数独の解生成と解に対する番号付け

戸神星也

理学部 情報科学科　　指導教官 渡辺治

2007年2月

目 次

第1章　はじめに

パズルの一種に「数独」というものがある.「数独」というのは図 1.1(左) のような盤面が与えられたときに，ルール (後述) にしたがって解を導くパズルである．例えば，この盤面に対しては図 1.1(右) が解となる.

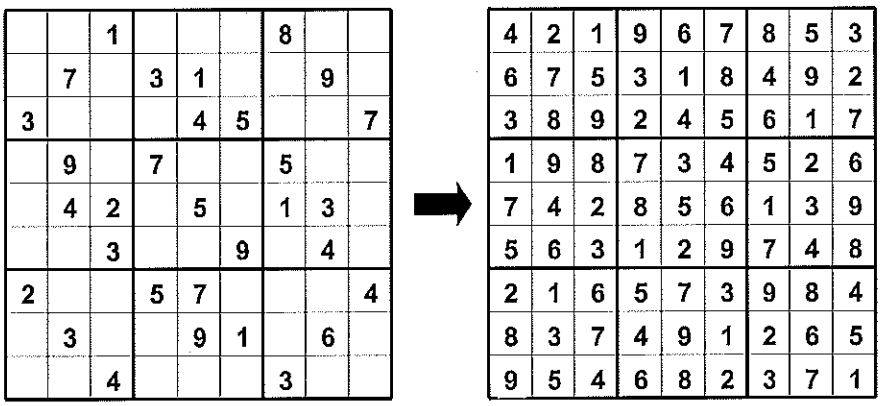

図 1.1: 3 × 3 数独の例

本研究では数独の解盤面 (図 1.1(右) は解盤面の一つである) に注目し，解盤面に対する番号付けを提案する．また，番号付けを効率よく行う方法を既存の研究を元に実践し，更にその改良方法を提案する.

背景

本研究の対象である数独は，**ペンシルパズル**の一種である．ペンシルパズルとは，紙に印刷された問題の図に鉛筆で書き込みながら解き進めていくという形態のパズルであり「スリザーリンク」,「カックロ」,「ナンバーリンク」,「ののぐらむ」等が有名である．数独はペンシルパズルの中で最もよく知られているパズルの一つであり，数独のみを扱う専門雑誌やゲームソフトの類が数多く世に出ている.

ペンシルパズルの特徴は,「解くのは難しいが解いた答えが正しいことは容易に確認できる」ということであり，これは NP 問題の性質に一致している．したがってペンシルパズルの多くは NP 問題であると予想される．実際に「数独」($n \times n$ 数独，後述) をはじめとした多くのペンシルパズルの NP 完全性が証明されている.

数独

次に，数独の説明をする．数独とは図1.1のような，既に埋まっているマスの数字を頼りに空いているマスを埋めていくパズルである．この際，一種の魔方陣を完成させるように数字を埋めていく．以下，正確なルールを記す．

- 空いている**マス**に1から9までどれかの数字を埋める
- 同じ**列**に1から9の数字をそれぞれ1つずつ埋める
- 同じ**行**に1から9の数字をそれぞれ1つずつ埋める
- 太線で囲まれた3×3のマスの集合を**ブロック**といい，
  同じブロックに1から9の数字をそれぞれ1つずつ埋める

問題としては図1.1(左)が与えられ，図1.1(右)のような解を導く．そして，右のように導かれた解を**解盤面**と呼ぶことにする．

また，マスの総数が$9 \times 9 (= 3^2 \times 3^2)$の場合に限らず，一般にマスの総数が$n^2 \times n^2$であれば数独を定義でき，実際に$16 \times 16 (n = 4), 25 \times 25 (n = 5)$といったサイズの数独も存在する．このときのルールは9×9の数独と同様であり，各列，各行，各ブロックに1からn^2までの数字をそれぞれ1つずつ埋めるというものである．このときのブロックは$n \times n$のマスからなる．ただ，これらは一般的なものではなく雑誌等に掲載されている数独も，サイズが$3^2 \times 3^2$のものが殆どである．本論文では$n^2 \times n^2$のマスからなる数独を$n \times n$数独と呼ぶ．また，単に数独といえば3×3数独を指すことにし，単に解盤面といえば3×3数独の解盤面を指すことにする．

ラテン方陣

数独の解盤面に関連して**ラテン方陣**というものがある．ラテン方陣とは図1.2で表されるようなものであり，数独の解盤面からブロックの制約条件を除いたものである．3×3数独の解盤面の集合は，9×9マスのラテン方陣の集合に真に含まれる．

ラテン方陣に関しては古くから数学的な研究がなされており，その中の一つにラテン方陣の**標準形**というものがある．ラテン方陣の標準形とは，$n \times n$マスのラテン方陣について1列目を上から順番に$1, 2, \cdots, n$とし，1行列を左から順番に$1, 2, \cdots, n$としたものであり，各列の1行目及び各行の1列目をソートすることで任意のラテン方陣は標準形のラテン方陣に変換可能である．なぜなら，列同士の交換及び行同士の交換ではラテン方陣の制約条件を崩さないからである．こういった標準形の概念が，後に述べる数独の解盤面の標準形の基になっている．ただし，数独にはブロックによる制約条件が加わるため標準形の定義はもう少し複雑になる．

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

図 1.2: ラテン方陣の例

ラテン方陣を解盤面とする，部分ラテン方陣完成問題というものも存在する．ルールは数独のルールからブロックによる制約を取り除いたものであり，Colbourn により NP 完全性が示されている [3]．また，八戸により $n \times n$ 数独の NP 完全性が示されている [2]．これはラテン方陣完成問題からの還元による．

存在しうる解盤面の数

解盤面として存在し得るパターンは何通りあるかという研究が F.Jarvis らによってなされており，$6670903752021072936960 \approx 6.67 \times 10^{21}$ 通りであることがわかっている [1]．解盤面の組み合わせの数を求めるのに，バックトラック法を用いてコンピュータを利用して求めるという方法が可能であるが，10^{21} というサイズの解盤面を全て列挙することは現在のコンピュータの能力では困難である．Jarvis らはコンピュータで列挙すべき範囲を大幅に縮小する方法 (主に，**同型変換**と呼ばれる手法を用いる) を提案し，組み合わせの総数を得た．これらについては第 2 章で詳しく取り扱う．

関連として 9×9 マスのラテン方陣の総数も S.F.Bammel，J.Rothstein らによって正確に数え上げられており，およそ $5524751496156892842531225600 \approx 5.525 \times 10^{27}$ であることがわかっている [4]．つまり，9×9 ラテン方陣のうちおよそ 0.00012%が 3×3 数独の解盤面となっている．

同型変換

同型変換については第 2 章で詳しく述べるが，大きく分けて 2 種類の同型変換が存在する．一つは**標準形**を用いたもので，もう一つは上 3 ブロックに対するものである．

例えばラテン方陣の組み合わせの総数を求めようとするとき，標準形のラテン方陣の

みを取り扱えばよい．9×9マスのラテン方陣の組み合わせの総数は，同サイズの標準形のラテン方陣の組み合わせの総数の9!×8!倍である．9!×8!というのは1列目と1行目に入る数字の組み合わせの総数である．これを用いれば，例えばラテン方陣の組み合わせを探索によって求めるとき，探索範囲は9!×8!分の1に狭めることができる．数独の解盤面に対しても同様に数独の標準形を定義して探索範囲を狭めることができる．これが標準形を用いた同型変換である．

また，ラテン方陣内に図1.3(左)のようなパターンが含まれるとき図1.3(左)の部分だけ図1.3(右)に変換してもそれはラテン方陣の性質を満たす．これを数独の解盤面に応用したものが上3ブロック(=上3行)に対する同型変換である．数独の場合はブロック単位で制約条件があるため，今回は上3ブロックについての変換を考えている．

図 1.3: ラテン方陣における変換の例

新しい同型変換

本研究では前節で述べた上3ブロックに対する同型変換で，今までに提案されていないものを発見した．これについては第3章で詳しく取り扱う．

解盤面に対する番号付け

本研究では存在する全ての解盤面に対して番号付けを行うことを提案する．全ての解盤面は6670903752021072936960(= N_0 とする)通りあり，それぞれの解盤面に対して1から N_0 までの番号を付けることを考える．番号付けの実現は，解盤面の組み合わせを求めるときと同様に，コンピュータを用いれば番号付けは容易にできる．しかし，単純な方法では現在のコンピュータの能力を考えると困難となる．具体的には最悪時実行時間が何万年とかかってしまう．

そこで，効率のよい番号付けを実現するために**同型変換**を利用して実行時間を大幅に短縮する方法を提案，実践した．この方法は，簡単に述べると同型変換とバックトラックによるものであり，この方法を用いることにより最悪時実行時間を5分程度まで縮めることに成功した．

また，更なる改良として探索すべき解盤面を全て記憶媒体に保存していくという手法を提案，実践した．この方法を用いることにより最悪時実行時間を10秒程度まで縮める

ことに成功した．この方法の問題点は，予め保存しておくべき記憶領域が膨大になってしまうことであり，記憶領域の縮小が課題となった．単純に探索すべき解盤面を保存しておくだけだと180TB程度の領域が必要な計算になってしまうが，Jarvisらによって提案された上3ブロックに対する同型変換，および本研究で発見した同型変換を用いることにより最終的には必要記憶領域を40GB程度にすることに成功した(bzip等の圧縮技術を利用すれば4GB程度にすることが可能)．これらについては第4章で詳しく取り扱う．

