筑波大学 情報学群 情報メディア創成学類

卒業研究論文

Origami Checkerboardパズルの最適解探索

大島 和輝

指導教員　三谷 純

2018 年 01 月

概要

折り紙の分野における未解決問題の 1 つに、Origami Checkerboard パズルの最適解の全列挙がある。このパズルの目的は、表裏 2 色の正方形の紙を折り、できるだけ短い手数で 3×3 のチェッカーボードパターンを作ることである。3×3 のパターンは全部で 50 種類あり、全パターンに対する手順は発見済みであったが最適解であるかどうかは明らかではなかった。本研究の目的は、コンピュータを用いた探索を行い全 50 パターンに対する最適解を列挙することがである。紙の状態と紙を折る操作をモデル化し、幅優先探索と深さ優先探索を組み合わせて総当り探索を行うプログラムを作成してスーパーコンピュータを用いて探索を行った。その結果、50 問全てに対する最適解の全列挙を達成した。また、探索の対象となった範囲で等間隔 45 度格子に沿った折り目で紙を折った際の独立した状態の個数の列挙も行った。

目 次

図 目 次

第1章　はじめに

折り紙の研究分野の一つに折りの幾何的構造をコンピュータで扱う計算折り紙(Computational Origami) という分野がある。計算折り紙の分野では折りに関する数学的性質を明らかにするための研究が行われている。この分野において、解が明らかでない折り紙の具体的な問題として Origami Checkerboard パズルの最適解の列挙がある。Origami Checkerboard パズルは表裏 2 色の正方形の紙から単純折りのみを使ってできるだけ短い手数でできるだけ大きい折り上がりになるように 3×3 のチェッカーボードパターンを作るパズルである。このパズルは 1990 年代から折り紙の研究者らが取り組んできた問題であり、全てのパターンに解となる手順が示されていたが、それらの解が最適解であるかどうかは不明であった。これを受け、本研究ではプログラムによって総当り探索を行い、Origami Checkerboard の各パターンに対する最適解を列挙することを目的とする。

この目的を達成するために、紙面上での折り線の位置や山折り谷折りのどちらか等の情報を用いて任意の細かさの等間隔 45 度格子に沿った折り線で紙を折る操作をモデル化し、幅優先探索と深さ優先探索を組み合わせて総当り探索を行うプログラムを作成した。また、プログラムの実行には JAIST の上原隆平教授の協力を得て、JAIST に設置されているスーパーコンピュータを利用した。

本論文では、まず計算折り紙やチェッカーボード折り紙に関する関連研究を紹介し、その後、Origami Checkerboard パズルの定義や紙の状態と紙を折る操作をコンピュータ上で表現するために設計したモデル、最適解列挙のための幅優先探索及び深さ優先探索の方法とそれら探索中における枝刈り等の高速化手法について説明する。最後に探索の結果とそれから得られる考察、及び残された課題や発展的な研究テーマについて述べる。

第2章　関連研究

本研究は計算折り紙の分野に属するが、計算対象となる折り紙は 45 度格子のチェッカボード折り紙である。ここでは、計算折り紙とチェッカボード折り紙のそれぞれについてこれまでに行われてきた研究をまとめる。

2.1　計算折り紙

計算折り紙の研究では主にコンピュータを利用して折り紙を設計したり、折り紙に関する問題を解くことを目的としている。

初期の折り紙設計においては、前川淳が設計した「悪魔」が代表的である。作品を構成する基本部品を展開図に配置することによって設計された。

Robert J. Lang が考案した tree 法 (circle/river 法とも呼ばれる)[1] は、目標とする形状の角の個数と位置に注目する。例えば、鳥は頭と両翼と尾の 4 つの角、虫であれば頭と 3 対の脚と尾の 8 つの角を持つと考える。この角の位置関係を木構造によって表し、その木構造をもとに展開図を生成する。tree 法における最適化は NP 困難であることが示された [2] が、ある程度の最適化であれば経験に基づいて行うことができる。

Erik D. Demaine らは、任意の多角形が接続された形状の平坦折り紙の展開図を単一の長方形の紙の上に作成する方法を示した [3]。この手法は紙の帯 (strip) を用いて所望の形状を折るのであるが、帯を用いるという性質上、折り始めの長方形の紙が細長い場合を除いて紙の利用効率が悪くなってしまうという問題がある。

舘知宏は 1 枚の紙から円板と同相の多面体モデルを折る手法である Origamizer[4, 5, 6] を発表した。最初の Origamizer はヒューリスティックな手法であり、実現可能な展開図が発見できず局所最適化に失敗する場合があるという問題があった。その後、水密 (watertight) な折り方という概念を導入して紙の境界をどこに配置すべきか定める手法を提案し、向き付け可能な多面多様体に対して実現可能な展開図を見つけることが保証された新たな Origamizer アルゴリズムが発表された。

また、曲面や曲線の折り紙を扱う手法及びそれを実装したものとして、三谷純の ORI-REVO[7, 8] と ORI-REF[9, 10] がある。ORI-REVO は回転体をベースとした形状を折るためのツールで、マウスによる操作または点列データのロードにより折り線を入力すると、その折り線を母線とするような紙で折れる回転体の近似形状の展開図が計算される。ORI-REF は可展面に対してユーザの入力をもとに鏡映反転を適用することで曲面を含む折り形状を出力する。David Haffman がこの鏡映反転による手法を用いた折り紙設計手法を使っていたことも知られてい

る [11]。

2.2 チェッカボード折り紙

チェッカボード模様のような折りあがりが等間隔の格子から構成される折り紙についての研究も行われているが、その数はあまり多くはない。

そもそも、チェッカーボードのパターン即ち市松模様を表裏 2 色の紙から折るテーマに関して最初に広まったものは、1993 年の Montroll の本に掲載された、8×8 のチェス盤の模様を折り紙で折る方法である。その後も 8×8 のチェッカボードパターンを折る方法は幾つか考案されており、知られているものとしては正方形格子のサイズが $64\times64, 40\times40, 36\times36, 32\times32$ のものがある。[12]

Erik D. Demaine らはシームレスなチェッカボードパターンをできるだけ小さいサイズの正方形格子から折る方法を考案し、$n\times n$ のチェッカボードパターンを半周長 $\frac{1}{2}n^2+O(n)$ の正方形の紙から折る方法を示した [12]。シームレスとは、折りあがりの時点でチェッカーボード模様を構成する各正方形の内部に折り目がない状態を指す。

また、厳密には折りあがりの形がチェッカボードになるわけではないが、平織り (tessellation) により生成される紙の各部の上下関係を利用してドットパターンを折る手法が山本陽平によって提案されている [13]。

第3章　Origami Checkerboardパズル

Origami Checkerboard パズルは表裏 2 色の正方形の紙を折って、3×3 の正方形で構成されるパターンを作るパズルである。このパズルは 1990 年代半ばごろからパズルや折り紙の分野で知られるようになった。パズルの対象となるパターンは 50 通りあり、現在それらすべてのパターンに解が見つかっている。

3.1　歴史

2.2 節「チェッカボード折り紙」で述べたとおり、1993 年の Montroll の本に掲載された 8×8 のチェス盤の模様を折り紙で折る方法がチェッカーボードのパターンを表裏 2 色の紙から折るテーマとして最初に広まった。

その後、1990 年代半ばに Serhiy Grabarchuk が図 3.1 に示す 3×3 のメッシュの市松模様を折る問題をパズル界に提起した [14]。

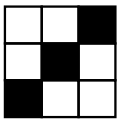


図 3.1: Grabarchuk が提起した市松模様。

この問題が提起されてから日本で図 3.1 以外の 50 通りのパターンが研究及び考察され、1998 年の第 3 回 G4G(Gathering for Gardner) での芦ヶ原伸之の発表を通じて世界に知られるようになった [15]。

現在では石井恵一郎により全 50 問が Grabarchuk 原作、佐々木節夫設問のパズルとして掲載 [16] され、公開された記録が更新されてきた。

3.2　パズルの定義

回転や鏡像を取り除くと、3×3 のパターンは表 3.1 に示すような、全 50 種類が列挙される。パターンと問題番号は [16] にまとめられているものに合わせた。図 3.1 で示した Grabarchuk が最初に提起したパターンは問題# 23 にあたる。

例えば、表 3.1 に示した問題番号# 2 のパターンが与えられた場合、図 3.2 の手順でそのパターンを折ることができる。

表 3.1: Origami Checkerboard パズルで与えられる 50 のパターン。

# 1	# 2	# 3	# 4	# 5	# 6	# 7	# 8	# 9	# 10
# 11	# 12	# 13	# 14	# 15	# 16	# 17	# 18	# 19	# 20
# 21	# 22	# 23	# 24	# 25	# 26	# 27	# 28	# 29	# 30
# 31	# 32	# 33	# 34	# 35	# 36	# 37	# 38	# 39	# 40
# 41	# 42	# 43	# 44	# 45	# 46	# 47	# 48	# 49	# 50

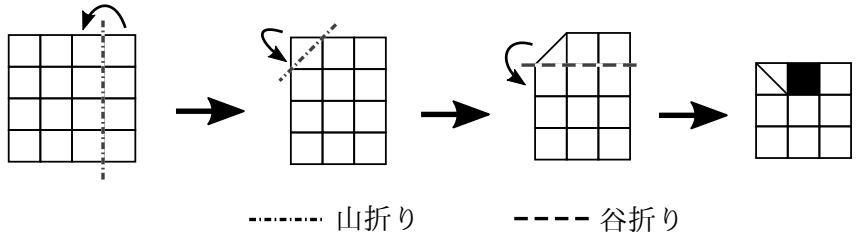


図 3.2: # 2 のパターンを折る手順の 1 つ。

当然 1 つのパターンに対して複数の手順が存在する場合が予想できるが、そういった場合には、折る回数がより短い手順の方が良い解であるとする。折る回数が同じ解が複数存在する場合は、同じ大きさの正方形の紙でそのパターンを折った時、最後の折りあがりが大きい方が良い解であるとする。よって、Origami Checkerboard パズルにおける最適解とは、最短であり、かつ、折りあがりのパターンのサイズが最大となる手順である。折る回数と最後の折りあがりのサイズが同じ手順はそのパターンに対する別解となる。

紙や折る操作についての条件は以下のように定められている。要約すると、紙の厚さは 0 で、180 度の単純折りのみが認められているということである。

- 紙の厚さは 0 とする。
- 1 回の折りでできる折り線は、その操作の時点で 1 つの線分でなくてはならない。
- 折る面は平面を保ち、その面で 180 度折る。
- 紙に重なりがあった場合、同時に任意の枚数折ってよい。

