

信州大学工学部

学士論文

ルービックキューブに対する誤り訂正符号の
群構造を用いた効率的な探索について

指導教員 西新 幹彦 准教授

学科 電子情報システム工学科
学籍番号 22T2803D
氏名 猪又 佑太郎

2026年2月9日

目次

1	はじめに	1
2	ルービックキューブの基本	1
3	誤り訂正符号	2
3.1	通信路モデルと符号化	2
3.2	距離と誤り訂正能力	2
4	符号語探索アルゴリズム	4
4.1	状態グラフの探索	4
4.2	貪欲法による符号語選択	5
4.3	禁止リスト法による距離検証の高速化	5
4.4	部分群構造に着目した符号語選択（提案手法）	6
5	探索性能の評価	6
5.1	評価条件とパラメータ設定	7
5.2	結果	7
6	考察	8
6.1	部分群構造の有効性について	8
6.2	禁止リスト法の限界性能	8
6.3	誤り訂正符号としての評価	9
7	まとめ	9
	謝辞	10
	参考文献	10
	付録 A 探索プログラム	11

1 はじめに

誤り訂正符号は、通信路上のノイズによる情報の欠損を防ぐための基盤技術として広く利用されている。一方で、世界的に有名な立体パズルであるルービックキューブは、その操作が群の構造をなすことから、数学的な研究対象とされてきた[1, 2]。本研究は、この二つの領域を融合し、ルービックキューブの巨大な状態空間を誤り訂正符号の新たな構成空間として活用する手法を提案するものである。

ルービックキューブのとりうる状態の総数は約 4.3×10^{19} 通りに達し、これは情報量にして約 65 ビットに相当する。これは、8 文字の英数字 (ASCII) パスワードが持つ情報量に匹敵する。つまり、何気なく机に置かれたキューブは、それだけで一つのパスワードを表現しているとも言える。

膨大な状態空間内の各状態を点、状態間の遷移を引き起こす回転操作を辺と見なすことで、ルービックキューブ上に巨大なグラフ構造（状態グラフ）を定義できる。本研究では、このグラフ上の 2 状態間の最短手数を「距離」として導入し、符号理論の枠組みを適用する。これにより、互いに一定の距離以上離れた状態の集合を「1 手操作の誤りを訂正可能な符号」として構成することを目指す。

しかし、このような巨大なグラフ上で大規模な符号を構成する試みは、計算量的な困難に直面する。符号語の候補が既存の符号語集合との最小距離条件を満たすか検証するプロセスは、符号語数 N に対して $O(N^2)$ の計算量を要し、単純な探索アルゴリズムでは大規模な符号の生成が現実的な時間内に完了しない。

この計算量のボトルネックを解消するため、本研究ではルービックキューブが持つ「群構造」に深く着目する。具体的には、特定の回転操作によって生成される部分群が、状態空間全体の中で疎に分布するという代数的性質を利用する。この性質を利用して符号語の候補を戦略的に選定することにより、距離検証における棄却率を下げ、実効的な探索効率を向上させる。本論文では、この群構造に基づいた探索戦略と、ハッシュテーブルを用いた高速な検証手法を組み合わせ、計算機資源の制約下で可能な限り大きな符号語集合を効率的に構成するアルゴリズムを提案し、その有効性を検証する。

2 ルービックキューブの基本

本章では、ルービックキューブの物理的な構造、状態の定義について述べ、本研究で用いる基本的な用語を定義する。

ルービックキューブは、各面が 3×3 に分割された立方体型のパズルである。6 つの面を持ち、それぞれの面は 9 つの領域（本研究ではステッカーと呼称する）に分割されている。ス

テッカーは 6 色存在し、それぞれの面を適切に回転させることで、6 面すべての色を揃えることができる。本研究では、この揃った状態を基本状態 e と呼称する。

6 つの面にはそれぞれ名前がついており、Up (上), Down (下), Right (右), Left (左), Front (前), Back (後) からなる。慣例に従い、それぞれ U 面, D 面, R 面, L 面, F 面, B 面と表す。ルービックキューブのある時点での色の配置を「状態」と呼ぶ。ルービックキューブは面を回転させることによって膨大な数の状態を取りうる。その組み合わせ数は約 4.3×10^{19} 通り（正確には $43,252,003,274,489,856,000$ 通り）であることが知られている [1, 3]。 $\log_2(43252003274489856000) \approx 65.23$ より、1 つのルービックキューブが持つ情報量は約 65 ビットとなる。

3 誤り訂正符号

本章では、本研究の基礎となる誤り訂正符号の概念と、その数学的な性質について述べる。

3.1 通信路モデルと符号化

ディジタル通信やデータの記録において、通信路上のノイズやメディアの劣化により、情報の一部が変化してしまう「誤り」が発生する場合がある。誤り訂正符号は、送信したい情報を冗長性を持たせることで、受信側で誤りを検出し、訂正することを可能にする技術である。

送信したいメッセージを m とし、符号化器によって符号語 c に変換されるとする。通信路を経て受信された語を r とする。ノイズが存在する場合、 c と r は必ずしも一致しない。復号器は、受信語 r から、最も確からしい送信語 c' を推定し、元のメッセージ m' を復元する。このとき、 $m = m'$ であれば誤り訂正が成功したことになる。

3.2 距離と誤り訂正能力

誤り訂正能力を議論する際には、符号語間の「近さ」や「遠さ」を定量的に評価する「距離」の概念を導入するとよい。集合 \mathcal{V} における任意の点 $x, y, z \in \mathcal{V}$ に対して、以下の 3 つの条件（距離の公理）を満たす関数 $d(x, y)$ を距離関数と呼ぶ。

1. **非負性:** $d(x, y) \geq 0$ であり、 $d(x, y) = 0$ となるのは $x = y$ のときに限る。
2. **対称性:** $d(x, y) = d(y, x)$ が成立する。
3. **三角不等式:** $d(x, z) \leq d(x, y) + d(y, z)$ が成立する。

一般的な符号理論では、等長 n の系列同士の異なるシンボルの個数を表す「ハミング距離」が用いられることが多い。しかし、本研究ではルービックキューブの状態空間を扱うため、操作の手数に基づいた距離を定義する必要がある。

まず、状態を変化させる「操作」を定義する。本研究では1操作を「6つの面のいずれかの1面を選び、その面を正面から見て時計回りもしくは反時計回りに90度だけ回転させる」こととする。この操作体系はQuarter Turn Metric(QTM)と呼ばれる。

この操作定義に基づき、本研究では状態 x から状態 y へ遷移するために必要な最小の操作回数を距離 $d(x, y)$ と定める。本論文ではこれをQT距離と呼ぶ。

符号 \mathcal{C} は、とりうる全ての状態の集合 \mathcal{V} の部分集合として定義される ($\mathcal{C} \subset \mathcal{V}$)。このとき、符号 \mathcal{C} に含まれる異なる2つの符号語間の距離の最小値を、その符号の最小距離 d_{\min} として、

$$d_{\min} \triangleq \min\{d(c_i, c_j) \mid c_i, c_j \in \mathcal{C}, c_i \neq c_j\} \quad (1)$$

と定義する。

受信した状態を最も距離が近い符号語に復号する最近傍復号を考えると、符号 \mathcal{C} がQT距離において t 以下の誤りを訂正可能であるための必要十分条件は、最小距離 d_{\min} が

$$d_{\min} \geq 2t + 1 \quad (2)$$

を満たすことである。この条件は、任意の符号語 c_i を中心とする半径 t の球 (c_i からの距離が t 以下である状態の集合) を考えたとき、異なる符号語 c_i, c_j を中心とする球が互いに交わらないための条件に由来する。すなわち、ある状態が距離 t 以下の誤りによって変化したとしても、その状態が属する半径 t の球はただ一つに定まるため、一意な復号が可能となる。この関係は本研究で採用するQT距離においても成立する。

すなわち、1つの誤りを訂正するためには、 $t = 1$ を代入して

$$d_{\min} \geq 3 \quad (3)$$

を満たす符号を構成する必要がある。本研究では、ルービックキューブの状態空間上において、任意の符号語間のQT距離が3以上となるような集合 \mathcal{C} を構成することを目的とする。これは、ある符号語 c が1手操作されて別の状態 r に変化したとしても (QT距離1の誤りが発生)、 r から距離1の範囲にある符号語は元の c 以外に存在しないため、一意に復号可能であることを意味する。

この誤り訂正能力と、構成可能な符号語の数 (符号サイズ $|\mathcal{C}|$) にはトレードオフの関係が存在する。 $d_{\min} \geq 3$ という条件は、任意の異なる2つの符号語 c_i, c_j を中心とする半径1の球が、互いに交わらないことを意味する。定義したQTMにおいて、ある状態から1手の操作で到達可能な状態は12通りであるため、1つの符号語を中心とする半径1の球に含まれる状態の数は、符号語自身を含めて $1 + 12 = 13$ 個となる。

ルービックキューブの状態グラフは各頂点の次数が一定であり、これらの球は全状態空間 \mathcal{V} の中に配置され、互いに素であるから、符号語の総数 $|\mathcal{C}|$ に関して以下の不等式 (スフィア

パッキング限界) が成立する.

$$|\mathcal{C}| \cdot 1 + 12 \leq |\mathcal{V}| \quad (4)$$

ここで $|\mathcal{V}|$ はルービックキューブの全状態数 (約 4.3×10^{19}) である. 実際に代入すると, $|\mathcal{C}| \leq 3.3 \times 10^{18}$ とわかる. これは約 62 ビットに当たる. この式は, 誤り訂正能力 (d_{\min}) を確保しようとすると, 符号語数 $|\mathcal{C}|$ に上限が課せられることを示している. 本研究では, この制約の中でできるだけ多くの符号語を持つ集合 \mathcal{C} を効率的に構成することを目指す.

4 符号語探索アルゴリズム

前章までの議論に基づき, 本章ではルービックキューブの状態空間を利用して $d_{\min} \geq 3$ の符号を構成するための具体的なアルゴリズムについて述べる. 本研究では, 単純な貪欲法に加え, 計算効率を向上させるために群構造を利用した新たな探索手法を提案する.

本研究における符号構成の手順は, 主に以下の二つのフェーズに大別される.

1. **状態グラフの探索フェーズ**: ルービックキューブの状態グラフを探索し, 基本状態 e からの最短距離が既知である状態のリスト \mathcal{L} を構築する.
2. **符号語選択フェーズ**: 構築された状態リスト \mathcal{L} から, 誤り訂正能力の条件を満たす符号語の集合 \mathcal{C} を選択し, そのサイズを最大化する.

状態グラフの探索フェーズでは, 幅優先探索 (Breadth First Search, 以下 BFS) を用い, 各状態が既に見つかっているかを確認しながら, 新規状態を状態リスト \mathcal{L} に追加する.

一方, 符号語の選択フェーズは, 既存の符号語リスト \mathcal{C} と, リスト \mathcal{L} 内の新しい候補との距離を検証するプロセスである. 符号サイズを $N = |\mathcal{C}|$ とするとき, すべての符号語ペア間の距離検証には $O(N^2)$ の計算量を要する. この計算量の急激な増加は, 符号サイズ N を大きくする上での深刻なボトルネックとなる.

そこで本章では, まず 4.1 節で基礎となる状態グラフの探索手法について述べる. 続く 4.2 節以降では, 符号語選択における計算コストの問題に対処するためのアルゴリズムとして, 基本となる貪欲法, 禁止リストを用いた高速化, および群構造を利用した候補絞り込みの 3 つのアプローチについて詳細に述べる.