本論文の構成

本論文の構成は次のようになっている．まず第2章ではJarvisらによるN_0を求める方法を紹介し，それに伴って数独の**標準形**の定義，及び2種類の同型変換(**標準化**という手法によるものと，上3ブロックに対するもの)についての説明をする．第3章では上3ブロックに対する同型変換で，今回新しく発見したものについて説明する．第4章では第2章，第3章で紹介した同型変換を用いて，解盤面に対する番号付け及び番号からの解盤面生成を高速に行う手法について解説する．最後に第5章で，結果と今後の課題について述べる．

第2章　数独の解盤面の数え上げ

3×3 数独の解盤面の場合の数は $6670903752021072936960 \approx 6.67 \times 10^{21}$ である．この数字はコンピュータで総当りで解を探す方法を用いれば求めることができる (**総当り計算**という表現を本論文では用いている)．しかしこれは現在のコンピュータの能力では現実的な方法でない．第 4 章でまた述べるが，バックトラック法を用いた場合，解盤面を導き出すためにはおよそ 10^{17}秒 $\approx 3 \times 10^{10}$年 ほどかかる．本章では，数独の解盤面を如何にして数え上げているのかを説明している．

2.1　準備

まず，本論文で用いる記号及び用語の定義を行う．

定義 2.1　3×3 数独の解盤面全体の集合を SUDOKU と表す

定義 2.2　$N_0 = |\mathrm{SUDOKU}|$ とする．

定義 2.3　数独内の枠で囲まれた部分を**マス**と呼ぶ．3×3 数独の場合は 81 のマスが存在する．また，マスに図 2.1 のようなラベルを付ける．例えば a_{54} は 3×3 数独の第 5 行第 4 列のマスを指す．

a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	a_{16}	a_{17}	a_{18}	a_{19}
a_{21}	a_{22}	a_{23}	a_{24}	a_{25}	a_{26}	a_{27}	a_{28}	a_{29}
a_{31}	a_{32}	a_{33}	a_{34}	a_{35}	a_{36}	a_{37}	a_{38}	a_{39}
a_{41}	a_{42}	a_{43}	a_{44}	a_{45}	a_{46}	a_{47}	a_{48}	a_{49}
a_{51}	a_{52}	a_{53}	a_{54}	a_{55}	a_{56}	a_{57}	a_{58}	a_{59}
a_{61}	a_{62}	a_{63}	a_{64}	a_{65}	a_{66}	a_{67}	a_{68}	a_{69}
a_{71}	a_{72}	a_{73}	a_{74}	a_{75}	a_{76}	a_{77}	a_{78}	a_{79}
a_{81}	a_{82}	a_{83}	a_{84}	a_{85}	a_{86}	a_{87}	a_{88}	a_{89}
a_{91}	a_{92}	a_{93}	a_{94}	a_{95}	a_{96}	a_{97}	a_{98}	a_{99}

図 2.1: マスに対するラベル

定義 2.4　3 × 3 数独内の太枠で囲まれた 3 × 3 の部分を**ブロック**と呼ぶ.

3×3 数独の場合は 9 のブロックが存在する. ブロックに図 2.2 のようなラベルを付ける. また図 2.2 で Bi と表記されているブロックを第 i ブロックと呼ぶ ($i = 1, \cdots, 9$).

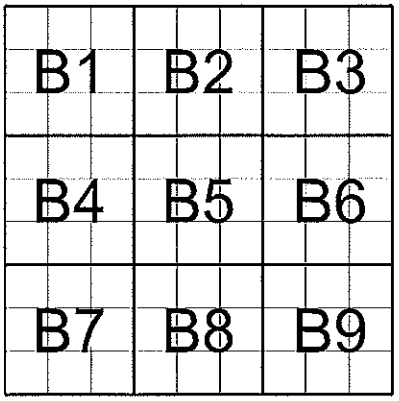

図 2.2: ブロックに対するラベル

いま, 上 3 行が図 2.3 のような数独の解盤面の総数が M であったとする. このとき, 上 3 行が図 2.4 のような数独の解盤面の総数は M であることがわかる. なぜなら, 図 2.3 の 1 列目と 2 列目を入れ替えたものが図 2.4 であり, 図 2.3 を上 3 行とする解盤面は 1 列目と 2 列目を入れ替えることによって図 2.4 を上 3 行とする解盤面となるからである (詳しい証明は後述).

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

図 2.3: 同型変換の例 (1)

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

図 2.4: 同型変換の例 (2)

このように図2.3から計算によりMが求まったとき，図2.4の場合を計算する必要はない．Mであることがわかるからである．本章の数え上げは最終的には総当り計算をして求めるものであるが，このように探索する範囲を狭めている．具体的には探索範囲を10^{12}分の1程に狭めている．詳しくは後で述べる．このような探索範囲を狭める手法を**同型変換**と呼び，Jarvisら[1]により提案されている．以下同型変換についての詳細を紹介していく．

2.2 同型変換(1)

2.2.1 第1ブロックに対する変換

第1ブロックを図2.5のように固定することを考えよう．

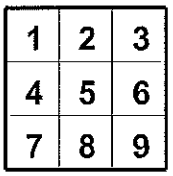

図 2.5: 固定された第1ブロック

定理 2.5 任意の$\{a_1, a_2, \cdots, a_9\} = \{1, 2, \cdots, 9\}$について，第1ブロックが図2.6と同じ形になっている解盤面の個数と，第1ブロックが図2.5と同じ形になっている解盤面の個数は同じである．

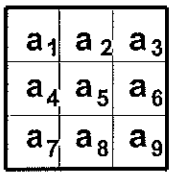

図 2.6: 任意の第1ブロック

証明 任意の$\{a_1, a_2, \cdots, a_9\} = \{1, 2, \cdots, 9\}$について，以下が成り立つ．

$$V = \{\text{解盤面} \mid \text{第1ブロックが図2.5の形}\}$$

$$W = \{\text{解盤面} \mid \text{第1ブロックが図2.6の形}\}$$

とする．このとき，関数fを次のように定義する．

入力：第1ブロックが図2.6となっている解盤面

出力： 入力された解盤面の全てのマスに対して $a_i \to i$ $(i = 1, \cdots, 9)$
の変換を加えたもの.

このとき $\forall w \in W$ について $f(w) \in V$ であり，かつ f は全単射であるから $|W| = |V|$.

□

以上より，第1ブロックが図2.5となっている解盤面のパターンが計算できれば，第1ブロックが図2.5以外の解盤面については計算する必要がないことがわかる．第1ブロックのパターンは $9! = 362880$ 通りあるため，この同型変換により探索範囲を362880分の1とすることができる.

また，ある解盤面から第1ブロックが図2.5となっている解盤面へ変換する操作を**第1ブロックによる標準化**と呼ぶ．変換の方法は定理2.5の証明で与えた f によるものである.

2.2.2 第1行に対する変換

第2ブロック内に対する変換

次に列交換による同型変換を示す.

1	2	3	a_1	a_2	a_3			
4	5	6	a_4	a_5	a_6			
7	8	9	a_7	a_8	a_9			

図 2.7: 同ブロック内の列交換 (1)

1	2	3	a_2	a_1	a_3			
4	5	6	a_5	a_4	a_6			
7	8	9	a_8	a_7	a_9			

図 2.8: 同ブロック内の列交換 (2)

定理 2.6 任意の $a_1, a_2, \cdots, a_9$ $(\{a_1, a_2, \cdots, a_9\} = \{1, 2, \cdots, 9\})$ について，
第2ブロックが図2.7と同じ形になっている解盤面の個数と，第2ブロックが図2.8と同じ形になっている解盤面の個数は同じである.

