

- $\alpha = \alpha'$, hence wlog $a_\alpha > a_{\alpha'}$ with respect to our monomial order. Then $LT(r - r') = cLT(r)$ where c is an appropriate constant, so that $LT(r - r')$ is not divisible by any $LT(g_i)$
- wlog $\alpha > \alpha'$ in which case $LT(r - r') = LT(r)$ and so $LT(r - r')$ is not divisible by any $LT(g_i)$

Both of these contradict the earlier assertion, so we have that $r = r'$ and consequently $g = g'$ so that the result of division by a Gröbner basis is unique. □

Corollary 6. *Let $I \subset k[x_1, ..., x_n]$ be a non-zero ideal, and let $G = \{g_1, ..., g_s\}$ be a Gröbner basis for I. Let $f \in k[x_1, ..., x_n]$. Then $f \in I$ if and only if the remainder on division of f by G is zero.*

Proof. If the remainder on division by G is zero, then by the previous proposition we have $f = a_1g_1 + ... + a_sg_s$ which implies that $f \in I$. On the other hand if $f \in I$ then $f = f + 0$ satisfies the two properties in the same proposition, and by the uniqueness argument we used in the proof, we have that 0 is the unique remainder of division by G. □

Now we have a way to tell whether a given polynomial is a member of a given ideal or not; the ideal membership problem. We find a Gröbner basis for the ideal, divide the polynomial by this generating set using the division algorithm, and check whether the remainder is 0. This will come in useful in the next section when we look at polynomials and ideals relevant to graphs and what we can tell about the graph by solving the ideal membership problem.

2.4.5 Gröbner Basis Algorithms

The standard procedure for testing whether a generating set $G = \{g_1, ..., g_s\}$ of an ideal $I \subset k[x_1, ..., x_n]$ is a Gröbner basis is attributed to Bruno Buchberger and relies on examining the remainders of so called S-polynomials upon multivariate division by the elements of the generating set.

Definition 7. Given $f, g \in k[x_1, ..., x_n]$, the S-polynomial of f and g is defined

$$S(f, g) = \frac{x^\gamma}{LT(f)}f - \frac{x^\gamma}{LT(g)}g$$

where $\gamma := lcm\{LM(f), LM(g)\}$.

Theorem 8. *Buchberger's criterion*

Let $I = (f_1, ..., f_s) \subset k[x_1, ..., x_n]$ be a non-zero ideal, and fix a monomial order. A basis $G = \{g_1, ..., g_s\}$ for I is a Gröbner basis if and only if for all pairs $i \neq j$ the remainder on division of $S(g_i, g_j)$ by G (listed in some order) is zero.

In fact, Buchberger went further than this and found a way to transform any generating set of I into a Gröbner basis:

Theorem 9. *Buchberger's algorithm*

Let $I = (f_1, ..., f_s) \subset k[x_1, ..., x_n]$ be a non-zero ideal. The basis $G_0 = \{f_1, ..., f_s\}$ for I can be transformed into a Gröbner basis in finitely many steps. At each step the old basis G_m is transformed into a new basis G_{m+1} by adding in new elements given by all non-zero $\overline{S(f_i, f_j)}^{G_m}$, where $f_i, f_j \in G_m$. When $G_m = G_{m+1}$ we have that G_m is a Gröbner basis for I.

The proofs of these results may be found in [5], they aren't excessively long or difficult, but are unnecessary to this project. For this reason we omit them.
This algorithm gives us a theoretic way to find a Gröbner basis for any ideal of a polynomial ring in rational coefficients. In real life, however, as Hinkelmann and Arnold mention [9], the coefficients and degrees of the polynomials in the Gröbner basis calculation can become staggeringly large resulting in computations that either exceed the available computer memory, or cannot be completed in a reasonable time frame. I found this out myself, as will be mentioned in section 5.4.

Although I used a different approach to circumvent these difficulties; Brickenstein and Dreyer mention an interesting approach related to Gröbner basis algorithms in their paper on fast Gröbner basis calculation using Boolean polynomials [4].

Definition 10. A Boolean polynomial, p, is defined to be a member of the quotient ring

$$Q = \mathbb{Z}_2[x_1, ..., x_n] \setminus \langle x_1^2 - x_1, ..., x_n^2 - x_n \rangle$$

so that it has the form $p = a_1 x_1^{v_{1,1}} ... x_n^{v_{1,n}} + ... + a_m x_1^{v_{m,1}} ... x_n^{v_{m,n}}$, where $m, n \in \mathbb{Z}$, $a_i \in \mathbb{Z}_2$. The polynomials $x_i^2 - x_i$ in the ideal we quotient by force $v_{i,j} \in \{0, 1\}$, $\forall i, j$.

What this means is that we can view a monomial in n variables as a binary string of length n, and polynomials can be viewed as lists of these strings. For example $x_1^3 x_3 x_4^2 x_5^5$ in a polynomial ring

of 5 variables, would be represented as $x_1x_3x_4x_5$ in Q, which can be thought of as 10001. Similarly; $x_1x_2 + x_2x_3 + x_1x_3$ can be thought of as $\{110, 011, 101\}$.

Arithmetic operations with Boolean polynomials are quite simple [9]; addition of two polynomials is just the concatenation of the lists of binary strings representing them. E.g. $f = x_1x_2$, $g = x_1x_3 + x_2$ then f is represented by $\{110\}$, g by $\{101, 010\}$ and so $f + g$ is given by $\{110, 101, 010\}$.

Multiplication of two Boolean polynomials is achieved by using the OR operator to see whether a given variable appears in either polynomial being multiplied. For instance $x_1x_2^2$ and $x_1^3x_3$ when multiplied in Q give: $x_1x_2^2 \cdot x_1^3x_3 = x_1x_2 \cdot x_1x_3 = x_1x_2x_3$. Note also that in Q the operation of multiplication coincides with the operation of finding the least common multiple (LCM).

Division of monomials is comes from considering them as binary strings bit-wise and noting that f divides g if and only if $g - f > 0$. So x_1x_2 does not divide x_1x_3 because in the second bit we get $0 - 1 = -1$ (clearly this bit-wise consideration takes place in $\mathbb{Z}$ else it would be meaningless).

Finally whether two monomials are coprime comes from looking at each monomial bit-wise (i.e. consider $x_1^{a_1}$in f, and $x_1^{b_1}$in g) and seeing whether $min\{a_i, b_i\} = 0$. If this is so for all i then the two monomials are coprime.

In the context of the Boolean polynomial ring we can perform Buchberger's criterion to test for a Gröbner basis, but more importantly we can implement Buchberger's algorithm in a way that avoids producing excessively large coefficients or orders. Arnold remarks that it is necessary to introduce extra variables to encode all the information; she gives the example of a Shidoku puzzle requiring 64 variables, rather than 16, in the Boolean ring [9] (this is also explained in [2]). The time saved in avoiding large coefficients and orders greatly outstrips that added by increasing the number of variables. Indeed Arnold found that the Boolean polynomial method solved a Shidoku roughly 100 times faster than Gröbner basis methods in the ring of rational polynomials with lexicographic order.

3 Gröbner Bases and Graph Colouring

As we hinted in the introduction, there is a link between Gröbner basis theory and graph colouring. In fact, it comes down to using the ideal membership problem, the theory of which we have just seen, in conjunction with some appropriate choices of ideals and polynomials.
The polynomial in question is the graph polynomial, f_G, which we defined in 2.1.2. The ideals we will use, however, need to be defined.

3.1 Ideals related to graph colouring

We now proceed to link ideals with graph colouring, via a method from Hillar [8]. Given a graph $G = (V, E)$ on n vertices, let F be an algebraically closed field of characteristic not dividing k, so that it contains k distinct k^{th} roots of unity. Also let $R = F[x_1, ..., x_n]$ to be the polynomial ring over F in variables $x_1, ..., x_n$. Let $\mathcal{H}$ be the set of graphs with vertices $\{1, ..., n\}$ consisting of a clique of size $k+1$ vertices and all other vertices isolated, that is to say, the set of all complete graphs built on a choice of $k+1$ vertices out of n. The ideals that will interest us are:

- $J_{n,k} = \langle f_H : H \in \mathcal{H} \rangle$
- $I_{n,k} = \langle x_i^k - 1 : i \in V \rangle$
- $I_{G,k} = I_{n,k} + \langle x_i^{k-1} + x_i^{k-2}x_j + ... + x_i x_j^{k-2} + x_j^{k-1} : \{i, j\} \in E \rangle$

They are related to graph colouring in a theorem we shall see shortly.