3.3 既知の解

Origami Checkerboard パズルの各パターンとそれらに対する既知の解が石井の Web サイト [16] にまとめられているのは前述のとおりである。石井の Web サイトによれば、既知の解の更新は 2004 年と 2005 年に 2 回ずつあり、また、2017 年には 8 回もの更新があった。

表 3.2 に 2017 年 12 月時点での既知の解の格子サイズと折りの手数をまとめる。最小の格子サイズと手数はそれぞれ 4×4 と 2 手 (問題番号 # 4, # 12, # 32) で、最大の格子サイズと手数はそれぞれ 9×9 と 6 手 (問題番号 # 45, # 50) である。また、石井の Web サイト [16] では、格子サイズと手数は同じであるが手順の異なる別解はまとめられておらず、1 つのパターンに対して 1 つの手順が挙げられている。

3.4 別解の定義

本研究における「別解」とは、格子サイズと折り手順のステップ数は同じであるが、折りあがりの紙の状態が異なる解を指す。

折り手順中に図 3.3 のような折る順番を入れ替えることが可能な独立した折りが複数含まれている場合、それらを入れ替えると、折りの手順自体は異なったものになる。しかし、それだけでは折りあがりの紙の状態は変化しないため、別解としてはカウントしていない。

表 3.2: 既知の解の格子サイズと折りの手数

問題番号	手数	格子サイズ	問題番号	手数	格子サイズ
1	2	5×5	26	6	5×5
2	3	4×4	27	3	5×5
3	4	5×5	28	3	4×4
4	2	4×4	29	4	5×5
5	3	5×5	30	3	5×5
6	4	5×5	31	4	5×5
7	5	5×5	32	2	4×4
8	4	4×4	33	5	5×5
9	5	4×4	34	5	5×5
10	3	5×5	35	4	5×5
11	4	5×5	36	5	5×5
12	2	4×4	37	4	5×5
13	4	4×4	38	5	5×5
14	3	5×5	39	6	5×5
15	4	4×4	40	6	5×5
16	4	5×5	41	3	4×4
17	4	5×5	42	4	5×5
18	4	5×5	43	5	6×6
19	5	9×9	44	4	5×5
20	5	6×6	45	6	9×9
21	5	9×9	46	5	6×6
22	4	7×7	47	5	9×9
23	5	5×5	48	6	5×5
24	4	5×5	49	5	5×5
25	5	9×9	50	6	9×9

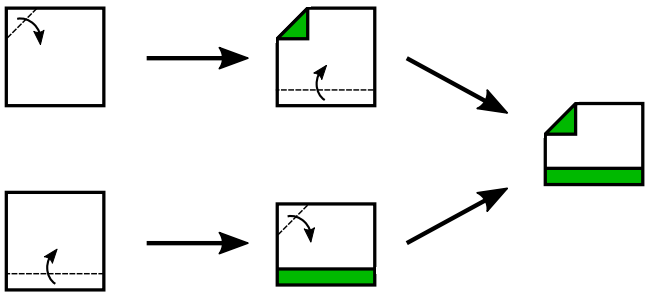


図 3.3: 折りの手順は異なるが折った後の状態が同じになる例。

第4章　探索手法

本研究では折りをシミュレートするプログラムによって Origami Checkerboard の最適解探索に取り組んだ。

探索には基本的に幅優先探索を使っているが幅優先探索をするためには紙の状態を保存するのに十分な量のメモリが必要となるため、メモリ節約のために探索の途中から深さ優先探索に切り替えることもできるようにした。

この章では、作成したプログラムの設計と実装を中心に Origami Checkerboard の探索手法についてまとめる。

4.1　探索時の条件と制約

本研究では、3 章「Origami Checkerboard パズル」で述べた Origami Checkerboard パズルそのもののルールに加えて、探索において以下のような 2 つの制限を設けた。

- 折り目は等間隔 45 度格子に沿うものとする。
- 探索範囲は、格子サイズ 4×4 から 9×9、手数は 6 手までとする。

まず、1 つ目の「折り目は等間隔 45 度格子に沿うものとする」という制限について説明する。本来 3.2 節「パズルの定義」で説明した Origami Checkerboard のルールでは、折り線の角度についての制限はないが、等間隔 45 度格子に沿わない折り目で紙を折ってしまうと、最終的な目標である 3×3 のパターンを作るのは難しいと考えられるためこの制限を導入した。

また、折り目が等間隔 45 度格子に沿うものとしても、格子サイズの下限と上限や、探索の最大手数はルールとは関係ない。しかし、格子サイズの範囲や探索の最大手数に制限を設けずに探索を行うのは現実的でない。そこで、3.3 節「既知の解」で述べた既知の解の最小及び最大格子サイズと最大手数から、探索範囲を 2 つ目の制限のように定めた。

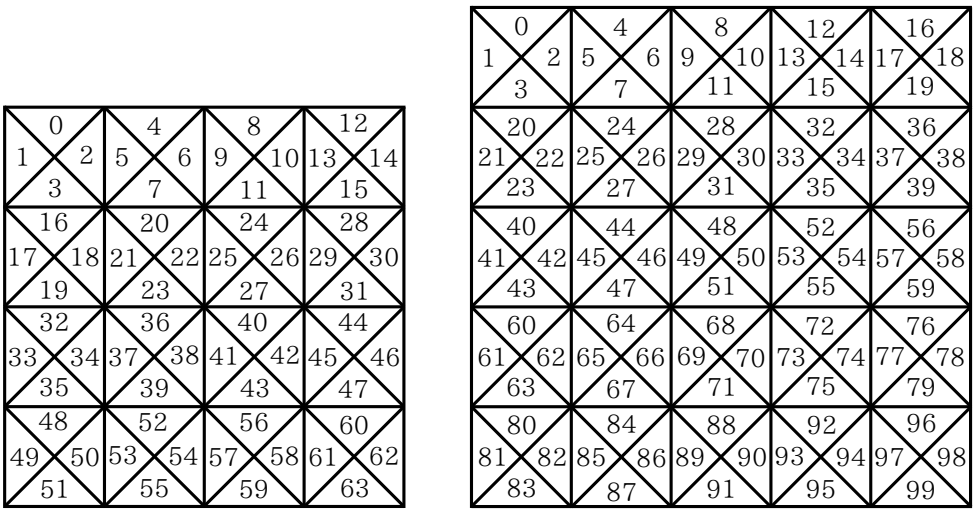
4.2　データ構造

プログラムによって折りを計算するためには、紙の状態を保持及び操作する必要がある。今回作成したプログラムでは、配列を用いて紙の状態を保存することとした。また、折り操作についてもプログラム内で表現する必要があるが、5 つのパラメータを用いて折りをデータ化した。

紙の状態を保持するデータ構造については 4.2.1 項「紙の状態の保持」で、折り操作の 5 つのパラメータについては 4.2.2 項「折りのパラメタライズ」でそれぞれ述べる。

4.2.1 紙の状態の保持

作成したプログラムでは、折り線は正方形の紙を分割する等間隔 45 度格子に沿っているものとして扱う。正方形をあるサイズの 45 度格子で分割すると、その正方形は複数の三角形で構成されることとなる。そこで、図 4.1 のようにこの三角形要素の 1 つ 1 つに通し番号を振り、配列に格納し、紙の状態を保持する。折りの操作は配列内でのデータの移動として表現される。



(a) 4 × 4 の 45 度格子　　(b) 5 × 5 の 45 度格子

図 4.1: 45 度格子の分割と通し番号の振り方の実例。左上から右へ進みながら右下まで全ての三角形要素に 0 から順に通し番号を振っていく。

三角形要素のデータを格納する配列は、各三角形要素が空間上のどの位置に存在するかを表す。そのため、配列の各要素は 45 度格子で分割された平面空間を構成する各三角形に対応している。

プログラムでは、この配列は (紙を構成する三角形要素の数 ×9) の大きさの 2 次元配列で表現されている。正方形の紙を 4 × 4 の格子に分割する場合では図 4.2 のようになる。中心の赤い部分に紙が配置されていることになるようにデータを初期化する。紙を折る処理を行うと、その結果、紙の一部分が赤い領域を飛び出す可能性がある。そのため、初期位置を取り囲むように紙と同サイズの配列領域を確保し、折られた紙の一部が飛び出した場合にもデータを保持できるようにしている。また、折る処理を行って紙が初期位置を飛び出た場合には、初期位置に収まるよう、配列内でデータを移動させる。2 次元配列となっているのは、「どの

正方形格子のマスか」と「正方形内の 4 つの三角形のどれか」という 2 つのパラメータで 45 度格子の各三角形の位置を表すようにしたためである。

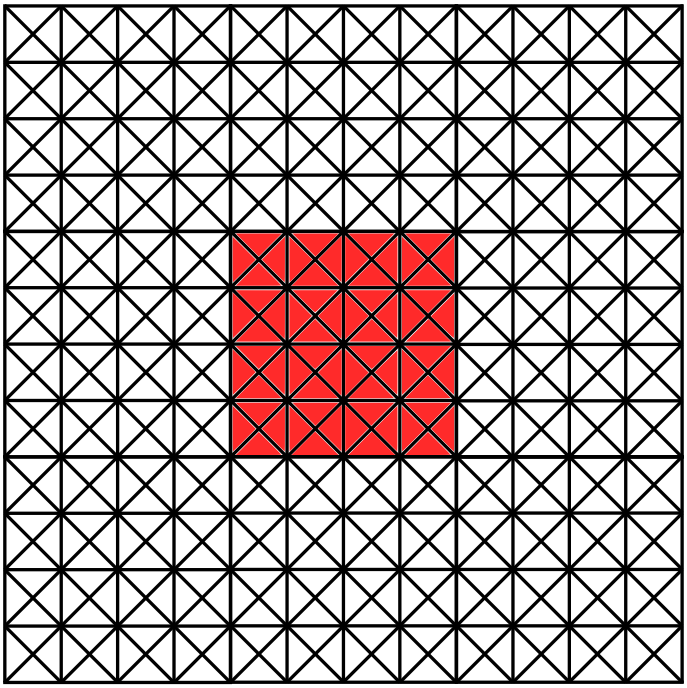


図 4.2: 4 × 4 の格子に分割された紙のデータを保持するための配列の概念図。

4.2.2 折りのパラメタライズ

ある折り方で紙を折る操作を実現するためには、折りをプログラムで表現する必要がある。折りを決定するのに必要なパラメータは以下の 5 つである。

1. 折りの方向 (水平、垂直、右斜め、左斜め)
2. 折りの位置
3. 山折りか谷折りか
4. 折り線のどちら側の紙を動かすか
5. 紙が重なっている場合、どこで紙を折るか

これらのパラメータのうち、1 から 4 は空間を分割する 45 度格子上の紙の位置が明らかであれば紙の状態によらず指定することが可能である。しかし、1 から 4 が同じであっても、5 つ目の「紙が重なっている場合、どこで紙を折るか」は紙の状態によって異なるため、これ