4.1 状態グラフの探索

本研究では, ルービックキューブの状態グラフ \mathcal{G} の探索アルゴリズムとして, BFS を採用した. BFS は, 探索の待機リストとしてキュー (FIFO) を用いるアルゴリズムであり, 探索開始ノード (基本状態 e) から QT 距離が k のノードを全て探し終えてから, 次に QT 距離が $k+1$ のノードの探索を開始する. これにより, 層ごとの網羅的な探索が保証される. し

たがって、あるノード x に初めて到達した際の探索経路長は、定義上、基本状態 e から x への最短 QT 距離 $d(e, x)$ となる。BFS の実装においては、各状態を数値化し、ハッシュテーブルを用いて既知の状態であるかを判定する。なお、探索範囲は計算機資源に応じて上限深度 D_{\max} で打ち切るものとする。

4.2 貪欲法による符号語選択

ベースラインとして、単純な貪欲法を用いる。この手法は、探索フェーズで得られた状態リスト \mathcal{L} を、あらかじめ定められた順序（インデックス順）で走査し、条件を満たすものを順次符号語として採用していく決定論的なアルゴリズムである。

具体的な手順は以下の通りである。

1. **初期化:** 符号語集合 \mathcal{C} を空集合とする。
2. **候補の選択:** 状態リスト \mathcal{L} から、まだ判定を行っていない状態 x をインデックスの昇順に 1 つ選択する。
3. **距離検証:** 選択された候補 x と、既に採用された全ての符号語 $c \in \mathcal{C}$ との QT 距離 $d(x, c)$ を BFS を用いて計算する。
4. **採用判定:** 全ての $c \in \mathcal{C}$ に対して $d(x, c) \geq 3$ が成立すれば x を採用し、そうでなければ棄却する。
5. **反復:** リスト \mathcal{L} の全ての候補に対して判定が終了するまで繰り返す。

4.3 禁止リスト法による距離検証の高速化

候補となる状態 x が条件を満たすか否かの判定を、都度の距離計算ではなく、禁止リストを用いた集合演算に置き換えることで高速化を図る。

ある符号語 c が採用された際、その c から QT 距離 2 以下の範囲にある全ての状態は、以降新たな符号語として採用できない。そこで、採用された符号語の近傍状態を禁止リスト \mathcal{F} に追加し、新たな候補 x が \mathcal{F} に含まれるか否かで判定を行う。

$$\mathcal{F} = \{y \in \mathcal{G} \mid \exists c \in \mathcal{C}, d(c, y) < 3\} \quad (5)$$

この手法により、判定プロセスを平均計算量 $O(1)$ のハッシュテーブル検索に帰着させることができると、メモリ使用量が増大するというトレードオフが存在する。

4.4 部分群構造に着目した符号語選択（提案手法）

本研究では、大規模な符号構成を可能にするため、ルービックキューブの群構造を利用した効率的な候補選択戦略を提案する。具体的には、特定の部分群 S に着目する。この S は、

$$S = R\langle U, F \rangle R^{-1} = \{RgR^{-1} \mid g \in \langle U, F \rangle\} \quad (6)$$

と定義される。ここで、 $\langle U, F \rangle$ は U 面と F 面の回転によって生成される部分群であり、 R は R 面回転による操作を表す。この操作によって生成される部分群 S の要素は、元の群構造を保つつつ、状態グラフ \mathcal{G} 上では互いに QT 距離がある程度離れた状態で埋め込まれることが期待される。

この部分群 S の要素を符号語候補として優先的に探索することで、以下の効果を狙う。

- 棄却頻度の減少: S の要素が互いに QT 距離を保つ傾向があれば、符号語候補が既存の符号語との距離検証で棄却となる頻度が減少し、探索効率が向上する。

この戦略に基づくアルゴリズムでは、状態リスト \mathcal{L} の中から、この部分群 S に属する状態のみを抽出し、符号語選択を行う。

5 探索性能の評価

本章では、第 4 章で提案した各アルゴリズムの評価方法について述べる。本評価の目的は、単純な貪欲法と、提案手法である部分群構造を利用した探索法の性能を比較し、さらに禁止リスト法による高速化が符号構成数と計算資源に与える影響を定量的に明らかにすることである。本評価は、大規模なグラフ探索と大量の距離計算を伴うため、計算機の処理能力が結果に大きく影響する。使用したハードウェア仕様およびソフトウェア環境を表 1 に示す。また、使用したソースコードは付録 A に示した。

表 1 実行環境および開発環境

項目	仕様
OS	Microsoft Windows 11 Pro 24H2
CPU	AMD Ryzen 7 7800X3D (8-Core, 4.2 GHz)
RAM	DDR5-4800 64GB
Compiler	GCC 15.2.0
Options	<code>-O3 -march=native -funroll-loops</code>

5.1 評価条件とパラメータ設定

本節では、比較を行う3つの手法と、それぞれのパラメータ設定について述べる。各手法における探索深度 D_{\max} は、予備調査において現実的な実行時間およびメモリ容量 (64GB) の制約を超えない範囲で最大となるように設定した。設定一覧を表2に示す。

表2 比較手法とパラメータ設定

手法名称	D_{\max}	特徴
(1) 単純な貪欲法	5	全状態空間を対象、貪欲選択
(2) 部分群法	16	共役部分群内のみ探索
(3) 部分群法 + 禁止リスト	26	手法(2)を禁止リストで高速化

1. 単純な貪欲法 ($D_{\max} = 5$):

ルービックキューブの全状態空間を対象とするベースライン手法である。全状態空間においては、QT距離が1増えるごとに状態数が指数関数的に増大するため、 $D_{\max} = 5$ と設定した。これ以上の深さでは候補リスト \mathcal{L} が膨大となり、計算が現実的な時間内で終了しないためである。

2. 部分群法 ($D_{\max} = 16$):

第4章で提案した、特定の共役部分群に属する状態のみを候補とする手法である。部分群内の状態分布は疎であるため、十分な数の候補を得るために探索範囲を $D_{\max} = 16$ まで拡張した。

3. 部分群法 + 禁止リスト ($D_{\max} = 26$):

手法(2)に加え、禁止リスト法を適用して距離検証を高速化した手法である。検証の高速化により、さらに多くの候補状態を処理可能となるため、メモリ容量の上限に達しない最大の深度として $D_{\max} = 26$ を設定した。

5.2 結果

各手法の性能を定量的に評価するため、以下の2つの指標を計測した。

- **符号語数 N** : 最終的に構成された符号集合 \mathcal{C} の要素数。本研究の主目的であり、この値が大きいほど優れた符号構成法であると言える。
- **実行時間**: プログラムの開始から符号構成が完了するまでの時間。

表3に、各手法における探索候補数、最終的に得られた符号語数 N 、および実行時間を示す。

手法(1)においては、計算時間の都合上 $D_{\max} = 5$ で探索を打ち切った。一方、手法(2)および(3)では、計算資源の許す範囲でそれぞれ $D_{\max} = 16$ および $D_{\max} = 26$ まで探索を行った。

表3 各手法における結果

手法	候補状態数 $ \mathcal{L} $	符号語数 N	実行時間 (秒)
(1) 単純な貪欲法 ($D_{\max} = 5$)	105,046	8,960	706.995
(2) 部分群法 ($D_{\max} = 16$)	152,076	152,076	415.214
(3) 部分群法 + 禁止リスト ($D_{\max} = 26$)	4,787,847	4,787,847	92.9

6 考察

6.1 部分群構造の有効性について

候補状態集合 \mathcal{L} に対する最終的な符号語数 N の割合（採用率）に着目して比較を行うと、手法(1)が約 8.53% に留まったのに対し、手法(2)および手法(3)では 100% を達成した。本研究の探索範囲内において、部分群 S に属する状態は一度も棄却されることなく、全て符号語として採用された。この結果は、当該部分群 S が全状態空間内において、互いに距離 3 以上を保って疎に分布している可能性があることを示唆する。また、アルゴリズムの観点からは、距離検証による棄却という計算コストの無駄を排除できたことを意味し、探索効率の向上に寄与したと言える。

6.2 禁止リスト法の限界性能

手法(3)は実行時間の面では圧倒的に高速であった。表3に示す条件 ($D_{\max} = 26$) では、約 478 万個の符号語をわずか 92.9 秒で構成することに成功している。

さらに、本研究では計算機の物理メモリ (64GB) を最大限まで利用した場合の限界性能についても検証を行った。探索深度を $D_{\max} = 29$ まで拡張し、禁止リストのサイズがメモリ上限に達してスワッピングが発生するまで探索を行った結果、最終的に 43,074,266 個の符号語を構成することに成功した。これ以上の探索はスワッピングの頻度増加による急激な速度低下を招いたことから、本アルゴリズムにおける符号語数の上限は、計算時間ではなくメモリ容量によって律速されることがわかる。

6.3 誤り訂正符号としての評価

本研究で得られた最大の符号語数は、前述の通り $N \approx 4.3 \times 10^7$ であった。これは、ルービックキューブの全状態空間 4.3×10^{19} に対しては依然としてわずかな割合であるが、符号間 QT 距離 3 を保証する集合（1 誤り訂正符号）としては、グラフ探索アプローチで構成されたものとして大規模なものである。

情報理論の観点からこの結果を評価すると、

$$\lfloor \log_2(43,074,266) \rfloor = 25 \quad (7)$$

より、本手法を用いることで 25 ビットの情報を、1 手の操作ミスを訂正可能な状態でルービックキューブ上に記録できることを意味する。

7 まとめ

本研究では、ルービックキューブの状態空間を通信路とみなし、任意の 1 操作による誤りを訂正可能な符号、すなわち符号語間の最小距離が 3 以上となる符号語集合を構成することを目的とした。

巨大なグラフ上の符号構成において、単純な貪欲法では符号語数 N に対して $O(N^2)$ の計算量を要し、大規模な符号の探索が困難であることが課題であった。これに対し本研究では、計算効率を向上させるために以下の二つのアプローチを提案・実装した。第一に、ルービックキューブの持つ代数的な群構造に着目し、特定の共役部分群に属する状態を優先的に探索候補とする手法である。第二に、ハッシュテーブルを用いた禁止リスト法により、グラフ探索を伴う重い距離計算を、定数時間 $O(1)$ のメモリアクセスに置き換える手法である。

性能評価の結果、単純な貪欲法と比較して、提案手法は高い探索効率を示した。特に部分群法においては、候補とした状態が既存の符号語と衝突することなく 100% の確率で採用され、群構造を利用して探索空間を適切に限定することの有効性が実証された。また、禁止リスト法を併用することで、約 4300 万個 (4.3×10^7) の符号語を持つ 1 誤り訂正符号の構成に成功した。これは情報量に換算すると約 25 ビットに相当し、ルービックキューブ上に 25 ビットの情報を、1 手の操作ミスを許容できる形式で埋め込み可能であることを意味する。

本研究の成果は、パズルという親しみやすい題材を通して符号理論の概念を具現化しただけでなく、巨大なグラフにおける符号探索問題に対して、対象の持つ対称性や代数的構造を利用する効果が有効であることを示した点において意義がある。

謝辞

本研究を進めるにあたり、終始懇切丁寧なご指導とご鞭撻を賜りました、指導教員である信州大学工学部 西新 幹彦 淄教授に深く感謝の意を表します。先生には、研究テーマの選定からアルゴリズムの構築に関して多大なるご助言とご配慮をいただきました。

参考文献

- [1] David Singmaster. *Notes on Rubik's 'Magic Cube'*. Enslow Publishers, 1981.
- [2] Tom Davis. *Group Theory via Rubik's Cube*. geometer.org, 2006.
- [3] T. Rokicki, H. Kociemba, M. Davidson, and J. Dethridge. The Diameter of the Rubik's Cube Group is Twenty. *SIAM Review*, 56(4):645–670, 2014.

付録 A 探索プログラム

本研究で使用した、ルービックキューブの探索を行う C 言語プログラムのソースコードを以下に示す。

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <time.h>
4 #include <ctype.h>
5 #include <omp.h>
6 #include "uthash.h"
7 #define U_FACE_IDX 0 // 上面 (Up)
8 #define L_FACE_IDX 1 // 左面 (Left)
9 #define F_FACE_IDX 2 // 正面 (Front)
10 #define R_FACE_IDX 3 // 右面 (Right)
11 #define B_FACE_IDX 4 // 背面 (Back)
12 #define D_FACE_IDX 5 // 下面 (Down)
13 typedef struct Cubedata{
14     char cube[6][9];
15     char rotation[40];
16     char depth;
17 }Cubedata;
18 char rotationindex[12] = "UFRDBLufrdbl";
19 void simpleprintcube(char cube[6][9]){
20     for(int i=0;i<6;i++){
21         for(int j=0;j<9;j++){
22             printf("%d,",cube[i][j]);
23         }
24         printf("\n");
25     }
26     return;
27 }
28 void printcube(char cube[6][9]){
29     for(int i=0;i<9;i++){
30         printf("%02d ",cube[0][i]);
31         if(i%3==2)printf("\n");
32     }
33     for(int i=0;i<36;i++){
34         printf("%02d ",cube[(i/3)%4+1][(i/12)*3+i%3]);
35         if(i%12==11)printf("\n");
36     }
37     for(int i=0;i<9;i++){
38         printf("%02d ",cube[5][i]);
39         if(i%3==2)printf("\n");
40     }
41     return;
42 }
43 char printcolor(int num){
```