Recall that we earlier defined a colouring as a function $c : V(G) \longrightarrow \{1, 2, ..., k\}$. Since we will work over the k^{th} cyclotomic field (e.g. see A.2), which is defined as $\mathbb{Q}[\zeta_k]$, the rational numbers with a k^{th} primitive root adjoined, the k-element set representing our colours will actually be the set of k^{th} roots of unity (see sections 5.3 and 5.4 for the reasoning behind this) .
The ideal $I_{n,k}$ ensures that each x_i (representing a vertex of the graph) takes the value of a k^{th} roots of unity. Since the k^{th} roots of unity represent our colours, this is equivalent to setting the condition that each vertex be coloured with a valid element from our k-element set of colours. What this actually means is that the variety $\mathbb{V}(I_{n,k})$ represents all possible (not necessarily proper!) colourings of the graph G, because it takes each vertex, in turn, and assigns to it a valid colour.
The ideal $I_{G,k}$ is even more useful since its variety corresponds to the proper colourings of G:

Lemma 11. $\mathbb{V}(I_{G,k})$ *corresponds exactly to the proper* k*-colourings of* G*. [8]*

Let $\underline{v} = (v_1, ..., v_n) \in \mathbb{V}(I_{G,k})$ and let $\{i, j\} \in E$ be an edge of G. we define

$$q_{ij} = \frac{x_i^k - x_j^k}{x_i - x_j}$$

Note that by definition;

$$q_{ij} = x_i^{k-1} + x_i^{k-2}x_j + ... + x_j^{k-1} \in I_{G,k} \tag{3.1}$$

If $v_i = v_j$ the colouring is improper and $q_{ij}(\underline{v}) = kv_i^{k-1} \neq 0$, by substituting into 3.1 and because v_i is a k^{th} root of unity, so $v \notin \mathbb{V}(I_{G,k})$. Therefore, the contrapositive gives us that if $\underline{v} \in \mathbb{V}(I_{G,k})$, it represents a proper colouring.

Conversely, let $\underline{v} = (v_1, ..., v_n)$ represent a proper colouring.

$$q_{ij}(\underline{v})(v_i - v_j) = \left(v_i^k - v_j^k\right) = 1 - 1 = 0$$

Again by 3.1 and because v_i, v_j are k^{th} roots of unity. Since we have a proper colouring, $v_i \neq v_j$ (recall $\{i, j\} \in E$) and hence it must be that $q_{ij}(\underline{v}) = 0$ since we are working over a field, which is an integral domain by definition. Therefore $\underline{v} \in \mathbb{V}(I_{G,k})$. □

Theorem 12. *The following statements are equivalent:*

1. *The graph* G *is not* k*-colourable.*
2. $\dim_k R/I_{G,k} = 0$.
3. *The constant polynomial* 1 *belongs to the ideal* $I_{G,k}$.
4. *The graph polynomial* f_G *belongs to the ideal* $I_{n,k}$.
5. *The graph polynomial* f_G *belongs to the ideal* $J_{n,k}$.

In fact this statement is excessive (the complete proof may be found in [8]), since we saw above that $\mathbb{V}(I_{G,k})$ corresponds exactly to the proper k-colourings of G then we are only really concerned with the proof of (1) $\iff$ (3):

Proof. By definition and Lemma 11, G is not k-colourable if and only if $\mathbb{V}(I_{G,k}) = \emptyset$. Furthermore, by definition we are working over an algebraically closed field so that we may invoke the weak Nullstellenstaz [10], which says that $\mathbb{V}(I_{G,k}) = \emptyset \iff I_{G,k} = k[x_1, ..., x_n]$, where $k[x_1, ..., x_n]$ is the polynomial

ring over the field, and in the number of variables, that we have chosen to work over. Now, clearly, $I_{G,k} = k[x_1, ..., x_n] \iff 1 \in I_{G,k}$, which gives us the required result. □

What we are now able to do is examine, using computational algebra tools, the variety $\mathbb{V}(I_{G,k})$ to see whether G is not k-colourable, or, if it is, to identify all the proper k-colourings.

4 Applications to Sudoku

Sudoku is a global phenomenon; a simple number game which involves placing the digits 0 to 9 in a grid following certain rules. The game originated in Japan in the late 80s, but has seen an explosion in popularity across the world since 2005 [6]. The standard Sudoku is a 9×9 grid comprised of nine 3×3 "sub-grids". Each sub-grid must contain the digits 0 to 9 (each digit occurring once, and only once) in such a way that when the whole 9×9 grid is considered, the digits 0 to 9 appear once, and only once, in each row and column.

Because Gröbner basis methods are quite computationally expensive, and I have access to limited resources, though my code is written for a general $n^2 \times n^2$ Sudoku most of my actual examples will focus around 4×4 Sudoku; known as Shidoku (from the Japanese *Shi*, meaning "four").

4.1 Clarification of terms

The terms row and column are obvious, however the term ***sub-grid*** may require a definition: it is a collection of n^2 cells (in an $n^2 \times n^2$ Sudoku) which form an $n \times n$ square. These are usually indicated by thicker lines separating different sub-grids, refer to Fig. 4.2 where the first sub-grid contains the numbers $0, 1, 4$ and 5.

Another term that will be of use is that of a ***band***. A band can be thought of as a row of sub-grids, that is; it is a collection of n horizontally connected sub-grids. A ***stack*** is the vertical equivalent of a band, and a ***chute*** is the generic term for stacks and bands.

The diagrams below demonstrate a correctly solved Sudoku, as well as visual representation of the terms in italics above; a sub-grid in purple, a row and column in blue and red respectively, and a band and stack in green and yellow respectively.

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

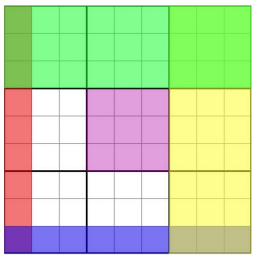


Figure 4.1: Sudoku terminology

4.2 Comparison of Gröbner basis methods with effective Sudoku solving methods

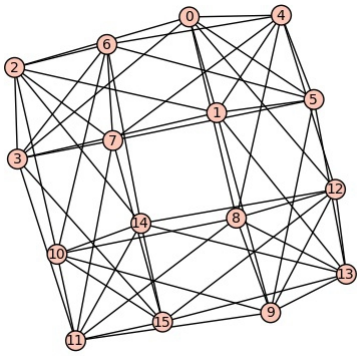
The theory of graph colouring and Gröbner basis methods can be applied to Sudoku, when it is modeled as a graph, to solve it. This is not the most efficient way to solve Sudoku because Gröbner basis calculations are very computationally taxing, and the method using the ideals above is quite brutish, given it formulates conditions at once for the entire Sudoku grid, which naturally introduces a lot of redundant conditions. Indeed, I found this out the hard way when my code surpassed the memory available to the PC whilst trying to solve the 9×9 Sudoku with no initial conditions.

Chi and Lange [12] talk about several methods to solve Sudoku, among which the most efficient is a process known as "Backtracking". Backtracking takes a partial solution to a Sudoku and tries to extend it, one cell at a time, into a full solution. The visual representation of this is as a tree, whose vertices are the partial solutions. The partial solutions increase in the amount of information they contain as we climb the tree. When an inconsistent solution is reached, the algorithm "backtracks" to the previous vertex and goes along the next branch. In this way the final product of the algorithm is a tree representing all possible solutions of the Sudoku.

Whereas my code was unable to solve a 9×9 Sudoku due to running out of memory, the Backtracking algorithm is able to solve a puzzle rated "difficult" in just under one hundredth of a second CPU time [12], on similar hardware. This only goes to show that the exploration of Sudoku in this project is not included for the sake of furthering the study of solving Sudoku, rather for the purpose of showing an interesting application of the theory of Gröbner bases applied to graph colouring methods.

4.3 The graph of a Sudoku

Each of the digits 0 to 9 can be thought of as our nine, distinct, colours. Each cell (there are 81 in a Sudoku) can be modeled as the vertex of a graph, and the edges of the graph link precisely those pairs of vertices which are in the same row, column or sub-grid and therefore cannot contain the same digit (colour). In this way, each proper colouring of the graph of the Sudoku corresponds to a legitimate solution of the Sudoku. The SAGE code to generate the graph of an $n^2 \times n^2$ is in appendix A.1. Below is a sample output of the code; the graph of a 4×4 Sudoku, it corresponds to the Sudoku with cells numbered as in the figure next to it. Note that in literature a 4×4 Sudoku is called a *Shidoku* (from the Japanese "shi", meaning "four"). I will adopt this term henceforth.



