Sudoku is Hard

E. Routledge

01 Nov 2022

Abstract

sudoku overview

Contents

Chapter 1

Introduction

Sudoku is a simple logic game, in the standard 9×9 (or $3 \times 3 \times 3 \times 3$) one must complete the grid such that every row, column and box contains the numbers 1 to 9, that is all, yet it is filled with mathematics. Through sudoku we can explore the connections between various areas of maths: complexity theory, graph theory, group theory and information theory.

1.1 History

1.2 Defining Sudoku Notation

Def^n: A valid sudoku puzzle is a function $S : i, j \to x$ for values i,j $\in \{1, ..., D^2\}$ and $x \in \{0, ..., D^2\}$ satisfying the following:

- for all $a, b, c \in \{1, ..., D^2\}$ with $S(a, b) \neq 0$ and $S(a, c) \neq 0$, then $S(a, b) \neq$ S(a,c)
- for all $a, b, c \in \{1, ..., D^2\}$ with $S(a, b) \neq 0$ and$S(c, b) \neq 0$, then $S(a, b) \neq$ S(c,b)
- for all $a, b, c, d \in \{1, ..., D^2\}$ with $a \bmod D = c \bmod D$, $b \bmod D = d \bmod D$, $S(a, b) \neq 0$ and $S(c, d) \neq 0$, then $S(a, b) \neq S(a, c)$

Def^n: A completed sudoku puzzle is a function $S : i, j \to x$ as above but with the added condition that $x \neq 0$.

Chapter 2

Classic solving techniques

Def^n: A forced cell is a value pair (a, b) such that $S(a, b)$ can only be a single value call this x as $\{1, ..., D^2\}/\{x\}$ are already present in $S(a, j)$ for $j \in \{1, ..., D^2\}/\{b\}$ or $S(i, b)$ for $i \in \{1, ..., D^2\}/\{a\}$ or $S(i, j)$ where $a \bmod D = i \bmod D$ and $b \bmod D = j \bmod D$.

define x wing define y wing

Chapter 3

Sudoku is Hard

Let's imagine a sudoku of size $D^2 \times D^2$. How big does D have to be for you to need more than a day to solve it? Maybe 6 or 10 or even just 4. Don't worry if you said a smaller number than your friends, this has nothing to do with your problem solving skills, even a computer finds sudoku hard. In fact just incrementing D by 1 leads to an exponential increase in compute time and the most optimal algorithms for solving sudoku are infeasible for 100×100.

We prove sudoku's hardness by transforming it into a known 'difficult' problem; we will use SAT, a problem that has plagued computer scientists for decades.

3.1 Computational Complexity

For those with a mathematical mind, outraged by the lack of definitions for 'difficulty' and 'hardness', let's take a detour into complexity theory.

3.1.1 Turing Machines

We are working on the boundaries between computer science and mathematics, to stray into computer science we need a rigorous definition of a computer, that's where the Turing Machine comes in. In Turing's paper *Computing Machinery and Intelligence* **CITE** the Turing Machine is introduced as a mathematical model of what is now known as a CPU, the difference beign that the theoretical machine has finite but unbounded memory. While the full definition isn't completely necessary for our discussion we include it for completeness.

Def^{n}: A **Turing Machine** $M = Q, \Gamma, b, \Sigma, \delta, q_0, F$ such that:

- Q is a set of states, with $q_0 \in Q$ being the initial state and $F \subseteq Q$ is the set of final states
- Γ finite set of tape alphabet symbols, with $b \in \Gamma$ being the blank symbol
- $\Sigma \subseteq \Gamma \backslash \{b\}$ the set of input symbols
- δ the set of transition functions, given the current state and symbol the transition function determines which state to progress to and wether to change the symbol, if the transition is undefined the machine halts

Anything computable should therefore have an instance of a turing machine with defined alphabet, states and state transitions. The input is the original contense of the tape and the output is the contense of the tape once the machine halts. **EXAMPLE, comparing two numbers**

We now consider a less tangible version of teh Turing Machine one involving non determinism. When

given a single state multiple transitions our new machine does not necessarily have a single transition, tehre may exist multiple avenues to explore. We can therefore explore all possible tranisitons at once by changing the set of state transitions δ from a function to a relation with each state transition explored on a seperate tape.

Def^{n}: A **non-deterministic** Turing Machine is the mathematical model of a CPU that can undertake any possible action in a single time step.

See figure 3.1 and figure 3.2 for a comparison between these two versions of Turing machines with solving a 4 by 4 sudoku with a brute force method (this method is discussed further in Big O Examples and Solving Techniques - Backtracking), each row will theoretically execute in the same time step.

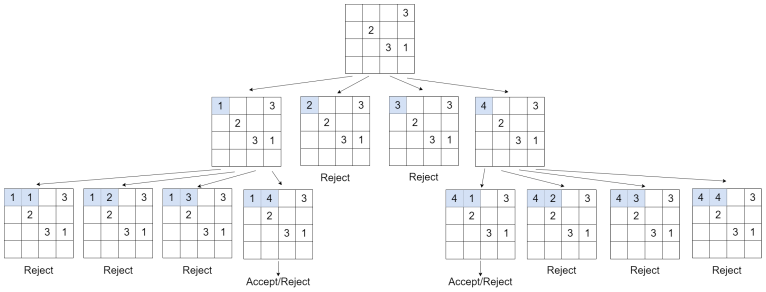

Figure 3.1: Non Deterministic Turing Machine with Brute Force Solving

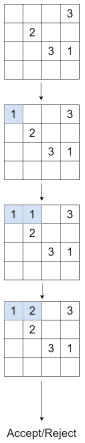

Figure 3.2: Deterministic Turing Machine with Brute Force Solving

3.1.2 Big O Notation

REDO SUBSECTION

Def^{n}: Let f be a function indicating the execution time for an algorithm and g a strictly positive function. $f(x) = O(g(x))$ if $\exists$ positive M and x_0 such that $|f(x)| \leq Mg(x)\ \forall\ x \geq x_0$. This is coined **Big O Notation**.

Def^{n}: A **Reduction**, $A \leq_p B$, is a transformation in polynomial time ($O(x^c)$) from problem A to B.