```

44     switch(num/9){
45         case 0: return 'W';
46         case 1: return 'R';
47         case 2: return 'B';
48         case 3: return 'O';
49         case 4: return 'G';
50         case 5: return 'Y';
51         default: return 'e';
52     }
53     return 'e';
54 }
55 void printcubecolor(char cube[6][9]){
56     for(int i=0;i<9;i++){
57         printf("%c",printcolor(cube[0][i]));
58         if(i%3==2)printf("\n");
59     }
60     for(int i=0;i<36;i++){
61         printf("%c",printcolor(cube[(i/3)%4+1][((i/12)*3)+i%3]));
62         if(i%12==11)printf("\n");
63     }
64     for(int i=0;i<9;i++){
65         printf("%c",printcolor(cube[5][i]));
66         if(i%3==2)printf("\n");
67     }
68 }
69 void printcubeh(char cube[6][9]){
70     for(int i=0;i<9;i++){
71         printf("%02d ",cube[0][i]+1);
72         if(i%3==2)printf("\n");
73     }
74     for(int i=0;i<36;i++){
75         printf("%02d ",cube[(i/3)%4+1][((i/12)*3)+i%3]+1);
76         if(i%12==11)printf("\n");
77     }
78     for(int i=0;i<9;i++){
79         printf("%02d ",cube[5][i]+1);
80         if(i%3==2)printf("\n");
81     }
82     return;
83 }
84 void copycube(const char src[6][9],char dist[6][9]){
85     for(int i=0;i<6;i++){
86         for(int j=0;j<9;j++){
87             dist[i][j]=src[i][j];
88         }
89     }
90     return;
91 }
92 void resetcube(char cube[6][9]){
93     for(int i=0;i<6;i++){

```

```

94         for(int j=0;j<9;j++){
95             cube[i][j]=i;
96         }
97     }
98     return;
99 }
100 void rotate_u(char cube[6][9]) {
101     char tempcube[6][9];
102     copycube(cube, tempcube);
103     cube[U_FACE_IDX][0] = tempcube[U_FACE_IDX][6];
104     cube[U_FACE_IDX][1] = tempcube[U_FACE_IDX][3];
105     cube[U_FACE_IDX][2] = tempcube[U_FACE_IDX][0];
106     cube[U_FACE_IDX][3] = tempcube[U_FACE_IDX][7];
107     cube[U_FACE_IDX][5] = tempcube[U_FACE_IDX][1];
108     cube[U_FACE_IDX][6] = tempcube[U_FACE_IDX][8];
109     cube[U_FACE_IDX][7] = tempcube[U_FACE_IDX][5];
110     cube[U_FACE_IDX][8] = tempcube[U_FACE_IDX][2];
111
112     int src_faces[] = {F_FACE_IDX, R_FACE_IDX, B_FACE_IDX, L_FACE_IDX};
113     int dst_faces[] = {L_FACE_IDX, F_FACE_IDX, R_FACE_IDX, B_FACE_IDX};
114
115
116
117     for (int i = 0; i < 4; ++i) {
118         for (int j = 0; j < 3; ++j) { // ステッカーインデックス 0, 1, 2 各面の上段()
119             cube[dst_faces[i]][j] = tempcube[src_faces[i]][j];
120         }
121     }
122     return;
123 }
124 void rotate_u_prime(char cube[6][9]){
125     char tempcube[6][9];
126     copycube(cube, tempcube);
127     cube[U_FACE_IDX][0] = tempcube[U_FACE_IDX][2];
128     cube[U_FACE_IDX][1] = tempcube[U_FACE_IDX][5];
129     cube[U_FACE_IDX][2] = tempcube[U_FACE_IDX][8];
130     cube[U_FACE_IDX][3] = tempcube[U_FACE_IDX][1];
131     cube[U_FACE_IDX][5] = tempcube[U_FACE_IDX][7];
132     cube[U_FACE_IDX][6] = tempcube[U_FACE_IDX][0];
133     cube[U_FACE_IDX][7] = tempcube[U_FACE_IDX][3];
134     cube[U_FACE_IDX][8] = tempcube[U_FACE_IDX][6];
135
136     int src_faces[] = {L_FACE_IDX, F_FACE_IDX, R_FACE_IDX, B_FACE_IDX};
137     int dst_faces[] = {F_FACE_IDX, R_FACE_IDX, B_FACE_IDX, L_FACE_IDX};
138
139
140     for (int i = 0; i < 4; ++i) {
141         for (int j = 0; j < 3; ++j) { // ステッカーインデックス 0, 1, 2 各面の上段()
142             cube[dst_faces[i]][j] = tempcube[src_faces[i]][j];
143         }
}

```

```

144     }
145     return;
146 }
147 void rotate_f(char cube[6][9]) {
148     char tempcube[6][9];
149     copycube(cube, tempcube);
150     cube[F_FACE_IDX][0] = tempcube[F_FACE_IDX][6];
151     cube[F_FACE_IDX][1] = tempcube[F_FACE_IDX][3];
152     cube[F_FACE_IDX][2] = tempcube[F_FACE_IDX][0];
153     cube[F_FACE_IDX][3] = tempcube[F_FACE_IDX][7];
154     cube[F_FACE_IDX][5] = tempcube[F_FACE_IDX][1];
155     cube[F_FACE_IDX][6] = tempcube[F_FACE_IDX][8];
156     cube[F_FACE_IDX][7] = tempcube[F_FACE_IDX][5];
157     cube[F_FACE_IDX][8] = tempcube[F_FACE_IDX][2];
158
159     int src_faces[] = {U_FACE_IDX, R_FACE_IDX, D_FACE_IDX, L_FACE_IDX};
160     int dst_faces[] = {R_FACE_IDX, D_FACE_IDX, L_FACE_IDX, U_FACE_IDX};
161
162     cube[3][0]=tempcube[0][6];
163     cube[3][3]=tempcube[0][7];
164     cube[3][6]=tempcube[0][8];
165
166     cube[5][2]=tempcube[3][0];
167     cube[5][1]=tempcube[3][3];
168     cube[5][0]=tempcube[3][6];
169
170     cube[1][8]=tempcube[5][2];
171     cube[1][5]=tempcube[5][1];
172     cube[1][2]=tempcube[5][0];
173
174     cube[0][6]=tempcube[1][8];
175     cube[0][7]=tempcube[1][5];
176     cube[0][8]=tempcube[1][2];
177
178     return;
179 }
180 void rotate_f_prime(char cube[6][9]){
181     char tempcube[6][9];
182     copycube(cube, tempcube);
183     cube[F_FACE_IDX][0] = tempcube[F_FACE_IDX][2];
184     cube[F_FACE_IDX][1] = tempcube[F_FACE_IDX][5];
185     cube[F_FACE_IDX][2] = tempcube[F_FACE_IDX][8];
186     cube[F_FACE_IDX][3] = tempcube[F_FACE_IDX][1];
187     cube[F_FACE_IDX][5] = tempcube[F_FACE_IDX][7];
188     cube[F_FACE_IDX][6] = tempcube[F_FACE_IDX][0];
189     cube[F_FACE_IDX][7] = tempcube[F_FACE_IDX][3];
190     cube[F_FACE_IDX][8] = tempcube[F_FACE_IDX][6];
191
192     int src_faces[] = {R_FACE_IDX, D_FACE_IDX, L_FACE_IDX, U_FACE_IDX};
193     int dst_faces[] = {U_FACE_IDX, R_FACE_IDX, D_FACE_IDX, L_FACE_IDX};

```

```

194
195
196     cube[0][6] = tempcube[3][0];
197     cube[0][7] = tempcube[3][3];
198     cube[0][8] = tempcube[3][6];
199
200     cube[1][8] = tempcube[0][6];
201     cube[1][5] = tempcube[0][7];
202     cube[1][2] = tempcube[0][8];
203
204     cube[3][0] = tempcube[5][2];
205     cube[3][3] = tempcube[5][1];
206     cube[3][6] = tempcube[5][0];
207
208     cube[5][2] = tempcube[1][8];
209     cube[5][1] = tempcube[1][5];
210     cube[5][0] = tempcube[1][2];
211
212     return;
213 }
214 void rotate_r(char cube[6][9]) {
215     char tempcube[6][9];
216     copycube(cube, tempcube);
217     cube[R_FACE_IDX][0] = tempcube[R_FACE_IDX][6];
218     cube[R_FACE_IDX][1] = tempcube[R_FACE_IDX][3];
219     cube[R_FACE_IDX][2] = tempcube[R_FACE_IDX][0];
220     cube[R_FACE_IDX][3] = tempcube[R_FACE_IDX][7];
221     cube[R_FACE_IDX][5] = tempcube[R_FACE_IDX][1];
222     cube[R_FACE_IDX][6] = tempcube[R_FACE_IDX][8];
223     cube[R_FACE_IDX][7] = tempcube[R_FACE_IDX][5];
224     cube[R_FACE_IDX][8] = tempcube[R_FACE_IDX][2];
225
226     int src_faces[] = {U_FACE_IDX, B_FACE_IDX, D_FACE_IDX, F_FACE_IDX};
227     int dst_faces[] = {B_FACE_IDX, D_FACE_IDX, F_FACE_IDX, U_FACE_IDX};
228
229     for (int i = 0; i < 4; ++i) {
230         for (int j = 0; j < 3; ++j) { // ステッカーインデックス 0, 1, 2 各面の上段()
231             if(dst_faces[i]==B_FACE_IDX)cube[dst_faces[i]][6-j*3] = tempcube
232 [src_faces[i]][j*3+2];
233             else if(src_faces[i]==B_FACE_IDX)cube[dst_faces[i]][j*3+2] =
234 tempcube[src_faces[i]][6-j*3];
235             else cube[dst_faces[i]][j*3+2] = tempcube[src_faces[i]][j*3+2];
236         }
237     }
238     return;
239 }
240 void rotate_r_prime(char cube[6][9]) {
241     char tempcube[6][9];
242     copycube(cube, tempcube);
243     cube[R_FACE_IDX][0] = tempcube[R_FACE_IDX][2];

```

```

242     cube[R_FACE_IDX][1] = tempcube[R_FACE_IDX][5];
243     cube[R_FACE_IDX][2] = tempcube[R_FACE_IDX][8];
244     cube[R_FACE_IDX][3] = tempcube[R_FACE_IDX][1];
245     cube[R_FACE_IDX][5] = tempcube[R_FACE_IDX][7];
246     cube[R_FACE_IDX][6] = tempcube[R_FACE_IDX][0];
247     cube[R_FACE_IDX][7] = tempcube[R_FACE_IDX][3];
248     cube[R_FACE_IDX][8] = tempcube[R_FACE_IDX][6];
249
250     int src_faces[] = {B_FACE_IDX, D_FACE_IDX, F_FACE_IDX, U_FACE_IDX};
251     int dst_faces[] = {U_FACE_IDX, B_FACE_IDX, D_FACE_IDX, F_FACE_IDX};
252
253     for (int i = 0; i < 4; ++i) {
254         for (int j = 0; j < 3; ++j) { // ステッカーインデックス 0, 1, 2 各面の上段()
255             if(dst_faces[i]==B_FACE_IDX)cube[dst_faces[i]][6-j*3] = tempcube
256 [src_faces[i]][j*3+2];
257             else if(src_faces[i]==B_FACE_IDX)cube[dst_faces[i]][j*3+2] =
258 tempcube[src_faces[i]][6-j*3];
259             else cube[dst_faces[i]][j*3+2] = tempcube[src_faces[i]][j*3+2];
260         }
261     }
262     return;
263 }
264 void rotate_l(char cube[6][9]) {
265     char tempcube[6][9];
266     copycube(cube, tempcube);
267     cube[L_FACE_IDX][0] = tempcube[L_FACE_IDX][6];
268     cube[L_FACE_IDX][1] = tempcube[L_FACE_IDX][3];
269     cube[L_FACE_IDX][2] = tempcube[L_FACE_IDX][0];
270     cube[L_FACE_IDX][3] = tempcube[L_FACE_IDX][7];
271     cube[L_FACE_IDX][5] = tempcube[L_FACE_IDX][1];
272     cube[L_FACE_IDX][6] = tempcube[L_FACE_IDX][8];
273     cube[L_FACE_IDX][7] = tempcube[L_FACE_IDX][5];
274     cube[L_FACE_IDX][8] = tempcube[L_FACE_IDX][2];
275
276     int src_faces[] = {U_FACE_IDX, F_FACE_IDX, D_FACE_IDX, B_FACE_IDX};
277     int dst_faces[] = {F_FACE_IDX, D_FACE_IDX, B_FACE_IDX, U_FACE_IDX};
278
279     for (int i = 0; i < 4; ++i) {
280         for (int j = 0; j < 3; ++j) { // ステッカーインデックス 0, 1, 2 各面の上段()
281             if(dst_faces[i]==B_FACE_IDX)cube[dst_faces[i]][8-j*3] = tempcube
282 [src_faces[i]][j*3];
283             else if(src_faces[i]==B_FACE_IDX)cube[dst_faces[i]][j*3] =
284 tempcube[src_faces[i]][8-j*3];
285             else cube[dst_faces[i]][j*3] = tempcube[src_faces[i]][j*3];
286         }
287     }
288     return;
289 }
290 void rotate_l_prime(char cube[6][9]) {
291     char tempcube[6][9];

```