証明 任意の $a_1, a_2, \cdots, a_9$ ($\{a_1, a_2, \cdots, a_9\} = \{1, 2, \cdots, 9\}$) について，以下が成り立つ．

$$V = \{\text{解盤面} \mid \text{第2ブロックが図2.7と同じ形}\}$$

$$W = \{\text{解盤面} \mid \text{第2ブロックが図2.8と同じ形}\}$$

とする．このとき，関数 f を次のように定義する．

入力： 第2ブロックが図2.7と同じ形となっている解盤面

出力： 入力された解盤面の第4列と第5列の要素をそれぞれの行において交換したもの

このとき $\forall w \in W$ について $f(w) \in V$ であり，かつ f は全単射であるから $|W| = |V|$．

□

この事実を拡張すれば，任意の $a_1, a_2, \cdots, a_9$ ($\{a_1, a_2, \cdots, a_9\} = \{1, 2, \cdots, 9\}$) に対して，第2ブロックが図2.9に示された6つのパターンとなっている解盤面の個数はそれぞれ等しいということがわかる．つまり，図2.9の6つのパターンのうちひとつが計算できれば，残る5つのパターンについては計算を省くことができる．

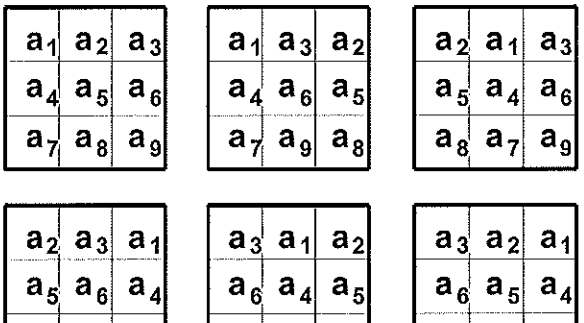

図 2.9: 6つのパターン

つまり，以下の定理が成り立つ．

定理 2.7 任意の $a_1, a_2, \cdots, a_9$ ($\{a_1, a_2, \cdots, a_9\} = \{1, 2, \cdots, 9\}$) について，第2ブロックが図2.9のいずれかと同じ形になっている解盤面の個数と，第2ブロックが図2.9の(別の)いずれかと同じ形になっている解盤面の個数は等しい．

第3ブロック内の変換

第3ブロックによる同型変換は，第2ブロックの場合と同様に図2.9の6つのパターンのうちひとつを計算すればよいというものである．証明等は第2ブロックのときとほぼ同じであるため省略する．また，以下の定理が成り立つ．

定理 2.8 任意の $a_1, a_2, \cdots, a_9$ ($\{a_1, a_2, \cdots, a_9\} = \{1, 2, \cdots, 9\}$) について，第3ブロックが図2.9のいずれかと同形になっている解盤面の個数と，第3ブロックが図2.9の(別の)いずれかと同形になっている解盤面の個数は等しい．

ブロックに対する変換

ブロック交換による同型変換を示す．

1	2	3	a_1	a_2	a_3	b_1	b_2	b_3
4	5	6	a_4	a_5	a_6	b_4	b_5	b_6
7	8	9	a_7	a_8	a_9	b_7	b_8	b_9

図 2.10: ブロック交換 (1)

1	2	3	b_1	b_2	b_3	a_1	a_2	a_3
4	5	6	b_4	b_5	b_6	a_4	a_5	a_6
7	8	9	b_7	b_8	b_9	a_7	a_8	a_9

図 2.11: ブロック交換 (2)

定理 2.9 任意の $a_1, \cdots, a_9, b_1, \cdots, b_9$ ($\{a_1, \cdots, a_9\} = \{1, \cdots, 9\}, \{b_1, \cdots, b_9\} = \{1, \cdots, 9\}$) について，第2，第3ブロックが図2.10と同形になっている解盤面の個数と第2，第3ブロックが図2.11と同形になっている解盤面の個数は同じである．

証明 任意の $a_1, \cdots, a_9, b_1, \cdots, b_9$ ($\{a_1, \cdots, a_9\} = \{1, \cdots, 9\}, \{b_1, \cdots, b_9\} = \{1, \cdots, 9\}$) について，以下が成り立つ．

$$V = \{\text{解盤面} \mid \text{第2，第3ブロックが図2.10と同形}\}$$

$$W = \{\text{解盤面} \mid \text{第2，第3ブロックが図2.11と同形}\}$$

とする．このとき，関数 f を次のように定義する．

入力： 第2，第3ブロックが図2.10と同形なっている解盤面

出力： 入力された解盤面の第4列と第7列の要素，第5列と第8列の要素，第6列と第9列の要素をそれぞれの行において交換したもの

このとき $w \in W$ について $f(w) \in V$ であり，かつ f は全単射であるから $|W| = |V|$．

□

つまり，図2.10を第2,第3ブロックとする解盤面の個数が計算できれば，図2.11を第2,第3ブロックとする解盤面については計算を省略できる．

第1行に対する同型変換

上記3つの同型変換をあわせて考えると，$a_{14} < a_{15} < a_{16}$，$a_{17} < a_{18} < a_{19}$，$a_{14} < a_{17}$ となっている解盤面の個数が計算できれば，そうでない解盤面については計算を省略できることがわかる．これを第1行に対する同型変換ということにする．探索範囲は第2,第3ブロック内の変換でそれぞれ6分の1に，ブロックに対する変換で2分の1としていて，これらは独立に変換可能であるので，第1行に対する同型変換では $6 \times 6 \times 2$，即ち72分の1に探索範囲を狭めている．

また，ある解盤面から $a_{14} < a_{15} < a_{16}$，$a_{17} < a_{18} < a_{19}$，$a_{14} < a_{17}$ となっている解盤面へ変換する操作を**第1行による標準化**と呼ぶ．変換は定理2.6，及び定理2.9の証明内で与えた関数 f によって行う．

2.2.3 第1列に対する変換

第1列に対する同型変換は，第1行に対する同型変換と考え方は全く同じである．前節と同様に，第4，第7ブロック内の変換，第4，第7ブロックの交換による変換をあわせて考えることにより前節とは独立して72分の1に探索範囲を狭めることができる．また，第1行による標準化と同様に第4，第7ブロック内の第1列の要素がそれぞれ昇順，かつ $a_{41} < a_{71}$ となっている解盤面に変換する操作を**第1列による標準化**と呼ぶ．

以上第1ブロック，第1行，第1列それぞれの同型変換をあわせて考えると探索範囲を $9! \times 72 \times 72 = 1881169920$ 分の1とすることができる．以下，解盤面の標準形について定義する．

定義 2.10 $a_{11} = 1, \cdots, a_{13} = 3, a_{21} = 4, \cdots, a_{33} = 9$，$a_{14} < a_{15} < a_{16}$，$a_{17} < a_{18} < a_{19}$，$a_{14} < a_{17}$，$a_{14} < a_{15} < a_{16}$，$a_{17} < a_{18} < a_{19}$，$a_{14} < a_{17}$ ($\{a_{14}, \cdots, a_{19}\} = \{4,5,6,7,8,9\}$，$\{a_{41}, \cdots, a_{91}\} = \{2,3,5,6,8,9\}$) であるような解盤面を**標準形**の解盤面と呼ぶ．また，任意の解盤面を標準形に変換することを**標準化**と呼ぶ．

標準化は，第1ブロックによる標準化，第1行による標準化，第1列による標準化をそれぞれ実行することにより実現可能である．

2.3 探索

2.2節より，標準形の解盤面の総数がわかれば，解盤面の総数がわかることになる．標準形の解盤面の総数に1881169920をかけたものが解盤面の総数になっているからである．

ただ，標準化だけでは探索範囲を十分に狭めているとはいいがたい．2.4節ではさらに探索範囲を狭めるための方法を示す．

上3行を埋めるパターンの数

ここでは，解盤面の一部となっているような第1,第2,第3ブロックのパターンが何通りあるかを議論する．この議論は探索範囲を狭める手法と直接は関係がないがこの結果は探索範囲をどの程度狭めることができているかの評価をする際に必要となる．

いま，2.2節で説明した標準形の数独のみを考えており，更に上の3ブロックについてのみ考えているので第1ブロックによる同型変換と第1行による同型変換を用いている．よって実際のパターン数はここで得られたものに$9! \times 72 = 26127360$をかけたものとなっている．

456,789型

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}		

図 2.12: 456,789型

まず，第2ブロックの第1行に第1ブロックの第2行がくる場合を考える．このとき第2ブロック第3ブロック内の第2行，第3行に入る数字の集合も一意に定まる．よってこのとき$(3!)^2 \times (3!)^2 = 1296$個のパターンが存在する．

457,689型

次に，上で考えた場合以外の型を考える．図2.13に示したのは一例であり第1行に入る数字の組み合わせは${}_5C_2 = 10$通りである (次項の表参照)．

1	2	3	4	5	7	6	8	9
4	5	6	{8, 9, a}			{7, b, c}		
7	8	9	{6, b, c}			{4, 5, a}		

図 2.13: 457,689 型

456,789	468,579
457,689	469,578
458,679	478,579
459,678	479,568
467,589	489,567

ただし 456,789 型については先に議論したので考えない．よって 9 通りのパターンが存在する．このとき第 2 ブロック第 3 ブロック内の第 2 行，第 3 行に入る数字の集合は 3 通りに定まる．$\{a, b, c\} = \{1, 2, 3\}$ であり，a の選び方が 3 通り存在するからである．よってこのとき $9 \times 3 \times (3!)^2 \times (3!)^2 = 34992$ 個のパターンが存在する．

よって，以下の定理が成り立つ．

定理 2.11 標準形の解盤面において，第 1 ブロックから第 3 ブロックのパターンは 36288 個存在する．

2.4 同型変換 (2)

この節では上で考察した 36288 個のパターンに対しての同型変換を紹介する．つまり，36288 個のパターンのうちあるパターンを含む解盤面の総数を計算することによって，別のパターンを含む解盤面の総数の計算を省く，という手法について述べる．

例えば，第 1 ブロックから第 3 ブロックの各列において第 1 行と第 2 行の数字をそれぞれ入れ替るといった，各列の要素の組を変えないような変換を施しても，それを含む数独の解盤面の総数は変わらない．なぜなら，第 4 行以降のマスの制約条件が一切変わらないためである．もちろん，この変換は標準形を崩す．しかし，この変換と標準化を合わせることによりある標準形から別の標準形への変換が可能となる場合がある．現在 7 つの同型変換が提案されており，以下それぞれの変換について説明する．