3.1.3 Sets of Difficulty

We care about decision problems, these are problems that given an input produce a 'yes' or 'no' answer. We will discuss three sets of these problems:

- P (Polynomial) is the the class of problems that can be solved in polynomial time by a deterministic Turing machine;

- NP (Non-deterministic Polynomial) is the class of problems that can be verified in polynomial time by a deterministic Turing machine or solved in polynomial time by a non-deterministic Turing machine;
- the NP-hard class are at least as hard as the hardest NP problem;
- the NP-complete set is the intersection of NP and NP-hard problems, these are the hardest problems in NP.

Problems in P are considered feasible and those in NP-complete are infeasible as their complexity scales exponentially with respect to the input size and as it is assumed they cannot be solved in polynomial time ($P \neq NP$) and are therefore infeesible for large inputs. [1]

So when we state sudoku is hard we are actually saying sudoku belongs to NP-complete. We cannot just prove sudoku belongs to NP as this also includes problems in P. [2]

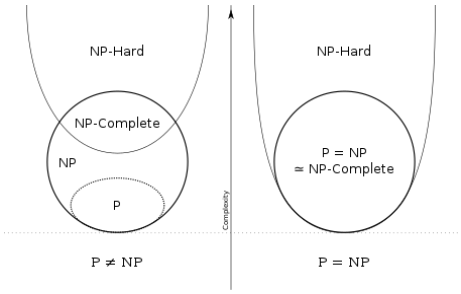

Figure 3.3: P, NP, NP-complete & NP-hard sets [1]**REDO IMAGE**

3.1.4 Proving NP completeness?

Call the problem we wish to prove is NP-complete x.

The set of NP-complete problems are defined as members of both the set of NP problems and the set of NP-hard problems.

First show there exists a verifier for x with a polynomial or less runtime, this is a algorithm that decides if a proposed solution to problem x is correct. This proves $x \in$ NP.

Then take a known NP-complete problem call this y, and reduce it to x, one does this by transforming the input of y to the input of x in polynomial time, we call this function $g : y \to x$. Assume there exists a polynomial time algorithm to solve x, f, then we could solve y in polynomial time too, $f(g(y))$. Therefore if the reduction exists then x is at least as hard as y and as y is NP-complete then x is at least as hard as the hardest question in NP meaning $x \in$ NP-hard.

Therefore $x \in$ NP-complete.

3.1.5 The First NP-complete Problem

If, as the above suggests, we require a NP-complete problem to prove a problem is NP-complete then we seem to have reached a paradox. Luckily we have the Cook-Levin Theorem.

Cook-Levin Theorem: SAT is NP-Complete. **CITE**

Terminology:

[1] We can only assume that $P \neq NP$ as this problem is yet to be proven, it is in fact one of the Millennium Prize problems.

[2] Due to Ladner's Theorem there exists problem $\in$ NP but $\notin$ NP-complete and $\notin$ P iff $P \neq NP$, these problems are called NP-intermediate.

$(a \vee b \vee c \vee \neg c) \wedge (a \vee c) \wedge (\neg a \vee b)$						
a	b	c	$(a \vee b \vee c \vee \neg c)$	$(a \vee c)$	$(\neg a \vee b)$	Full Clause
F	F	F	T	F	T	F
F	F	T	T	T	T	T
F	T	F	T	F	T	F
F	T	T	T	T	T	T
T	F	F	T	T	F	F
T	F	T	T	T	F	F
T	T	F	T	T	T	T
T	T	T	T	T	T	T

Table 3.1: An example of a SAT clause set with all possible truth assignments explored and valid assignments highlighted.

- Boolean Variable: a variable that can can be true or false ($a = T$ or $a = F$).
- Literal: a boolean variable or it's negation (if $a = T$ then its negation $\neg a = F$).
- $\wedge$: an operation that outputs true when all operands are true, false otherwise ($a \wedge b = T$ iff $a = T$ and $b = T$).
- $\vee$: an operation that outputs true when at least one operand is true, false otherwise ($a \vee b = T$ iff $a = T$ or $b = T$).
- Clause: multiple literals operated on by $\vee$s, a set of clauses are joined by $\wedge$s.
- Truth Assignment : assignment of true and false values to each boolean variable.

Def^n: SAT is the following decision problem. Given a set of boolean variables B and a collection of clauses C does a valid truth assignment exist that satisfies all clauses in C?

Given $B = \{a, b, c\}$ and $C = \{(a \vee b \vee c \vee \neg c), (a \vee c), (\neg a \vee b)\}$ a valid truth assignments exists. See the table 3.1 for all valid assignments.

We now have a NP-complete problem to reduce other problems to.

3.1.6 Examples

Let's start putting complexity into the context of sudoku. Our input is a grid of size $n \times n$ ($n = D^2$), we will investigate an algorithms run time in comparison to this variable.

Constant time, $O(1)$: This includes functions that take the same time no matter the input size, for example accessing the value of a cell in the sudoku grid, even as the grid increases in size, grid(x,y) takes constant time.

Linear time, $O(n)$: Consider a function that when given a grid and a grid coordinate (x, y) outputs a boolean value; true if the value of the grid at this coordinate is valid (does not repeat in the row, column or box) and false otherwise. Let's consider the inner workings of the function:

- First we assume the comparison between two values takes a single unit of time $O(1)$
- We must compare the value at the given coordinate with each value on the row, observe this is performed $n - 1$ times and therefore this operation takes $O(n)$ time.
- We then compare the value at the given coordinate with each value on the column, as per the previous argument this has complexity $O(n)$.

- Finally we compare the value with the remainder of the box value that have not been compared previously, this takes $n - (\sqrt{n} - 1) - (\sqrt{n} - 1) - 1 = n - 2\sqrt{n} + 1$ comparison operations which has a complexity of $O(n)$.

Overall this function takes $(n-1)+(n-1)+(n-2\sqrt{n}+1) = 3n-2\sqrt{n}-1$ comparisons and therefore calculates the boolean variable in linear time ($O(n)$). See algorithm 1.