0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Figure 4.2: A Shidoku and its graph

4.4 The minimal clues problem

Generally when a Sudoku puzzle is set, say in a newspaper, it is presented in such a way that it has a unique solution. That is, given the partial solution supplied, there is only one way to complete the solution. We say that the partial solution extends uniquely to a solution. Varying levels of difficulty are then achieved by giving different amounts of information in the partial solution supplied to the reader.

A question that has long interested Sudoku aficionados is what the minimal number of clues in a partial solution is, such that there is a unique solution to this Sudoku. In the 9×9 case this has long been suspected to be 17, indeed just under $50,000$ puzzles with 17-clue partial solutions with an unique solution were collected by Gordon Royle [15] in his search to find a unique puzzle with a 16-clue partial solution. This conjecture was finally proved by Gary McGuire [14] and published in 2012, through a method that was broadly brute force with some clever tweaks, showing that no 16-clue partial solution extends uniquely to a solution.

The Shidoku case lends itself much more to basic computation by hand. The minimal number of clues a unique Shidoku can have can be shown to be 4 fairly easily by the following method:

- If we have a partial solution with less than 3 clues in it then at least 2 of the "colours" are unused, and however we extend this partial solution, we can always swap the two originally unused colours to get two isomorphic but non-identical Shidoku solutions which both came from the same partial-solution
- Clearly there is a finite number of partial solutions with 3 clues; an obvious upper bound is 4^{16}.

This is, in fact, far too large because of the rules of Sudoku

- Furthermore, we only pay attention to partial solutions with 3 clues all of which are different "colours" otherwise we could easily swap the two unused colours, as argued above, such that the partial solution does not extend uniquely
- We identify these partial solutions and show that none of them extends uniquely to a solution of the Shidoku by using the SAGE code given in the appendices
- We display a partial solution with 4 clues that does extend uniquely

If we have 3 clues, then clearly one of the following three cases must occur:

1. All 3 clues are in the same sub-grid
2. 2 clues are in the same sub-grid and the 3^{rd} is in a different sub-grid
3. All 3 clues are in different sub-grids

Considering the partial solutions in this way is helpful because it avoids the possibility of counting the same configuration of 3 clues twice, since the overlap of configurations between the three cases above is clearly empty by definition.
Furthermore, we are only interested in non-isomorphic configurations of the 3 clues, that is to say that if one configuration can be transformed into another via the action of elements of $Aut(G)$ then it is a waste of time to check both of these partial-solutions. The reason is that if one of them extends uniquely (or not) then obviously the other must too, as the complete solutions of both can be transformed into each other via the action of the same elements of $Aut(G)$. Recall that these elements of $Aut(G)$ include (but are not limited to):

- Row swaps within a band
- Column swaps within a stack
- Reflection in any line of symmetry of the grid
- Band or Stack swaps

Finally, whether or not these partial-solutions actually extend to a valid solution of the Shidoku is not of immediate concern to us, since the SAGE code will be able to flag that error. More importantly, if

this happens it still means that the solution does not extend uniquely, which is what we're trying to prove.

We now consider the three cases above to see what configurations of clues in partial solutions we are interested:

If all 3 clues are in the same sub-grid, then we only need to concern ourselves with the following configuration:

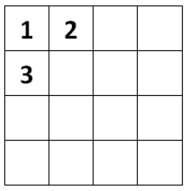
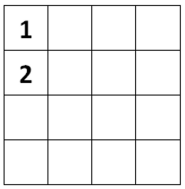


Figure 4.3: Shidoku minimal clues I

It doesn't take much work to see that through a row or column swap we can force the empty cell in the sub-grid to be in any of the four spaces in the sub-grid, so all four configurations are actually isomorphic and we only need to examine one.

Running this partial colouring through SAGE using a combination of the code given in the appendices tells us that this partial solution extends to 12 non-isomorphic solutions of the Shidoku.

If 2 clues are in the same sub-grid and the 3^{rd} in a different sub-grid, then the 2 clues can (by symmetry and w.l.o.g.) be arranged in the following way:



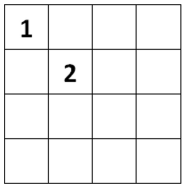


Figure 4.4: Shidoku minimal clues IIa

Going from here, it isn't hard to see that putting the 3^{rd} clue anywhere within the diagonally opposite sub-grid is equivalent to putting it in the bottom-right cell after row/column swaps if necessary. Furthermore, in the left-hand diagram above, if we place the 3^{rd} clue in the upper-right sub-grid,

through row and column swaps all four choices are isomorphic to putting it in the top-right cell, in the bottom-left sub-grid the a row swap gives means we only need to check the two positions in the bottom row. For the right-hand diagram above, by symmetry putting the clue in the top-right and bottom-left sub-grids is the same thing, so the configurations where the 3^{rd} clue is in the bottom row, left stack, interest us.

Overall, there are 7 configurations and they extend as follows:

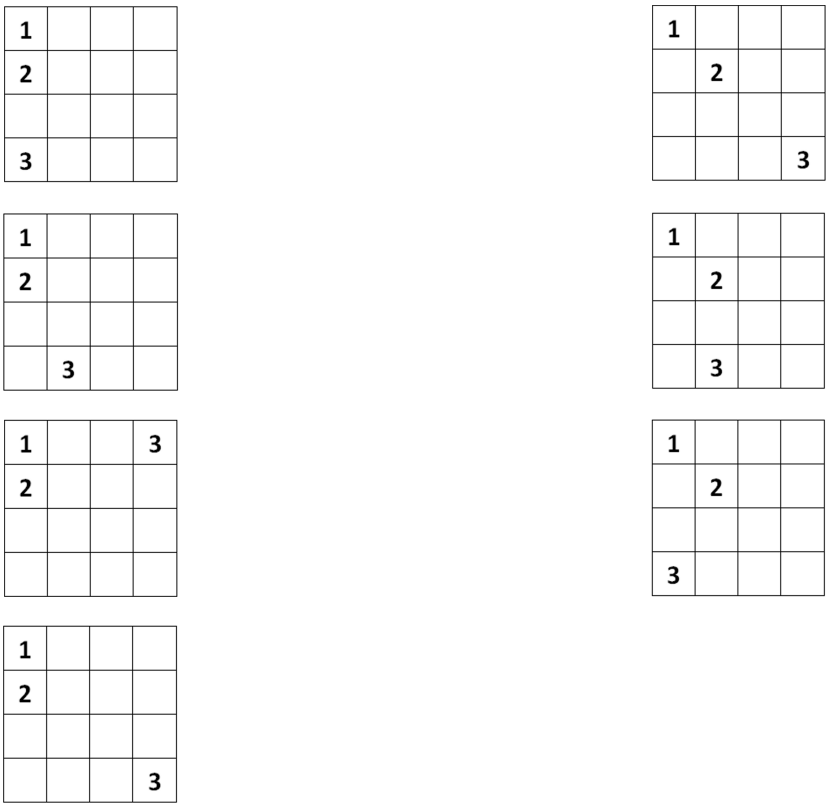


Figure 4.5: Shidoku minimal clues IIb

In the final case all 3 clues are in different sub-grids. Clearly this means that 2 of the clues must be in diagonally opposite sub-grids, if we put the first clue in the top-left cell w.l.o.g. then through row and column swaps we see that the clue in the bottom-right sub-grid can be placed in the bottom-right cell:

by symmetry we only need to consider the 3^{rd} clue in the bottom-left sub-grid, and again only in three of the four cells, as follows:

1			
			2

Figure 4.6: Shidoku minimal clues IIIa

1			
3			
			2

1			
	3		
			2

1			
3			2

Figure 4.7: Shidoku minimal clues IIIb

We see now that all possible partial solutions with 3 clues, up to isomorphism, do not extend uniquely to a Shidoku solution. All that remains to do is show that a partial solution of 4 clues that extends uniquely does exist. Consider the partial solution below

1			
		4	
	3		
			2

Figure 4.8: Shidoku 4 clues unique extension

Clearly the bottom-left and top-right cells are immediately soluble, and hence the bottom-right and top-left sub-grids; so far there has been no choice. From here we observe that there is always a row or column with one empty cell, and hence complete it uniquely. Finally we arrive at this unique extension of the partial solution:

1	4	2	3
3	2	4	1
2	3	1	4
4	1	3	2

Figure 4.9: Shidoku 4 clues unique extension, solved

4.5 The automorphism group of the graph of a Shidoku

Given the graph of a Shidoku, I can use the inbuilt SAGE [16] function `.automorphism_group()` to generate $Aut(G)$. This, however, returns a list of all 128 elements of the group, as a subgroup of the permutation group on 16 letters, rather than a symbolic presentation of the group. By using the `.gens()` SAGE function I can obtain the following generators for $Aut(G)$ (note that SAGE handles permutation groups with strictly positive integers so the cell we originally labeled 0 is here labeled 16, though this makes no change to the transformations):

$$\begin{aligned}
a: &= (8\,12)\,(9\,13)\,(10\,14)\,(11\,15)\\
b: &= (2\,3)\,(6\,7)\,(10\,11)\,(14\,15)\\
c: &= (1\,3)\,(2\,16)\,(4\,6)\,(5\,7)\,(8\,10)\,(9\,11)\,(12\,14)\,(13\,15)\\
d: &= (1\,4)\,(2\,8)\,(3\,12)\,(6\,9)\,(7\,13)\,(11\,14)\\
e: &= (1\,16)\,(4\,5)\,(8\,9)\,(12\,13)
\end{aligned}$$

If we observe the amended numbering of the Shidoku square:

16	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Figure 4.10: Renumbering of Shidoku for $Aut(G)$

It's not hard to see that the generators correspond to the following transformations:
a is a swap of the 3^{rd} and 4^{th} rows, b is a swap of the 3^{rd} and 4^{th} columns, c is a swap of the 1^{st} and 3^{rd}, and 2^{nd} and 4^{th} columns (a swap of the 1^{st} and 2^{nd} stacks), d is a reflection in the leading diagonal and e is a swap of the 1^{st} and 2^{nd} columns.
Putting the above generators into the MAGMA computer algebra package, the `FPgroup()` command yields the following presentation of the group:

$$Aut(G) = \left\langle a, b, c, d, e \,|\, a^2, b^2, c^2, d^2, e^2, abab, acac, ecbc, adbd, bebe, (cd)^4 \right\rangle$$

Arnold et. al. [1] found the structure of this same group to be:

$$H_4 = \langle r, s, t \,|\, r^4, s^2, t^2, trtr, rsr^2tsr^3t, tststs, srsr^3srsr^3 \rangle$$

The Magma `IsIsomorphic()` command confirms that $Aut(G) \cong H_4$ (it is recommended to convert both groups into permutation groups via `PermutationGroup()` before testing for isomorphism), which seems to suggest that our version of the permutation group has superfluous generators; in fact it's easy to see that, for example, the relation $adbd = 1$ implies that $a = dbd$, so that we could get rid of the generator a.

In fact, we can observe the following:

$$\begin{aligned} adbd &= 1 \iff adb = d \iff dadb = 1 \iff b = dad \\ ecbc &= 1 \iff e \cdot (cbc) = 1 \iff e = cbc \end{aligned}$$

so that we can get rid of the generators b and e.
This now means we can offer the following, simpler, presentation:

$$Aut(G) = \left\langle a, c, d \,|\, a^2, c^2, d^2, (ad)^4, acac, (dadc)^4, (cd)^4 \right\rangle$$

These are, by no means, the same presentation of the automorphism group, since many such presentations exist. However, we know have a concise group theoretic definition of the group that dictates which transformations of the Shidoku board preserve the rules of Sudoku.

4.6 Uniqueness of the colouring, up to isomorphism

Generating all proper colourings of graph is not where our interest in this matter stops. If we find one solution to a Sudoku and then rotate it by 180^o then we ought not to claim that this is a different solution, since it doesn't truly contain a different configuration of colours (digits) - there is nothing new here. Consider these two Shidoku solutions:

The pattern of the colourings looks very similar. In fact the case here is that these colourings are isomorphic. What this means is that there is a bijection from $\{1, ..., k\}$ to itself (or rather, a permutation $s \in$

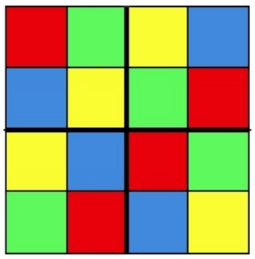


Fig. 4.3

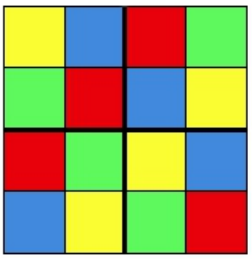


Fig 4.4

Figure 4.11: Isomorphic Shidoku example

S_k, where k is the number of colours) such that the second colouring is the image of the first under this permutation. In this case the map is $\{red \longrightarrow yellow, yellow \longrightarrow red, blue \longrightarrow green, green \longrightarrow blue\}$, and it happens to be its own inverse.

So, we have established that isomorphic colourings do not interest us; we want one representative from each class of isomorphic colourings. But how do we, first and foremost, identify the number of total solutions and non-isomorphic solutions, and furthermore, extract the relevant solutions?

I will first describe a method by Arnold et. al. using Gröbner bases, because it is applicable to larger Sudoku puzzles, as well as relevant to this project. I will then demonstrate how many Shidoku solutions there are up to isomorphism. I will then explain how I used the automorphism group of the graph of the Shidoku, G, to find these solutions; a detailed explanation of the code supplied in appendix A.3 is in section 5.1.3.

Above I have used the roots of unity approach to describing the information in a Sudoku using equations, and then using Gröbner basis methods to solve them. Arnold describes an alternative characterisation specific to Shidoku which is called the *sum product Shidoku system [2]*. It generates equations for the Shidoku based on the fact that the cells in any sub-grid must add to 10 and multiply to give 24 (Since the cells have values $1, 2, 3, 4$).

Arnold labels the vertices of her Shidoku from a to p and gives the following Gröbner basis for the set of equations from the *sum product system*:

If we were to begin with a basic counting argument (as I will demonstrate later) to try to enumerate the number of Shidoku solutions, we might say that there are 4 choices for cell a, 3 for cell b, 2 for cell c and hence 1 for cell d. Furthermore, there are 2 choices for cell e (which is in the same row as a

$$
\begin{aligned}
p_1 &= a^4 - 10a^3 + 35a^2 - 50a + 24 \\
p_2 &= b^3 + b^2a + lowerterms \\
p_3 &= c^2 + bc + lowerterms \\
p_4 &= d + c + b + a - 10 \\
p_5 &= e^2 + lowerterms \\
p_6 &= f + e + b + a - 10 \\
p_7 &= g^2 - gb - ga + ab \\
p_8 &= h + g - b - a \\
p_9 &= i^2 + lowerterms \\
p_{10} &= j^2 - je - ja + ae \\
p_{11} &= 18k + lowerterms \\
p_{12} &= 18l + lowerterms \\
p_{13} &= m + i + e + a - 10 \\
p_{14} &= n + j - e - a \\
p_{15} &= 18o + lowerterms \\
p_{16} &= 18p + lowerterms \\
p_{17} &= 9gj + lowerterms
\end{aligned}
$$

Figure 4.12: Gröbner basis for the *sum product system*

and b) and this then decides the 1 choice for cell f; the 4^{th} cell in that row. So for these first 6 cells we have a total of $4 \cdot 3 \cdot 2 \cdot 1 \cdot 2 \cdot 1 = 48$ different configurations. Arnold observes that this is also the product of the powers of the leading terms of the first six polynomials in the Gröbner basis for the *sum product Shidoku system.*

If we blindly continue with this method it would seem that there are

$$4 \cdot 3 \cdot 2 \cdot 1 \cdot 2 \cdot 1 \cdot 2 \cdot 1 \cdot 2 \cdot 2 \cdot 1 \cdot 1 \cdot 1 \cdot 1 \cdot 1 \cdot 1 \cdot 1 = 384 \tag{4.1}$$

Shidoku solutions. There is a problem here however. When we work through the enumeration of Shidoku solutions below, we will see that upon completing a given number of cells, certain choices give rise to a different number of solutions and that the consequent number of Shidoku solutions is 288. Arnold explains that this "branching effect" comes from the fact that the polynomial p_{17} in the Gröbner basis has a leading term that relies on two variables, rather than one.