2.4.1 列交換系の変換

行交換による変換

行交換とは第1行から第3行までの任意の行を入れ替える変換であり，これは各列の要素の組を変えない，即ち第4行以降のマスの制約条件を変えない変換である．図2.14は行交換の一例である．変換の前後で別のパターンとなっている．

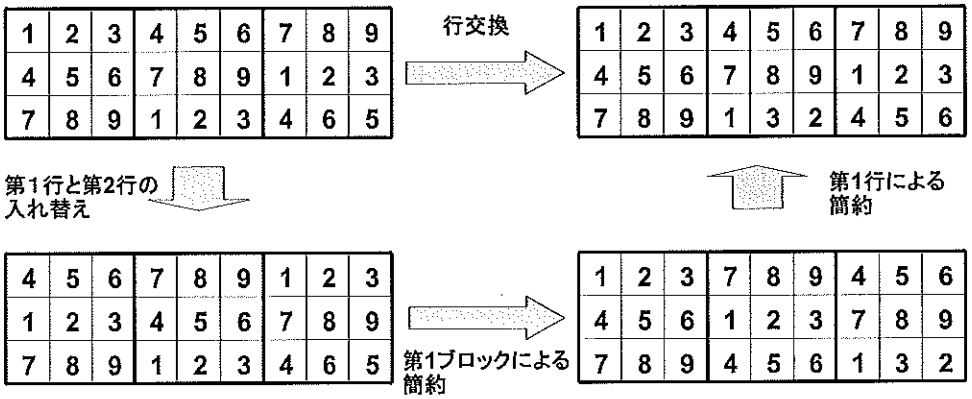

図 2.14: 行交換の例

列交換による変換

列交換とは第1列から第3列までの任意の行を入れ替える同型変換である．解盤面の上3ブロックのみを考えたとき，そこに列交換を施す場合，交換の前と後とでそれを含む解盤面の個数は変化しない．証明は定理2.6のときと同様である．

ブロック交換による変換

ブロック交換とは第1ブロックから第3ブロックまでの任意のブロックを入れ替える変換である．解盤面の上3ブロックのみを考えたとき，そこにブロック交換を施す場合，交換の前と後とでそれを含む解盤面の個数は変化しない．証明は定理2.9のときと同様である．

2.4.2 $m \times n$ 交換系の変換

次に，常に変換が可能とは限らない同型変換について紹介する．列交換系とは異なり局所的な変換を行うのが特徴である．

2×2 交換による変換

定義 2.12 以下の関数で実行される同型変換を 2×2 交換と呼ぶ.

2x2$[x1, x2, y1, y2]$

input: 標準形の解盤面

output: 標準形の解盤面

1. 入力された解盤面について，2から5を実行し，盤面を出力
2. 第 $x1$ 列について，第 $y1$ 行と第 $y2$ 行を入れ替える
3. 第 $x2$ 列について，第 $y1$ 行と第 $y2$ 行を入れ替える
4. 第1ブロックによる標準化を行う
5. 第1行による標準化を行う

ただし，この変換を実行した後も解盤面が出力されるためには

$a_{y1x1} = a_{y2x2}$, $a_{y1x2} = a_{y2x1}$ である必要がある.

図 2.15 が 2×2 交換の例である.

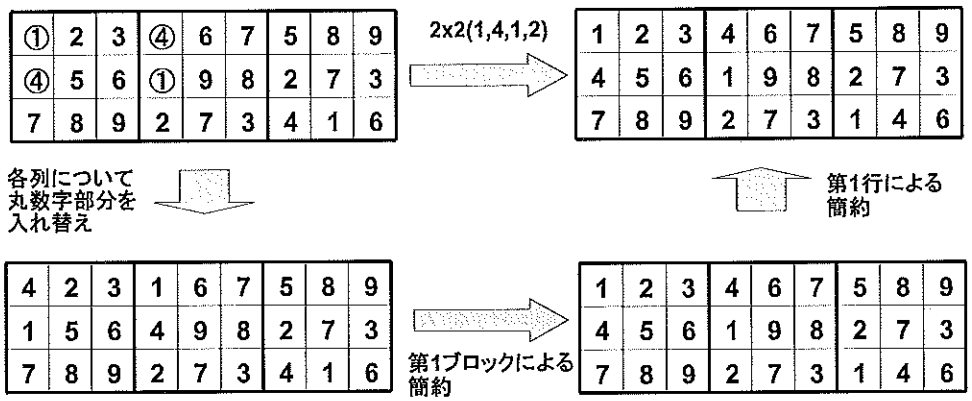

図 2.15: 2×2 交換の例

3×2 交換による変換

定義 2.13 以下の関数で実行される同型変換を 3×2 交換と呼ぶ.

3x2$[x1, x2, x3, y1, y2]$

input: 標準形の解盤面

output: 標準形の解盤面

1. 入力された解盤面について，2から6を実行し，盤面を出力
2. 第 $x1$ 列について，第 $y1$ 行と第 $y2$ 行を入れ替える
3. 第 $x2$ 列について，第 $y1$ 行と第 $y2$ 行を入れ替える

4. 第 $x3$ 列について，第 $y1$ 行と第 $y2$ 行を入れ替える
5. 第 1 ブロックによる標準化を行う
6. 第 1 行による標準化を行う

ただし，この変換を実行した後も解盤面が出力されるためには
$\{a_{y1x1}, a_{y1x2}, a_{y1x3}\} = \{a_{y2x1}, a_{y2x2}, a_{y2x3}\}$ である必要がある．

4×2 交換による変換

定義 2.14 以下の関数で実行される同型変換を 4×2 交換と呼ぶ．

4x2$[x1, x2, x3, x4, y1, y2]$

input: 標準形の解盤面

output: 標準形の解盤面

1. 入力された解盤面について，2 から 7 を実行し，盤面を出力
2. 第 $x1$ 列について，第 $y1$ 行と第 $y2$ 行を入れ替える
3. 第 $x2$ 列について，第 $y1$ 行と第 $y2$ 行を入れ替える
4. 第 $x3$ 列について，第 $y1$ 行と第 $y2$ 行を入れ替える
5. 第 $x4$ 列について，第 $y1$ 行と第 $y2$ 行を入れ替える
6. 第 1 ブロックによる標準化を行う
7. 第 1 行による標準化を行う

ただし，この変換を実行した後も解盤面が出力されるためには
$\{a_{y1x1}, a_{y1x2}, a_{y1x3}, a_{y1x4}\} = \{a_{y2x1}, a_{y2x2}, a_{y2x3}, a_{y2x4}\}$ である必要がある．

図 2.16 が 4×2 交換の例である．

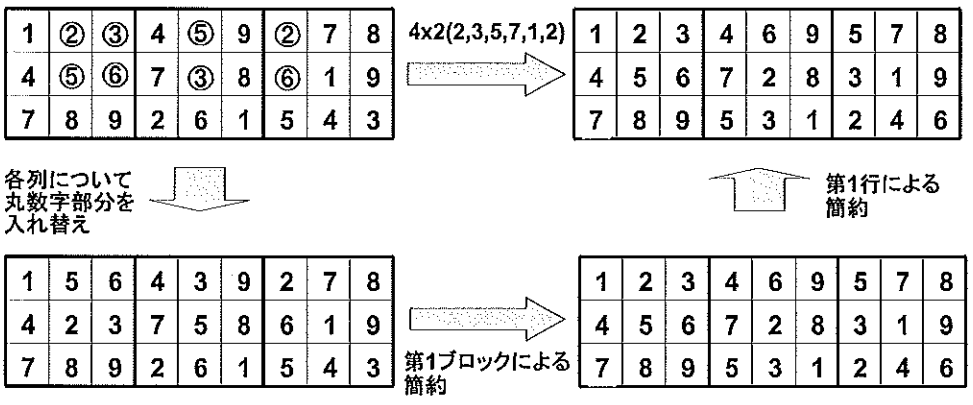

図 2.16: 4×2 交換の例

2×3 交換による変換

定義 2.15 以下の関数で実行される同型変換を 2×2 交換と呼ぶ.

2x3$[x1, x2]$

input: 標準形の解盤面

output: 標準形の解盤面

1. 入力された解盤面について，2から5を実行し，盤面を出力
2. 各行について，第 $x1$ 列と第 $x2$ 列を入れ替える
3. 第1ブロックによる標準化を行う
4. 第1行による標準化を行う

ただし，この変換を実行した後も解盤面が出力されるためには
$\{a_{1x1}, a_{2x1}, a_{3x1}\} = \{a_{1x2}, a_{2x2}, a_{3x2}\}$ である必要がある.

図 2.17 が 2×3 交換の例である.