を適切に指定するためのアルゴリズムが必要である。このアルゴリズムは 4.3 節「紙を折るアルゴリズム」で述べる。

また、折りのパラメータの値を使って折り方を表記する場合は以下に示す方法を用いる。1 つの折り方は **[アルファベット 3 文字][数字],[数字]** という形式で表される。表記の例を図 4.3 に、表記に使われる記号の一覧とその説明を表 4.1 にそれぞれ示す。また、動かす側と折りの位置の指定は具体的には図 4.4 と図 4.5 に示すようになっている。

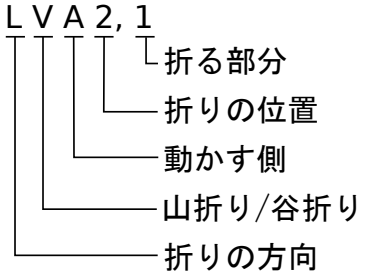


図 4.3: 折り方の表記の例。

表 4.1: 折り方の表記で使う記号一覧。

項目	記号	意味
折りの方向	H	折り線は水平。
	V	折り線は垂直。
	U	折り線は右上から左下への斜めの線分。
	L	折り線は右下から左上への斜めの線分。
山折り/谷折り	M	山折りで折る。
	V	谷折りで折る。
動かす側 (図 4.4)	A	折る時に A の側を移動させる。
	B	折る時に B の側を移動させる。
折りの位置	数字	折りの位置を表す数字 (図 4.5)。
折る部分	数字	動かす部分に振られた通し番号。

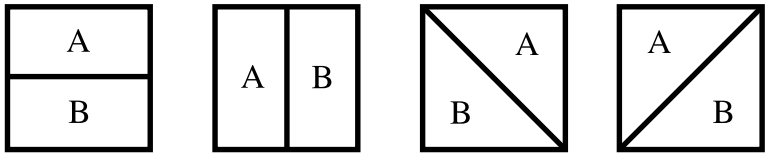


図 4.4: 折り線と動かす側の指定。

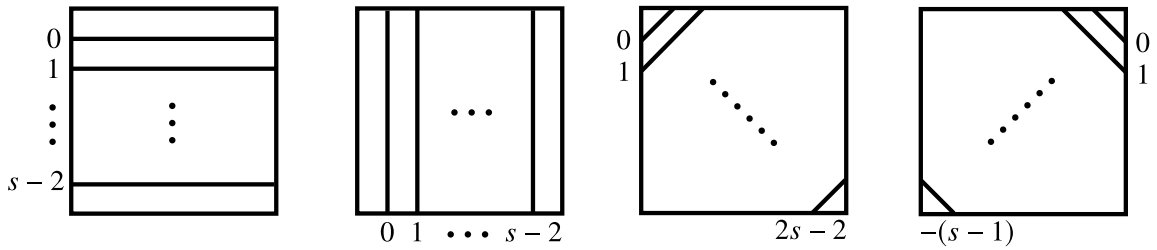


図 4.5: 折りの位置を指定する際に使う番号の振り方。s は格子サイズである。

4.3 紙を折るアルゴリズム

紙を折る処理は指定された折り方によって紙の状態を更新する処理である。よって、紙を折るためには、折る処理と折り方を指定する処理の 2 つが必要である。この節ではこれら 2 つのアルゴリズムについてまとめる。

折り方を指定するためには 4.2.2 項「折りのパラメタライズ」で述べた 5 つのパラメータを指定すれば良い。このうちの 1,3,4 にあたる「折りの方向」「山折りか谷折りか」「折り線のどちら側の紙を動かすか」は他のデータ構造とも独立したパラメータであるので、指定するのに特に工夫は必要ない。パラメータ 2 の「折りの位置」については、紙の各部分のデータを配列に格納するために空間を 45 度格子で分割するアプローチをとっているため、図 4.5 に示したように、45 度格子による各分割線に通し番号を振り、このパラメータを指定できるようにした。

5 つ目の「紙が重なっている場合、どこで紙を折るか」の指定は、独立して折れる紙の各部分に 1 から始まる通し番号を振り、その番号を指定することで実現する。0 は折られない側に振る番号として使う。この通し番号を振る処理をナンバリングと呼ぶこととする。折り方によって独立して折れる紙の部分は変わってくるため、ナンバリング処理を行う前に「折りの方向」「折りの位置」「折り線のどちら側の紙を動かすか」が決定されている必要がある。そのため、プログラムでは、1 から 4 のパラメータを決定した後、ナンバリング処理を行い、その結果から追加でどこで紙を折るかを指定して、折る処理に渡す折り方を決定するようになっている。

ナンバリング処理の例を図 4.6 に示す。事前に指定されたパラメータから、指定された折り線から見て折られる側の部分についてナンバリング処理を行う。図 4.6 では、折り線の手前側の部分である。例では、折られる側には独立して折れる部分が 3 つ存在するため、その各部分に通し番号を振る。この結果を用いて 5 つ目のパラメータとして通し番号のいずれかを指定することで、折りの指定が完了する。

ナンバリング処理は再帰呼び出しを用いて実装されている。まず、各三角形要素に折られる側に位置するかどうかのフラグを設定する。次に折られる側に位置する 1 つの三角形要素に対して通し番号を振り、再帰呼び出しを用いて隣接する三角形要素にも同じ通し番号を振っていく。この時、隣接する三角形要素の折られる側かどうかのフラグをチェックし、折られる側にない場合はその三角形要素に対しての再帰呼び出しは行わない。この再帰呼び出しによ

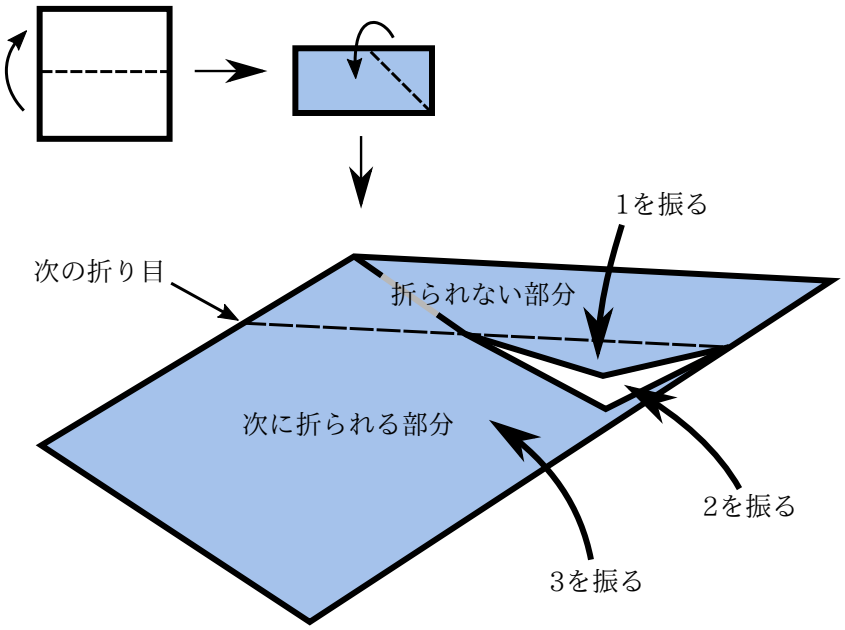


図 4.6: ナンバリング処理の例。

る番号振りを折られる側に位置する三角形要素全てに通し番号が振られるまで繰り返し、ナンバリング処理を完了する。

ナンバリング処理において注意しなくてはいけないのは、図 4.7 に示すような場合である。図 4.7 はある状態の紙の断面図の例であり、図中の同じ色の部分は同じ通し番号が振られていることを表している。(a) の青い部分は折られる側にある紙の一部分としては独立しているが緑の部分に囲われており青い部分のみで折ることはできず、青い部分が折られる場合には必ず緑の部分も折られていることとなる。そのため、(a) のように同じ通し番号の別の部分で挟まれている部分があれば、(b) のような結果になるよう、挟まれている部分の通し番号を挟んでいる部分と同じにする処理を行っている。

(a) 誤ったナンバリング　　(b) 正しいナンバリング

図 4.7: 紙の部分として独立していても独立して折れない部分に対する誤ったナンバリング (a) と正しいナンバリング (b)。

ここまでの手順によってどういった折りで紙を折るかが決定できるので、後は折る処理を実際に実行するのみである。折る処理は図 4.8 のように行われる。図 4.8 は通し番号 3 の部分

で谷折りする時に三角形要素を移動する例である。谷折りの場合は指定された通し番号より上にある部分を、山折りの場合は指定された通し番号より下にある部分を、それぞれ折り線について線対称の位置にある配列の要素に逆の重なり順になるように移動させる。

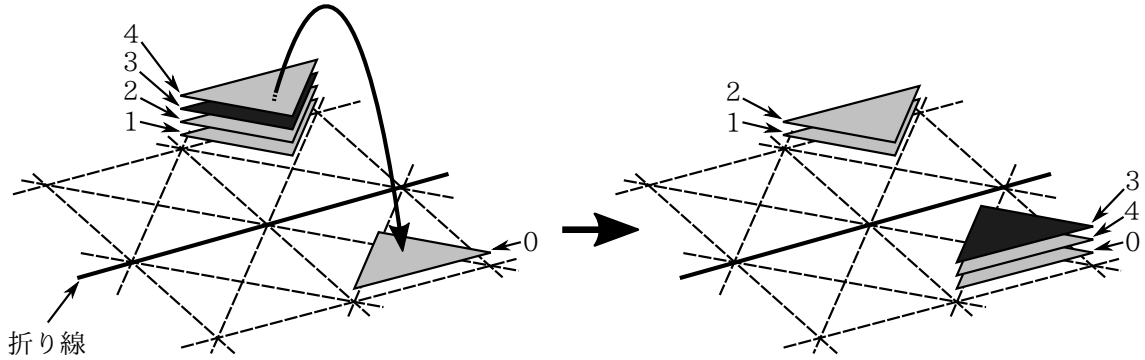


図 4.8: 折る処理におけるデータ移動の例。

ここまで紙を折る処理について述べてきたが、実際には折る処理自体の前に、注目している折り線で紙を折ることができるかどうかを判定する必要がある。折り線が指定された時、その折り線で折ろうとする前に、折り線の両側に紙が存在するかを調べ、そうである場合にはその折り線で折れると判断する。

4.4 探索の方法

以上に述べた内容で、紙を折る操作をプログラム化してコンピュータ上で実行できるので、折る操作を繰り返すことで手順の最適解探索を行うことができる。

探索は幅優先探索に基づいて行われるが、幅優先探索におけるメモリ使用量の問題から、探索途中で深さ優先探索に切り替えることが可能なプログラムを作成した。また、探索速度向上のために深さ優先探索の並列化も行った。