```

288     copycube(cube, tempcube);
289     cube[L_FACE_IDX][0] = tempcube[L_FACE_IDX][2];
290     cube[L_FACE_IDX][1] = tempcube[L_FACE_IDX][5];
291     cube[L_FACE_IDX][2] = tempcube[L_FACE_IDX][8];
292     cube[L_FACE_IDX][3] = tempcube[L_FACE_IDX][1];
293     cube[L_FACE_IDX][5] = tempcube[L_FACE_IDX][7];
294     cube[L_FACE_IDX][6] = tempcube[L_FACE_IDX][0];
295     cube[L_FACE_IDX][7] = tempcube[L_FACE_IDX][3];
296     cube[L_FACE_IDX][8] = tempcube[L_FACE_IDX][6];
297
298     int src_faces[] = {F_FACE_IDX, D_FACE_IDX, B_FACE_IDX, U_FACE_IDX};
299     int dst_faces[] = {U_FACE_IDX, F_FACE_IDX, D_FACE_IDX, B_FACE_IDX};
300
301     for (int i = 0; i < 4; ++i) {
302         for (int j = 0; j < 3; ++j) { // ステッカーインデックス 0, 1, 2 各面の上段()
303             if(dst_faces[i]==B_FACE_IDX)cube[dst_faces[i]][8-j*3] = tempcube
304             [src_faces[i]][j*3];
305             else if(src_faces[i]==B_FACE_IDX)cube[dst_faces[i]][j*3] =
306             tempcube[src_faces[i]][8-j*3];
307             else cube[dst_faces[i]][j*3] = tempcube[src_faces[i]][j*3];
308         }
309     }
310     return;
311 }
312 void rotate_d(char cube[6][9]) {
313     char tempcube[6][9];
314     copycube(cube, tempcube);
315     cube[D_FACE_IDX][0] = tempcube[D_FACE_IDX][6];
316     cube[D_FACE_IDX][1] = tempcube[D_FACE_IDX][3];
317     cube[D_FACE_IDX][2] = tempcube[D_FACE_IDX][0];
318     cube[D_FACE_IDX][3] = tempcube[D_FACE_IDX][7];
319     cube[D_FACE_IDX][5] = tempcube[D_FACE_IDX][1];
320     cube[D_FACE_IDX][6] = tempcube[D_FACE_IDX][8];
321     cube[D_FACE_IDX][7] = tempcube[D_FACE_IDX][5];
322     cube[D_FACE_IDX][8] = tempcube[D_FACE_IDX][2];
323
324     int src_faces[] = {L_FACE_IDX, F_FACE_IDX, R_FACE_IDX, B_FACE_IDX};
325     int dst_faces[] = {F_FACE_IDX, R_FACE_IDX, B_FACE_IDX, L_FACE_IDX};
326
327     for (int i = 0; i < 4; ++i) {
328         for (int j = 0; j < 3; ++j) { // ステッカーインデックス 0, 1, 2 各面の上段()
329             cube[dst_faces[i]][j+6] = tempcube[src_faces[i]][j+6];
330         }
331     }
332     return;
333 }
334 void rotate_d_prime(char cube[6][9]) {
335     char tempcube[6][9];
336     copycube(cube, tempcube);

```

```

336     cube[D_FACE_IDX][0] = tempcube[D_FACE_IDX][2];
337     cube[D_FACE_IDX][1] = tempcube[D_FACE_IDX][5];
338     cube[D_FACE_IDX][2] = tempcube[D_FACE_IDX][8];
339     cube[D_FACE_IDX][3] = tempcube[D_FACE_IDX][1];
340     cube[D_FACE_IDX][5] = tempcube[D_FACE_IDX][7];
341     cube[D_FACE_IDX][6] = tempcube[D_FACE_IDX][0];
342     cube[D_FACE_IDX][7] = tempcube[D_FACE_IDX][3];
343     cube[D_FACE_IDX][8] = tempcube[D_FACE_IDX][6];
344
345     int src_faces[] = {F_FACE_IDX, R_FACE_IDX, B_FACE_IDX, L_FACE_IDX};
346     int dst_faces[] = {L_FACE_IDX, F_FACE_IDX, R_FACE_IDX, B_FACE_IDX};
347
348     for (int i = 0; i < 4; ++i) {
349         for (int j = 0; j < 3; ++j) { // ステッカーインデックス 0, 1, 2 各面の上段()
350             cube[dst_faces[i]][j+6] = tempcube[src_faces[i]][j+6];
351         }
352     }
353     return;
354 }
355 void rotate_b(char cube[6][9]){
356     char tempcube[6][9];
357     copycube(cube, tempcube);
358
359
360     cube[B_FACE_IDX][0] = tempcube[B_FACE_IDX][6];
361     cube[B_FACE_IDX][1] = tempcube[B_FACE_IDX][3];
362     cube[B_FACE_IDX][2] = tempcube[B_FACE_IDX][0];
363     cube[B_FACE_IDX][3] = tempcube[B_FACE_IDX][7];
364     cube[B_FACE_IDX][5] = tempcube[B_FACE_IDX][1];
365     cube[B_FACE_IDX][6] = tempcube[B_FACE_IDX][8];
366     cube[B_FACE_IDX][7] = tempcube[B_FACE_IDX][5];
367     cube[B_FACE_IDX][8] = tempcube[B_FACE_IDX][2];
368     int src_faces[] = {R_FACE_IDX, D_FACE_IDX, L_FACE_IDX, U_FACE_IDX};
369     int dst_faces[] = {U_FACE_IDX, R_FACE_IDX, D_FACE_IDX, L_FACE_IDX};
370
371     cube[0][2]=tempcube[3][8];
372     cube[3][8]=tempcube[5][6];
373     cube[5][6]=tempcube[1][0];
374     cube[1][0]=tempcube[0][2];
375
376     cube[0][1]=tempcube[3][5];
377     cube[3][5]=tempcube[5][7];
378     cube[5][7]=tempcube[1][3];
379     cube[1][3]=tempcube[0][1];
380
381     cube[0][0]=tempcube[3][2];
382     cube[3][2]=tempcube[5][8];
383     cube[5][8]=tempcube[1][6];
384     cube[1][6]=tempcube[0][0];
385
386     return;

```

```

386 }
387 void rotate_b_prime(char cube[6][9]) {
388     char tempcube[6][9];
389     copycube(cube, tempcube);
390     cube[B_FACE_IDX][0] = tempcube[B_FACE_IDX][2];
391     cube[B_FACE_IDX][1] = tempcube[B_FACE_IDX][5];
392     cube[B_FACE_IDX][2] = tempcube[B_FACE_IDX][8];
393     cube[B_FACE_IDX][3] = tempcube[B_FACE_IDX][1];
394     cube[B_FACE_IDX][5] = tempcube[B_FACE_IDX][7];
395     cube[B_FACE_IDX][6] = tempcube[B_FACE_IDX][0];
396     cube[B_FACE_IDX][7] = tempcube[B_FACE_IDX][3];
397     cube[B_FACE_IDX][8] = tempcube[B_FACE_IDX][6];
398
399     int src_faces[] = {U_FACE_IDX, R_FACE_IDX, D_FACE_IDX, L_FACE_IDX};
400     int dst_faces[] = {R_FACE_IDX, D_FACE_IDX, L_FACE_IDX, U_FACE_IDX};
401
402     cube[0][2]=tempcube[1][0];
403     cube[1][0]=tempcube[5][6];
404     cube[5][6]=tempcube[3][8];
405     cube[3][8]=tempcube[0][2];
406
407     cube[0][1]=tempcube[1][3];
408     cube[1][3]=tempcube[5][7];
409     cube[5][7]=tempcube[3][5];
410     cube[3][5]=tempcube[0][1];
411
412     cube[0][0]=tempcube[1][6];
413     cube[1][6]=tempcube[5][8];
414     cube[5][8]=tempcube[3][2];
415     cube[3][2]=tempcube[0][0];
416     return;
417 }
418 void rotate_char(char cube[6][9], char rotation[], int len){
419     for(int i=0;i<len;i++){
420         switch(rotation[i]){
421             case 'U':rotate_u(cube);break;
422             case 'u':rotate_u_prime(cube);break;
423             case 'R':rotate_r(cube);break;
424             case 'r':rotate_r_prime(cube);break;
425             case 'F':rotate_f(cube);break;
426             case 'f':rotate_f_prime(cube);break;
427             case 'D':rotate_d(cube);break;
428             case 'd':rotate_d_prime(cube);break;
429             case 'L':rotate_l(cube);break;
430             case 'l':rotate_l_prime(cube);break;
431             case 'B':rotate_b(cube);break;
432             case 'b':rotate_b_prime(cube);break;
433             default:printf("[Error] rotate error: undefined rotation");break
434         };
435     }
436 }

```

```

435     }
436     return;
437 }
438 //char rotationindex[12]={"UFRDBLufrdbl";
439 char lut_2t[156][6][9];
440 int lut_create(){
441     for(int i=0;i<12;i++){
442         char cube_state[6][9];
443         resetcube(cube_state);
444         rotate_char(cube_state,&rotationindex[i],1);
445         copycube(cube_state,lut_2t[i]);
446
447         for(int j=0;j<12;j++){
448             resetcube(cube_state);
449             rotate_char(cube_state,&rotationindex[i],1);
450             rotate_char(cube_state,&rotationindex[j],1);
451             copycube(cube_state,lut_2t[12+i*12+j]);
452         }
453     }
454
455     return 0;
456 }
457 void rotate_char_lut(char cube[6][9],char rotation[]){
458     char tempcube[6][9];
459     copycube(cube,tempcube);
460     int num=0;
461     for(int i=0;i<12;i++){
462         if(rotation[0]==rotationindex[i]&&strlen(rotation)==1){
463             num=i;break;
464         }
465         for(int j=0;j<12;j++){
466             if(rotation[0]==rotationindex[i]&&rotation[1]==rotationindex[j])
467             {
468                 num=12+i*12+j;
469             }
470         }
471     }
472     for(int i=0;i<6;i++){
473         for(int j=0;j<9;j++){
474             cube[i][j]=tempcube[lut_2t[num][i][j]/9][lut_2t[num][i][j]%9];
475         }
476     }
477
478     return;
479 }
480 void cubedata_print(Cubedata data[],int indexnum){
481     printcube(data[indexnum].cube);
482     printf("  data[%d],rotation='%s',depth=%d\n",indexnum,data[indexnum].
483     rotation,data[indexnum].depth);
484
485     return;
486 }

```

```

483 }
484 void cubedata_list(Cubedata data[], int len){
485     for(int i=0;i<len;i++){
486         printf("data[%d], rotation=%s, depth=%d\n",i,data[i].rotation,data[i]
487             ].depth);
488     }
489 }
490 int cubedata_allclear(Cubedata data[], int len){
491     for(int i=0;i<len;i++){
492         resetcube(data[i].cube);
493         for(int j=0;j<40;j++)data[i].rotation[j]='\0';
494         data[i].depth=-1;
495     }
496     return 0;
497 }
498 int cubedata_delete(Cubedata data[], int indexnum){
499     resetcube(data[indexnum].cube);
500     for(int j=0;j<40;j++)data[indexnum].rotation[j]='\0';
501     data[indexnum].depth=-1;
502
503     return 0;
504 }
505 void cubedata_write(Cubedata data[], int indexnum, Cubedata src){
506     copycube(src.cube,data[indexnum].cube);
507     strcpy(data[indexnum].rotation,src.rotation);
508     data[indexnum].depth=src.depth;
509     return;
510 }
511 Cubedata cubedata_create(const char cube[6][9],char rotation[],char depth){
512     Cubedata data;
513     copycube(cube,data.cube);
514     strcpy(data.rotation,rotation);
515     data.depth=depth;
516     return data;
517 }
518 #define BFS_TARGET_DEPTH 29
519 #define MAX_DATA 100000000
520 Cubedata *maindata=NULL;
521 #define HASH_TABLE_SIZE 225519191
522
523 // --- 1. ハッシュテーブルの構造定義とグローバル変数 ---
524
525 #include <stdlib.h> // malloc, のために追加free
526
527 // ハッシュテーブルの各エントリが持つノードの構造体
528 // このノードが連結リスト（チェイン）を形成する
529 typedef struct HashNode {
530     int maindata_idx;           // 配列におけるノードのインデックスmaindata
531     struct HashNode* next;    // 同じハッシュ値を持つ次のノードへのポインタ

```