Algorithm 1 Validate an Entry

```
procedure ValidateEntry(grid, (x,y), n)
    for i = 1 to n do                                          ▷ Check column
        if i ≠ x then
            if grid(i,y) = grid(x,y) then
                return False
            end if
        end if
    end for
    for i = 1 to n do                                          ▷ Check row
        if i ≠ y then
            if grid(x,i) = grid(x,y) then
                return False
            end if
        end if
    end for
    for i = 1 to √n do                                         ▷ Check box
        for j = 1 to √n do
            if x DIV √n + i = x and y DIV √n + j = y then
                if grid(x DIV √n + i, y DIV √n + j) = grid(x,y) then
                    return False
                end if
            end if
        end for                          ▷ DIV refers to integer division e.g. 7 DIV 3 = 2
    end for
    return True
end procedure
```

Polynomial time, $O(n^t)$: Consider a function that when given a partially complete sudoku grid returns true if the grid is valid, otherwise it returns false. Using the linear time algorithm we just described we can repeat this for every value within the grid. See algorithm 2.

Algorithm 2 Validate a Grid

```
procedure Validate(grid, n)
    for i = 1 to n do          ▷ Loop through all (i,j) pairs to validate all squares of the grid
        for j = 1 to n do
            if ValidateEntry(grid, (i,j), n) = False then
                return False
            end if
        end for
    end for
    return True
end procedure
```

We call ValidateEntry n^2 times so our complexity is $O(n \times n^2) = O(n^3)$ this is polynomial and therefore still considered feesible as n increases.

Exponential time, $O(a^n)$: Consider a brute force algorithm to solve sudoku, all we need to do is cycle through values 1 to n for all squares rejecting those that create an invalid sudoku grid and outputting a valid grid if one is found, otherwise false if the sudoku cannot be solved. See algorithm 3.

Algorithm 3 Brute Force Sudoku Solver

```
procedure BruteForceSolve(grid, n)
    if grid is complete then
        if Validate(grid) = True then
            return grid
        else
            return False
        end if
    end if
    (x,y) = location of first empty cell in grid
    for i = 1 to n do
        grid(x,y) = i
        if BruteForceSolve(grid, n) ≠ False then
            return grid
        end if
    end for
    return False
end procedure
```

This algorithm refers to itself, this is called recursion and shall be explored further in chapter 4 when we discussion solving techiques. For now let us see explore this specific algorithm;

- Assume we only have 1 empty square then we try the values 1 to n and for each we check if the grid is valid, this takes $O(n \times n^3)$.
- Now assume we have 2 empty squares we try 1 to n and for every option we have to do the same as the first bullet point which takes $O(n \times n \times n^3)$.
- A pattern forms, for every empty square we must times the complexity by n, we have at most n^2 empty squares so the upper bound is $O(n^{n^2+3})$.

This algorithm has a complexity that is a bit above exponential as the base is dependent on n too however it is not quite factorial complexity so we will call it exponential. This is infeasible for large values of n and does not belong to P. So solving sudoku is hard, case closed - not quit, this is only one example of a sudoku solver there may exist more efficient algorithms so we need to disprove this. [3]

3.2 Existance is Hard

Checking if a solution to sudoku exists is NP-complete, let us define the decision problem:

$$\Phi(S) = \begin{cases} \text{True if a completion exists} \\ \text{False if a completion does not exist.} \end{cases} \tag{3.1}$$

Our question is does there exist a function Φ that when given an instance of the problem will, in polynomial time or less, return True if it can be solved and False otherwise.

[3] BogoSort is a sorting algorithm that randomises a list until it is in the correct order, this has an unbounded run time, but there exists sorting algorithms with complexity $O(n\log n)$. **CITE**

3.2.1 Proof Outline

The verifier must be shown to be $\in$ P, this means the Sudoku decision problem belongs to the set NP.

Then we need a reduction from sudoku to a known NP-complete problem to prove sudoku is also NP-hard. We will be creating a chain of reductions: **Sudoku** $\geq_p$ **Latin Square** $\geq_p$ **Triangulated Tripartite** $\geq_p$ **3SAT** $\geq_p$ **SAT**.

As the Sudoku decision problem is a member of NP and NP-hard it is NP-complete by definition.

Note: Theoretically any problem in the set NP-complete can be reduced to Sudoku and therefore this reduction is not unique, however, it is the most intuitive way. Some readers may question why we are not looking at a reduction to a Graph n^2-Colouring problem but in section **cite** we explore this is the wrong direction of reduction.

3.2.2 Verification is Easy

Given a sudoku grid S, we have an algorithm to determine:

$$\Psi(S) = \begin{cases} \text{True if the puzzle is complete} \\ \text{False if the puzzle is not complete.} \end{cases} \tag{3.2}$$

This algorithm is an extension of algorithm 2 from the complexity examples, we simply add a for loop to the end to check that $\forall S(i,j) \neq 0$, in other words there are no empty cells. To find the complexity algorithm we add n^2 to the complexity of the original validation algorithm, giving $O(n^2+n^3) = O(n^3)$. This is polynomial time, therefore $\Psi \in$P.

3.2.3 Sudoku $\geq_p$ Latin Square

Def^n: A valid Latin Square puzzle is a function $L : i, j \to x$ for values $i, j \in \{1, .., D\}$ and $x \in \{0, ..., D\}$ satisfying the following:

- for all $a, b, c \in \{1, ..., D\}$ with $L(a, b) \neq 0$ and $L(a, c) \neq 0$ then $L(a, b) \neq L(a, c)$
- for all $a, b, c \in \{1, ..., D\}$ with $L(a, b) \neq 0$ and $L(c, b) \neq 0$ then $L(a, b) \neq L(c, b)$

It is complete or solved if for all $i, j \in \{1, ..., D\}$, $L(i, j) \neq 0$.

By observation we see this is a superset of the sudoku puzzle, we just add the restrictions that the dimension must be a square number and also add the third property of the sudoku puzzle defintion.

What is the Latin Square decision problem? Given a latin square puzzle $L(,)$, can the function be augmented, by changing only the value of the function for value pairs i, j that previously gave $L(i, j) = 0$, to get a complete latin square puzzle?

Proof idea: We must reduce a given latin square grid of size $D \times D$ to a sudoku grid size $D^2 \times D^2$ that is solvable iff the Latin square is.

Lemma: Let S_l be a Sudoku problem with the following construction

$$S_l(i,j) = \begin{cases} 0 & \text{when } (i,j) \in L_s \\ ((i-1 \bmod n)n + \lfloor i-1/n \rfloor + j - 1) \bmod n^2 + 1 & \text{otherwise} \end{cases} \tag{3.3}$$

where $L_s = \{(i,j) | \lfloor i-1/n \rfloor = 0 \text{ and } (j \bmod n) = 1\}$. Then there exists an augmentation S'_l to complete the sudoku puzzle if and only if the square L such that $L(i, j/n) = S'_l(i, j) - 1/n + 1$ for all $(i, j) \in L_s$ is a Latin square.