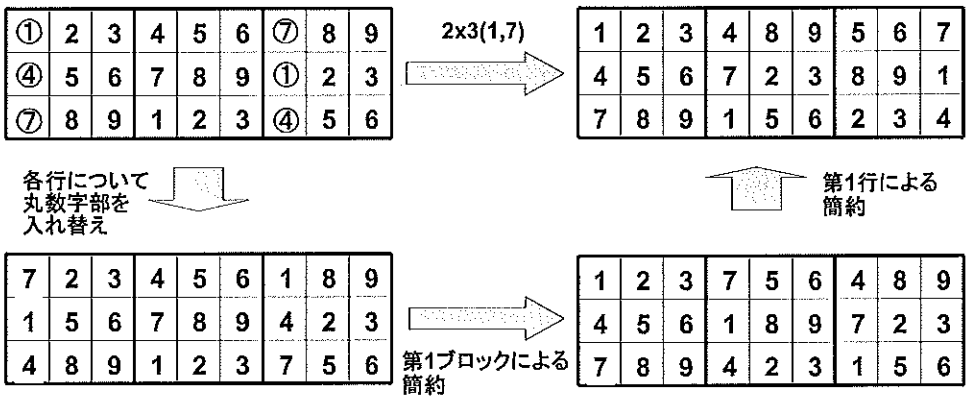

図 2.17: 2×3 交換の例

まとめ

以上7つの同型変換を用いると実際に計算すべきパターンは 71 にまで落ちる．この結果はコンピュータを用いて導き出したものである.

まとめ

2.2節では探索範囲を1881169920分の1に，2.4節では探索範囲を36288分の71に狭めている．よってこれらの結果を合わせると探索範囲は $1881169920 \times 36288/71 \approx 10^{12}$ 分の1となっている．このレベルまで探索範囲を絞ることができれば現実的に解盤面を全生成することが可能である．

実際にコンピュータ (スペック等を追記予定) を用いて解生成を行ったところ，6時間ほどで $N_0 = 6670903752021072936960 \approx 6.67 \times 10^{21}$ という結果が得られた．解生成には以下のアルゴリズムを用いた．

N_0 を求めるアルゴリズム

1. 先に述べた標準形の上3ブロックのうち71パターンについて **2** から **4** を実行し，それぞれ4で得られた値を足し合わせる
2. それぞれ同パターンの数独が何通りあるかを調べる
3. 標準形のもののみ全て生成し，その個数をカウントする
4. **2**，**3** で得られた値をかけ合わせる
5. **1** で得られた値に1881169920をかけたものを出力する

定理 2.16 $N_0 = 6670903752021072936960$ である．

また，なぜ2.4節の変換で調べるべきパターンの数が71に落ちるのか，ということに関してはよくわかっていない．実際に71パターンの各状態について，それを含む解盤面の総数を調べると解盤面の総数のパターンは44個しかないことがわかる．即ち同じ数だけ解盤面を”生成する”パターンがいくつか存在している，ということである．

次章では上3ブロックに関する新しい同型変換について紹介する．

第3章　数え上げの改良

この章では, 2.4節で紹介した上3ブロックに関する7つの同型変換の他に新たに発見した同型変換について紹介する.

3.1　3×2×3交換による変換

3×2×3交換というものを定義する. この交換はそれぞれの列に対しての交換を行うのみであるので, 解盤面の上3ブロックを考えたとき, 交換の前と後とでそれを含む解盤面の数は変化しない.

定義 3.1　以下の関数で実行される変換を3×2×3 交換と呼ぶ.

3x2x3[$y1, y2, y3, y4, y5, y6$]

input: 標準形の解盤面

output: 標準形の解盤面

1. 入力された解盤面について, 2から5を実行し, 盤面を出力
2. 第1ブロックについて, 第$y1$行以外の行同士をそれぞれ入れ替える
3. 第2ブロックについて, 第$y2$行以外の行同士をそれぞれ入れ替える
4. 第3ブロックについて, 第$y3$行以外の行同士をそれぞれ入れ替える

ただし, この変換を実行した後も解盤面が出力されるためには
$\{a_{y11}, a_{y12}, a_{y13}\} = \{a_{y24}, a_{y25}, a_{y26}\} = \{a_{y37}, a_{y38}, a_{y39}\}$ である必要がある.

図3.1が3×2×3 交換の例である.

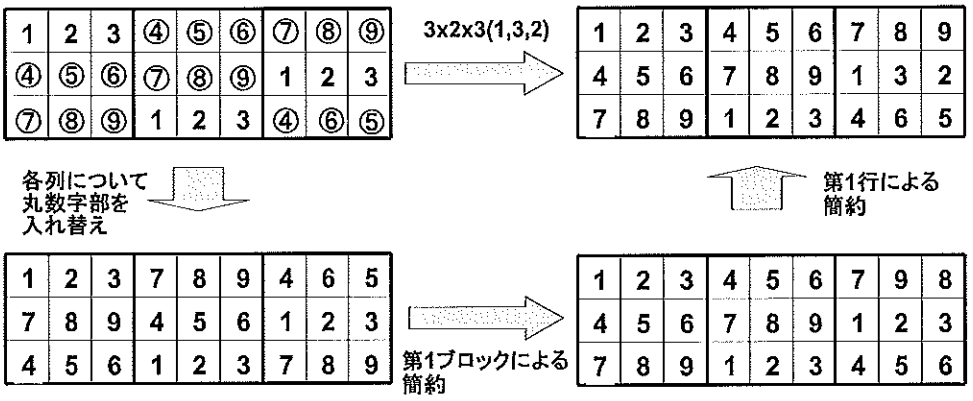

図 3.1: 3×2×3 交換の例

まとめと課題

2.4 節で紹介した 7 つの変換とこの変換を組み合わせることにより，調べるべきパターンの数は 36288 から 62 にまで落ちる．

ただ，前章の最後でも述べたが，このパターンの数は 44 まで落ちる可能性がある (44 未満となることはない)．44 まで落ちる同型変換があるかどうかはまだ分かっていない．

第4章　解の生成と番号付け

本章ではコンピュータにおいて，任意の解盤面を生成する方法について提案する．まず，ある非負整数から解盤面を出力する関数 index_to_sudoku と，ある解盤面から非負整数を出力する関数 sudoku_to_index を定義する．

定義 4.1　関数 index_to_sudoku を以下のように定義する

input: $n \in \mathbf{N}(0 \leq n < N_0)$

output: $s \in \mathrm{SUDOKU}$

但し，$\forall a \forall b \in \mathrm{SUDOKU}$ について，

$a \neq b \Rightarrow \mathrm{index\_to\_sudoku}(a) \neq \mathrm{index\_to\_sudoku}(b)$

定義 4.2　関数 sudoku_to_index を以下のように定義する

input: $s \in \mathrm{SUDOKU}$

output: $n \in \mathbf{N}(0 \leq n < N_0)$

但し，$\mathrm{sudoku\_to\_index}(\mathrm{index\_to\_sudoku}(n)) = n$

系 4.3　$\mathrm{sudoku\_to\_index}^{-1} = \mathrm{index\_to\_sudoku}$

4.1　解盤面の生成

ここでは index_to_sudoku の実現方法において述べる．先に基本的な手法を紹介し，次にその手法の改良法について提案していく．

4.1.1　基本的な手法

まず基本的な手法を紹介する．アルゴリズムは n 個の解盤面を順番に生成していき，最後に生成された数独を出力するというものである．解盤面の生成にはバックトラック法を用いる．

バックトラック法による解生成を説明する．最初は何も埋められていない解盤面が与えられ，次に $a_{11}, \cdots, a_{19}, a_{21}, \cdots, a_{99}$ の順に盤面を埋めていく．盤面を埋める際は数独の制約条件に矛盾しない数字を埋めていく．途中で埋める数字がなくなれば1つ前に戻って (バックトラックをして) 別の数字を埋める．この操作を盤面が全て埋まるまで繰り

返していく．盤面が埋まるとそれが解盤面となっている．因みにこのアルゴリズムでは，$n-1$回目までは解盤面が完成してもバックトラックをし，次の解盤面を生成する．これによりn番目の解盤面の生成を実現している．

以下はindex_to_sudokuプログラムの擬似コードである．

```
//データ: count,max_countは静的な整数
function index_to_sudoku(index)
//入力: indexは非負整数
  count := 0
  max_count := index
  search(1,1)
end-function

//データ: tableは非負整数の静的2次元配列, サイズは9*9
function search(i, j)
//入力: i,j,count,max_countは非負整数
  if i = 10 then
    count := count + 1
  end-if
  if count = max_count then
    結果を出力, プログラムの終了
  end-if
  for k = 1...9
    if table[i][j]にkを代入可能 then
      table[i][j] := k
      if j = 9 then
        search(1,j+1)
      else
        search(i+1,j)
      end-if
    end-if
  end-for
end-function
```

この方法の問題点は，実行時間である．家庭用PCにおいて1億$(=1.0\times10^8)$個の生成におよそ300秒の時間がかかる．本節のアルゴリズムでは入力にN_0が与えられたとき，$300\times N_0/(1.0\times10^8)\approx2.0\times10^{16}$(秒)$\approx6.3\times10^8$(年) というとてつもない時間がかかる．

次節以降では index_to_sudoku プログラムの最悪時実行時間をいかにして短くしていくか, ということについて考えていく. 因みに本節のアルゴリズムにおいては入力に N_0 を与えたときが最も実行時間が悪くなる. このことは順番に解を生成していく, というアルゴリズムの特性から明らかである.