The way to account for this branching is based on considering the monomials formed by products of the variables a to p that are not divisible by the leading term of any polynomial in the Gröbner basis; the proof of this fact is omitted as I merely intend to give a flavour of the methodology here. We saw that the letters a to p represent the cells of the Shidoku. Any monomial that is the product of powers of these letters has form $a^{r_1} \cdot b^{r_2} \cdot ... \cdot p^{r_{16}}$ where, for example, r_1 can take the values $0, 1, 2, 3$ but not greater than or equal to 4 since the leading term of p_1 is $a_4$and this would then divide such a monomial. If we proceed in this way we see that the number of such monomials is exactly 384 as calculated above. However, we now account for the fact that the leading term of p_{17} is gj, which will divide any monomial where r_7 and r_{10} (the powers of g and j respectively) are not 0. Since r_7 and r_{10} must take values 0 or 1 (this can be seen by looking at 4.1) we see that the monomials that must be removed are those where $r_7 = r_{10} = 1$. The number of such monomials is $4 \cdot 3 \cdot 2 \cdot 1 \cdot 2 \cdot 1 \cdot 1 \cdot 1 \cdot 2 \cdot 1 \cdot 1 \cdot 1 \cdot 1 \cdot 1 \cdot 1 \cdot 1 = 96$,

so that the actual number of Shidoku solutions is $384 - 96 = 288$.

The following sequence of diagrams describes a practical way to show the number of Shidoku solutions is 288. However, I have gone a step further and arbitrarily labeled the top-left sub-grid in a specified way. This cuts the number of solutions by a factor of $4! = 24$, but since we are only interested in non-isomorphic solutions (i.e. those that aren't the images of each other under the action of an element of S_4) then this colouring of the top-left sub-grid essentially rids us of duplicate, isomorphic solutions. In the diagrams below, the numerals represent the number of choices for a given cell, given the colours in the other cells of the Shidoku.

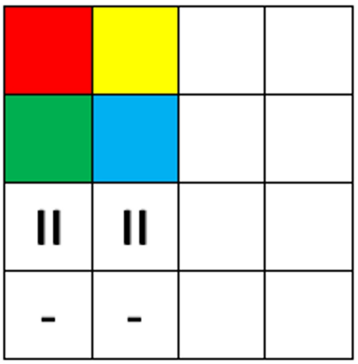


Figure 4.13: Shidoku enumeration I

Clearly we have two choices for each of the cells with numerals above. If we choose, say, yellow/red, then the bottom two cells are automatically chosen as blue/green. For this reason we will consider the choice of yellow/red and blue/green as the same for now. Thus we see that there are two choices of pairs for the cells with numerals above (they would be cells 8 and 9 by our standard Shidoku numbering method as in 4.3), these are yellow/red or yellow/green. Both choices are shown below in the left hand and right hand boxes, respectively.

For the choice yellow/red we see that we can have two choices for cell number 3, w.l.o.g. we colour it green and then there are a further two choices for cell number 6; this determines all other cells' colours. For the choice yellow/green (right-hand side), however, this is not the case. Once we choose, w.l.o.g., to colour cell 3 in green, we see that there is no longer a choice between yellow and red for cell 6 because if we colour it yellow, then no colour can legitimately be placed in cell 14.

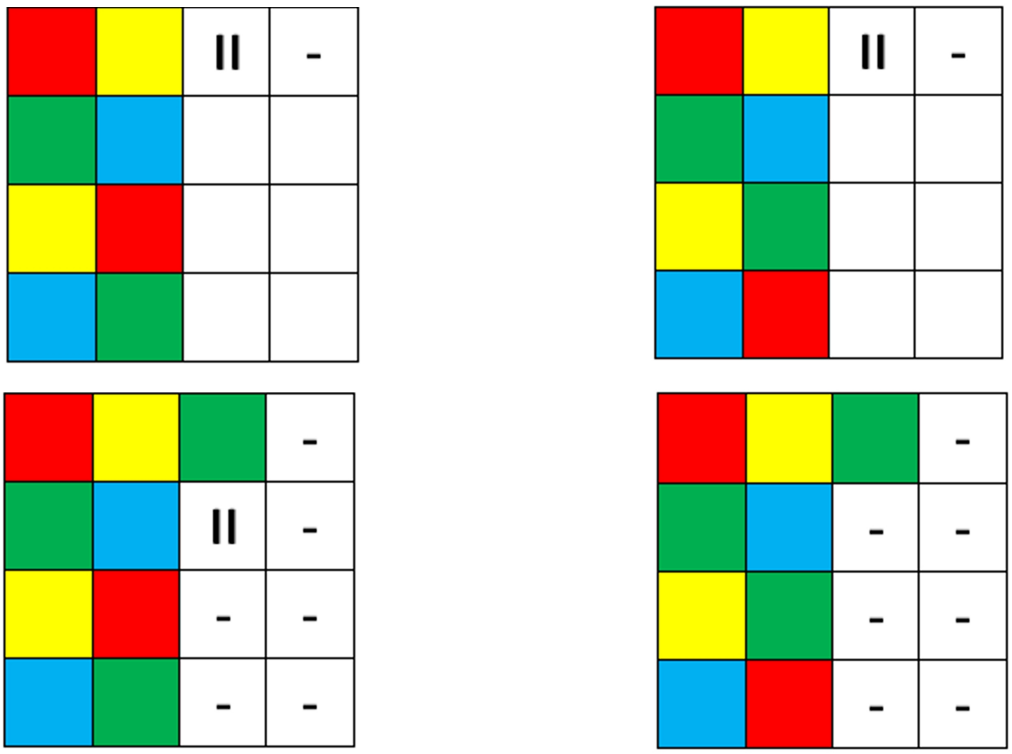


Figure 4.14: Shidoku enumeration II

To summarise: After colouring the top-left sub-grid arbitrarily, we have four choices of pairs for cells 8 and 9 that will decide how the two left-hand sub-grids are coloured. Two of these choices (yellow/red and blue/green) give us a further two choices for cell 3 and two choices for cell 6. The other two choices (yellow/green and blue/red) give us a further two choices for cell 3 and nothing more.
Hence the total number of Shidoku solutions (up to isomorphism) is $2 \cdot 2 \cdot 2 + 2 \cdot 2 = 8 + 4 = 12$.

We can think of these 12 as classes of Shidoku that are non-isomorphic and actually represent $4! = 24$ Shidoku each, hence the earlier value for the total number of Shidoku solutions; $288 = 12 \cdot 24$. How do we actually isolate these non-isomorphic solutions? The code in appendix A.3 does this for general Sudoku, and is explained in the following section, however it relies on the following method:

- Take the automorphism group, H, of the graph, G, of the Sudoku. By the definition in section 2.3 this is the group of all permutations of the vertices (hence a subgroup of S_{n^4}) that preserve edge relations between vertices
- Each element of H is, therefore, a map between valid Sudoku solutions. In certain cases, some of

these will be equivalent to relabellings of the colours of the cells, and so will not interest us. Note that this means that the 12 representatives of the classes of non-isomorphic Shidoku solutions may be different if the variety calculated in appendix A.2 is calculated in a different order, but this doesn't matter as each of these solutions is a *representative* of a class up to isomorphism

- Given a pair of colourings of a Shidoku in the form $(B_0, ..., B_{15})$ and $(C_0, ..., C_{15})$, we can test whether they are isomorphic by looking at how many values $(k+1)\cdot B_i + C_i$ takes as i ranges across the cells. For isomorphic colourings it should take exactly k values; hence for non-isomorphic colourings it must not take k values

- The reasoning behind this is two colourings are isomorphic if and only if there is a permutation in S_k taking one to the other. This happens if and only if each colour is uniquely mapped to to another colour (possibly itself) and since all k colours are used in the colouring of Sudoku/Shidoku, we can expect exactly k distinct pairs (B_i, C_i) where $s : B_i \longrightarrow C_i$ for some $s \in S_k$.

5 Computational Implementation of Sudoku as a Graph Colouring Problem

Now that the relevant preliminaries and results have been stated and proved, we move on to implementing them using the computer algebra package SAGE. SAGE is an open-source software based on the Python language with access to an extensive array of mathematical packages, including ones that handle graph theory and Gröbner basis calculations, that are necessary to implement the above theory. As discussed previously, I aim to apply the above theory in the context of solving Sudoku by representing it as a graph and approaching the colouring via Gröbner basis methods. In this section I offer an explanation of the SAGE code (found in appendix A), to help the reader bridge the gap between the Gröbner basis colouring method theory and its practical implementation in the code to colour a given Sudoku. I also offer an account of the difficulties which I had to overcome to get there.

5.1 Companion to the Code

5.1.1 Generating the graph of an $n^2 \times n^2$ Sudoku

As mentioned above, the code to generate the graph of an $n^2 \times n^2$ Sudoku can be found in appendix A.1. Note that I intend to label the vertices of the graph corresponding to, say, a Shidoku as in Fig. 4.2; starting from 0 and going up to $n^4 - 1$.