ここでは、最適解探索時に行われている処理と探索の効率化のための種々の最適化についてまとめる。

4.4.1 幅優先探索

幅優先探索は本研究における探索処理の最初のステップである。前述のとおり、幅優先探索の次のステップとして深さ優先探索も行うことができるが、探索量に対して十分なメモリがある場合は幅優先探索のみで終わることもある。

幅優先探索は図 4.9 に示すように queue を用いて実装した。探索前に初期の queue q_0 に 1 手も折られていない初期状態の紙のデータを格納し、指定された幅優先探索の手数 N まで q_{n-1} から紙の状態を 1 つ取り出し、その状態で可能な折り方全てについてを列挙し、列挙したそれぞれの折り方で折った結果を q_n に格納する。この時、$1 < n \leq N$ である。

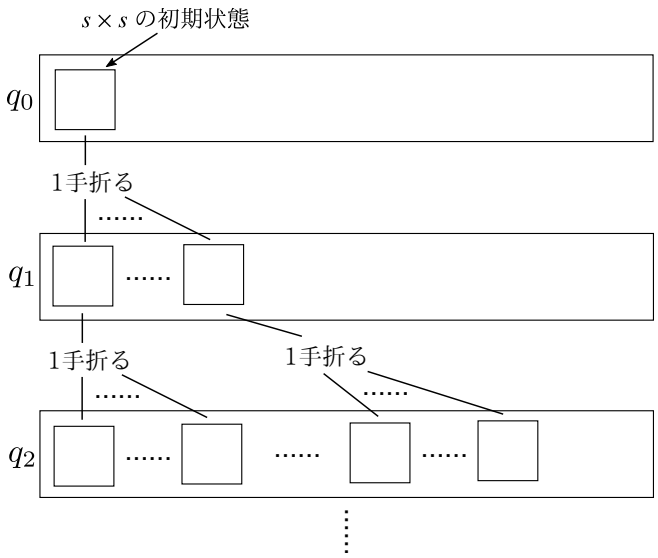


図 4.9: queue を利用した幅優先探索。本研究の探索範囲は $s = 4, 5, 6, 7, 8, 9$ である。

既知の状態を利用した枝刈り

探索中には、図 4.10(図 3.3 再掲) のように、折りの手順は異なるが折った後の状態が同じになる組が複数出現する。

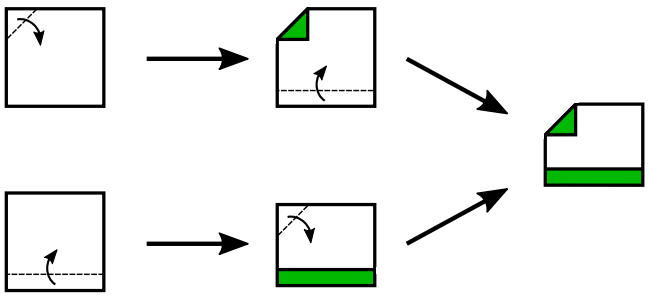


図 4.10: 折りの手順は異なるが折った後の状態が同じになる例 (図 3.3 再掲)。

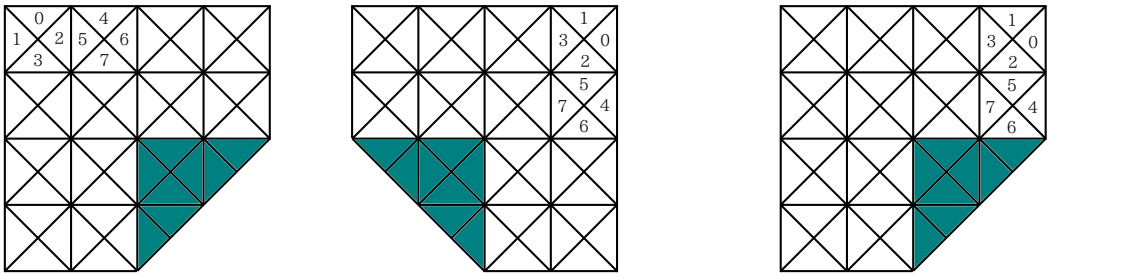
同じ状態に対する探索を複数回行うのは無駄であるので、これを避けるために、幅優先探索中には枝刈りを行っている。この枝刈りにおいては、紙の回転と反転、鏡像も考慮しており、既知の状態と実質的に同じ状態のものは全てそこで探索を打ち切るようにしている。枝刈りで考える項目は以下に示す 4 点である。

1. 紙の鏡像 (2 通り)
2. 紙の裏返し (2 通り)

3. 紙の 90 度ごとの回転 (4 通り)

4. 紙を構成する三角形要素に振った通し番号の 90 度ごとの回転 (4 通り)

枝刈りで考慮する項目のうち、項目 3 と 4 は似ているが、図 4.11 に示すような違いがある。3 の「紙の 90 度ごとの回転」では紙全体が回転するため、図 4.11(a) の状態から時計回りに 90 度回転した場合、回転の結果は (b) のようになる。一方、4 の「紙を構成する三角形要素に振った通し番号の 90 度ごとの回転」では、紙自体は回転させず、通し番号のみを 90 度回転させる。よって、図 4.11(a) を項目 4 の方法で時計回りに 90 度回転させた場合、(c) が回転の結果として得られる。



(a) 回転前の状態　(b) 紙の時計回り90度の回転結果　(c) 通し番号の時計回り90度の回転結果

図 4.11: 枝刈りで考慮する 2 つの回転。回転前の状態は 1 度も折られていない正方形の紙の右下を谷折りした状態とする (a)。通し番号は 0 から 7 までのみ示した。

それぞれの項目に対して考えられる状態数を掛け合わせると $2 \times 2 \times 4 \times 4 = 64$ であるので、実質的に同じと言える紙の状態は 1 つでも、プログラム中では 64 通りに表現され得るということがわかる。

紙の状態を一意に決定するには、紙を構成する各三角形要素について以下の情報がわかっていればよい。

- 初期位置 (最初に振られる通し番号)
- 現在の位置
- 現在位置での重なり順
- 表裏

そこで、連続したメモリ領域に、通し番号順に各三角形要素の現在位置、重なり順、表裏を格納して、その結果得られるバイト列を紙の状態を一意に表す情報として扱うこととした。

データ量節約のために、ビットパッキングも行っている。現在位置を表す値の最大値は、通し番号の最大値と同じであるから、今回の探索条件では $9 \times 9 \times 4 = 324$ かつ $2^8 = 256 < 324 < 2^9 = 512$ より、現在位置は 9 ビットあれば保持できる。重なり順は、最大 6 手探索することから紙は

最大 $2^6 = 64$ 層に重なる可能性がある。よって、6 ビットの領域があればその三角形要素が何層目にあるか表すことができる。表裏は 2 通りの値なので 1 ビットの値で表現できる。以上の計算より、1 個の三角形要素に対して $9 + 6 + 1 = 16$ ビットの領域を用意して各三角形要素の情報をビットパッキングにより保存している。また、1 個の三角形要素あたり 16 ビット即ち 2 バイトの領域を要するため、格子サイズを s とした時、1 つの紙の状態を保存するには $2 \times 4s^2$ バイトの領域が必要となる。

枝刈りをするためには、折った結果の紙の状態に対して得られたビット列を保存する機能と、現在注目している紙の状態のビット列と同じものが既に出現して保存されていないか照合する機能が必要になる。単に 2 次元配列として保存しただけでは、メモリ効率も悪く、照合にも時間がかかる。そこで、メモリ効率と照合にかかる時間の長さの観点から、紙の状態を表すビット列はトライ木によって保存することとした。

辞書等で文字列を扱う場合、トライ木の枝は文字を表し、各ノードにそれまで根から辿ってきた枝の文字を繋げた文字列が辞書に含まれているかどうかのフラグを設ける。しかし、本研究では、メモリ使用量を減らすために 1 度の探索で異なった格子サイズの探索を同時に行わないことにした。これにより、トライ木に格納するデータ列の長さは常に一定となるため、前述の「含まれているかどうかのフラグ」は不要となった。

紙のサイズを利用した枝刈り

紙を折っていくと段々と紙のサイズが小さくなっていく。ここで、紙の縦と横のどちらかまたはその両方が格子 2 マス分の幅以下しかない場合を考えると、その状態から 1 手折るだけでは、紙の縦と横の両方を 3 にすることはできないということが言える。

なぜなら、Origami Checkerboard パズルでは、紙を折ることはできるが広げることはできないから、垂直または水平に折る場合、縦か横の幅が 2 以下になった時点でその長さは 3 以上に増えることはなく、また、斜めに折る場合は幅が 1 または 2 の部分が必ず残ってしまうからである。

これを利用して、(探索最大手数 – 1) 手折った段階で、紙の縦と横のどちらかまたはその両方が格子 1 マス分の幅しかない状態が出現した場合は枝刈りを行うようにした。

折り手順の保存

本研究の目的は各パターンに対する最適な手順を見つけることであるので、パターンを探索する以外に、最終的にそのパターンに到達するまでの手順も出力できなくてはならない。

手順の保存には紙の状態を保存するトライ木の末尾のノードへのポインタを利用する。図 4.12 のように、状態 S_b で折り f_a で折った結果が S_c である場合、元の状態、折った結果の状態、折りの組で表されるデータ (q, r, f_a) を用意し、これを保存する。この時、紙の状態はトライ木の末尾ノードへのポインタで表すので、S_b と S_c そはれぞれ q と r で表される。また、例えば、S_c からさらに折り f_b で折った結果 S_a が 3×3 のパターンを成していたとする。この場