```

532     } HashNode;
533
534     // ハッシュテーブル本体。各要素はチェインの先頭ノードへのポインタ
535     static HashNode** hash_table = NULL;
536
537
538
539     // 以前の visited_states は不要になるので削除します
540     // static int visited_states[HASH_TABLE_SIZE] = {0}; // ← 削除
541
542     // メモリ解放用の関数 (プログラム終了時に呼ぶ)
543     void free_hash_table() {
544         if (hash_table == NULL) return;
545         for (int i = 0; i < HASH_TABLE_SIZE; i++) {
546             HashNode* current = hash_table[i];
547             while (current != NULL) {
548                 HashNode* temp = current;
549                 current = current->next;
550                 free(temp);
551             }
552             hash_table[i] = NULL;
553         }
554     }
555
556 /**
557 * @brief キューブの状態からハッシュ値を計算する (Polynomial rolling hash)
558 * @param cube 計算対象のキューブ状態
559 * @return unsigned int 計算されたハッシュ値
560 */
561 unsigned int calculate_hash(const char cube[6][9]) {
562     unsigned int hash = 0;
563     // は一般的に使われる素数31
564     const unsigned int prime = 31;
565
566     for (int i = 0; i < 6; i++) {
567         for (int j = 0; j < 9; j++) {
568             // (hash * prime + value) % size
569             // 言語のCは負の値を返すことがあるため、正の値になるように調整%
570             hash = (hash * prime + cube[i][j]);
571         }
572     }
573     return hash % HASH_TABLE_SIZE;
574 }
575 #define FNV_PRIME_32 167777619
576 #define FNV_OFFSET_BASIS_32 2166136261U
577
578 unsigned int calculate_hash_fnv1a(const char cube[6][9]) {
579     // FNV-1で推奨される初期値と素数a (32-bit)
580     unsigned int hash = 0x811c9dc5;
581     const unsigned int FNV_PRIME = 0x01000193;

```

```

582
583     for (int i = 0; i < 6; i++) {
584         for (int j = 0; j < 9; j++) {
585             // (1) ハッシュ値と入力データをするXOR
586             hash ^= (unsigned char)cube[i][j];
587             // (2) 素数を掛ける
588             hash *= FNV_PRIME;
589         }
590     }
591
592     hash ^= (hash >> 16);
593     hash *= 0x85ebca6b;
594     hash ^= (hash >> 13);
595     hash *= 0xc2b2ae35;
596     hash ^= (hash >> 16);
597
598     return hash % HASH_TABLE_SIZE;
599 }
600
601
602 /**
603 * @brief 幅優先探索(BFS)を用いて、初期状態から指定された深さまでの全ての状態を生成する
604 *
605 * グローバル配列「maindata」をキューとして使用し、探索結果をその配列に直接格納します。
606 * この実装は、同じ面を連続して回転させるような無駄な手順を枝刈り（スキップ）することで、
607 * 探索の効率を向上させています。
608 *
609 * @return int 生成された総ノード数
610 */
611 //short edge_from_idx[MAX_DATA];
612 //short edge_to_idx[MAX_DATA];
613 int edgenum=0;
614
615 /**
616 * @brief 幅優先探索(BFS)を用いて、初期状態から指定された深さまでの全ての状態を生成する
617 *
618 * この関数は、探索結果として以下のつのグローバル変数を更新します。3
619 * 1. maindata[]: 発見したユニークなノード状態()の情報を格納する配列
620 * 2. edge_from_idx[], edge_to_idx[]: ノード間の接続情報エッジ()をインデックスで記録する
621 * 配列
622 * 3. edgenum: 発見したエッジの総数
623 *
624 * @return int 生成された総ノード数
625 */
626 int bfs_searchall() {
627     int head = 0; // キューの先頭 読み込み位置()
628     int tail = 0; // キューの末尾 書き込み位置()
629
630     // グローバル変数を初期化

```

```

631     edgenum = 0;
632     memset(visited_states, 0, sizeof(visited_states));
633
634     // 1. 初期ノードの準備
635     char initial_cube[6][9];
636     resetcube(initial_cube);
637     Cubedata initial_node = cubedata_create(initial_cube, "", 0);
638
639     // 2. 初期ノードをキューと訪問済みリストに追加
640     unsigned int initial_hash = calculate_hash_fnv1a(initial_node(cube));
641
642     // visited_states には maindata のインデックスを格納+1 は未訪問を意味するため(0)
643     visited_states[initial_hash] = tail + 1;
644
645     cubedata_write(maindata, tail, initial_node);
646     tail++;
647
648     // 3. 探索ループ キューが空になるまで()
649     while (head < tail) {
650         // ★ 変更点 1: 現在処理しているノードの「インデックス」を取得
651         // これがエッジの始点(from)になる。
652         int current_idx = head;
653         Cubedata current_node = maindata[current_idx];
654         head++; // キューを進める
655
656         // 深さ制限に達したら、このノードからは探索しない
657         if (current_node.depth >= BFS_TARGET_DEPTH) {
658             continue;
659         }
660
661         short child_depth = current_node.depth + 1;
662
663         // 4. 通りの回転を試して子ノードを生成12
664         for (int i = 0; i < 12; i++) {
665             // 配列が満杯なら探索を中断
666             if (tail >= MAX_DATA || edgenum >= MAX_DATA) {
667                 fprintf(stderr, 警告": または配列が満杯です。探索を中断します。
maindataedge\n");
668                 return tail;
669             }
670
671             char current_move_char = rotationindex[i];
672
673             // 子ノードの状態を生成
674             char temp_child_cube[6][9];
675             copycube(current_node(cube), temp_child_cube);
676             rotate_char(temp_child_cube, &current_move_char, 1);
677
678             // 子ノードのハッシュを計算し、訪問済みかチェック
679             unsigned int child_hash = calculate_hash_fnv1a(temp_child_cube);

```

```

680     int existing_node_idx = visited_states[child_hash] - 1; // して
681     // ベースのインデックスに戻す-10
682
683     // ★ 変更点 2: エッジの始点を「インデックス」で記録
684     edge_from_idx[edgenum] = current_idx;
685
686     if (existing_node_idx >= 0) {
687         // ケースA: 子ノードは既に訪問済みの場合
688
689         // ★ 変更点 3A: エッジの終点を「既存ノードのインデックス」で記録
690         edge_to_idx[edgenum] = existing_node_idx;
691         edgenum++;
692
693         continue; // この子ノードはキューに追加せず、次の回転へ
694
695     } else {
696         // ケースB: 子ノードが未訪問の場合
697
698         // ★ 変更点 3B: エッジの終点を「これから追加する新しいノードのインデック
699         // ス(tail)」で記録
700         edge_to_idx[edgenum] = tail;
701         edgenum++;
702
703         // 新しいノードを訪問済みリストに登録
704         visited_states[child_hash] = tail + 1;
705
706         // に格納するために、回転履歴文字列を作成maindata
707         char child_rotation_history[30];
708         strcpy(child_rotation_history, current_node.rotation);
709         int current_len = strlen(child_rotation_history);
710         if (current_len < sizeof(child_rotation_history) - 1) {
711             child_rotation_history[current_len] = current_move_char;
712             child_rotation_history[current_len + 1] = '\0';
713         }
714
715         // 新しいノードをキューの末尾に追加
716         Cubedata child_node = cubedata_create(temp_child_cube,
717         child_rotation_history, child_depth);
718         cubedata_write(maindata, tail, child_node);
719         tail++; // キューの末尾を更新
720     }
721
722     return tail; // 生成された総ノード数を返す
723 }/*
724 /**
725 * @brief 幅優先探索(BFS)を用いて、初期状態から指定された深さまでの全ての状態を生成する
726 *
727 * ハッシュ衝突を正しく処理するチェイニング法(連鎖法)を用いたハッシュテーブルを使用します。
728 */

```

```

728 * @return int 生成された総ノード数
729 */
730 void check_hash_table_duplicates() {
731     printf("Checking for duplicates in the hash table chains...\n");
732     for (int i = 0; i < HASH_TABLE_SIZE; i++) {
733         HashNode* p1 = hash_table[i];
734         if (p1 == NULL || p1->next == NULL) continue;
735
736         // チェイン内の各ノードを比較
737         while (p1 != NULL) {
738             HashNode* p2 = p1->next;
739             while (p2 != NULL) {
740                 if (memcmp(maindata[p1->maindata_idx].cube, maindata[p2->
741 maindata_idx].cube, sizeof(maindata[0].cube)) == 0) {
742                     printf("DUPLICATE STATE FOUND IN HASH CHAIN! hash_key=%d
743 , idx1=%d, idx2=%d\n",
744                         i, p1->maindata_idx, p2->maindata_idx);
745                 }
746                 p2 = p2->next;
747             }
748             p1 = p1->next;
749         }
750     }
751     printf("Hash table check finished.\n");
752 }
753 int bfs_searchall_with_collision_handling() {
754     int head = 0; // キューの先頭 読み込み位置()
755     int tail = 0; // キューの末尾 書き込み位置()
756
757     // グローバル変数を初期化
758     edgenum = 0;
759     free_hash_table(); // 以前の実行結果が残らないようにクリア
760
761     // 1. 初期ノードの準備
762     char initial_cube[6][9];
763     resetcube(initial_cube);
764     Cubedata initial_node = cubedata_create(initial_cube, "", 0);
765
766     // 2. 初期ノードをキューに追加し、ハッシュテーブルに登録
767     // に追加maindata
768     cubedata_write(maindata, tail, initial_node);
769
770     // ハッシュテーブルに登録
771     unsigned int initial_hash = calculate_hash_fnv1a(initial_node(cube));
772     HashNode* new_hash_node = (HashNode*)malloc(sizeof(HashNode));
773     if (new_hash_node == NULL) {
774         fprintf(stderr, "メモリ確保エラー\n");
775         return 0;
776     }
777     new_hash_node->maindata_idx = tail; // 現在のインデックスは0

```

```

776     new_hash_node->next = hash_table[initial_hash];
777     hash_table[initial_hash] = new_hash_node;
778
779     tail++;
780
781     // 3. 探索ループ キューが空になるまで()
782     while (head < tail) {
783         int current_idx = head;
784         Cubedata current_node = maindata[current_idx]; // コピーが発生するが、ここ
784         では許容
785         head++;
786
787         if (current_node.depth >= BFS_TARGET_DEPTH) {
788             continue;
789         }
790
791         short child_depth = current_node.depth + 1;
792
793         // 4. 通りの回転を試して子ノードを生成12
794         for (int i = 0; i < 12; i++) {
795             if (tail >= MAX_DATA || edgenum >= MAX_DATA) {
796                 fprintf(stderr, "警告： または配列が満杯です。探索を中断します。
796                 maindataedge\n");
797                 return tail;
798             }
799
800             char current_move_char = rotationindex[i];
801
802             char temp_child_cube[6][9];
803             copycube(current_node(cube, temp_child_cube);
804             rotate_char(temp_child_cube, &current_move_char, 1);
805
806             // --- ここからが最重要部分 ---
807
808             // 4a. 子ノードのハッシュを計算し、訪問済みかチェック
809             unsigned int child_hash = calculate_hash_fnv1a(temp_child_cube);
810
811             // 4b. チェインをたどって、本当に同じ状態が存在するか確認
812             int existing_node_idx = -1;
813             HashNode* current_chain = hash_table[child_hash];
814             while (current_chain != NULL) {
815                 // ハッシュ値が同じでも、実際のデータが同じかで比較memcmp
816                 if (memcmp(maindata[current_chain->maindata_idx].cube,
816                 temp_child_cube, sizeof(temp_child_cube)) == 0) {
817                     existing_node_idx = current_chain->maindata_idx;
818                     break;
819                 }
820                 current_chain = current_chain->next;
821             }
822
823             // 4c. エッジを記録

```