Note: The fact we have a formula to generate a valid sudoku for any size D^2 is interesting and we should explore if this can be done for a $M \times N$ sudoku too. (explored in section 3). Figure 3.4 gives examples of generated sudokus from this formula.

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

$n = 2$

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

$n = 4$

Figure 3.4: Formula Generation of Valid Sudoku

Proof:

First we must show $S_l(i,j) = ((i-1 \bmod n)n + \lfloor i-1/n \rfloor + j - 1) \bmod n^2 + 1$ forms a complete and valid sudoku puzzle.

When $i = [1, ..., n^2]$ then:

$$0 < \lfloor i-1/n \rfloor < n-1 \tag{3.4}$$
$$0 < i-1 \bmod n < n-1 \tag{3.5}$$
$$0 < (i-1 \bmod n)n + \lfloor i-1/n \rfloor < n^2 - n \tag{3.6}$$
$$0 < (i-1 \bmod n)n + \lfloor i-1/n \rfloor + j - 1 < n^2 - 1 \tag{3.7}$$
$$1 < ((i-1 \bmod n)n + \lfloor i-1/n \rfloor + j - 1) \bmod n^2 + 1 < n^2 \tag{3.8}$$
$$1 < S_l(i,j) < n^2 \tag{3.9}$$

Note $\lfloor i-1/n \rfloor$ gives the row coordinate when indexed at 0 in which the larger box that (i,j) belongs to starts and $i-1 \bmod n$ gives the row within that box when indexed at 0. Therefore $(\lfloor i-1/n \rfloor, i-1 \bmod n)$ will take all value pairs of integers between 0 and $n-1$.

When j is fixed (particular column), assume two cells have the same value, that is $S_l(i,j) = S_l(i',j)$ then

$$(i-1 \bmod n)n + \lfloor i-1/n \rfloor + j - 1 = (i'-1 \bmod n)n + \left\lfloor i'-1/n \right\rfloor + j - 1 \tag{3.10}$$
$$(i-1 \bmod n)n + \lfloor i-1/n \rfloor = (i'-1 \bmod n)n + \left\lfloor i'-1/n \right\rfloor \tag{3.11}$$

from the above $i = i'$. No cell on a column has the same value.

When i is fixed (particular column) assume two cells have the same value, that is $S_l(i,j) = S_l(i,j')$ implies $j - 1 = j' - 1 (\text{mod } n)$ therefore $j = j'$.

For the third condition fix $\lfloor i-1/n \rfloor$. (i-1 mod n,j) takes all value pairs of integers 0 to n-1 so if a cell has the same value as another within the n by n square $S_l(i,j) = S_l(i',j')$ implying $(i-1 \bmod n, j) = (i'-1 \bmod n, j')$ which means $i = i'$ and $j = j'$. Therefore S_l is a valid and complete sudoku puzzle.

Now consider which integers fill the blanks in L_s. For $(i,j) \in L_s$, $S_l(i,j) - 1 = ((i-1 \bmod n)n + j - 1) \bmod n^2$ as $jmodn = 1$, $j - 1modn = 0$ therfore $S_l(i,j) - 1$ is divisible by n so $S_l - 1/n + 1$ gives integers between $[1, ..., n]$. Therefore $L(i,j) \in [0, ..., n]$.

We must validate the Latin square conditions. The row constraint in S_l ensures $S'(i,j) = S'(i,j') \implies j = j'$, $S'(i,j) - 1/n + 1 = S'(i,j') - 1/n + 1 \implies j = j'$, $L(i,j/n) = S'(i,j'/n) \implies j = j'$ is equivalent

to the row constraint of L. The column constraint of S_l is equivalent to the column constraint of L. The small square constraint of S_l is equivalent to the column constraint of L. □

3.2.4 Latin Square $\geq_p$ Triangulate A Tripartite Graph

Def^n: A graph $G = (V, E)$ is tripartite if a partition V_1, V_2, V_3 exists such that the vertices are split into three sets with no edges between vertices that belong to the same set, i.e for all $(v_i, v_j) \in E$ if $v_i \in V_i$ then $v_j \notin V_i$.

Def^n: A triangulation T of a graph is a way to divide edges into disjoint subsets T_i, each forming a triangle ($T_i = \{(v_1, v_2), (v_2, v_3), (v_3, v_1)\}$).

If a tripartite graph can be triangulated it must be uniform, that is: every vertex in V_1 (or V_2 or V_3) has the same number of neighbour in V_2 and V_3 (or the respective sets).

What is the Triangulated Tripartite decision problem? Given a graph G that is tripartite (can be split into 3 subgroup, within these subgroups vertices should not share edges) can it be triangulated?

Theorem: Completing a Latin square with dimensions n by n is equivalent to triangulating a tripartite graph $G = V_1, V_2, V_3$.

Proof:

Intuitively, we map a graph to a Latin square L through the following: given tripartite graph G=(V,E) label vertices in V_1 with distinct lables $\{r_1, ...r_n\}$, label vertices in V_2 with distinct lables $\{c_1, ...c_n\}$ and label vertices in V_3 with distinct lables $\{e_1, ...e_n\}$. Add edges such that:

- If $L(i, j) = 0$ then add the edge (r_i, c_j)
- If for all $i \in [0, ..., n]$ and constant j, $L(i, j) \neq k$ then add the edge (r_i, e_k)
- If for all $j \in [0, ..., n]$ and constant i, $L(i, j) \neq k$ then add the edge (c_j, e_k)

This graph has a triangulation iff $L(i, j)$ can be solved.

EXAMPLE

Let us show every uniform tripartite graph can be transformed to the above formulation of a Latin square.

First we need an intermediate that is a generalisation of a latin square

Def^n: A Latin framework LF for tripartite graph G, size (r,s,t) is a r by s array with values [1,...,t]. With constraints:

- Each row/column contain each element only once.
- If $(r_i, c_j) \in E$ then LF(i,j)=0 else LF(i,j)= k, $k \in [1, ..., t]$
- If $(r_i, e_k) \in E$ then $\forall j \ LF(i, j) \neq k$
- If $(c_j, e_k) \in E$ then $\forall i \ LF(i, j) \neq k$

If r=s=t then LF is a latin square (formulation above) which can be completed iff G has a triangle partition.