Observe, also, that this code is written in the form of a SAGE function and is intended to be used, along with other functions, by a main function to generate the list of all permissible, proper colourings. This will be exhibited later. The principle behind this code is as follows:

We begin, in lines 3 to 5, by defining an empty graph and adding n^4 vertices to it. There are still, however, no edges yet. Notice that this function takes, as input, n, which is the size of the side of the sub-grid rather than the whole grid. Therefore to generate the usual 9×9 Sudoku, I would call it with $n = 3$, thus we would have $3^4 = 81$ vertices, which is correct.

In lines 6 to 14 we set up 3 nested loop variables L_2, L_3 and L_4 all ranging from 0 to n^2. L_2 cycles us through each row of the Sudoku (remember it's equivalent to the graph in Fig. 4.1), L_3 cycles us through each row element and L_4, again, cycles us through each row element. Between lines 9 and 13 we set up variables u and v, the goal is to add an edge between vertices u and v to the edge set of G. In line 9, u is defined based on the row (due to L_2) and position in the row (due to L_3), we then use L_4 to define v as each vertex in the same row as u, successively, and add the appropriate edge to the

edge set of G. Note that the if/else clause is there to ensure that when we define v from u, we don't define v as a vertex that is actually in the next row; it works as a modulo n^2 (the number of cells in a row) operator. In essence, this ensures that (referring to Fig. 4.1 and Fig. 4.2) we join, say, vertex 1 to vertices 2, 3 and 0, rather than 2, 3 and 4, which would be incorrect.

In lines 15 to 23 this process is repeated for the columns of the Sudoku. The code has the same structure, but for some minor alterations to indexing to account for the fact that we're putting edges in between elements in the same column, not the same row, now. This time the if/else clause ensures we don't define vertex y from vertex x in a way that means it's not in the vertex set of G. That is to say, the clause functions as a modulo n^4 (the number of cells in the whole Sudoku) operator, ensuring that, say, vertex 5 gets joined to vertices 9, 13 and 1, rather than 9, 13 and 17, which would not only be wrong but would introduce new vertices into our graph.

Finally, lines 24 to 33 repeat this process a third time to add edges between all vertices that are in the same sub-grid. There is a notable difference in this section, however, in the approach. The first loop variable, N_2, takes us through each of the n^2 sub-grids. We then define an empty list into which we will put the vertices that occur in this sub-grid. Since there isn't such a regular pattern to the numbering of vertices in a sub-grid as opposed to a row or column (compare the first row of Fig. 4.2 - $0, 1, 2, 3$, and the first column - $0, 4, 8, 12$, with the first sub-grid - $0, 1, 4, 5$) we have to be a little careful with the population of this list. The second loop variable, N_3, cycles us through the rows within the specified sub-grid and the third loop variable, N_4, takes us through the elements in each of those rows (within the sub-grid).

In line 28, the definition of *elt* as each successive element of a given sub-grid works as follows:

$\frac{N_2-(N_2\,(mod\,n))}{n} \cdot n^3$ puts us in the correct band, given the sub-grid (numbering the sub-grids from the top and left).

$(N_2\ (mod\,n)) \cdot n$ moves us along the top row of the selected band, to the upper-left element in the desired sub-grid.

$(n^2) \cdot N_3$ shifts us down by a row for each iteration of N_3, and N_4 moves us along the row, within that sub-grid.

Now that we've added edges for each pair of points in the same row, column or sub-grid, we have a graph which completely embodies the relationships of cells in a Sudoku and the problem of solving the Sudoku is now translated into the problem of finding proper colourings of this graph.

5.1.2 Colouring the graph of an $n^2 \times n^2$ Sudoku

Now that we have the graph of a Sudoku, we can colour it using Gröbner basis methods. The code in appendix A.2 does this using the ideals described in Hillar's paper [8] that were discussed in section 3.1. Furthermore, my code takes a list of triples which are user defined by another function and describe a partial colouring of the graph in the form *(row, column, colour)*, and incorporates them into the solution process by appending algebraic characterisations of the information contained to the relevant ideal. The code returns a list of dictionaries, each of which describes a proper colouring of graph G, given the constraints in list M.

In lines 5 and 6 we define the polynomial ring R in as many variables as our graph has vertices, over the cyclotomic field of degree k; the number of colours. Since the the vertices of the graph match the cells of our Sudoku, solving equations in these variables will identify a colour for each cell. Building our ring over the cyclotomic field (the field consisting only of k roots of unity) is done because the k roots of unity can be used to describe our k colours, so the only valid root of an expression in our ring is one of the k colours.

Lines 7 to 23 create the ideals $I_{n,k}$ and $J = \langle x_i^{k-1} + x_i^{k-2}x_j + ... + x_i x_j^{k-2} + x_j^{k-1} : \{i,j\} \in E \rangle$, though we're not yet ready to add them to get $I_{G,k}$. First we must algebraically account for the user-inputted conditions in list M. This is done simply by adding equations to the ideal J of the form $x_i - k_i$, where x_i is the variable in ring R corresponding to the cell in question, and k_i is the element of the cyclotomic ring of order k representing the colour we want for this cell. This process is what occurs from line 24 onwards, the idea for which came from Gago-Vargas et. al. [6].

The first thing that happens is that we build the list *Condcell* (later cast as a set) from the triples in list M which holds information in the form *(cell,colour)* where the cell is numbered in the usual way (top to bottom, left to right). This formulation of the partial colouring (if it is supplied) is easier to use in certain parts of the code.

The set *Cells* gives us the identifying numbers (note: without duplicates) of the cells that have conditions on them. The set *Col* gives the used colours, and we use this to build the set *UnCol*, in lines 34 to 38, which holds the colours that are unused in the partial colouring.

In lines 39 to 41 we take the partial colouring, if it exists, and formulate it algebraically to be added to the ideal J. If, however, there are no conditions given, lines 42 to 45 see to it that the top-left sub-grid is coloured arbitrarily with the n^2 colours. Note that this is legitimate since we are, later, going to

whittle down the colourings to those that are non-isomorphic. Therefore, solving one sub-grid makes no difference. The reason for doing this is to reduce the computational cost; the effect of which will be discussed in section 5.4.

Lines 46 to 64 perform the following action; they take the conditions given in the partial colouring, check whether they all fall into one sub-grid and then fill in the rest of that sub-grid arbitrarily with the unused colours. This is legitimate as discussed above. In lines 47 to 51 we go through the triples describing each condition and determine which sub-grid the cell in question falls into. The sub-grids are, *in this case*, identified by the number of their top-left cell (in a 4×4 Sudoku they would be $0, 2, 8$ and 10, rather than $0, 1, 2$ and 3). These sub-grid numbers are then put into a set to eradicate duplicates; if the set has size 1 then clearly all our conditions fall into one sub-grid and we can fill in the gaps.

This process occurs in lines 53 to 64; we take the only element of the set *Sgrid*, which is the top-left cell of the sub-grid in question, and use it as a starting point to iterate through the rows and columns of the sub-grid we've identified as relevant, putting all its cells in a set. Once we difference this set with the set *Cells*, which holds the cell numbers of the cells in the partial colouring, we get the cells in our relevant sub-grid that don't yet have a colour. Finally, it's a simple matter of going through this new set (of uncoloured cells), giving each one a random colour from *UnCol*, the set of unused colours, and adding the algebraic formulation of this colouring to the ideal J.

We finish the process described in section 3.1 by building the ideal $I_{G,k}$ in line 66, and then finding the variety generated by it, which corresponds to the proper colourings of the graph G.

5.1.3 Isolating non-isomorphic colourings of the $n^2 \times n^2$ Sudoku

In lines 3 to 5 we define, once more, the field we've been working over; we'll need the generators later. Then we generate the automorphism group of the graph through the in-built SAGE function. Line 7 gives us a list of the generators of this permutation group, which we'll iterate later.

In lines 8 to 13 we map the colours, represented in each dictionary in list V, from the form c^i for $0 \leq i < k$ to the form $i + 1$; the colours are now integers in the range $1, ..., k$ which is the usual representation. In line 18 we take the first dictionary in V and create from it a list of integers representing the colours of the vertices in the form $[x_0, ..., x_{k-1}]$.

What we now do, in lines 21 to 42 is go through each dictionary (proper colouring) in V with each element of the permutation group acting on it in turn, represent it as a list and compare this list

(colouring) to the colourings already in list M (the list of colourings that are confirmed as mutually non-isomorphic). If this latest colouring is found to be non-isomorphic (as described in the previous section), it is added to list M.