```

824     //edge_from_idx[edgenum] = current_idx;
825
826     if (existing_node_idx >= 0) {
827         // ケースA: 子ノードは既に訪問済みの場合
828         //edge_to_idx[edgenum] = existing_node_idx;
829         edgenum++;
830
831     } else {
832         // ケースB: 子ノードが未訪問の場合
833
834         // 新しいノードを追加maindata
835         // 回転履歴はメモリを食うので、ここでは空にしておくことも可能
836         // 必要なら後で復元する
837
838         char child_rotation_history[30];
839         strcpy(child_rotation_history, current_node.rotation);
840         int current_len = strlen(child_rotation_history);
841         if (current_len < sizeof(child_rotation_history) - 1) {
842             child_rotation_history[current_len] = current_move_char;
843             child_rotation_history[current_len + 1] = '\0';
844         }
845
846         Cubedata child_node = cubedata_create(temp_child_cube,
847         child_rotation_history, child_depth);
848         cubedata_write(maindata, tail, child_node);
849
850         // 新しいノードの情報をハッシュテーブルのチェインの先頭に追加
851         HashNode* new_node_for_hash = (HashNode*)malloc(sizeof(
852             HashNode));
853         if (new_node_for_hash == NULL) {
854             fprintf(stderr, "メモリ確保エラー\n");
855             return tail;
856         }
857         new_node_for_hash->maindata_idx = tail;
858         new_node_for_hash->next = hash_table[child_hash];
859         hash_table[child_hash] = new_node_for_hash;
860
861         // エッジの終点を記録
862         //edge_to_idx[edgenum] = tail;
863         edgenum++;
864
865         // キューの末尾を更新
866         tail++;
867     }
868     // --- ここまでが最重要部分 ---
869 }
870 check_hash_table_duplicates();
871 return tail; // 生成された総ノード数を返す
872 }

```

```

872 // 配列をファイルに書き込む関数Cubedata
873 // data: 書き込む配列へのポインタCubedata
874 // num_elements: 配列内の要素数
875 // filename: 書き込むファイルの名前
876 // 戻り値: 成功した場合は0-1
877 int write_cubedata_to_file(const Cubedata data[], int num_elements, const
878     char *filename) {
879     FILE *outfile;
880
881     // ファイルをバイナリ書き込みモードで開く ("wb")
882     outfile = fopen(filename, "wb");
883     if (outfile == NULL) {
884         perror("Error opening file for writing");
885         return -1; // ファイルオープン失敗
886     }
887
888     // を使用して配列全体を一度に書き込むfwrite
889     // fwrite書き込むデータのポインタ(, 各要素のサイズ, 要素数, ファイルポインタ)
890     // 戻り値は実際に書き込まれた要素数
891     size_t items_written = fwrite(data, sizeof(Cubedata), num_elements,
892     outfile);
893
894     if (items_written != num_elements) {
895         fprintf(stderr, "Error writing data to file. Wrote %zu of %d items.\n",
896         items_written, num_elements);
897         fclose(outfile);
898         return -1; // 書き込みエラー
899     }
900
901     // ファイルを閉じる
902     if (fclose(outfile) != 0) {
903         perror("Error closing file");
904         return -1; // ファイルクローズエラー データは書き込まれている可能性あり()
905     }
906
907     printf("Successfully wrote %d Cubedata elements to %s\n", num_elements,
908     filename);
909     return 0; // 成功
910 }
911
912 // オプション() 配列をファイルから読み込む関数Cubedata テスト用()
913 int read_cubedata_from_file(Cubedata data[], int num_elements_to_read, const
914     char *filename) {
915     FILE *infile;
916
917     // ファイルをバイナリ読み込みモードで開く ("rb")
918     infile = fopen(filename, "rb");
919     if (infile == NULL) {
920         perror("Error opening file for reading");
921         return -1;

```

```

917     }
918
919     // を使用してファイルからデータを読み込むfread
920     size_t items_read = fread(data, sizeof(Cubedata), num_elements_to_read,
921     infile);
922
923     if (items_read != num_elements_to_read) {
924         // ファイルの終端に達した場合やエラーの場合がある
925         if (feof(infile)) {
926             fprintf(stderr, "Error reading data: unexpected end of file.
927             Read %zu of %d items.\n", items_read, num_elements_to_read);
928         } else if (ferror(infile)) {
929             perror("Error reading data from file");
930         }
931         fclose(infile);
932         return -1;
933     }
934
935     // ファイルを閉じる
936     if (fclose(infile) != 0) {
937         perror("Error closing file after reading");
938         // データは読み込んでいる可能性があるので、ここではエラーを返さない選択肢もある
939     }
940
941     printf("Successfully read %zu Cubedata elements from %s\n", items_read,
942     filename);
943     return 0; // 成功
944 }
945
946 /*
947 int dotter(Cubedata data[], int elements, const char *filename){
948
949     char nodename[elements][30];
950
951     // generate node
952     strcpy(nodename[0], "e");
953     for(int i=1;i<elements;i++){
954         strcpy(nodename[i], data[i].rotation);
955     }
956
957     FILE *outfile;
958
959     // ファイルをバイナリ書き込みモードで開く ("wb")
960     outfile = fopen(filename, "w");
961     if (outfile == NULL) {
962         perror("Error opening file for writing");
963         return -1; // ファイルオープン失敗
964     }
965
966     fprintf(outfile, "digraph G {\n");

```

```

964     fprintf(outfile, "  graph [overlap=scale];\n");
965     fprintf(outfile, "  node [style=filled];\n");
966     for(int i=0; i<elements;i++){
967         if(strlen(nodename[i])==1)fprintf(outfile, "  %s [fillcolor=red];\n",
968         ,nodename[i]);
969         if(strlen(nodename[i])==2)fprintf(outfile, "  %s [fillcolor=blue];\n",
970         ,nodename[i]);
971         if(strlen(nodename[i])==3)fprintf(outfile, "  %s [fillcolor=
972         lightblue];\n",nodename[i]);
973         if(strlen(nodename[i])==0)break;
974     }
975
976     for(int i=1; i<elements;i++){
977         fprintf(outfile, "  %s -> %s;\n",edgefrom[i],edgeto[i]);
978         if(strlen(edgefrom[i])==0&&strlen(edgeto[i])==0)break;
979     }
980     fprintf(outfile, "}\n");
981     fclose(outfile);
982     printf("Successfully wrote %d Cubedata elements to %s\n", elements,
983     filename);
984     printf("[DOTTER] Success.");
985     return 0;
986 }
987 */
988 */
989 int dotter(Cubedata data[], int node_count, int edge_count, const char *
990             filename){
991     FILE *outfile;
992
993     outfile = fopen(filename, "w");
994     if (outfile == NULL) {
995         perror("Error opening file for writing");
996         return -1;
997     }
998
999     fprintf(outfile, "graph G {\n");
1000     fprintf(outfile, "  graph [overlap=scale, splines=true];\n");
1001     fprintf(outfile, "  node [style=filled, shape=circle, fixedsize=true,
1002             width=0.6];\n");
1003
1004     // --- ノード定義 ---
1005     // 初期ノード "e" 空文字列の代わりに()
1006     fprintf(outfile, "  e [label=\"\", fillcolor=gold];\n");
1007
1008     // 深さごとのノード
1009     for(int i = 1; i < node_count; i++){
1010         const char* color;
1011         switch(data[i].depth) {
1012             case 1: color = "lightcoral"; break;
1013             case 2: color = "lightblue"; break;

```

```

1008         case 3: color = "lightgreen"; break;
1009         default: color = "gray"; break;
1010     }
1011     // ノード名に回転履歴を、ラベルにも同じものを表示
1012     fprintf(outfile, " %s [label=\"%s\", fillcolor=%s];\n", data[i].
1013             rotation, data[i].rotation, color);
1014 }
1015
1016 // --- エッジ定義 ---
1017 for(int i = 0; i < edge_count; i++){
1018     // ノード名が空文字列の場合は "e" として扱う
1019     const char* from_node = (strlen(edgefrom[i]) == 0) ? "e" : edgefrom[i];
1020     const char* to_node = (strlen(edgeto[i]) == 0) ? "e" : edgeto[i];
1021     fprintf(outfile, " %s -- %s;\n", from_node, to_node);
1022 }
1023
1024 fprintf(outfile, "}\n");
1025 fclose(outfile);
1026 printf("Successfully wrote %d nodes and %d edges to %s\n", node_count,
1027        edge_count, filename);
1028 printf("[DOTTER] Success.\n");
1029 return 0;
1030 }
1031 */
1032 // 各ノードが持つ隣接ノードのリスト
1033 /*
1034 typedef struct AdjacencyList {
1035     int neighbors[12]; // 隣接ノードの配列におけるインデックスを格納maindata
1036     int neighbor_count;
1037 } AdjacencyList;
1038
1039 // 全ノード分の隣接リスト
1040 AdjacencyList adj[MAX_DATA];
1041
1042 void build_adjacency_list(int node_count, int edge_count) {
1043     // 初期化
1044     for (int i = 0; i < node_count; i++) {
1045         adj[i].neighbor_count = 0;
1046     }
1047
1048     // エッジ情報インデックス()から直接リストを構築
1049     for (int i = 0; i < edge_count; i++) {
1050         int from_idx = edge_from_idx[i];
1051         int to_idx = edge_to_idx[i];
1052
1053         // from -> to を追加 (重複チェック付き)
1054         char exists = 0;
1055         for (int k = 0; k < adj[from_idx].neighbor_count; k++) {
1056             if (adj[from_idx].neighbors[k] == to_idx) {

```

```

1055             exists = 1;
1056             break;
1057         }
1058     }
1059     if (!exists && adj[from_idx].neighbor_count < 12) {
1060         adj[from_idx].neighbors[adj[from_idx].neighbor_count++] = to_idx
1061     ;
1062     }
1063
1064     // to -> from を追加 (重複チェック付き)
1065     exists = 0;
1066     for (int k = 0; k < adj[to_idx].neighbor_count; k++) {
1067         if (adj[to_idx].neighbors[k] == from_idx) {
1068             exists = 1;
1069             break;
1070         }
1071     }
1072     if (!exists && adj[to_idx].neighbor_count < 12) {
1073         adj[to_idx].neighbors[adj[to_idx].neighbor_count++] = from_idx;
1074     }
1075     printf("Adjacency list has been built successfully.\n");
1076 }
1077
1078 // とのにおけるインデックスを引数とするcube1cube2maindata
1079 int is_distance_at_least_3_improved(int idx1, int idx2) {
1080
1081     // 距離 0 のチェック
1082     if (idx1 == idx2) return 0; // false
1083
1084     // 距離 1 のチェック
1085     // の隣接ノードにが含まれているか調べるidx1idx2
1086     for (int i = 0; i < adj[idx1].neighbor_count; i++) {
1087         if (adj[idx1].neighbors[i] == idx2) {
1088             return 0; // false, 距離1
1089         }
1090     }
1091
1092     // 距離 2 のチェック
1093     // の各隣接ノードidx1(neighbor1)について、その隣接ノード(neighbor2)にが含まれているか調
1094     // べるidx2
1095     for (int i = 0; i < adj[idx1].neighbor_count; i++) {
1096         int neighbor1_idx = adj[idx1].neighbors[i];
1097
1098         // の隣接ノードを調べるneighbor1
1099         for (int j = 0; j < adj[neighbor1_idx].neighbor_count; j++) {
1100             int neighbor2_idx = adj[neighbor1_idx].neighbors[j];
1101             if (neighbor2_idx == idx2) {
1102                 return 0; // false, 距離2
1103             }
1104         }
1105     }
1106 }
```

```

1103     }
1104 }
1105
1106 // 上記のいずれにも当てはまらなければ、距離は以上3
1107     return 1; // true
1108 }/*
1109
1110 // greedy() が生成するデータ
1111 int codeword_indices[MAX_DATA]; // 見つけた符号語におけるインデックスを格納maindata
1112 int codeword_count = 0; // 見つけた符号語の総数
1113
1114 int reverse_rotation(char in[]) {
1115
1116 // --- ^^e2^^91^^a0 安全性のためのチェック ---
1117 if (in == NULL) {
1118     return -1;
1119 }
1120
1121 // --- ^^e2^^91^^a1 文字列の反転処理(前回と同じ) ---
1122 int length = strlen(in);
1123 if (length == 0) {
1124     return 0; // 空文字列なら何もしない
1125 }
1126
1127 int start = 0;
1128 int end = length - 1;
1129 char temp;
1130
1131 while (start < end) {
1132     temp = in[start];
1133     in[start] = in[end];
1134     in[end] = temp;
1135
1136     start++;
1137     end--;
1138 }
1139
1140 // --- ^^e2^^91^^a2 大文字と小文字の入れ替え処理(ここからが追加部分) ---
1141 for (int i = 0; i < length; i++) {
1142     // 現在の文字を取得
1143     char c = in[i];
1144
1145     // もし大文字なら小文字に変換
1146     if (isupper(c)) {
1147         in[i] = tolower(c);
1148     }
1149     // もし小文字なら大文字に変換
1150     else if (islower(c)) {
1151         in[i] = toupper(c);
1152     }

```