Lemma: For tripartite graph G=(V,E) with $|V_1| = |V_2| = |V_3| = n$ (uniform), there's a Latin

framework of (n,n,2n).

Define LF an n by n array. For $(r_i, c_j) \in E$ $LF(i,j) = 0$ *else* $LF(i,j) = 1 + n + ((i+j) \bmod n)$. *LF is a latin framework as the first two bullet points of the definition hold by construction and as* $1 + n \leq LF(i,j) \leq 2n$ *LF will never equal a value in* $1, ...n$ *and therefore the last two bullet points hold. The size is trivial.* □

Lemma: Given latin framework LF(n,n,2n) for uniform tripartite graph G, we can extend the latin framework to have size (n,2n,2n).

First we have a few denotions: $R(k)$ *= the number of times* k *appears in* L *plus half* $|e_k|$; $S_i = \{k | k \notin LF(i,j) \forall j \cap (r_i, e_k) \notin E\}$; $M = \{k | R(k) = r + s - t\}$. *We show sets* $S_1, ...S_r$ *have a system distinct representative (**DEFINE**) containing all elements of* M, *we then add this system as the* $(s+1)st$ *column and repeat until we have* $2n$ *columns.*

*Using Hoffman and Kuhn's theorem **CITE** we need only show that* $S_1, ...S_r$ *have a system distinct representative and that for every* $M' \subseteq M$ *at least* $|M'|$ *of sets* $S_1, ...S_r$ *have non empty subsections with* M'.

First choose any m *sets such that* $1 \leq m \leq r$. *As* G *is uniform each set has t-s elements, so* m *sets together have* $m(t-s)$ *cardinality. Each value* $1, ..., t$ *appears at least r+s-t times in* LF, *so note each value appears in at most* $t - s$ *of the sets* S_i. *Consider the union of the* m *sets, this contains some* p *elements so we have* $p(t-s) \geq m(t-s)$ *therfore* $p \geq m$. *So any* m *sets have at least* m *elements in their union and by the P Hall theorem **CITE** a system distinct representative exists.*

Next take $M' \subseteq M$ *and assume there are* p *sets in* $S_1, ...S_r$ *that have a nonempty intersection with* M'. *Each set has* $t - s$ *elements and together have caridinality* $p(t-s)$, *each element of* M *appears in exactly* $r - (r + s - t) = t - s$ *of the* $s_i s$, *therefore* $|M'|(t-s) \leq p(t-s)$ *so* $|M'| \leq p$. *At least* $|M'|$ *sets have nonempty intersections with* M'.

The Hoffman and Kuhn theorem holds and therefore a system distinct representative exists and can be added to the end. We repeat this n times. □

Lemma: Latin framework (n,2n,2n) for grpah G, can be extended to (2n,2n,2n).

We can transpose the array and do the same as the previous lemma. □

Note: we can find a system distinct representative using the Hopcroft-Karp **CITE** algorithm which solves bipartite matching in polynomial time.

Given a tripartite graph G, if it is not uniform then no triangulation exists, else we apply above to produce a latin framework of size (2n,2n,2n) in polynomial time. This is a Latin square which can be completed iff G has a triangulation. The latin square problem has been reduced to the triangulating a tripartite graph problem. □

3.2.5 Triangulated Tripartite $\geq_p$ 3SAT

What is 3SAT? With a set of boolean variables B and a collection of clauses C, with at most 3 literals (a literal is any $b \in B$ or its negation $\bar{b}$) in each, does a valid truth assignment exist that satisfies C?

$$\phi(C, B) = \begin{cases} \text{True if a truth assignment exists} \\ \text{False if a truth assignment does not exist.} \end{cases} \tag{3.12}$$

This decision problem is therefore an enforced limitation of SAT as defined in the section Computational Complexity.

Proof:

This reduction is a little trickier as we need to introduce the Holyer graph H, this graph has the topology of torus.

Def^n: The Holyer graph $H_{3,p}$ is the set of vertices $V = \{(x_1, x_2, x_3) \in \mathbb{Z}_p^3 \| x_1 + x_2 + x_3 \equiv 0(modp)\}$ and an edge exists between vertices (x_1, x_2, x_3) and (y_1, y_2, y_3) if distinct i,j and k exist such that:

- $x_i \equiv y_i (\text{mod } p)$
- $x_j \equiv y_j + 1 (\text{mod } p)$
- $x_k \equiv y_k - 1 (\text{mod } p)$

See figure 3.5 for an example.

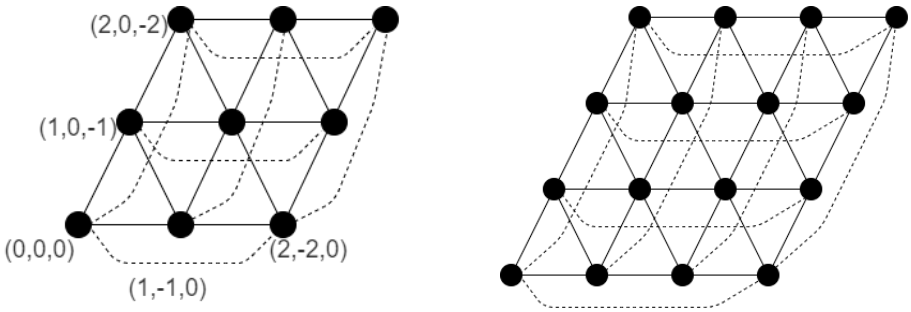

Figure 3.5: $H_{3,2}$ and $H_{3,3}$, the torus is embedded in the 2 dimensional plane, dotted lines link vertices that are the "same".

This graph is tripartite if and only if $p \equiv 0 \pmod 3$, this is demonstrated by a 3-colouring (a graph is tripartite if and only if it is 3-colourable) in figure 3.6.