Finally the complete list of non-isomorphic, proper colourings of the graph G is returned.

5.2 A brief inquiry into term orders

The default monomial order in SAGE varies across different types of polynomial ring, but is generally degree reversed lexicographic. The reason behind this is that it is generally accepted that *degrevlex* is most often the fastest of the term orders when computing Gröbner bases [17].

I experimented with my own SAGE code for solving a Shidoku with the top left sub-grid coloured in (as explained above, this can be done without loss of generality), and found the following results, averaged over eight timings, for three different monomial orderings.

Monomial order	CPU time (average) / s	Wall time (average) / s
lex	58.45	83.46
deglex	56.50	80.99
degrevlex	58.59	83.81

Note that *deglex* (or *grlex*) is a monomial order such that for $\alpha, \beta \in \mathbb{Z}^n_{\geq 0}$, $\alpha >_{grlex} \beta$ if $|\alpha| > |\beta|$ or if $|\alpha| = |\beta|$ and $\alpha >_{lex} \beta$.

Interestingly, my results fall into a minority, according to Stoutemyer. However, we do witness a strong performance by *deglex* which, according to Stoutemyer, is the next best at being most often fastest.

At this time length, and with a difference of less than 4%, it seems that changing monomial orders is a very minor concern when, as shall be seen in section 5.4, there are other, far more significant methods of speeding up the process of solving a Shidoku.

5.3 A brief inquiry into polynomial rings

There are a number of polynomial rings in which one can encode the information held in a Sudoku; the most popular choices being $\mathbb{Q}[x_1, ..., x_n]$ as in [6] and [2], where the colours are encoded as integers $1, ..., k$, or $\mathbb{C}[x_1, ..., x_n]$ which is central in [8] and which I based my code own code on, where the colours are encoded as the k, k^{th} roots of unity.

There is not such a significant difference between these two methods, merely that the some of the

polynomials in the ideals whose varieties we seek differ slightly. As will be seen in section 5.4, I had to use the polynomial ring over the cyclotomic field $(\mathbb{Q}[\zeta_k])[x_1, ..., x_n]$ because I had problems with $\mathbb{C}[x_1, ..., x_n]$. This is a minor difference from the method in Hillar and Windfeldt's paper as I was still using the k^{th} roots of unity to encode the colours.

An interesting alternative was seen in section 2.4.5, described in [9] and [2], where we use the field $Q = \mathbb{Z}_2[x_1, ..., x_n] \setminus \langle x_1^2 - x_1, ..., x_n^2 - x_n \rangle$ which results in more variables to represent all the information of the system (since our variables may now only take the values 0 and 1), but allows certain systems to be solved where this was not possible in the fields mentioned above.

5.4 Difficulties encountered during the computational implementation

I encountered many difficulties over the course of the project, in the computational implementation of the project, each one taught me something new and influenced the direction of my approach.

The very first problem I encountered was trying to calculate the varieties of the ideals in Hillar's paper [8] that describe proper and improper colourings of the graph. This method hinges on using a polynomial ring over the complex field, since the k^{th} roots of unity represent the k colours. I found, however, that the varieties calculated by SAGE over this complex polynomial ring were incomplete. This problem was very unusual and my supervisor, Dr. A. M. Kasprzyk, and other colleagues that use SAGE, suggested it may be a bug in SAGE. Fortunately I was able to circumvent this problem by using the cyclotomic field ($\mathbb{Q}$ with the k^{th} roots appended), since it was only the k^{th} roots I really needed. In this field the varieties were calculated correctly.

The code that is supplied in the appendices for generating the graph of a Sudoku and finding its non-isomorphic colourings I wrote for general n. Much to my dismay, I found that running it for $n = 3$ (the case of a 9×9 Sudoku) caused SAGE to run out of memory and terminate the process before it found the solutions. For this reason I had to focus my studies on Shidoku instead. This is not entirely a negative note, however, as it opened up lots of other interesting areas that I may not have considered otherwise, such as the nature of the automorphism group of the graph.

Even in the case of the Shidoku ($n = 2$), SAGE ran out of memory before it could find all colourings. At this point I was seriously concerned about whether I would be able to use any of the code I had written; fortunately my supervisor suggested a technique to help the code along, which I later found used in literature as well, which was to colour the top left sub-grid arbitrarily. Since only non-isomorphic

colourings interested me, this was not a problem, and it did mean that the Shidoku could be solved now, in about 60s CPU time.
Later I would find that this sort of problem was commonplace [9] and that it is symptomatic of using Gröbner bases which are notoriously computationally expensive.

Whilst I was still struggling to get my code to run on $n = 2, 3$ without the added help of colouring in the top left sub-grid, I tried using different monomial orders; this made no difference, and even after I coloured the sub-grid it made little difference, as can be seen in section 5.2. I also experimented with rearranging the order of the polynomials Hillar suggested in the ideal $I_{G,k}$, particularly after I had added additional conditions such as $x_0 - c^2$ (colour the 0^{th} cell with the 3^{rd} colour), but again the speed change was not noticeable.

Finally, I had intended to use the same methods as with Sudoku, to try and solve a Kakuro puzzle. despite choosing a very simple puzzle I encountered the same problem of running out of memory as with Sudoku initially. Since, however, Kakuro does not have a uniform shape, and the cross-sum clues are individual to each puzzle, it does not lend itself as nicely to computational automation as Sudoku does, so I stopped working in this area and refocused all my attention on Sudoku.

6 Conclusion and Suggestions for Further Work

Studying the applications of Grobner bases to graph colouring first hand through computation using SAGE, as well as through reading the literature, reveals the fact that Grobner bases are not the optimal tool for solving Sudoku. One must note, however, that this was merely one application of the theory described in this project. For a start, the link between Grobner bases and graph theory is, in itself, interesting and worthy of study. Furthermore, although the method may have failed, as I described in section 5.4, in certain cases, when it did work it yielded a concise way of identifying all proper colourings of the given graph. I, personally, may not have been successful in some cases I studied, but as others' papers show (c.f. section 2.4.5), there are methods that can be used to improve the application of Grobner basis methods and yield better results still.

Overall, studying the ways in which Grobner bases can be linked with graph theory and group theory has been very interesting for me, and given more time to explore this area I would expand my knowledge in both depth and breadth as follows:
First of all I would like to explore performing Grobner basis calculations in the Boolean polynomial ring, to see if I could have more success with the puzzles I wasn't able to solve using my current method (i.e. the regular Sudoku). Further to this I would like to explore where the tipping point of not being able to solve a given Sudoku is. For example, I found that a Shidoku could be solved in around 1 minute of CPU time with 4 clues in the top-left sub-grid, but could not be solved without any clues; how many clues would it take for my code to be able to solve a Sudoku (both in the cyclotomic and Boolean polynomial rings), how many clues would it take for larger Sudoku puzzles like 16×16?
Secondly, I would like to expand my application of these methods to other number puzzles. Gago-Vargas et. al. give a brief introduction to this [6], for instance kakuro (which I already attempted and was unsuccessful with) as well as killer Sudoku, among others. For these sorts of puzzles the basic techniques are very similar, but I would be interested to find what effect the small nuances between the puzzles have on the possibility to solve them using Grobner basis methods, as well as just how far one can go in applying Grobner basis methods to number puzzles; Gago-Vargas et. al. even mention applications to games such as minesweeper!