4.1.2 改良法 (1)

この節では index_to_sudoku の改良アルゴリズムを提案する. 2.3 節で紹介した標準化を用いて探索範囲を狭めている. つまり, 図 4.1(左) の解盤面を標準化すると図 4.1(右) の解盤面となる. 図 4.1(左) 以外にも標準化により図 4.1(右) となる解盤面は存在し, 2.3 節でも触れたが, 標準化により同一の解盤面となる解盤面の集合はそれぞれ $9! \times 72^2 = 1881169920$ 個ある. 即ち最悪時実行時間が 1881169920 分の 1 程度になる, と見積もることができる.

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

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

図 4.1: 標準化の例

index_to_sudoku 改良アルゴリズム (1)

1. index に対応する第 1 ブロックの番号を求める
2. index に対応する第 1 行の番号を求める
3. index に対応する第 1 列の番号を求める
4. 標準形盤面について解盤面を探索
5. 4 で求まった解盤面について 1,2,3 で求めた値をもとに解盤面を変換
6. 5 の結果を出力

以下, アルゴリズムの各ステップに関する説明である.

第1ブロックの番号を求める

第1ブロックの形のパターンは9!通り存在するので, 第1ブロックが同じ形の解盤面はそれぞれ $N_0/9! = N_1$ ずつ存在する. **index_to_sudoku** の入力として非負整数 n が与えられるとすると, $0 \leq n \leq N_1 - 1$ の N_1 個の解盤面は第1ブロックがそれぞれ同じ形の盤面を出力し, $N_1 \leq n \leq 2N_1 - 1$ の N_1 個の解盤面は第1ブロックがそれぞれ同じ形の盤面を出力し, といったように決めていく. 以上の理由から, 第1ブロックの番号を $\lfloor n/N_1 \rfloor$ と定めることができる. まとめると, 第1ブロックの番号が同じ解盤面同士は第1ブロックの形が同じであり, この番号は0から $9! - 1$ までの値を取り得る.

以下詳しい番号付けの方法を述べる. 第1ブロックの各数字を $1, \cdots, 9$ の順列だと考える. 例えば図4.1(左) なら $3,5,8,9,2,6,7,1,4$ という順列と考える. 更にこの順列に, 下表のように番号をつけることを考える. インデックスに対応する順列が求める第1ブロックの形である. 例えば, 第1ブロックの番号が100000の盤面のうち1つが図4.1(左) である.

index	順列
0	1,2,3,4,5,6,7,8,9
1	1,2,3,4,5,6,7,9,8
⋮	⋮
362779	9,8,7,6,5,4,3,2,1

第1行, 第1列の番号を求める

第1ブロックによる標準化が行われた解盤面を考える. 第1行, 第1列の並び方のパターンはそれぞれ $(3!)^2 \times 2 = 72$ 通り存在する. 即ち第1行, 第1列の並び方のパターンはあわせて $72^2 = 5184$ 通り存在し, 第1行, 第1列の並び方が同じ解盤面はそれぞれ $N_1/5184 = N_2$ ずつ存在する. **index_to_sudoku** の入力として非負整数 n が与えられるとすると, $\lfloor (n\%N_1)/N_2 \rfloor$ の値が第1行, 第1列の並び方を示しているとみなせる. 因みに, $0 \leq (n\%N_1)/N_2 < 5184$ である. この値が第1行, 第1列の並び方を示すわけだが, $\lfloor \lfloor (n\%N_1)/N_2 \rfloor /72 \rfloor$ が第1行の並び方を, $\lfloor (n\%N_1)/N_2 \rfloor \%72$ が第1列の並び方をそれぞれ示しているとみなすことにする. ただし % 演算子は剰余演算子である.

この0から71の数がどの並び方を表しているのかを以下で示す. まず, 第1行に関して説明する. 第2, 第3ブロックのうち, 第1行に4を含むブロックの第1行の数字の集合を $\{a,b,c\}$ $(a < b < c)$, そうでない方のブロックの第1行の数字の集合を $\{d,e,f\}$ $(d < e < f)$ としたときに, その順列の状態に次項の表のように番号をつけることを考える. ($\{a,b,c,d,e,f\} = \{4,5,6,7,8,9\}, a = 4$ である) 例えば第1行の番号が10の盤面の1つが図4.1(左) である. 図4.1(左) に第1ブロックによる標準化を施すと第1行は1,2,3,4,6,5,9,7,8となり, この順列は次項の表でインデックスが10となっている.

第1列に関しても同様で, 例えば第1列の形が60の盤面が図4.1(左)である. 図4.1(左)に第1ブロックによる標準化を施すと第1列は1,4,7,9,3,6,2,5,8となり, この順列は次項の表でインデックスが60となる.

index	順列
0	a,b,c,d,e,f
1	a,b,c,d,f,e
⋮	⋮
5	a,b,c,f,e,d
6	a,c,b,d,e,f
⋮	⋮
35	c,b,a,f,e,d
36	d,e,f,a,b,c
⋮	⋮
71	f,e,d,c,b,a

解盤面の探索

標準化された解盤面を順に生成していく. このアルゴリズムはベースのアルゴリズムとほぼ同一のものである. ただ第1ブロック, 第1行, 第1列に関しては標準形の一部となっているのものしか生成しないように注意する.

index_to_sudokuの入力として非負整数nが与えられると, $(n\%N_1)\%N_2$番目の解盤面を探索によって見つける.

解盤面の変換

解盤面探索で得られた盤面をもとに, まず第1行, 第1列の番号にあわせて盤面を変換する. 即ち, 第1行が表の$\lfloor\lfloor(n\%N_1)/N_2\rfloor/72\rfloor$番目の順列に, 第1列が表の$\lfloor(n\%N_1)/N_2\rfloor\%72$番目の順列になるように解盤面に変換を施す.

次に, 第1ブロックの形にあわせて盤面を変換する. 第1ブロックを上表の$\lfloor n/N_1\rfloor$番目の順列に変換する. (それに従いその他のブロックも変換する.) この結果が**index_to_sudoku**の出力となる.

実行時間の解析

最も時間のかかるポイントは明らかに探索の部分である. この改良法では探索範囲が4.1.1節の方法に比べ1881169920分の1となっている. 即ち最悪時実行時間もほぼ

1881169920分の1となっていてこれはおおよそ10^7(秒) $\approx$ 123(日)である. 現実的な実行時間に近づいたが, まだこれでは不十分である.

擬似コード

このアルゴリズムの擬似コードを紹介する.

```
//データ: A,_A,B,C,count,max_count は非負整数 (静的変数)
function index_to_sudoku(index)
//入力: index は非負整数
  A = index / N1
  _A = index % N1
  B = (_A / N2) / 72
  C = (_A / N2) % 72
  count = 0
  max_count := _A % N2
  search(1,1,1,_A % N2)
end-function

function search(i, j, count, max_count)
//入力: i,j は非負整数
//先に紹介した search と内容はほぼ同一だが,
//出力の際に A,B,C をもとに盤面を変換した上で出力
end-function
```

4.1.3 改良法 (2)

この節では index_to_sudoku の更なる改良アルゴリズムを提案する. 2.4.1節で議論した上3ブロックのパターンそれぞれについて予め上3ブロックを固定したときの解盤面の個数を保存しておくことにより探索範囲を狭めるというものである.

index_to_sudoku 改良アルゴリズム (2)

1. index に対応する第1ブロックの番号を求める
2. index に対応する第1行の番号を求める
3. index に対応する第1列の番号を求める
4. index に対応する上3ブロックの番号を求める
5. 標準形盤面について解盤面を探索

6. 5 で求まった解盤面について **1,2,3** で求めた値をもとに解盤面を変換
7. 6 の結果を出力

以下, アルゴリズムの詳細な説明であるがステップ **1,2,3** については前節と同じであるため省略する.

上 3 ブロックの番号を求める

上 3 ブロックの形が 36288 のパターンのどれに属するかを調べる. 予め下表のように上 3 ブロックの形とそのブロックを含み, かつ標準化された盤面の個数がわかっているとする. また, 上 3 ブロックの表現として第 2, 第 3 ブロックの各行を書き下す形で行っている. 第 1 ブロックは標準化されていることにより常に同じ数字が埋まっている.

上 3 ブロック	サイズ
456789,789123,123456	108374976
456789,789123,123465	102543168
456789,789123,123546	102543168
⋮	⋮
489567,732981,651432	97477096

表 4.1: 探索数のテーブル

4.1.2 節の方法では, **index_to_sudoku** の入力として非負整数 n が与えられるとすると, 標準形盤面のうち, $(n\%N_1)\%N_2$ 番目の盤面を探索していた. 例えば $(n\%N_1)\%N_2 = 108374977$ である場合, 標準形の盤面のうち 108374977 番目の盤面を探索していた. しかし今回の方法では, 例えば $(n\%N_1)\%N_2 = 108374977$ である場合, 上 3 ブロックが 456789,789123,123456 である場合の探索を行う代わりに, 108374977 − 108374976 の計算を行う. (上 3 ブロックが 456789,789123,123456 である盤面の探索をしたことにする.) そして上 3 ブロックが 456789,789123,123465 である場合の 108374977 − 108374976 = 1 番目の盤面を探索する. 即ち探索すべき番目の盤面が上表の 36288 の項目のどれに該当するかを調べ, 該当箇所がわかればそこで初めて探索する, という方法をとる. どれに該当するかを調べる方法は, 先の例のように, $(n\%N_1)\%N_2$ から上表のサイズを上から順に引いていき, 初めて負となる場所が該当箇所である.