```

1153         // アルファベット以外 (数字、記号、スペースなど) は何もしない
1154     }
1155
1156     // --- ^^e2^^91^^a3 成功を通知 ---
1157     return 0;
1158 }
1159
1160 int compare_cubedata(const void* a, const void* b) {
1161     Cubedata* cubeA = (Cubedata*)a;
1162     Cubedata* cubeB = (Cubedata*)b;
1163     return memcmp(cubeA->cube, cubeB->cube, sizeof(cubeA->cube));
1164 }
1165 int bfs_uf() {
1166     int head = 0; // キューの先頭 読み込み位置()
1167     int tail = 0; // キューの末尾 書き込み位置()
1168
1169     // グローバル変数を初期化
1170     edgenum = 0;
1171     free_hash_table(); // 以前の実行結果が残らないようにクリア
1172
1173     // 1. 初期ノードの準備
1174     char initial_cube[6][9];
1175     resetcube(initial_cube);
1176     Cubedata initial_node = cubedata_create(initial_cube, "", 0);
1177
1178     // 2. 初期ノードをキューに追加し、ハッシュテーブルに登録
1179     // に追加maindata
1180     cubedata_write(maindata, tail, initial_node);
1181
1182     // ハッシュテーブルに登録
1183     unsigned int initial_hash = calculate_hash_fnv1a(initial_node.cube);
1184     HashNode* new_hash_node = (HashNode*)malloc(sizeof(HashNode));
1185     if (new_hash_node == NULL) {
1186         fprintf(stderr, "メモリ確保エラー\n");
1187         return 0;
1188     }
1189     new_hash_node->maindata_idx = tail; // 現在のインデックスは0
1190     new_hash_node->next = hash_table[initial_hash];
1191     hash_table[initial_hash] = new_hash_node;
1192
1193     tail++;
1194
1195     // 3. 探索ループ キューが空になるまで()
1196     while (head < tail) {
1197         int current_idx = head;
1198         Cubedata current_node = maindata[current_idx]; // コピーが発生するが、ここ
1199         // では許容
1200         head++;
1201
1202         if (current_node.depth >= BFS_TARGET_DEPTH) {

```

```
1202         continue;
1203     }
1204
1205     short child_depth = current_node.depth + 1;
1206
1207     // 4. 通りの回転を試して子ノードを生成12
1208     for (int i = 0; i < 2; i++) {
1209         if (tail >= MAX_DATA || edgenum >= MAX_DATA) {
1210             fprintf(stderr, "警告: または配列が満杯です。探索を中断します。
1211 maindataedge\n");
1212             return tail;
1213         }
1214
1215         char current_move_char = rotationindex[i];
1216
1217         char temp_child_cube[6][9];
1218         copycube(current_node(cube, temp_child_cube);
1219         rotate_char(temp_child_cube, &current_move_char, 1);
1220
1221         // --- ここからが最重要部分 ---
1222
1223         // 4a. 子ノードのハッシュを計算し、訪問済みかチェック
1224         unsigned int child_hash = calculate_hash_fnv1a(temp_child_cube);
1225
1226         // 4b. チェインをたどって、本当に同じ状態が存在するか確認
1227         int existing_node_idx = -1;
1228         HashNode* current_chain = hash_table[child_hash];
1229         while (current_chain != NULL) {
1230             // ハッシュ値が同じでも、実際のデータが同じかで比較memcmp
1231             if (memcmp(maindata[current_chain->maindata_idx].cube,
1232             temp_child_cube, sizeof(temp_child_cube)) == 0) {
1233                 existing_node_idx = current_chain->maindata_idx;
1234                 break;
1235             }
1236             current_chain = current_chain->next;
1237         }
1238
1239         // 4c. エッジを記録
1240
1241         if (existing_node_idx >= 0) {
1242             // ケースA: 子ノードは既に訪問済みの場合
1243             edgenum++;
1244
1245         } else {
1246             // ケースB: 子ノードが未訪問の場合
1247
1248             // 新しいノードを追加maindata
1249             // 回転履歴はメモリを食うので、ここでは空にしておくことも可能
1250             // 必要なら後で復元する
1251         }
1252     }
1253 }
```

```
1250     char child_rotation_history[40];
1251     strcpy(child_rotation_history, current_node.rotation);
1252     int current_len = strlen(child_rotation_history);
1253     if (current_len < sizeof(child_rotation_history) - 1) {
1254         child_rotation_history[current_len] = current_move_char;
1255         child_rotation_history[current_len + 1] = '\0';
1256     }
1257
1258     Cubedata child_node = cubedata_create(temp_child_cube,
1259     child_rotation_history, child_depth);
1260     cubedata_write(maindata, tail, child_node);
1261
1262     // 新しいノードの情報をハッシュテーブルのチェインの先頭に追加
1263     HashNode* new_node_for_hash = (HashNode*)malloc(sizeof(
1264     HashNode));
1265     if (new_node_for_hash == NULL) {
1266         fprintf(stderr, "メモリ確保エラー\n");
1267         return tail;
1268     }
1269     //printf("[BFS-UF] %s\n", child_rotation_history);
1270     new_node_for_hash->maindata_idx = tail;
1271     new_node_for_hash->next = hash_table[child_hash];
1272     hash_table[child_hash] = new_node_for_hash;
1273
1274     // エッジの終点を記録
1275     edgenum++;
1276
1277     // キューの末尾を更新
1278     tail++;
1279 }
1280 // --- ここまでが最重要部分 ---
1281
1282 check_hash_table_duplicates();
1283 return tail; // 生成された総ノード数を返す
1284 }
1285 int koukanshi(char cube[6][9], char rotation1[], char rotation2[]){
1286     char temp_rotation[30];
1287     rotate_char(cube, rotation1, strlen(rotation1));
1288     rotate_char(cube, rotation2, strlen(rotation2));
1289
1290     strcpy(temp_rotation, rotation1);
1291     reverse_rotation(temp_rotation);
1292     rotate_char(cube, temp_rotation, strlen(temp_rotation));
1293     strcpy(temp_rotation, rotation2);
1294     reverse_rotation(temp_rotation);
1295     rotate_char(cube, temp_rotation, strlen(temp_rotation));
1296     return 0;
1297 }
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2289
2290
2291
2292
2293
2294
2295
2296
2297
2297
2298
2299
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2389
2390
2391
2392
2393
2394
2395
2396
2397
2397
2398
2399
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2489
2490
2491
2492
2493
2494
2495
2496
2497
2497
2498
2499
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2589
2590
2591
2592
2593
2594
2595
2596
2597
2597
2598
2599
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2619
2620
2621
2622
2623
2624
2625
2626
2627
2628
2629
2629
2630
2631
2632
2633
2634
2635
2636
2637
2638
2639
2639
2640
2641
2642
2643
2644
2645
2646
2647
2648
2649
2649
2650
2651
2652
2653
2654
2655
2656
2657
2658
2659
2659
2660
2661
2662
2663
2664
2665
2666
2667
2668
2669
2669
2670
2671
2672
2673
2674
2675
2676
2677
2678
2679
2679
2680
2681
2682
2683
2684
2685
2686
2687
2688
2689
2689
2690
2691
2692
2693
2694
2695
2696
2697
2697
2698
2699
2699
2700
2701
2702
2703
2704
2705
2706
2707
2708
2709
2709
2710
2711
2712
2713
2714
2715
2716
2717
2718
2719
2719
2720
2721
2722
2723
2724
2725
2726
2727
2728
2729
2729
2730
2731
2732
2733
2734
2735
2736
2737
2738
2739
2739
2740
2741
2742
2743
2744
2745
2746
2747
2748
2749
2749
2750
2751
2752
2753
2754
2755
2756
2757
2758
2759
2759
2760
2761
2762
2763
2764
2765
2766
2767
2768
2769
2769
2770
2771
2772
2773
2774
2775
2776
2777
2778
2778
2779
2780
2781
2782
2783
2784
2785
2786
2787
2788
2789
2789
2790
2791
2792
2793
2794
2795
2796
2797
2797
2798
2799
2799
2800
2801
2802
2803
2804
2805
2806
2807
2808
2809
2809
2810
2811
2812
2813
2814
2815
2816
2817
2818
2819
2819
2820
2821
2822
2823
2824
2825
2826
2827
2828
2829
2829
2830
2831
2832
2833
2834
2835
2836
2837
2838
2839
2839
2840
2841
2842
2843
2844
2845
2846
2847
2848
2849
2849
2850
2851
2852
2853
2854
2855
2856
2857
2858
2859
2859
2860
2861
2862
2863
2864
2865
2866
2867
2868
2869
2869
2870
2871
2872
2873
2874
2875
2876
2877
2878
2878
2879
2880
2881
2882
2883
2884
2885
2886
2887
2888
2888
2889
2890
2891
2892
2893
2894
2895
2895
2896
2897
2898
2899
2899
2900
2901
2902
2903
2904
2905
2906
2907
2908
2909
2909
2910
2911
2912
2913
2914
2915
2916
2917
2918
2919
2919
2920
2921
2922
2923
2924
2925
2926
2927
2928
2929
2929
2930
2931
2932
2933
2934
2935
2936
2937
2938
2939
2939
2940
2941
2942
2943
2944
2945
2946
2947
2948
2949
2949
2950
2951
2952
2953
2954
2955
2956
2957
2958
2959
2959
2960
2961
2962
2963
2964
2965
2966
2967
2968
2969
2969
2970
2971
2972
2973
2974
2975
2976
2977
2978
2978
2979
2980
2981
2982
2983
2984
2985
2986
2987
2988
2989
2989
2990
2991
2992
2993
2994
2995
2996
2997
2997
2998
2999
2999
3000
3001
3002
3003
3004
3005
3006
3007
3008
3009
3009
3010
3011
3012
3013
3014
3015
3016
3017
3018
3019
3019
3020
3021
3022
3023
3024
3025
3026
3027
3028
3029
3029
3030
3031
3032
3033
3034
3035
3036
3037
3038
3039
3039
3040
3041
3042
3043
3044
3045
3046
3047
3048
3049
3049
3050
3051
3052
3053
3054
3055
3056
3057
3058
3059
3059
3060
3061
3062
3063
3064
3065
3066
3067
3068
3069
3069
3070
3071
3072
3073
3074
3075
3076
3077
3078
3078
3079
3080
3081
3082
3083
3084
3085
3086
3087
3088
3089
3089
3090
3091
3092
3093
3094
3095
3096
3097
3097
3098
3099
3099
3100
3101
3102
3103
3104
3105
3106
3107
3108
3109
3109
3110
3111
3112
3113
3114
3115
3116
3117
3118
3119
3119
3120
3121
3122
3123
3124
3125
3126
3127
3128
3129
3129
3130
3131
3132
3133
3134
3135
3136
3137
3138
3139
3139
3140
3141
3142
3143
3144
3145
3146
3147
3148
3149
3149
3150
3151
3152
3153
3154
3155
3156
3157
3158
3159
3159
3160
3161
3162
3163
3164
3165
3166
3167
3168
3169
3169
3170
3171
3172
3173
3174
3175
3176
3177
3178
3178
3179
3180
3181
3182
3183
3184
3185
3186
3187
3188
3189
3189
3190
3191
3192
3193
3194
3195
3195
3196
3197
3198
3199
3199
3200
3201
3202
3203
3204
3205
3206
3207
3208
3209
3209
3210
3211
3212
3213
3214
3215
3216
3217
3218
3219
3219
3220
3221
3222
3223
3224
3225
3226
3227
3228
3229
3229
3230
3231
3232
3233
3234
3235
3236
3237
3238
3239
3239
3240
3241
3242
3243
3244
3245
3246
3247
3248
3249
3249
3250
3251
3252
3253
3254
3255
3256
3257
3258
3259
3259
3260
3261
3262
3263
3264
3265
3266
3267
3268
3269
3269
3270
3271
3272
3273
3274
3275
3276
3277
3278
3278
3279
3280
3281
3282
3283
3284
3285
3286
3287
3288
3289
3289
3290
3291
3292
3293
3294
3295
3296
3297
3297
3298
3299
3299
3300
3301
3302
3303
3304
3305
3306
3307
3308
3309
3309
3310
3311
3312
3313
3314
3315
3316
3317
3318
3319
3319
3320
3321
3322
3323
3324
3325
3326
3327
3328
3329
3329
3330
3331
3332
3333
3334
3335
3336
3337
3338
3339
3339
3340
3341
3342
3343
3344
3345
3346
3347
3348
3349
3349
3350
3351
3352
3353
3354
3355
3356
3357
3358
3359
3359
3360
3361
3362
3363
3364
3365
3366
3367
3368
3369
3369
3370
3371
3372
3373
3374
3375
3376
3377
3378
3378
3379
3380
3381
3382
3383
3384
3385
3386
3387
3388
3389
3389
3390
3391
3392
3393
3394
3395
3396
3397
3398
3398
3399
3400
3401
3402
3403
3404
3405
3406
3407
3408
3409
3409
3410
3411
3412
3413
3414
3415
3416
3417
3418
3419
3419
3420
3421
3422
3423
3424
3425
3426
3427
3428
3429
3429
3430
3431
3432
3433
3434
3435
3436
3437
3438
3439
3439
3440
3441
3442
3443
3444
3445
3446
3447
3448
3449
3449
3450
3451
3452
3453
3454
3455
3456
3457
3458
3459
3459
3460
3461
3462
3463
3464
3465
3466
3467
3468
3469
3469
3470
3471
3472
3473
3474
3475
3476
3477
3478
3478
3479
3480
3481
3482
3483
3484
3485
3486
3487
3488
3489
3489
3490
3491
3492
3493
3494
```