A SAGE Code

A.1 Code that generates the graph of an n^2 by n^2 Sudoku

```
1. def Sudoku(n):
2.         "Returns a graph on n**2 x n**2 vertices representing a Sudoku (n is
   the size of the side of the sub-grids)"
3.         G=Graph()
4.         for L1 in range(n**4):
5.                 G.add_vertices([L1])
6.         for L2 in range(n**2):
7.                 for L3 in range(n**2):
8.                         for L4 in range(n**2):
9.                                 u=((n**2)*L2)+L3
10.                                if (u+L4)<((L2+1)*(n**2)):
11.                                        v=u+L4
12.                                else:
13.                                        v=u+L4-(n**2)
14.                                G.add_edge(u,v)
15.        for M2 in range(n**2):
16.                for M3 in range(n**2):
17.                        for M4 in range(n**2):
18.                                x=(M2)+(n**2)*M3
19.                                if (x+(n**2)*M4)<(n**4)-(n**2)+M2+1:
```

```
20.                                     y=x+(n**2)*M4
21.                             else:
22.                                     y=(x+(n**2)*M4)-(n**4)
23.                             G.add_edge(x,y)
24.         for N2 in range(n**2):
25.                 S=list()
26.                 for N3 in range(n):
27.                         for N4 in range(n):
28.                                 elt=(((N2-(N2%n))/n)*(n**3)) + ((N2%n)*n) +
    (n**2)*N3 + N4
29.                                 S.append(elt)
30.                 for s in S:
31.                         for t in S:
32.                                 G.add_edge(s,t)
33.         return G
```

A.2 Code that generates all proper colourings of the graph of an n^2 by n^2 Sudoku

```
1. def proper_col(G,n,M):
2.         "Takes a Graph, G, integer, n, and list, M, and returns a list of
   dictionaries, each of which describes a proper (n**2)-colouring of G given
   conditions in list M"
3.         N=G.num_verts()
4.         k=n**2
5.         F.<c>=CyclotomicField(k)
6.         R=PolynomialRing(F,N,"x",order='degrevlex')
7.         I=list()
8.         for i in range(N):
9.                 f=((R.gen(i))^k)-1
10.                I.append(f)
11.        Ink=R.ideal(I)
12.        a=G.num_edges()
13.        E=G.edges()
14.        J=list()
15.        for i in range(a):
16.                F=E[i]
17.                F0=F[0]
18.                F1=F[1]
```

```
19.                 j=R.zero()
20.                 for g in range(k):
21.                         j1=(R.gen(F0)^(k-1-g))*(R.gen(F1)^g)
22.                         j=j+j1
23.                 J.append(j)
24.         Condcell=list()
25.         CellsL=list()
26.         ColL=list()
27.         for M1 in M:
28.                 cell=(M1[0]*(n**2))+M1[1]
29.                 Condcell.append((cell,M1[2]))
30.                 CellsL.append(cell)
31.                 ColL.append(M1[2])
32.         Cells=set(CellsL)
33.         Col=set(ColL)
34.         UnColL=list()
35.         for M0 in range(k):
36.                 UnColL.append(M0)
37.         UnCol=set(UnColL)
38.         UnCol.difference_update(Col)
39.         if len(M)!=0:
40.                 for i in range(len(M)):
```

```
41.                         J.append(R.gen((Condcell[i])[0])
42.                         -(c**((Condcell[i])[1])))
43.         else:
44.                 for i in range(n):
45.                         for j in range(n):
46.                                 J.append(R.gen(j+((n**2)*i))-(c**(j+(n*i))))
47.         SgridL=list()
48.         for M3 in M:
49.                 bandno=(M3[0]-(M3[0]%n))/n
50.                 stackno=(M3[1]-(M3[1]%n))/n
51.                 sgridTL=bandno*(n**3)+stackno*n
52.                 SgridL.append(sgridTL)
53.         Sgrid=set(SgridL)
54.         if len(Sgrid)==1:
55.                 sgridno=Sgrid.pop()
56.                 ScellsL=list()
57.                 for M4 in range(n):
58.                         for M5 in range(n):
59.                                 ScellsL.append(sgridno+(M4*(n**2))+M5)
60.                 Scells=set(ScellsL)
61.                 Scells2=Scells.difference(Cells)
62.                 for i in range(len(Scells2)):
```

```
63.                         colour=UnCol.pop()
64.                         cellTemp=Scells2.pop()
65.                         J.append(R.gen(cellTemp)-(c**(colour)))
66.         J1=R.ideal(J)
67.         IGk=Ink+J1
68.         V=IGk.variety()
69.         return V
```

A.3 Code that returns all non-isomorphic, proper colourings of graph G

```
1. def col_uptoiso(G,V,k):
2.         "Takes a Graph, G, the list of varieties that represents all proper
   colourings (taking into account given conditions) of G and the number of
   colours in the colouring, and outputs a list of proper colourings up to
   isomorphism"
3.         vert=G.num_verts()
4.         F.<c>=CyclotomicField(k)
5.         R=PolynomialRing(F,vert,"x",order='degrevlex')
6.         H=G.automorphism_group()
7.         elements=H.list()
8.         map=dict()
9.         for i in range(k):
10.                map[c**i]=i+1
11.        for v in V:
12.                for P in v.iterkeys():
13.                        v[P]=map[v[P]]
14.        M=list()
15.        N=list()
16.        K=list()
17.        for j in range(vert):
18.                K.append((V[0])[R.gen(j)])
19.        M.append(K)
```

```
20.         N.append(V[0])
21.         for v in V:
22.                 for h in elements:
23.                         l=list()
24.                         ints=[(x+1) for x in range(vert)]
25.                         authelp=h(ints)
26.                         M1=list()
27.                         N1=list()
28.                         l=[v[R.gen(i)] for i in range(vert)]
29.                         lh=h(l)
30.                         count=0
31.                         for m in M:
32.                                 L=list()
33.                                 for j in range(vert):
34.                                         P=(k+1)*(m[j])+(lh[j])
35.                                         if P not in L:
36.                                                 L.append(P)
37.                                 if len(L)==k:
38.                                         count=count+1
39.                         if count==0:
40.                                 M.append(l)
41.                                 N.append(v)
```

```
42.                                 break

43.         return M
```

References

[1] Elizabeth Arnold, Rebecca Field, Stephen Lucas, and Laura Taalman, *Minimal Complete Shidoku Symmetry Groups*, arXiv:1302.5949v1, October 2011.

[2] Elizabeth Arnold, Stephen Lucas, and Laura Taalman, *Grobner Basis Representations of Sudoku*, March 2009, http://educ.jmu.edu/ taalmala/arnold_lucas_taalman.

[3] Bela Bollobas, *Graph Theory An Introductory Course*, Graduate Texts in Mathematics, Springer-Verlag, 1979.

[4] Micheal Brickenstein and Alexander Dreyer, *Grobner-free normal forms for Boolean polynomials*, Journal of Symbolic Computation **48** (2013), 37–53.

[5] David Cox, John Little, and Donal O'Shea, *Ideals, Varieties, and Algorithms*, Undergraduate Texts in Mathematics, Springer-Verlag, 1992.

[6] Jesus Gago-Vargas, Isabel Hartillo-Hermoso, Jorge Martin-Morales, and Jose Maria Ucha-Enriquez, *Sudokus and grobner bases: not only a Divertimento*, http://www.ricam.oeaw.ac.at/Groebner-Bases-Bibliography/gbbibZ_files/publication_1180.pdf.

[7] Georges Gonthier, *Formal Proof - The Four-Color Theorem*, Notices of the AMS **55** (2008), no. 11, 1382–1393.

[8] Christopher J. Hillar and Troels Windfeldt, *Algebraic characterization of uniquely vertex colorable graphs*, Journal of Combinatorial Theory, Series B **98** (2008), 400–414.

[9] Franziska Hinkelmann and Elizabeth Arnold, *Fast Grobner Basis Computation for Boolean Polynomials*, arXiv:1010.2669v1, October 2010.

[10] Alexander M. Kasprzyk, *Computational Commutative Algebra Notes*, 2012, http://magma.maths.usyd.edu.au/users/kasprzyk/teaching/math3353/pdf/LectureNotes.pdf.

[11] Hans Kurzweil and Bernd Stellmacher, *The Theory of Finite Groups*, Universitext, Springer, 2004.

[12] Kenneth Lange and Eric C. Chi, *Techniques for Solving Sudoku Puzzles*, arXiv:1203.2295v3, May 2013.

[13] L. Lovasz, *Bounding the Independence Number of a Graph*, Annals of Discrete Mathematics **16** (1982), 213–223, https://www.math.ucdavis.edu/ delo-era/MISC/BIBLIOTECA/trunk/Lovasz/Lovaszindependencenumber.pdf.

[14] Gary McGuire, Bastian Tugemann, and Gilles Civario, *There is no 16-Clue Sudoku: Solving the Sudoku Minimum Number of Clues Problem*, arXiv:1201.0749v1, January 2012.

[15] Gordon Royle, *Minimum sudoku*, http://school.maths.uwa.edu.au/ gordon/sudokumin.php.

[16] William A. Stein, *Sage mathematics software (version 4.8.0)*, 2012, http://www.sagemath.org.

[17] David R. Stoutemyer, *Subtotal ordering - a pedagogically advantageous algorithm for computing total degree reverse lexicographic order*, arXiv:1203.1295v1, March 2012.

[18] National Chiao Tung University, *Topological Graph Theory*, http://www.math.nctu.edu.tw/hlfu/getCourseFile.php?CID=162&type=browser.