解盤面の探索

4.1.2節で説明した探索と基本的には同じ方法. ただし上3ブロックは固定されており, かつ探索範囲は標準形の盤面のみに絞っている.

実行時間の解析

改良法 (1) に対して最大探索範囲が36288分の1となっている. 即ち最悪時実行時間もおおよそ36288分の1となっており, これは $10^7/36288 \approx 275$(秒) である.

4.1.4 改良法 (3)

この節では, 前節の改良法に対し更に実行時間を短くする方法を提案する. アルゴリズムの流れ, 詳細はは前節の改良アルゴリズム (2) の場合とほぼ同じであるが, ステップ5において探索すべき全ての盤面を予め保存しておくという点が異なる.

探索盤面の保存

探索すべき盤面を予めファイルに保存しておく. 上3ブロックそれぞれの場合に関して, 探索で生成され得る盤面の値をファイルに保存しておく. ファイルの内容は, 例えば $a_{41}, \cdots, a_{49}, a_{51}, \cdots, a_{59}, \cdots, a_{91}, \cdots, a_{99}$ の $6 \times 9 = 54$ 個の数の並びとする. ただし $a_{41}, a_{51}, a_{61}, a_{71}, a_{81}, a_{91}$ に関しては標準形, 即ち" 昇順" に並んでいる.

以下は盤面の値を保存したファイルの例 (上3ブロックが456789,789123,123456型の場合) である. それぞれの行に関して54個の数が記されている. このようなファイルを上3ブロックのパターンの数 (36288個) 用意する.

盤面列挙ファイル (最初の10行)

```
214365897365897214897214365531642978642978531978531642
214365897365897214897214365531642978648971532972538641
214365897365897214897214365531642978672938541948571632
214365897365897214897214365531642978678931542942578631
214365897365897214897214365531642978942578631678931542
214365897365897214897214365531642978948571632672938541
214365897365897214897214365531642978972538641648971532
214365897365897214897214365531642978978531642642978531
214365897365897214897214365531648972642971538978532641
214365897365897214897214365531648972648972531972531648
```

この手法を用いれば探索に要する時間は非常に少なくなる. なぜなら, 前節の方法のように上3ブロックを固定して n 番目の解盤面を探索して見つける代わりに, 上3ブロックの形から該当ファイルを見つけ出しそこから n 行目の解盤面を見つければよいからである.

問題点はファイルサイズである. ファイルはおよそ 10^8 行からなり, 各ファイルサイズは $55(\text{Byte}) \times 10^8 \approx 5\text{GB}$ となる. 即ち総ファイルサイズは $5(\text{GB}) \times 36288 \approx 180(\text{TB})$ となり, 現在の家庭用コンピュータ環境を考えると現実的な方法とは言い難い.

列挙ファイルの軽量化 (1)

先に示した列挙ファイルで不必要な部分を削除する. まず, $a_{91}, \cdots, a_{99}$ の部分は不要である. なぜなら, 各 $j = 1, \cdots, 9$ について a_{9j} の値は $a_{1j}, \cdots, a_{8j}$ の値から一意に定まるからである.

また, 補題4.4より第8, 第9ブロックのうち第8, 第9行の値は不要である. 先の議論と合わせると $a_{84}, \cdots, a_{89}, a_{91}, \cdots, a_{99}$ の部分は不要であることがわかる. ただ, こうすると列挙ファイル内に同一の行が出てくる場合がある. (2パターン生成されることがあるため.) 前の行と同一の行が出てきた場合は, そこから解の探索で生成される2番目の盤面が該当行の盤面と決めておけば一意性に問題はない.

補題 4.4 第8, 第9ブロックの第8, 第9行が未確定である解盤面について, そこから生成しうる解盤面の個数は高々2個である.

証明 第7ブロックおよび第7行まで確定している (マスが埋まっている) 解盤面について考える. このとき, 第7ブロックの数字を図のように固定しても一般性を失わない.

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}		

図 4.2: 下3ブロックの一例 (1)

まず, 図4.2のように第7ブロックの第8行が第8ブロックの第7行にくる場合を考える. このとき, 図4.2のように解盤面は一意に定まる. (上の6ブロックの値は既に決まっているため.)

また, その他の場合, 即ち図4.3のような場合には解盤面が一意に決まるとはいえない. 例えば, $\{8,9,a\}, \{7,b,c\}, \{6,b,c\}, \{4,5,a\}$ の代わりに $\{8,9,c\}, \{7,b,a\}, \{6,b,a\}, \{4,5,c\}$

1	2	3	4	5	7	6	8	9
4	5	6	{ 8, 9, a }			{ 7, b, c }		
7	8	9	{ 6, b, c }			{ 4, 5, a }		

図 4.3: 下 3 ブロックの一例 (2)

という組み合わせでも成立しうる場合が存在する. (存在するかどうかは上 6 ブロックの値に依存する.) ただ, 定まる解盤面は多くてもこの 2 つのパターンに限られる. (上の 6 ブロックの値は既に決まっているため.) □

更に, $a_{63}, a_{66}, a_{49}, \cdots, a_{79}$ の値も $a_{91}, \cdots, a_{99}$ の場合と同様にその他のマスから明らかに一意に定まるので不要である.

以上の議論から 54 のマスのうち $15 + 6 = 21$ 個の値については不要となった. 即ち列挙ファイルには 1 行あたり 33 個で済み, 必要なデータ量はそれだけ少なくなる. (それでもまだ現実的とは言いがたいが.)

列挙ファイルの軽量化 (2)

列挙ファイルの例を見ればわかるが, それぞれの行について共通する部分が非常に多い. だから, それぞれの行は前の行との差分のみを表示すればデータサイズの削減が図れる. 実際に先に表示したデータファイルにこの操作を施すと以下のようになる.

盤面列挙ファイル, 差分版 (最初の 10 行)

```
21436589736589721489721436553164297864297853197853164 2
8971532972538641
72938541948571632
8931542942578631
942578631678931542
8571632672938541
72538641648971532
8531642642978531
8972642971538978532641
8972531972531648
```

この手法を用いればファイルサイズはおよそ 65% 削減できる. 列挙ファイルの軽量化 (1) とあわせれば (実験的には) およそ 87% 削減でき, 必要ファイルサイズは 23TB 程と

なる.

列挙ファイルの軽量化 (3)

2.4 節の上 3 ブロックに対する同型変換を用いて必要ファイルサイズを減らす方法を提案する.

例えば, 図 2.14 によると解盤面全体のうち, 上 3 ブロックが 456789,789123,123465 となっている盤面と 456789,789132,123456 となっている盤面の個数は等しいことがわかる (上 3 ブロックの表記法は 4.1.3 節を参照のこと). だから, 上記 2 つのパターンについては番号付けの際にどちらか 1 つのファイルが存在すれば事足りる.

つまり, 36288 個ある標準化された上 3 ブロックのパターンのうち (2.4 節及び 3 章で求めた)62 個についてのファイルを持っていれば残りの 36226 個のファイルについては持つ必要がなくなる.

該当ファイルが存在しないパターンについては, 7 つの変換の組み合わせで変換可能なパターンの該当ファイルのうちで存在するファイルを代わりに探索する. このときの変換を g とする (このようなファイルは必ず 1 つ存在する). 探索で得られた解盤面に変換 g^{-1} を施したものを探索結果とすればよい.

各パターンがどのように変換すれば列挙ファイルが存在するパターンへ変換できるかは, 予め調査し調査結果をファイルに保存するなどしておく必要がある. ただし, それぞれの変換の詳細を見てわかるとおり, 7 つの変換のうちどの変換も第 4 行以降が変換の影響を受けるのは第 1 ブロックに対する同型変換と第 1 行に対する同型変換の際のみである. よって任意の変換の組み合わせによる変換も数字の付け替えと行の入れ替えで表すことができる. 数字の付け替え方は $9! = 362880$ 通り, 行の入れ替え方は 72 通りあるので, 任意の変換は 1 から $9! \times 72 = 26127360$ の整数で表すことができる. 変換方法をファイルに保存しても, ファイルのサイズは高々36288(行) × 8(行あたり最大文字数) $\approx$ 300000(Byte) $\approx$ 300(MB) であり, それぞれの列挙ファイルの大きさに対して無視できる程度の大きさである.

この手法を用いると必要ファイルサイズがおよそ 62/36288 となり, 軽量化 (1),(2) と合わせることにより必要ファイルサイズはおよそ 23(TB) × 62/36288 $\approx$ 39(GB) となる. これは家庭用 PC の HDD に収まるレベルであり, 現実的といえる. 各ファイルについて bzip2 圧縮などの技術を用いればファイルサイズはより節約でき, 4GB 程度に収まる.

4.2 解に対する番号付け

次に, **sudoku_to_index** の実現方法において述べる. 先に基本的な手法を紹介し, 次にその手法の改良法について提案していく.