```

1298     char cube_state[6][9]; // キーとなるキューブの状態
1299     UT_hash_handle hh; // に必須のメンバ。これがないと動かないuthash
1300 } forbidden_item;
1301
1302 #define KYOYAKU_DEPTH 1
1303 int kyoyaku_search(char kyoya[]){
1304     clock_t start_time = clock();
1305     int word_count=0;
1306     //1: とからのみ生成される状態を全て記録するrotation1rotation2DEPTH
1307     int node_count = bfs_uf();
1308     //build_adjacency_list(node_count, edgenum);
1309     //2: で生成された状態を全て1で共役するkyoyaku
1310     char inverse_rotation[30];
1311     char temp_rotation[30];
1312     for(int j=1;j<node_count;j++){
1313         Cubedata tempcube=maindata[0];
1314         strcpy(inverse_rotation,kyoya);
1315         reverse_rotation(inverse_rotation);
1316         snprintf(temp_rotation,30,"%s%s%s",kyoya,maindata[j].rotation,
1317         inverse_rotation);
1318         rotate_char(tempcube.cube,temp_rotation,strlen(temp_rotation));
1319         cubedata_write(maindata,j,tempcube);
1320
1321         strcpy(maindata[j].rotation,temp_rotation);
1322     }
1323
1324     //3: で生成できた状態同士が距離にならないか調べる。ならなかったら符号語リストに追加23
1325     char tempcompare[150][6][9];
1326     char rotation[2];
1327     int count;
1328     short is_duplicate;
1329     for(int i=0;i<node_count;i++){
1330         count=1;
1331         //if(i%(node_count/20)==0)printf("[K-Search] %d/%d\n",i,node_count);
1332
1333         //ここから
1334
1335         for(int j=0;j<150;j++){
1336             copycube(maindata[i].cube,tempcompare[j]);
1337         }
1338         for(int j=0;j<12;j++){
1339             rotate_char(tempcompare[count],&rotationindex[j],1);
1340             count++;
1341         }
1342         for(int j=0;j<12;j++){
1343             for(int k=0;k<12;k++){
1344                 if (j == k + 6 || k == j + 6) continue;
1345
1346                 rotation[0]=rotationindex[j];rotation[1]=rotationindex[k];

```

```

1347         rotate_char_lut(tempcompare[count], rotation);
1348         count++;
1349     }
1350 }
1351 is_duplicate=0;
1352 //#pragma omp parallel for schedule(dynamic,8)
1353 for(int j=0;j<codeword_count;j++){
1354     if(is_duplicate==1) continue;
1355     if(maindata[i].rotation[1]==maindata[codeword_indices[j]].rotation[1]) continue;
1356
1357     //printf("[K-Search] Compared %s & %s\n",maindata[i].rotation,
1358     maindata[j].rotation);
1359     for(int k=0;k<count;k++){
1360         if(is_duplicate==1) continue;
1361         if(memcmp(tempcompare[k], maindata[codeword_indices[j]].cube
1362 , sizeof(tempcompare[k]))==0){
1363             //#pragma omp atomic write
1364             is_duplicate=1;
1365             break;
1366         }
1367     }
1368 }
1369 if(is_duplicate==0){
1370     codeword_indices[codeword_count]=i;
1371     //printf("[K-Search] Codeword append:%d",i);
1372     codeword_count++;
1373     if (codeword_count % 1000 == 0) {
1374         double elapsed = (double)(clock() - start_time) /
1375         CLOCKS_PER_SEC;
1376         printf("%d,%.3f\n", codeword_count, elapsed);
1377     }
1378 }
1379 // 3.1: 生成された状態同士を全て検証する必要はない。どのように、先頭の文字が異なるもの同士の組み
1380 合わせと、原点の組み合わせを調べれば良い。AAAAABAAA
1381
1382 printf("%d/%d codewords found.",codeword_count,node_count);
1383 return 0;
1384 }
1385 int kyoyaku_search_hash(char kyoya[]){
1386     clock_t start_time = clock();
1387     int word_count=0;
1388     //1: とからのみ生成される状態を全て記録するrotation1rotation2DEPTH
1389     int node_count = bfs_uf();
1390     //build_adjacency_list(node_count, edgenum);
1391     //2: で生成された状態を全て1で共役するkyoyaku
1392     char inverse_rotation[40];

```

```

1392     char temp_rotation[40];
1393     for(int j=1;j<node_count;j++){
1394         Cubedata tempcube=maindata[0];
1395         strcpy(inverse_rotation,kyoya);
1396         reverse_rotation(inverse_rotation);
1397         sprintf(temp_rotation,40,"%s%s%s",kyoya,maindata[j].rotation,
1398         inverse_rotation);
1399         rotate_char(tempcube.cube,temp_rotation,strlen(temp_rotation));
1400         cubedata_write(maindata,j,tempcube);
1401         strcpy(maindata[j].rotation,temp_rotation);
1402     }
1403     //3: で生成できた状態同士が距離にならないか調べる。ならなかったら符号語リストに追加23
1404     // ハッシュテーブルの先頭を指すポインタ。最初はにしておく。NULL
1405     forbidden_item *forbidden_set = NULL;
1406
1407     // 最初の符号語として原点(maindata[0])を追加
1408     codeword_indices[0] = 0;
1409     codeword_count = 1;
1410
1411     // --- 原点の近傍（距離以内）を生成して、2に追加forbidden_set ---
1412     char tempcompare[150][6][9];
1413     char rotation[2];
1414     int count = 1;
1415
1416     // maindataの状態をコピー[0]
1417     copycube(maindata[0].cube, tempcompare[0]);
1418
1419     // 距離の状態を生成1
1420     for (int j = 0; j < 12; j++) {
1421         copycube(maindata[0].cube, tempcompare[count]);
1422         rotate_char(tempcompare[count], &rotationindex[j], 1);
1423         count++;
1424     }
1425     // 距離の状態を生成2
1426     for (int j = 0; j < 12; j++) {
1427         for (int k = 0; k < 12; k++) {
1428             if (j == k + 6 || k == j + 6) continue;
1429             //参考:char rotationindex[12]="UFRDBLufrdbl";
1430             copycube(maindata[0].cube, tempcompare[count]);
1431             rotation[0] = rotationindex[j]; rotation[1] = rotationindex[k];
1432             rotate_char_lut(tempcompare[count], rotation);
1433             count++;
1434         }
1435     }
1436
1437     // 生成した距離0, 1, の全ての状態をハッシュテーブル（禁止リスト）に追加2
1438     for (int k = 0; k < count; k++) {
1439         forbidden_item *item = malloc(sizeof(forbidden_item));
1440         memcpy(item->cube_state, tempcompare[k], sizeof(item->cube_state));

```

```

1441         HASH_ADD(hh, forbidden_set, cube_state, sizeof(item->cube_state),
1442 item);
1443 }
1444
1445 // --- メインの探索ループ (0(N^2)から0(N)へ) ---
1446 for (int i = 1; i < node_count; i++) {
1447     if (i % (node_count / 20) == 0) printf("[K-Search] %d/%d\n", i,
1448 node_count);
1449
1450     forbidden_item *found_item = NULL;
1451     // 候補が禁止リストi(forbidden_set)に入っているか高速に検索
1452     HASH_FIND(hh, forbidden_set, maindata[i].cube, sizeof(maindata[i].
1453 cube), found_item);
1454
1455     if (found_item == NULL) {
1456
1457         // 見つからなかった場合 => 新しい符号語として採用！
1458         codeword_indices[codeword_count] = i;
1459         codeword_count++;
1460         if (codeword_count % 10000 == 0) {
1461             double elapsed = (double)(clock() - start_time) /
1462 CLOCKS_PER_SEC;
1463             printf("%d,%.3f\n", codeword_count, elapsed);
1464         }
1465         // この新しい符号語の近傍（距離以内）を計算して、禁止リストに追加する2
1466         // 上の原点の近傍を計算したコードとほぼ同じ（）
1467         count = 1;
1468         copycube(maindata[i].cube, tempcompare[0]);
1469         // 距離1
1470         for (int j = 0; j < 12; j++) {
1471             copycube(maindata[i].cube, tempcompare[count]);
1472             rotate_char(tempcompare[count], &rotationindex[j], 1);
1473             count++;
1474         }
1475         // 距離2
1476         for (int j = 0; j < 12; j++) {
1477             for (int k = 0; k < 12; k++) {
1478                 if (j == k + 6 || k == j + 6) continue;
1479
1480                 copycube(maindata[i].cube, tempcompare[count]);
1481                 rotation[0] = rotationindex[j]; rotation[1] =
1482 rotationindex[k];
1483                 rotate_char_lut(tempcompare[count], rotation);
1484                 count++;
1485             }
1486         }
1487
1488         // 新しい禁止状態をハッシュテーブルに追加
1489         for (int k = 0; k < count; k++) {

```

```

1486 // 念のため、追加する前にもう一度チェック（必須ではないが安全）
1487 HASH_FIND(hh, forbidden_set, tempcompare[k], sizeof(
1488 tempcompare[k]), found_item);
1489 if (found_item == NULL) {
1490     forbidden_item *item = malloc(sizeof(forbidden_item));
1491     memcpy(item->cube_state, tempcompare[k], sizeof(item->
1492 cube_state));
1493     HASH_ADD(hh, forbidden_set, cube_state, sizeof(item->
1494 cube_state), item);
1495 }
1496 }
1497 printf("[K-Search] %d/%d codewords found.", codeword_count, node_count);
1498 // --- 最後に、ハッシュテーブルで確保したメモリを全て解放 ---
1499 forbidden_item *current_item, *tmp;
1500 HASH_ITER(hh, forbidden_set, current_item, tmp) {
1501     HASH_DEL(forbidden_set, current_item); // テーブルからエントリを削除
1502     free(current_item); // メモリを解放
1503 }
1504
1505     return 0;
1506 }
1507 int main(){
1508
1509     if (hash_table == NULL) {
1510         hash_table = calloc(HASH_TABLE_SIZE, sizeof(HashNode*));
1511         if (hash_table == NULL) {
1512             fprintf(stderr, "Failed to allocate hash_table\n");
1513             return 1;
1514         }
1515     }
1516     maindata = malloc(sizeof(Cubedata) * MAX_DATA);
1517     kyoyaku_search_hash("R");
1518     return 0;
1519 }
1520 /*
1521 int main(){
1522     printf("Starting BFS search for depth %d...\n", BFS_TARGET_DEPTH);
1523
1524     // 1. でノードとエッジをインデックスで生成BFS
1525     int node_count = bfs_uf();
1526
1527     printf("BFS finished. Found %d nodes and %d edges.\n", node_count,
1528     edgenum);
1529
1530     printf("Checking for duplicates...\n");
1531     //isuuchekall(node_count);
1532     placeholder(node_count);

```

```

1532 // をソートmaindata
1533 //qsort(maindata, node_count, sizeof(Cubedata), compare_cubedata);
1534
1535 int duplicate_count = 0;
1536
1537 for (int i = 0; i < node_count - 1; i++) {
1538     if (memcmp(maindata[i].cube, maindata[i+1].cube, sizeof(maindata[i].cube)) == 0) {
1539         printf("Duplicate found at index %d and %d after sorting.\n", i, i+1);
1540         // どんな状態が重複しているか表示してみる
1541         // printcube(maindata[i].cube);
1542         duplicate_count++;
1543     }
1544 }
1545 //printf("Total duplicates found: %d\n", duplicate_count);
1546 // 2. エッジ情報から隣接リストを構築
1547 //printf("Building adjacency list...\n");
1548 //build_adjacency_list(node_count, edgenum);
1549
1550 // オプション() グラフ描画
1551 // dotter(maindata, node_count, edgenum, "dot/graph.dot");
1552
1553 // 3. アルゴリズムを実行Greedy
1554 //greedy(node_count);
1555 return 0;
1556 }
1557 */

```

Listing 1 ルービックキューブ探索プログラム