合、手数が現在の探索で明らかになっている最短手数以下であるかを確認し、そうであれば合致していたパターンの解を保存する領域へ S_a のアドレス p を保存する。

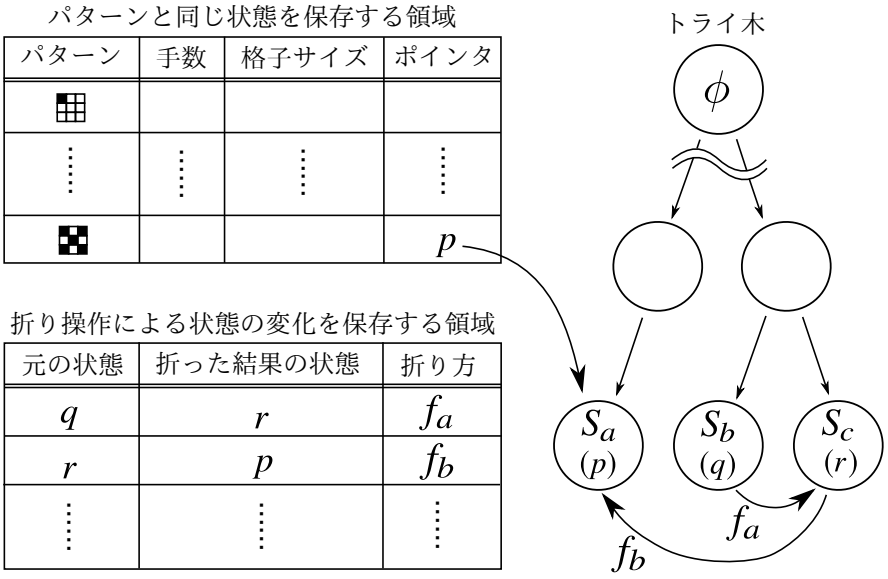


図 4.12: 解の手順の保存方法。トライ木に保存された状態 S_a, S_b, S_c それぞれを示すために末尾ノードのアドレス p, q, r を利用する。

手順を復元する際には、解を保存する領域からポインタの値 p を取り出し、折った結果のポインタが p と等しい即ち (q, p, f) であるような元の状態、折った結果の状態、折りの組を探し、その時の折り f を別の領域に記憶しておく。その後、該当する組が見つからなくなるまで、再帰的に折った結果のポインタが q であるような組を探して、記憶された折り手順を逆に辿ることでそのパターンへ至る折り手順を復元できる。

4.4.2 深さ優先探索

実際に幅優先探索を行うプログラムを実行した結果、探索中に出現する状態が非常に多く、メモリ不足で探索プログラムが途中で終了してしまう事象に見舞われた。この問題に対応するため、幅優先探索の途中から深さ優先探索に切り替えて探索を行う機能を実装した。

メモリ使用量を抑えることがこの深さ優先探索の目的であるため、探索の途中では既知の状態のメモリへの保存は一切行わず、幅優先探索の結果得られている既知の状態のみを利用して枝刈りを行う。

また、探索の高速化のために深さ優先探索の並列化を行った。図 4.13 に示すように、幅優先探索で得られた結果を配列に保存し、その配列に保存されている紙の状態 1 つ 1 つを取り出し逐次的に深さ優先探索を行う処理を並列化した。深さ優先探索の途中で 3×3 のパターンが見つかった場合は、解答保存領域をロックし解を書き込む。

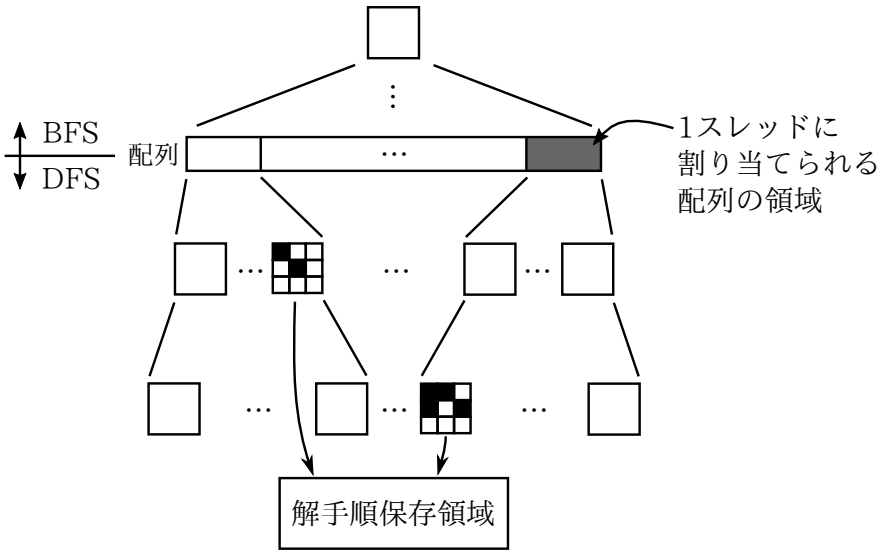


図 4.13: 途中から並列化された深さ優先探索に切り替える。

探索の並列化においては、同一のメモリ領域への書き込みが複数のスレッドから同時に行われる際のロックが頻繁に行われるとパフォーマンスの低下に繋がるという問題がある。しかし、この深さ優先探索では既知の状態のメモリへ追加で紙の状態を保存しないため、探索中のメモリへの書き込みは 3×3 のパターンが見つかりその手順即ち解を保存する時のみである。加えて、探索中出現する紙の状態と比べて 3×3 のパターンに至る手順の数は非常に小さい。このことから、そもそもにロックが起こる頻度が低く、並列化の結果十分な高速化を達成することができた。高速化の評価については 5.3 節「探索時間」で述べる。

4.5 実装と実行

プログラムの実装は C++で行った。

紙の状態を保存する配列の実装には std::vector を用いた。また、三角形要素のデータは紙の状態を保存する配列とは別の 1 次元配列に保存する。そして、その 1 次元配列の各要素へのポインタを紙の状態を保存する配列に保存するようにして、データ移動を行う際のコストを下げ、かつ、隣接する三角形要素への参照が簡単にできるようにした。

並列化には実装の簡単さから、OpenMP を使った。

プログラムの実行には JAIST のスーパーコンピュータを利用した。スーパーコンピュータの仕様を表 4.2 に示す。

SGI UV3000 にはジョブ管理システムが導入されており、計算を実行する際には適切な実行キューにジョブを投入する必要がある。本研究で主に利用したジョブキューの仕様を表 4.3 に示す。

表 4.2: 計算に用いたスーパーコンピュータの仕様。

名称		SGI UV3000
全体	ブレード数	128
	CPU	71.27TFLOPS (256CPU、1536 コア)
	メモリ	32TB (16GB × 2048DIMM)
	ディスク	SGI Infinite Storage 5100 (160TB)
ブレード	CPU	Intel Xeon E5-4655 v3 × 2 CPU
	メモリ	16GB DDR4 × 2DIMM × 4ch per CPU

表 4.3: 本研究で主に利用したジョブキューの仕様。

キュー名	コア数	メモリ	持続時間	優先度
SMALL	12〜48	256GB〜1TB	7 days	130
MEDIUM	48〜96	1TB〜2TB	3 days	90
LARGE	96〜192	2TB〜4TB	14 days	70

4.6 解の集計

探索結果のログファイルを読み込んで各格子サイズにおける各パターンに対する最短手数と別解の個数、それらの解の手順を集計するプログラムを作成した。

幅優先探索のみで探索が終了する場合は、解が重複することはない。しかし、後半の探索に深さ優先探索を用いた場合は、それまでの幅優先探索で得られた結果をもとにした枝刈りしか行わないため、折り手順に交換可能な部分が含まれていて折りあがりの状態が同じになる手順が解として複数得られる可能性がある。

これらの解の組は 3.4 節「別解の定義」で述べたように、別解としては扱わないため、解の状態が同じになる手順が見つかった場合には、ログファイルに一番最初に出現した手順以外を読み飛ばす機能を集計プログラムに実装した。

第5章　結果と考察

作成したプログラムを用いて最適解の探索を行った結果、探索対象とした 4×4 から 9×9 の格子サイズ全てで 6 手までの探索を達成した。

その結果、50 個の全パターンに対して最適解が得られ、既知の解は全て最適解のうちの 1 つであることが明らかになった。また、既知の最適解と同じ格子サイズと手数であるが、折りの手順が異なる別解の一覧も同時に得ることができた。

この章では作成したプログラムにより 50 問の各パターンに対する最適解を求めた結果とその結果に対する考察を述べるほか、プログラムの実装と実行時間等についての評価も行う。

5.1　探索結果

4×4 から 9×9 のそれぞれの格子サイズに対して探索を行った結果得られた各パターンに到達する折り手順の手数と別解の個数を表 5.1 に示す。

探索の結果、50 個の全パターンに対して最適解が得られた。得られた最適解のうち、既知の解よりより良い解であるものは 1 つもなく、既知の解は全て最適解のうちの 1 つであることが明らかになった。また、全 50 パターンを折ることができる最小の格子サイズは 9×9 であることも結果から読み取れる。

1,19,21,22,25,43,46 の各パターンに対しては、手数は最適解よりも長くなるが、格子サイズは最適解よりも小さい手順が得られた。

表 5.1 から読み取れるように、別解が得られた問題と格子サイズの組み合わせもあった。別解の例として、# 6 に対して得られた 3 つの別解の一覧を図 5.1 に示す。

5.2　紙の状態数

今回の探索によって各パターンに対する最適解の他に、4×4 から 9×9 それぞれの格子サイズからスタートして紙を折っていった時の n 手目における独立した紙の状態数を数え上げることができた。その結果を表 5.2 に示す。

括弧で囲われた数値は紙の縦または横の幅が 1 の場合を含まない個数であることを表す。4.4.1 項「幅優先探索」で述べた紙のサイズを利用した枝刈りによって紙の縦または横の幅が 1 の場合は除外されており、カウントされていないためである。

独立した状態数を数えるためには幅優先探索を行う必要がある。しかし、途中から深さ優先探索に切り替える場合に比べて全ての探索を幅優先探索で行うためには大量のリソースを

表 5.1: 探索結果。各欄の表記が $s(c)$ のものは、s が手数を c が別解の個数をそれぞれ表す。太字になっているものが探索が完了した範囲での最適解である。また、解が見つからなかった場合は空欄とした。