4.2.1 基本的な手法

基本的なアルゴリズムは，入力盤面と一致するまで解盤面を順番に生成していき，一致したらそれが何番目に生成された盤面かを出力するというものである．解盤面の生成にはバックトラック法を用いる．

この手法はindex_to_sudokuの場合と同様，入力によっては膨大な時間がかかる．以下このアルゴリズムの改良法を示していく．

4.2.2 改良法 (1)

この節ではsudoku_to_indexの改良アルゴリズムを提案する．このアルゴリズムは4.1.1節のindex_to_sudoku改良アルゴリズム (1) に基づく．

sudoku_to_index改良アルゴリズム (1)

1. 第1ブロックの番号を求め，(入力) 盤面に第1ブロックによる標準化を施す
2. 第1行の番号を求め，盤面に第1行による標準化を施す
3. 第1列の番号を求め，盤面に第1列による標準化を施す
4. 盤面に対応するインデックスを求める
5. 4で求めたインデックスを1,2,3で求めた値をもとに変換
6. 5の結果を出力

ステップ1,2,3についてはindex_to_sudoku改良アルゴリズム (1) の場合とほぼ同じである．ステップ4については前節の基本的な手法と同じくバックトラック法を用いる．このとき，標準形の盤面のみを探索すればよいので最大探索範囲は基本的な手法の場合と比べて1881169920分の1程度になっている．ステップ5で求めるインデックスを計算する．1から4で計算された値をそれぞれA, B, C, Dとすると，計算される値は$A \times N_1 + (72 \times B + C) \times N_2 + D$となる．この結果はindex_to_sudoku改良アルゴリズム (1) の結果に対応している．

4.2.3 改良法 (2)

4.1.1節のindex_to_sudoku改良アルゴリズム (2) に基づく改良法を簡単に示す．

sudoku_to_index改良アルゴリズム (2)

1. 第1ブロックの番号を求め，(入力) 盤面に第1ブロックによる標準化を施す
2. 第1行の番号を求め，盤面に第1行による標準化を施す
3. 第1列の番号を求め，盤面に第1列による標準化を施す

4. 入力盤面に対する上3ブロックのパターンを求める
5. 盤面に対応するインデックスを求める
6. 5で求めたインデックスを1,2,3で求めた値をもとに変換
7. 6の結果を出力

ステップ4で上3ブロックのパターンを求めるというのは入力盤面について上3ブロックのパターンがどれなのかを調べるということである．つまり，4.1.3節で示した表から(上から順番に一致するかどうか調べていくことにより)該当箇所を見つける．この際，パターンが一致しないときはそのパターンのサイズを5の出力結果に加えていく．例えば上3ブロックが456789,789123,123546である場合，ステップ5の出力に108374976と102543168を加える．

この手法で4.2.2節の場合に比べて最大探索範囲を36288分の1程度にすることが可能である．

4.2.4 改良法(3)

この節では4.1.3節の場合と同様に盤面を列挙したファイルを用意する方法を考える．用意するファイルは4.1.3節で示したものと全く同一のものとする．

今回は該当盤面がどの行に存在するかという探索を行う必要がある．この際1行ずつ比較してもよいが，2分探索法を用いることができる．列挙ファイルを，常に数字を小さい順に当てはめていくバックトラック法で生成すれば，列挙ファイルは行に関して昇順となり，2分探索法を用いることができる．この方法を用いれば探索の際，1億行のファイルに対しても，比較回数は$\log 10^8 = 8 \times \log 10 \approx 26$回程度で済む．

軽量化に関しても4.1.3節で紹介したものと同じ技術を用いることができ，必要な列挙ファイルの総量を(非圧縮で)45GB程度に抑えることができる．

まとめ

以上index_to_sudokuおよびsudoku_to_indexのアルゴリズムを紹介した．実際にプログラムを作成し，*http://doorgod.org/sudoku/*で公開中である(2007/02/28現在)．現在実行時間は10秒程度であり，これは列挙ファイルの入力や圧縮ファイルの解凍がネックとなっている．ファイルの細分化の工夫や，非圧縮ファイルの用意などで実行時間を更に短くする余地はある．

第5章　おわりに

まとめ

一般に普及している 3×3 数独の解盤面に注目し，解盤面に番号付けを行うという問題を考えた．$n \times n$ 数独はNP完全であることが八登 [2] により証明されているが，$n = 3$ 程度ならコンピュータを用いて十分に早く解を得られる．ただ，解の総数は $n = 3$ の場合でも膨大であるため，本研究では解き方のアルゴリズムではなく解答そのものに注目した．解盤面に1から N_0 までの番号を重複なくもれなく付けたときに，解盤面から番号を高速に探し出す方法，及び番号から解盤面を高速に探し出す方法をそれぞれ提案し，実装した．3×3 数独の解盤面の総数は実際に Jarvis ら [1] により数え上げられていて，本研究の高速化については彼らが提案した方法を多く利用している．この方法は同型変換と呼ばれるもので，これにより解盤面を探索する際，その探索範囲を狭めることが可能となる．これらの方法を用いて，本研究では実際に解盤面と番号の対応付けを高速に行うプログラムを作成した．標準化，及び上3行に対するパターン数の保存を用いた方法だと平均実行時間は5分程度で必要ファイル容量は1MB程度となり，さらに探索されるべき盤面を予め保存しておくという方法では平均実行時間は10秒程度で必要ファイル容量は40GB(圧縮すれば4GB)程度となった．これらを実行するプログラムを *http://doorgod.org/sudoku/* で公開中である (2007/2/28現在)．

今後の課題

3.2節でも触れたが，2.4節で紹介した7つの同型変換，3.1節で紹介した1つの同型変換の他にもこの種の同型変換が存在する可能性がある．これら8つの変換は，36288個ある (標準化された) 上3行のパターンのうち，62個を調べればよいということを示している．しかし，上3行のパターンに対してそれぞれその形を含む解盤面の個数を数えると，その個数は44通りとなる．言い換えれば，表4.1の右欄の数字が44通りしかないということである．しかし実際に8つの変換を利用するとき，62個のパターンを調べる必要がある．他の同型変換が存在し，調べるべきパターンが61個以下 (ただし44個以上) になる可能性もある．そしてそのような同型変換が存在するのではないかと考えられる．

また，本論文では触れていないが「本質的に異なる解盤面」という考え方がある [8]. これは，数字の付け替え，反転，同ブロック内の行の交換等の変換を示し，これらの変換をどう組み合わせても変換できない解盤面同士を「本質的に異なる解盤面」と呼ぶというものであり，ここで出てくる変換は同型変換に非常に近い．ただ，この考え方では標準化の様に変換が綺麗に記述できないため番号付けには相当の工夫を要するだろう，という理由で今回の番号付けの実践にはこの考え方は用いていない．しかし，番号付けのアプローチとして「本質的に異なる解盤面」をベースに考えてみるのは興味深い．

最後に，今回定義した番号は解盤面と番号が 1 対 1 の対応になっていることは保障されるが，この番号付けは手法により変わる．例えば，4.1.2 節及び 4.2.2 節で紹介した方法を用いれば，膨大な時間がかかるが解盤面は昇順に (探索順に) 番号付けすることが可能である．一方，最終的なプログラム (公開しているもの) は 4.1.4 節及び 4.2.4 節の方法で実践しているが，これは昇順になってはいない．列挙ファイルの軽量化に伴う変換により順番が崩れてしまうからである．ただ，先に述べたが 3.2 節のような変換を更に発見し，36288 個ある (標準化された) 上 3 行のパターンのうち，44 個を調べればよいということになれば，この手法ではこれ以上の改良は存在しないので 4.1.4 節及び 4.2.4 節で示した方法をより確かな (改良されにくい) ものとする事ができる．

謝辞

最後に，本研究を進めるにあたり，１年間終始あたたかい御指導と厳しい意見を賜りました渡辺治教授に心から感謝を申し上げます．そして，本研究を進めるにあたり多くのアドバイスを頂いた河内亮周助手にも感謝の意を申し上げます．また，本研究の論文作成，発表にあたって貴重な時間を割いて面倒をみて下さいました山本直樹氏，牧野格三氏をはじめ研究室の先輩方，共に研究を進めるにあたり苦労を分かち合った安藤友則氏，坂口亮氏にも感謝いたします．

参考文献

[1] Bertram Felgenhauer and Frazer Jarvis. Enumerating possible Sudoku grids. *http://www.afjarvis.staff.shef.ac.uk/sudoku/sudoku.pdf*, 2005

[2] T. Yato and T. Seta. Complexity and completeness of finding another solution and its application to puzzles. *IEICE Trans. Fundamentals*, 1052-1060, 2003.

[3] C. J. Colbourn. The complexity of completing partial Latin squares. *Discrete Appl.Math.*, 25-30, 1984.

[4] S. F. Bammel,J. Rothstein. The number of 9×9 Latin squares. *Discrete Mathematics 11*, 93-95, 1975.

[5] J. A. Bate and G. H. J van Rees. The size of the smallest srong critical set in a Latin square. *Ars Combinatorica*, 73-83, 1999.

[6] H. Simonis. Sudoku as a Constraint Problem. *Fourth Int. Works. Modelling and Reformulating Constraint Satisfaction Problems*, 13-27, 2005.

[7] D. Eppstein. Nonrepetitive paths and cycles in graphs with application to Sudoku. *ACM Computing Research Repository cs.DS/0507053*, 2005.

[8] Ed Russell and Frazer Jarvis.Mathematics of Sudoku II. *http://www.afjarvis.staff.shef.ac.uk/sudoku/russell_jarvis_spec2-1.pdf*, 2005