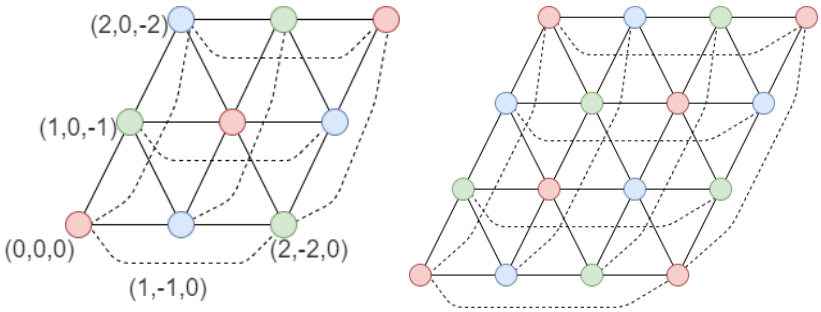

Figure 3.6: 3-colouring of $H_{3,2}$ and $H_{3,3}$

Def^n: $H_{3,p}$ has only two triangulations, termed a true and a false triangulation, see figure 3.7.

Note: We connect graphs together by taking a set of vertices in G_1 and making them the 'same' as a set of vertices in G_2, sets are the same size.

Def^n: We will connect out graph with F-patches and T-patches, see figure 3.8.

Let's turn an instance of 3SAT into an instance of triangulating a tripartite graph through the following transformation process: (select p large enough to prevent patch overlap and $p \equiv 0 \pmod 3$)

- For $b_i \in B$ create $H_{3,p}$ called G_{b_i}.
- For all $c_j \in C$, for each literal $l_{i,j}$ $j \in [1, 2, 3]$ create $H_{3,p}$ called $G_{i,j}$.
- If $l_{i,j} = b_k$ connect an F-patch in G_{b_k} to an F-patch in $G_{i,j}$, else if $l_{i,j} = \neg b_i$ connect a F-patch

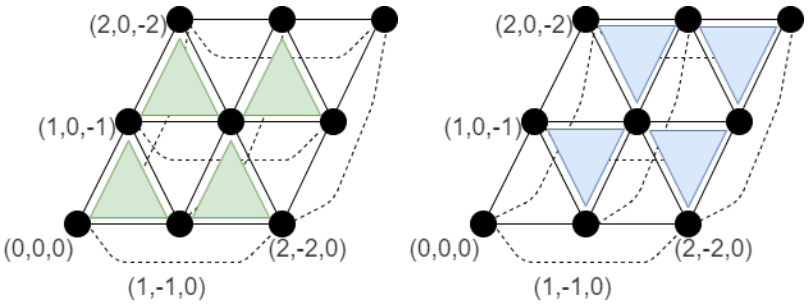

Figure 3.7: True triangulation and a False triangulation on a $H_{3,2}$ graph. Notice the edges between (0,0,0), (1,0,-1) and (1,-1,0) uniquely determine the triangulation; if they belong to the same triangle then it is a true triangulation and false otherwise.

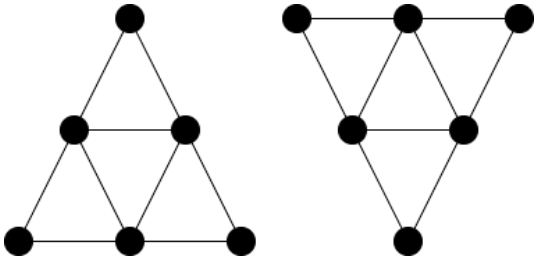

Figure 3.8: F-patch and a T-patch

in $G_{i,j}$ to a T-patch in G_{b_k}.

- For each i connect one F-patch from each $G_{i,1}$, $G_{i,2}$ and $G_{i,3}$ then delete the centre triangle.
- $G = \{G_{b_i} \mid b_i \in B\} \cup \{G_{i,j} \mid c_j \in C \text{ and } i \in [1,..,3]\}$

We now need to prove the graph produced by this transformation can be triangulated if and only if there is a truth assignment satisfying the 3SAT formula.

Assume a triangulation of G exists, consider a H within the construction of G. H is eitehr a true triangulation or a false triangulation. Now assume $l_{i,j}$ is b_k and consider the join between $G_{i,j}$ and G_{b_k} as this joins two F-patches we get at least one true triangulation: if $G_{i,j}$ is a true triangulation this acounts for all edges near the joining patch but the actual patch can be attributed to G_{b_k} which can be triangulated either way; if both are false triangulations the connecting patch is forced to belong to both $G_{i,j}$ and G_{b_k} which is a contradiction. (Figure 3.9)

Similarly if $l_{i,j} = \neg b_i$ then $G_{i,j}$ is a false triangulation or G_{b_k} is a true triangulation. (Figure 3.10)

Next the join between clause graphs allow for one false triangulation and the rest are true triangulations. As the centre of the patch is missing a single $G_{i,j}$ must take the outer edges of the patch by being a false triangulation. (Figure 3.11)

If G can be triangulated a truth assignment exists such that variable b_k is true if G_{b_k} has a true partition otherwise it is false.

If there exists a truth assignment we can triangulate G_{b_k} according to this truth assignment and this will allow for the whole graph to be triangulated.

This transformation takes place in polynomial time and therefore Triangulated Tripartite $\geq_p$ 3SAT. □

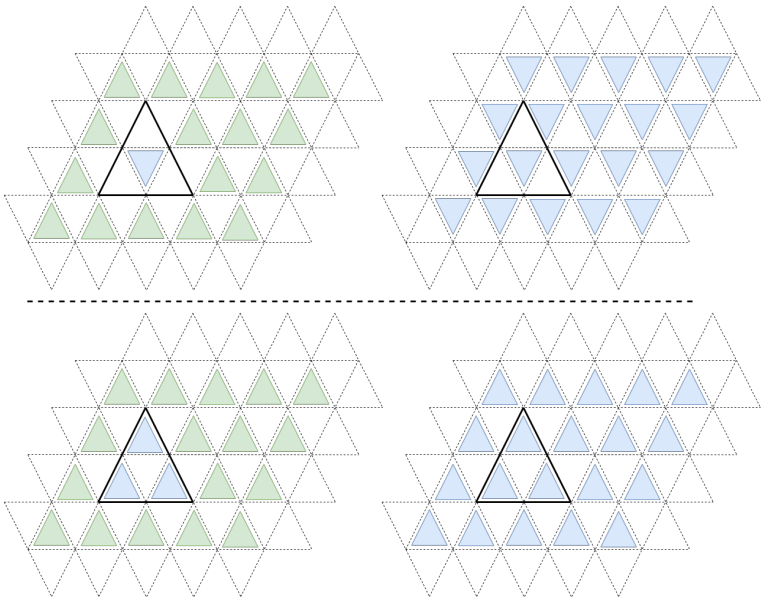

Figure 3.9: Graphs connected by two F-patches, the only complete triangulations are shown, one of each or two true triangulations.