#	既知の解		探索時の格子サイズ						#	既知の解		探索時の格子サイズ					
	手数	格子	4×4	5×5	6×6	7×7	8×8	9×9		手数	格子	4×4	5×5	6×6	7×7	8×8	9×9
1	2	5×5	3(1)	**2(1)**	3(2)	4(99)	3(1)	4(21)	26	6	5×5		**6(1)**		6(6)		6(4)
2	3	4×4	**3(1)**	3(3)	4(23)	4(1)	3(1)	4(16)	27	3	5×5		**3(1)**	3(1)	5(1)	5(1)	4(24)
3	4	5×5		**4(3)**	4(2)	5(1)	5(1)	4(3)	28	3	4×4	**3(2)**	3(5)	4(24)	4(15)	3(2)	5(1)
4	2	4×4	**2(1)**	2(1)	4(128)	4(114)	2(1)	4(22)	29	4	5×5		**4(5)**	4(2)	5(1)	5(1)	5(1)
5	3	5×5		**3(1)**	4(6)	5(1)	5(1)	4(6)	30	3	5×5		**3(1)**	4(28)	4(1)	5(1)	5(1)
6	4	5×5		**4(3)**	5(1)	5(1)	6(56)	5(5)	31	4	5×5		**4(1)**	5(1)	5(1)	5(7)	5(24)
7	5	5×5		**5(1)**	5(1)	5(1)	5(13)	5(13)	32	2	4×4	**2(1)**	3(4)	3(2)	4(27)	2(1)	4(9)
8	4	4×4	**4(2)**	4(2)	4(1)	5(1)	4(2)	6(54)	33	5	5×5		**5(2)**	5(1)	6(5)	5(1)	5(1)
9	5	4×4	**5(1)**	5(1)	5(1)	5(1)	5(1)	5(17)	34	5	5×5		**5(3)**	6(35)	6(9)	6(7)	6(10)
10	3	5×5		**3(1)**	4(10)	4(1)	5(1)	4(7)	35	4	5×5		**4(1)**	5(1)	5(1)	5(5)	5(13)
11	4	5×5		**4(1)**	5(1)	6(4)	6(47)	4(2)	36	5	5×5		**5(3)**	5(1)	5(2)	6(24)	5(3)
12	2	4×4	**2(1)**	2(1)	3(9)	4(221)	2(1)	3(3)	37	4	5×5		**4(9)**	4(9)	4(3)	5(1)	5(1)
13	4	4×4	**4(1)**	4(8)	4(3)	4(1)	4(1)	5(1)	38	5	5×5		**5(1)**	6(49)	5(3)	5(1)	6(13)
14	3	5×5		**3(1)**	4(2)	4(3)	5(1)	5(65)	39	6	5×5		**6(1)**	6(7)	6(11)	6(1)	6(1)
15	4	4×4	**4(1)**	5(1)	5(1)	5(5)	4(1)	5(1)	40	6	5×5		**6(1)**	6(3)		6(4)	6(13)
16	4	5×5		**4(3)**	5(1)	5(1)	5(1)	5(1)	41	3	4×4	**3(1)**	3(1)	5(68)	5(1)	3(1)	5(21)
17	4	5×5		**4(1)**	4(1)	5(1)	5(1)	5(1)	42	4	5×5		**4(3)**	4(5)	5(1)	5(1)	5(1)
18	4	5×5		**4(5)**	5(10)	5(1)	5(1)	5(29)	43	5	6×6		6(3)	**5(1)**	6(24)	6(48)	5(2)
19	5	9×9		6(3)	6(7)	6(9)	6(7)	**5(3)**	44	4	5×5		**4(1)**	5(1)	5(1)	5(8)	6(20)
20	5	6×6			**5(1)**	5(3)	5(1)	5(2)	45	6	9×9						**6(1)**
21	5	9×9		6(1)	6(9)	6(14)	6(14)	**5(1)**	46	5	6×6		6(2)	**5(2)**	6(17)	6(28)	5(8)
22	4	7×7		5(1)	5(15)	**4(2)**	5(10)	5(26)	47	5	9×9				6(8)	6(8)	**5(5)**
23	5	5×5		**5(1)**	6(2)	6(14)	6(1)	6(13)	48	6	5×5		**6(4)**	6(15)	6(17)	6(9)	6(41)
24	4	5×5		**4(1)**	4(1)	5(1)	5(1)	5(1)	49	5	5×5		**5(2)**	5(1)	6(43)	6(11)	5(35)
25	5	9×9		6(1)		6(1)	6(1)	**5(2)**	50	6	9×9						**6(1)**

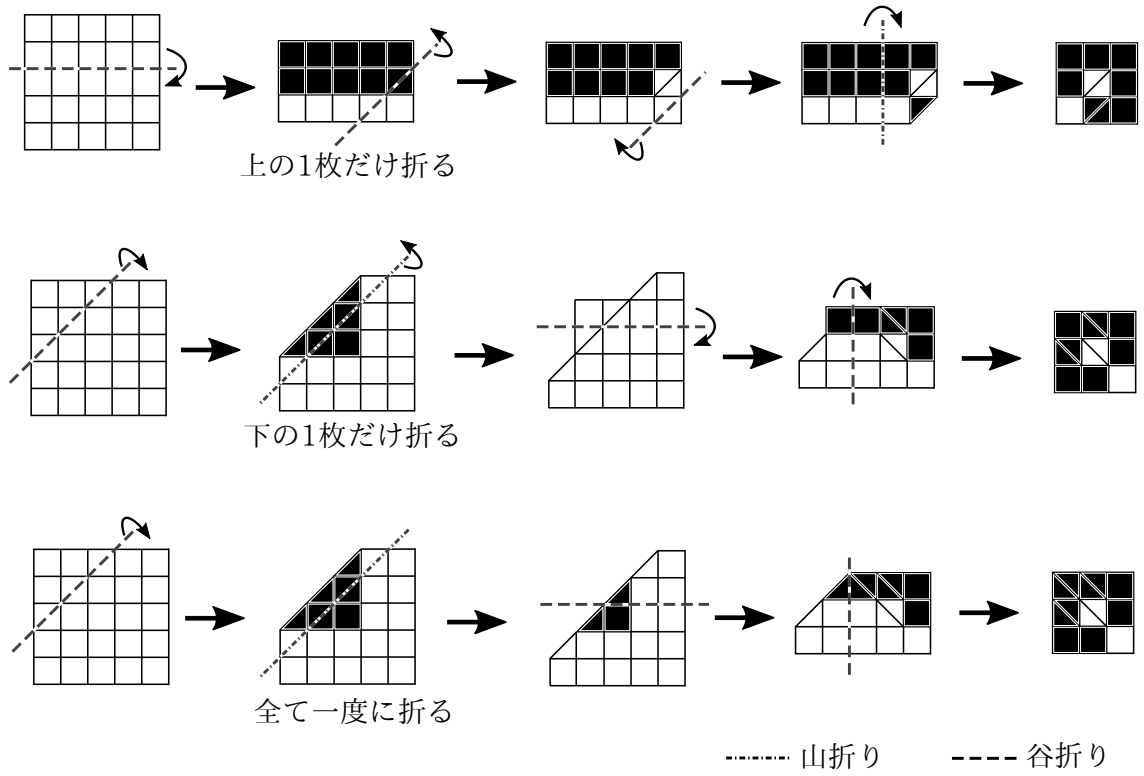


図 5.1: # 6 の別解の一覧。

必要とする。そのため、より大きい格子サイズ、より長い手数においては独立した状態数を全て列挙できない場合もあった。そのような格子サイズと手数の組み合わせは表では空欄とした。

表 5.2: 独立した紙の状態数。

手数	格子サイズ					
	4×4	5×5	6×6	7×7	8×8	9×9
1	6	7	9	10	12	13
2	100	165	252	351	474	607
3	1,936	4,381	8,364	14,204	22,323	33,024
4	37,526	117,676	282,798	585,277	1,075,344	1,834,254
5	589,148	2,784,214	8,642,541	21,947,979		
6	(4,431,474)	(31,882,460)				

独立した状態数と折りの回数の関係をグラフにプロットした結果を図 5.2 に示す。表 5.2 に掲載の 4×4 の 6 手目の状態数は紙の縦または横の幅が 1 の場合を含まないためグラフには含めなかった。独立した紙の状態数は折りの回数が増加すするにつれて指数的に増加することがわかる。

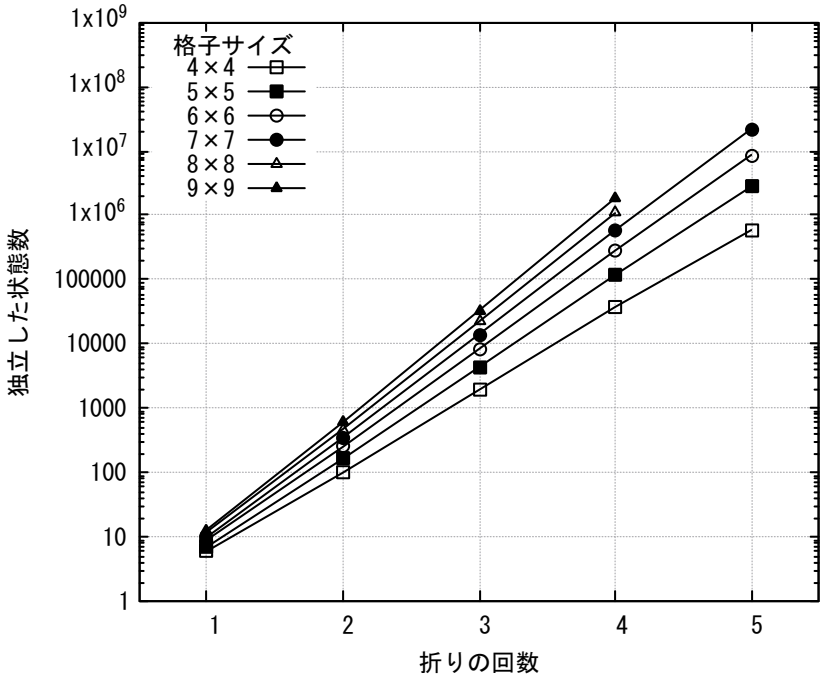


図 5.2: 独立した紙の状態数と折りの回数の関係。

また、独立した状態数と格子サイズの関係をグラフにプロットした結果を図 5.3 に示す。このプロットでも図 5.2 と同様に 4×4 の 6 手目の状態数は除外した。独立した紙の状態数は格子サイズが大きくなるのに従ってほぼ指数的に増加するが、増加の度合いは少しずつ減少す

ることがわかる。

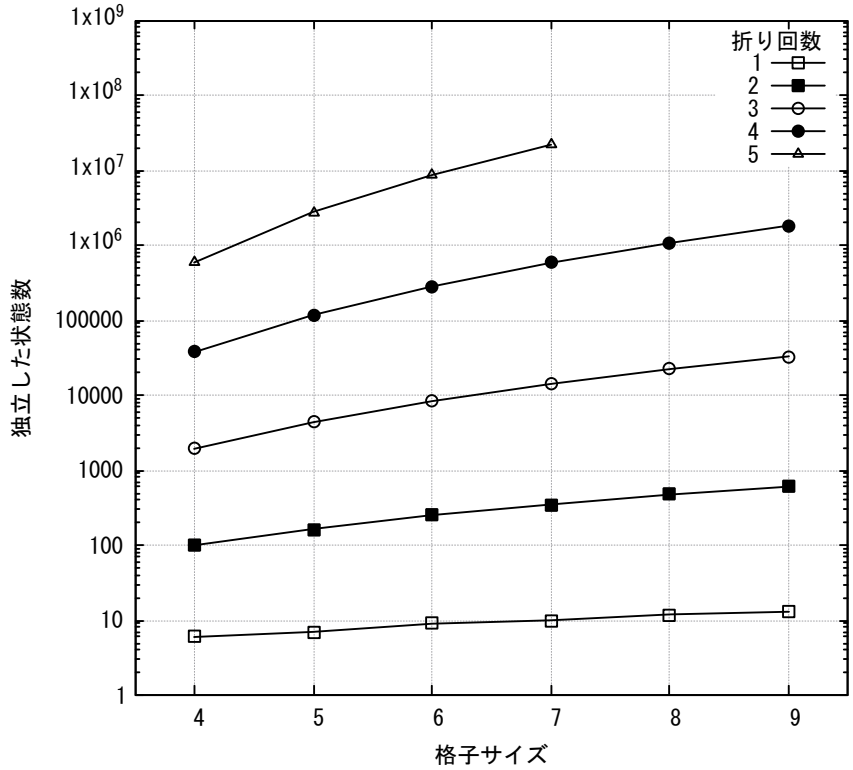


図 5.3: 独立した紙の状態数と格子サイズの関係。

5.3 探索時間

この節では、最適解の探索にかかった時間を計測した結果とその結果に対する考察をまとめる。計算時間は、格子サイズ、深さ優先探索の手数、OpenMP のスレッド数のそれぞれを変えた場合について計測した。