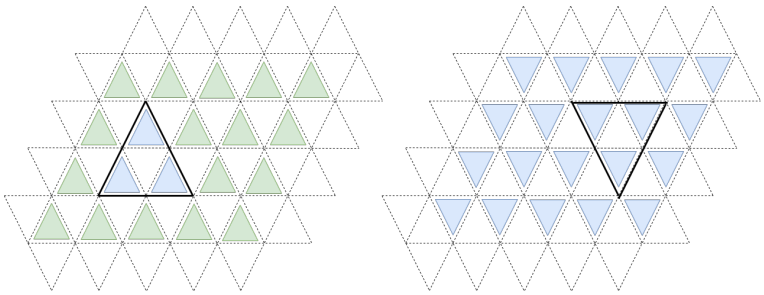

Figure 3.10: Graphs connected by a T-patch and a F-patch, the only complete triangulation is show.

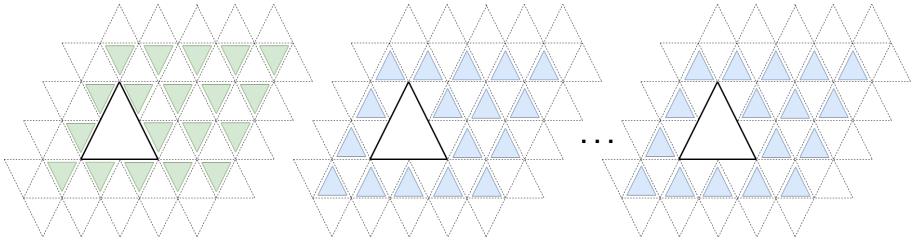

Figure 3.11: Graphs connected by F-patches with the centre removed, the only complete triangulation is show, exactly one must be a false triangualtion.

3.2.6 3SAT is NP-Complete

Proof:

Given a truth assignment t check each clause is satisfied, if all are satisfied return True else False, this algorithm is at most the length of C multiplied by the length of B. $O(BC)$ is polynomial, a polynomial verifier exists.

Given a SAT instance with the input sets of B and C. C is in conjunctive normal form (every clause set can be converted to an equivalent set in CNF form [2]) such that $\forall c \in C$ and for some $b_1, ..., b_n \in B$, $c = b_1 \vee b_2 \vee ... \vee b_n$. For each $c \in C$ with more than 3 literals we can transform these to a new set of clauses of length 3.

$(a \vee b) \wedge (\neg a \vee \neg b)$

a	b		$(a \vee b)$	$(\neg a \vee \neg b)$	$(a \vee b) \wedge (\neg a \vee \neg b)$
F	F		F	T	F
F	T		T	T	T
T	F		T	T	T
T	T		T	F	F

Figure 3.12: Truth Assignment Example with Highlighted Valid Assignment

For $c = b_1 \vee b_2 \vee ... \vee b_n$ we introduce a new literal: a_1 to give $b_1 \vee b_2 \vee a_1$, $\bar{b_1} \vee a_1$, $\bar{b_2} \vee a_1$ and $a_1 \vee b_3 \vee ... \vee b_n$. Then $a_1 \vee b_3 \vee ... \vee b_n$ becomes $b_3 \vee b_4 \vee a_2$, $\bar{b_3} \vee a_2$, $\bar{b_4} \vee a_2$ and $a_1 \vee a_2 \vee b_5 \vee ... \vee b_n$. This continues at most $n/2$ times to give $a_1 \vee ... \vee a_{n/2}$ or $a_1 \vee ... \vee a_{n/2} \vee b_n$ if n is odd.

Because we can convert a clause larger than 3 into multiple clauses of at most 3 literals in linear time $(O(n/2 + n/4 + ...) = O(n))$ this means we can reduce SAT to 3SAT in polynomial time.

As SAT is NP-complete by the Cook-Levin Theorem, this proves 3SAT is NP-Complete. □

3.2.7 Wrap Up

$\Phi \in$ NP as $\exists$ a verifier Ψ when given an instance S can determine if it is complete/solved in polynomial time, $\Psi \in$ P.

$\Phi \in$ NP-hard as an instance of SAT can be reduced to an instance of 3SAT in polynomial time which can be reduced to an instance of Triangulating a Tripartite Graph in polynomial time which can be reduced to an instance of Latin Square in polynomial time which can be reduced to an instance of Sudoku (Φ) in polynomial time.

EXAMPLE

As $\Phi \in$ NP and $\Phi \in$ NP-hard, $\Phi \in$ NP-complete. □

Determining if a sudoku grid S has a completion is hard and infeasible for large D.

3.2.8 Sudoku $\geq_p$ Graph Colouring

Intuitive but wrong.

3.2.9 Dimension Analysis

Motivate - a natural question, not conclusive as from definition of NP-complete class we know there is multiple ways to create a reduction.

A SAT instance with A clauses and B variables is transformed into a 3SAT instance with clauses and $B + b$ variables where $b = \Sigma_{c \in C'} |c| - 3$, $C' = \{c \in C || c| > 3\}$.

A 3SAT instance with C clauses and D variables is transformed into a Triangulating a Tripartite Graph instance with $p(D + 3C)$ nodes where $p \equiv 0 (\text{mod} 3)$.

A Triangulating a Tripartite Graph instance with E nodes is transformed into a Latin Square instance with dimension $\frac{2/3}{E}$.

A Latin Square instance with dimension F is transformed into a Sudoku with dimension F^2.

3.3 Solving Sudoku is Hard

We have only focused on decision problems with a boolean yes or no answer so far, but, when it comes to being given a sudoku most people assume it is solvable and then search for a solution so let us change our perspective to search problems; is it hard to solve sudoku?

$$\Gamma(S) = S' \text{ if } \exists \text{ a completed } S \text{ else } 0 \tag{3.13}$$

Where S' is a completed version of S.

It is intuitive that the search problem is harder than the decision problem but let's quantify this into our concepts of P and NP. If $P \neq NP$ then both are infeesible to solve as the search problem is harder than an NP-complete decision problem. However, we have hope, if $P = NP$ the corresponding search problem to an NP decision problem Γ can be solved in polynomial time.