5.3.1 格子サイズと探索時間の関係

表 5.3 に示す条件で格子サイズを変えた場合の探索時間を計測した。格子サイズが 8×8 と 9×9 の場合に探索できた最長手数に合わせて、探索手数は 5 手までとした。探索時間の計測結果を表 5.4 に示す。また、探索時間をプロットしたグラフを図 5.4 に示す。

図 5.4 より、格子サイズに対して探索時間は指数的に増加することがわかる。このことは、5.2 節「紙の状態数」で述べたように、格子サイズが増えると状態数がほぼ指数的に増加するという事実と合致している。

表 5.3: 格子サイズを変えて探索時間を計測した時の条件。

項目	条件
探索手数	6 手
深さ優先探索探索の手数	後半 2 手
ジョブキュー	MEDIUM
CPU コア数	96
スレッド数	12288

表 5.4: 格子サイズを変えて探索時間を計測した結果。

格子サイズ	計算時間 [分]
4×4	1.45
5×5	6.95
6×6	35.07
7×7	140.20
8×8	440.23
9×9	1134.58

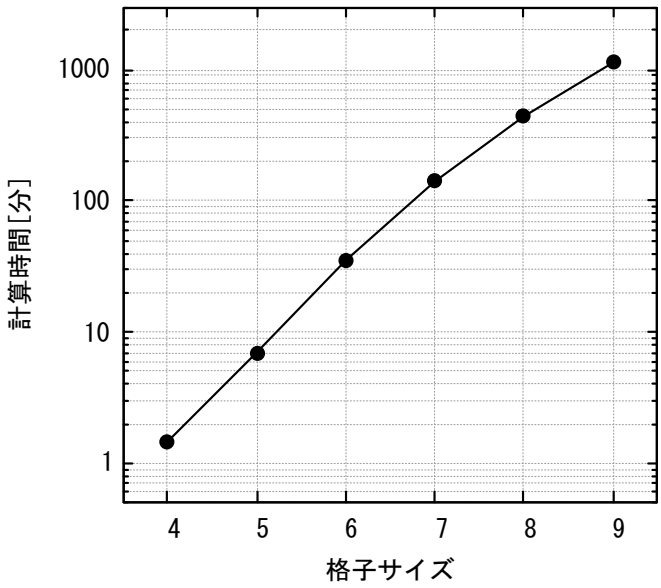


図 5.4: 格子サイズの変更に対する計算時間の変化。

5.3.2 深さ優先探索の手数と探索時間の関係

表 5.5 に示す条件で深さ優先探索の手数を変えた場合の探索時間を計測した。探索時間の計測結果を表 5.6 に示す。

表 5.5: 深さ優先探索の手数を変えて探索時間を計測した時の条件。

項目	条件
格子サイズ	4×4
探索手数	6 手
ジョブキュー	SMALL
CPU コア数	48
スレッド数	6144

表 5.6: 深さ優先探索の手数を変えて探索時間を計測した結果。

深さ優先探索の手数	計算時間 [分]
0	117
1	33
2	2
3	5
4	50
5	610

結果から、最も探索時間が短い深さ優先探索のステップが 2 手の場合の計算速度は 2 分と、全てのステップを幅優先探索で行った場合の 117 分の 58.5 分の 1 であり、並列化された深さ優先探索によって探索が大幅に高速化されたことがわかる。

また、深さ優先探索の手数を 0 手から増やしていくのに従って計算時間が短縮されるが、深さ優先探索の手数がある値 (ここでは 2 手) より大きくなると、逆に計算時間が長くなることもわかる。

深さ優先探索の手数を増やしていくと計算時間が短縮されるのは、幅優先探索はシングルスレッドで行われるのに対して、深さ優先探索は並列化によりマルチスレッドで実行されるためである。探索終盤には探索対象の紙の状態数が非常に大きなものとなるが、深さ優先探索の段階ではそれらを並列に探索できるため、より短時間で探索が終了する。

一方で、深さ優先探索の手数を増やしすぎると探索時間が長くなってしまうことは、深さ優先探索が並列化の恩恵を受けられていないことを示している。深さ優先探索の段階では既知の状態の追加は行わず枝刈りで参照されるのは過去に出現した状態のみであるから、深さ優先探索自体の探索効率は良いとは言えない。深さ優先探索の手数を増やすと、幅優先探索に対して効率の悪い深さ優先探索で探索を行う割合も増えてしまう。さらに、深さ優先探索の根となる状態数即ち独立して計算できる部分木の数が減り、並列化の程度も低下する。例えば、4×4 の格子サイズで、後半 2 手を深さ優先探索する場合は根となる状態数は 37,526 個あるが、後半 5 手を深さ優先探索する場合では 6 個と、部分木の数がスレッド数よりも少なくなってしまう。このような理由で、並列化による効率化で深さ優先探索自体の効率の悪さを改善できない場合には、深さ優先探索の手数を増やしても探索時間が増加すると考えられる。

5.3.3 OpenMP のスレッド数と探索時間の関係

表 5.7 に示す条件で OpenMP のスレッド数を変えた場合の探索時間を計測した。スレッド数の変更はプログラム実行前に環境変数 `OMP_NUM_THREADS` の値を設定することで行った。探索時間の計測結果を表 5.8 に示す。

表 5.7: OpenMP のスレッド数を変えて探索時間を計測した時の条件。

項目	条件
格子サイズ	4 × 4
探索手数	6 手
深さ優先探索探索の手数	後半 2 手
ジョブキュー	MEDIUM
CPU コア数	96

表 5.8: OpenMP のスレッド数を変えて探索時間を計測した結果。

スレッド数	CPU コア数比	計算時間 [秒]
1	1/96	4547
96	1	1135
768	8	312
1536	16	170
2304	24	147
3072	32	134
4608	48	117
6144	64	111
9216	96	107
12288	128	101

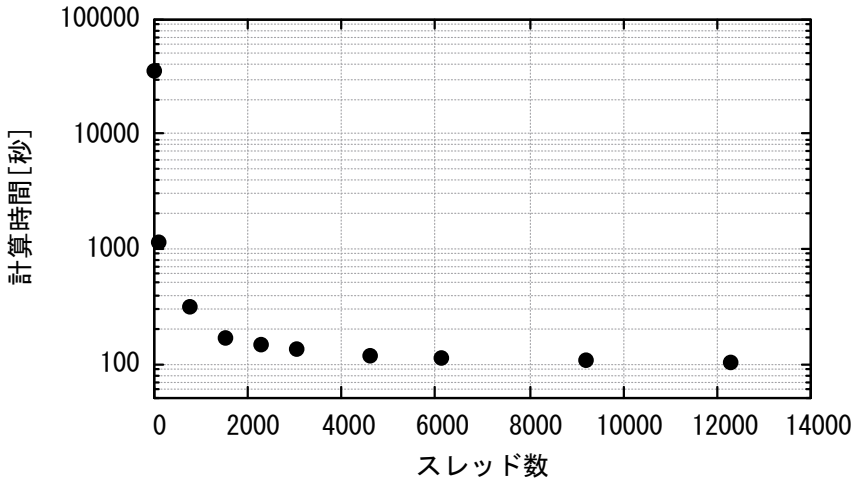


図 5.5: スレッド数と探索時間の関係。

計測結果から、スレッド数を増やすと探索効率が向上し、探索時間が短くなるが、探索効率の向上の度合いは段々と低下していくことがわかる。

また、スレッド数 $N = 96$ の時、1 スレッドの場合に比べて計算速度はほぼ 4 倍になったこと

から、並列化可能部分の割合 P は、アムダールの法則をもとに、式 5.1 のように推定できる。

$$P = \frac{\frac{1}{4} - 1}{\frac{1}{96} - 1} \simeq 0.758 \tag{5.1}$$

推定より、時間計測を行った格子サイズ 4×4 の計算では、並列化された深さ優先探索が行われた後半 2 手の探索が全体の約 75.8% を占めていると考えられる。

5.4 他のパズルとの関連

2004 年に Erik D. Demaine と Martin L. Demaine により、Origami Checkerboard パズルのルールを拡張し、スタートとなる正方形の紙を分割した格子の各マスとゴールとなるチェッカボードの各マスに模様を配置して、指定されたとおりに模様が配置されるように紙を折るというパズルが発表された [17]。スタートとなる紙の模様は図 5.6 に、目的のチェッカーボードパターン (2 種) は図 5.7 にそれぞれ示すようなものであった。模様は回転や反転の区別がつけば任意のもので良いのであるが、ここでは便宜的に文字「R」で表す。図 5.7 からもわかるように、このパズルが目標とする紙の状態は、Origami Checkerboard における# 23 のパターンに各マスの模様の向きという条件を加えたものである。

このパズルのゴールとなる紙の状態のうち、模様が縦に並ぶものは、今回の探索で得られた# 23 の解を使って図 5.8 に示す手順であると求められる。

紙を 90 度ずつ回転させれば、# 23 と同じ手順でも最終的な模様の配置は 4 通りに変化する。しかし、その中に模様が縦に並ぶものはあったが、横に並ぶものはなかった。よって、模様が横に並ぶ状態は今回の探索範囲には含まれておらず、折るためには単純折り以外の折り方が必要であることが予想される。

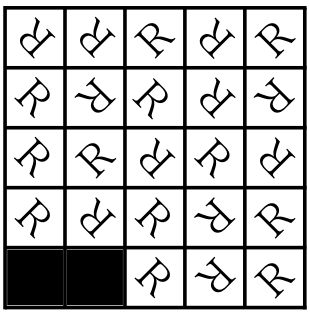


図 5.6: Origami Checkerboard パズルのルールを拡張したパズルの初期状態。このパズルが発表された時には左下の 2 マスにはパズルの説明が書かれていたため模様は割り当てられていない。また、裏面は白紙である。