Theorem:

ADD THEOREM

3.4 Determining Uniqueness is Hard

Def^n: The Sudoku Uniqueness problem is: Given a partially completed sudoku grid S does only a single completion exist?

$$\Gamma(S) = \begin{cases} \text{True if only a single completion exists} \\ \text{False if multiple completions or none exist.} \end{cases} \tag{3.14}$$

This is NP-hard (no polynomial verifier exists), NP-complete reduction exists.

It is hard to determine if a puzzle has a unique solution. *TO COMPLETE: FIND PAPER WITH PROOF*

Chapter 4

Solving Techniques

4.1 Backtracking

The standard way to solve a 9×9 sudoku puzzle is by the backtracking algorithm. This is a brute force method with a few optimisations. One can expect to find this algorithm in a computer science course introduction to recursion, that is to say it is not a complex concept and while useful for the usual sizes, as soon as we increase to 16×16 this becomes infeesible.

Algorithm 4 Backtracking

```
procedure BACKTRACKING(grid)
    for row do
        for column do
            if grid(row,column) = 0 then
                try a value in this position
                Backtracking(grid with new value)
                if successful then
                    return grid
                else:
                    try another value
                end if
                if no values left to try then
                    return False
                end if
            end if
        end for
    end for
    return grid
end procedure
```

Why does brute force not work for larger examples? It will work *TO DO: PROVE ALG CORRECTNESS* but due to the complexity of the problem (point back to sudoku is hard chapter) it is infeesible.

4.2 Simulated Annealing

Based on metalurgy

Algorithm 5 Simulated Annealing

```
procedure SimAnnealing(grid, schedule, f)
    current = initialise state
    for t = 1to inf do
        T=schedule[t]
        if T ≤ ϵ then
            return current
        else
            choose successor at random
            ΔE = f(successor) - f(current)
            if ΔE ≥ 0 then
                current = succ
            else choose with probability e^(ΔE/T)
                current = successor
            end if
        end if
    end for
end procedure
```

4.2.1 Convergence

4.2.2 Speed of Convergence

[**?**] one of 100 most cited papers, one of the first AI algs

Chapter 5

Group theory

5.1 Starting Simple 4×4

Let us analyse Shidoku which is a specific set of sudokus with dimensions 4 by 4 the smallest non trivial sudoku puzzle. Only 2 fundamentally different. One has 96 identical, other has 192. Why not the same amount?

5.2 Equivalence Classes

5.3 6×6

Define Rodoku Define which sudoku sizes can exist

812

5.4 8×8

5.5 9×9

5,472,730,538

Chapter 6

Other

6.1 Other Related Problems

6.1.1 Latin Squares

- A latin square is an n by n matrix filled with n characters that must not repeat along columns or rows.
- Reduced Form - f first row and column is in the natural order
- Equivalence classes
- Number of n by n latin squares is bounded
- Latin squares can be considered a bipartite graph
- Agronomic Research
- Latin hypercube

6.2 Magic Squares

- A magic square is a matrix of numbers with each column, row and diagonal summing to the same value, this value is known as a magic constant and the degree is the number of columns/rows.
- A normal magic square is one containing the integers 1 to n^2.
- Magic Squares with repeating digits are considered trivial.
- Semimagic squares omit the diagnonal sums also summing to the magic constant.
- Truly thought to be magic Shams Al-ma'arif.
- Generation, there exists not completely general techniques. Diamond Method
- Associative Magic Squares
- Pandiagonal Magic Squares
- Most-Perfect Magic Squares
- Equivalence classes for $n <= 5$ but not for higher orders.

- The enumeration of most perfect magic squares of any order.
- 880 distinct magic squares of order four
- Normal magic squares can be constructed for all values except 2
- Preserving the magic property when transformed
- Methods of construction
- Multiplicative magic squares - produce infinite
- Sator square
- magic square of squares - Parker Square is a failed example of this

6.2.1 Greco-Latin Squares

- Two orthogonal latin squares super imposed, such that the pairs of values are unique.
- Group based greco latin squares
- Eulers interest came from construction of magic squares
- Exists for all but 2 and 6.

6.3 Generating Techniques

A polynomial generation algorithm without requiring a uniqueness checker which we have proven to be np-complete and therefore infeesible for large n.

6.4 17 is the Magic Number

4 for shidoku

6.4.1 Sparsity - information theory

Bomb sudoku/latin squares - Additional rule: the same number can not occur in adjacent or diagonally adjacent squares.

6.5 Topology

6.5.1 Torus

6.6 Polynomials & Constraint Programming

Use of polynomials Roots of unity Grobner Basis

Bibliography

[1] https://en.wikipedia.org/wiki/NP_(complexity)

[2] Artificial Intelligence: A modern Approach Archived 2017-08-31 at the Wayback Machine [1995...] Russell and Norvig

[3] https://www.researchgate.net/publication/251863893_A_New_Algorithm_for_Generating_Unique-Solution_Sudoku

[4] https://fse.studenttheses.ub.rug.nl/22745/1/bMATH_2020_HoexumES.pdf.pdf

[5] http://web.math.ucsb.edu/p̃adraic/mathcamp_2014/np_and_ls/mc2014_np_and_ls_lecture3.pdf, http://web.math.ucsb.edu/p̃adraic/mathcamp_2014/np_and_ls/mc2014_np_and_ls_lecture4.pdf

[6] https://scholar.rose-hulman.edu/cgi/viewcontent.cgi?article=1398&context=rhumj

[7] https://onlinelibrary.wiley.com/doi/10.1002/(SICI)1520-6610(1996)4:6¡405::AID-JCD3¿3.0.CO;2-J

[8] http://joas.agrif.bg.ac.rs/archive/article/59

[9] https://www.semanticscholar.org/paper/Permutation-arrays-for-powerline-communication-and-Colbourn-Kløve/7e69cfdbd2082463c66de698da1e326f0556d1d4

[10] http://www.multimagie.com/English/SquaresOfSquaresSearch.htm

[11] https://plus.maths.org/content/anything-square-magic-squares-sudoku

[12] https://link.springer.com/book/10.1007/978-1-4302-0138-0

[13] an Laarhoven, P.J.M., Aarts, E.H.L. (1987). Simulated annealing. In: Simulated Annealing: Theory and Applications. Mathematics and Its Applications, vol 37. Springer, Dordrecht. https://doi.org/10.1007/978-94-015-7744-1_2