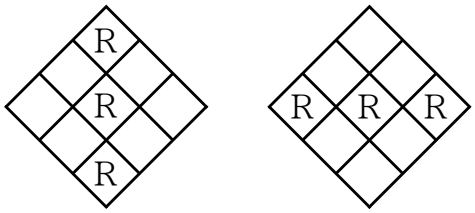


図 5.7: Origami Checkerboard パズルのルールを拡張したパズルの目的となるパターン。

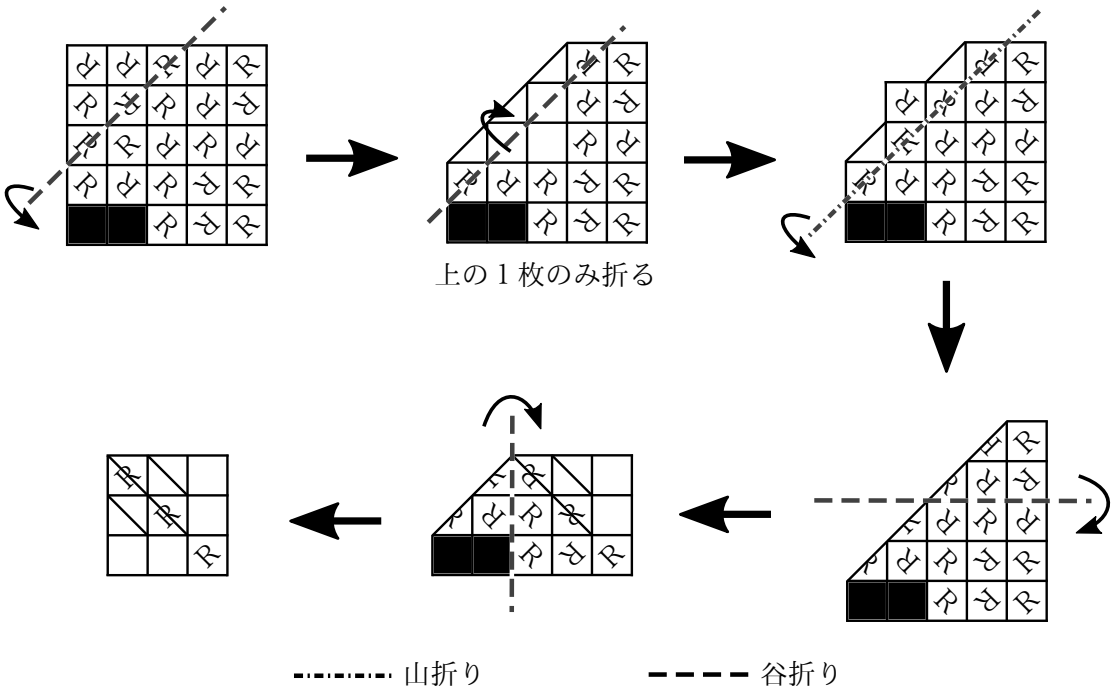


図 5.8: 図 5.6 に示す Origami Checkerboard パズルを拡張したパズルの目的のパターンのうち、模様が縦に並ぶものは# 23 の解の手順で折ることができる。

第6章　まとめ

本研究では図 3.1 に示した 50 種類の 3×3 のチェッカーボードパターンを折るための最適解を求めた。Origami Chekcerboard パズルに対するこれまでの成果として、全てのパターンに解となる手順が示されていたが、それらの解が最適解であるかどうかは不明であった。この問題を解決するために、本研究で最適解探索を行った結果、既知の解をより最適な解で更新することはできなかったが、格子サイズ 4×4 から 9×9 の範囲で全 50 パターンに対する最適解の全列挙を達成した。

計算にはスーパーコンピュータを用い、探索においては既知の状態を利用した枝刈り等の他、幅優先探索を途中から並列化された深さ優先探索に切り替える等の工夫を行った。これにより、探索速度を向上させ同時に探索範囲を拡大させることができた。

また、今後の研究対象及び Origami Checkerboard パズルの拡張として以下のようなテーマが考えられる。

- 格子サイズが 10×10 以上の領域の探索。
- 4×4 以上のチェッカーボードパターンに至る手順。
- パターン以外に模様も考えるパズル。

1 つ目の「格子サイズが 10×10 以上の領域の探索」は文字の通りで、今回の探索の最大格子サイズの 9×9 を超える格子サイズ 10×10 以上の正方形の紙からスタートする領域に既知の解よりも最適な解は存在するのかという問題である。最適解を更新するためにはより短い手順でなくてはならないという条件から、最適解が n 手であれば $n-1$ 手までの範囲の探索が行えれば良い。そのため、最適解とされている既知の手順が短い問題に対しては大きい格子サイズでも探索する手数を短く設定して探索を行うことは可能であると考えられる。一方で、既知の解の手数が長く、深く探索を行うことが求められる場合や、さらに大きい格子サイズに対して探索を行う場合は、やはり計算リソースに対して計算量が膨大になり探索が困難になったり、探索ができても長い時間が必要になったりすると考えられる。

実際により大規模な探索をするための方法の例としては、幅優先探索の終了後の状態全てではなく、それらを分割した一部のグループに対して並列化された深さ優先探索を行い、プログラムの複数回の実行で全状態に対する探索を完了させるというやり方が考えられる。現在の探索プログラムの実装では、n 手目から深さ優先探索を行う時は幅優先探索で得られた $n-1$ 手目での紙の状態すべてを対象としている。そのため、$n-1$ 手目の状態数が多い場合には深さ優先探索が現実的な時間内に終了しない可能性がある。しかし、この機能が実装で

きれば、深さ優先探索を実行されるプログラム単位で分割することができるようになるため、より大きい格子サイズの探索も可能になると考えられる。

ただし、この方法では幅優先探索終了時点での状態数自体が多くメモリが不足すると行った場合に対応できない。メモリ不足の問題を解決するには、探索時に幅優先探索の手数を減らす方法の他、ビットパッキング等のプログラミングにおけるメモリ節約手法をより積極的に採用する等の方法が考えられる。

2 つ目の「4×4 以上のチェッカーボードパターンに至る手順」は、3×3 より大きいチェッカボードのサイズをゴールとする手順に対する探索である。ゴールとなるチェッカボードのサイズが大きくなると、そのパターンに至るのに必要な格子サイズも大きくなると考えられるので、本研究における 3×3 のパターンの時よりも探索は困難になると考えられるが、4×4 以上のパターンを折る手順に関する成果は殆ど存在しないため、可能な範囲のみであっても探索を行うことは価値があると考えられる。

3 つ目の「パターン以外に模様も考えるパズル」は 5.1 節「探索結果」で紹介した、Origami Checkerboard パズルを拡張して各マスの模様も考慮するようにしたパズルのことである。本研究で各パターンに対する手順は明らかになった。手順が明らかであれば、スタート時の紙のどの部分が折り上がりのパターンのどこに来るかも求めることができるので、本研究の結果を利用して可能な模様の配置の一覧も列挙することができる。そのような列挙はまだ行っていないが、列挙を行うプログラムを作成することは大きな困難ではないと考えられる。

謝辞

本研究の遂行にあたり、全体を通して多くの的確かつ丁寧な助言、指導をいただいた三谷純教授に深く感謝の意を表します。

JAIST の上原隆平教授には JAIST 訪問の機会を与えていただいた他、プログラムの設計や実装に関して重要なアドバイスをいただきました。心より感謝いたします。

研究室の方々及び JAIST の上原研究室の皆様との議論では多くの示唆や刺激を得ることができました。ありがとうございました。

度々帰省した時に励ましの声をかけてくれた実家の両親と祖父母にも感謝しています。

参考文献

[1] Robert J. Lang, A computational algorithm for origami design, *Proceedings of the 12th Annual ACM Symposium on Computational Geometry*, pp.98-105, 1996.

[2] Erik D. Demaine, Sándor P. Fekete, and Robert J. Lang, Circle packing for origami design is hard, *Origami^5 : Proceedings of the 5th International Conference on Origami in Science, Mathematics and Education*, pp.609-626, 2010.

[3] Erik D. Demaine, Martin L. Demaine, and Joseph S. B. Mitchell, Folding flat silhouettes and wrapping polyhedral packages: New results in computational origami, *Computational Geometry: Theory and Applications*, Volume 16, Issue 1, pp.3-21, 2000.

[4] Tomohiro Tachi, Origamizing polyhedral surfaces, *IEEE Transactions on Visualization and Computer Graphics*, Volume 16, Issue 2, pp.298-311, 2010.

[5] Erik D. Demaine and Tomohiro Tachi, Origamizer: A Practical Algorithm for Folding Any Polyhedron, *33rd International Symposium on Computational Geometry (SoCG 2017)*, July 4-7, 2017, to appear.

[6] Tomohiro Tachi, Software: Origamizer, 2008. `http://www.tsg.ne.jp/TT/software/`.

[7] Jun Mitani, A Design Method for 3D Origami Based on Rotational Sweep, *Computer-Aided Design & Applications*, Volume 6, Issue 1, pp.69-79, 2009.

[8] Jun Mitani, ORI-REVO A Design Tool for 3D Origami of Revolution, 2011, `http://mitani.cs.tsukuba.ac.jp/ori_revo/`.

[9] Jun Mitani and Takeo Igarashi, Interactive Design of Planar Curved Folding by Reflection, *Pacific Conference on Computer Graphics and Applications - Short Papers*, pp.77-81, 2011.

[10] Jun Mitani, ORI-REF: A Design Tool for Curved Origami based on Reflection, 2011, `http://mitani.cs.tsukuba.ac.jp/ori_ref/`.

[11] Erik D. Demaine, Martin L. Demaine, and Duks Koschitz, Reconstructing David Huffman's Legacy in Curved-Crease Folding, *Origami5: Proceedings of the 5th International Conference on Origami in Science, Mathematics and Education (OSME 2010)*, pp.39-52, 2010.

[12] Erik D. Demaine, Martin L, Demaine, Goran Konjevod, and Robert J. Lang, Folding a Better Checkerboard, *Proceedings of the 20th Annual International Symposium on Algorithms and Computation (ISAAC 2009)*, Lecture Notes in Computer Science, Volume 5878, pp.1074-1083, 2009.

[13] 山本陽平, 三谷純, 平織りによるドット絵の表現, 折り紙の科学, Volume 6, No.1, 2017, http://origami.gr.jp/OSME/Origami-no-Kagaku06.html

[14] Serhiy Grabarchuk, The New Puzzle Classics: Ingenious Twists on Timeless Favorites, Sterling, 2005.

[15] 上原隆平, 私信による, 2017.

[16] 石井恵一郎, Origami Checkerboard, 2004, http://puzzlewillbeplayed.com/Origami/OrigamiCheckerboard/

[17] Erik D. Demaine and Martin L, Demaine, CSAIL Annual Meeting 2004 Puzzle, 2004, http://erikdemaine.org/puzzles/CSAIL2004/