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，送信したい情報に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，

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

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

$$d_{\min} \geq 2t + 1 \tag{2}$$

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

$$d_{\min} \geq 3 \tag{3}$$

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．

$$|\mathcal{C}| \cdot 1 + 12 \leq |\mathcal{V}| \tag{4}$$

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 のノードを全て探索し終えてから，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．

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

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が，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，

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

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，元の群構造を保ちつつ，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	`-O3 -march=native -funroll-loops`

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 $\lvert\mathcal{L}\rvert$	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，

$$\lfloor \log_2(43,074,266) \rfloor = 25 \tag{7}$$

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．

巨大なグラフ上で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こと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 准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．

```
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <ctype.h>
#include <omp.h>
#include "uthash.h"
#define U_FACE_IDX 0  // 上面 (Up)
#define L_FACE_IDX 1  // 左面 (Left)
#define F_FACE_IDX 2  // 正面 (Front)
#define R_FACE_IDX 3  // 右面 (Right)
#define B_FACE_IDX 4  // 背面 (Back)
#define D_FACE_IDX 5  // 下面 (Down)
typedef struct Cubedata{
    char cube[6][9];
    char rotation[40];
    char depth;
}Cubedata;
char rotationindex[12]="UFRDBLufrdbl";
void simpleprintcube(char cube[6][9]){
    for(int i=0;i<6;i++){
        for(int j=0;j<9;j++){
            printf("%d,",cube[i][j]);
        }
        printf("\n");
    }
    return;
}
void printcube(char cube[6][9]){
    for(int i=0;i<9;i++){
        printf("%02d ",cube[0][i]);
        if(i%3==2)printf("\n");
    }
    for(int i=0;i<36;i++){
        printf("%02d ",cube[(i/3)%4+1][((i/12)*3)+i%3]);
        if(i%12==11)printf("\n");
    }
    for(int i=0;i<9;i++){
        printf("%02d ",cube[5][i]);
        if(i%3==2)printf("\n");
    }
    return;
}
char printcolor(int num){
```

```
    switch(num/9){
        case 0:return 'W';
        case 1:return 'R';
        case 2:return 'B';
        case 3:return 'O';
        case 4:return 'G';
        case 5:return 'Y';
        default:return 'e';
    }
    return 'e';
}
void printcubecolor(char cube[6][9]){
    for(int i=0;i<9;i++){
        printf("%c",printcolor(cube[0][i]));
        if(i%3==2)printf("\n");
    }
    for(int i=0;i<36;i++){
        printf("%c",printcolor(cube[(i/3)%4+1][((i/12)*3)+i%3]));
        if(i%12==11)printf("\n");
    }
    for(int i=0;i<9;i++){
        printf("%c",printcolor(cube[5][i]));
        if(i%3==2)printf("\n");
    }
}
void printcubeh(char cube[6][9]){
        for(int i=0;i<9;i++){
        printf("%02d ",cube[0][i]+1);
        if(i%3==2)printf("\n");
    }
    for(int i=0;i<36;i++){
        printf("%02d ",cube[(i/3)%4+1][((i/12)*3)+i%3]+1);
        if(i%12==11)printf("\n");
    }
    for(int i=0;i<9;i++){
        printf("%02d ",cube[5][i]+1);
        if(i%3==2)printf("\n");
    }
    return;
}
void copycube(const char src[6][9],char dist[6][9]){
    for(int i=0;i<6;i++){
        for(int j=0;j<9;j++){
            dist[i][j]=src[i][j];
        }
    }
    return;
}
void resetcube(char cube[6][9]){
    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
    [src_faces[i]][j*3+2];
232             else if(src_faces[i]==B_FACE_IDX)cube[dst_faces[i]][j*3+2] =
    tempcube[src_faces[i]][6-j*3];
233             else cube[dst_faces[i]][j*3+2] = tempcube[src_faces[i]][j*3+2];
234         }
235     }
236     return;
237 }
238 void rotate_r_prime(char cube[6][9]) {
239     char tempcube[6][9];
240     copycube(cube, tempcube);
241     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
    [src_faces[i]][j*3+2];
256             else if(src_faces[i]==B_FACE_IDX)cube[dst_faces[i]][j*3+2] =
    tempcube[src_faces[i]][6-j*3];
257             else cube[dst_faces[i]][j*3+2] = tempcube[src_faces[i]][j*3+2];
258         }
259     }
260     return;
261 }
262 void rotate_l(char cube[6][9]) {
263     char tempcube[6][9];
264     copycube(cube, tempcube);
265     cube[L_FACE_IDX][0] = tempcube[L_FACE_IDX][6];
266     cube[L_FACE_IDX][1] = tempcube[L_FACE_IDX][3];
267     cube[L_FACE_IDX][2] = tempcube[L_FACE_IDX][0];
268     cube[L_FACE_IDX][3] = tempcube[L_FACE_IDX][7];
269     cube[L_FACE_IDX][5] = tempcube[L_FACE_IDX][1];
270     cube[L_FACE_IDX][6] = tempcube[L_FACE_IDX][8];
271     cube[L_FACE_IDX][7] = tempcube[L_FACE_IDX][5];
272     cube[L_FACE_IDX][8] = tempcube[L_FACE_IDX][2];
273
274     int src_faces[] = {U_FACE_IDX, F_FACE_IDX, D_FACE_IDX, B_FACE_IDX};
275     int dst_faces[] = {F_FACE_IDX, D_FACE_IDX, B_FACE_IDX, U_FACE_IDX};
276
277     for (int i = 0; i < 4; ++i) {
278         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()
279             if(dst_faces[i]==B_FACE_IDX)cube[dst_faces[i]][8-j*3] = tempcube
    [src_faces[i]][j*3];
280             else if(src_faces[i]==B_FACE_IDX)cube[dst_faces[i]][j*3] =
    tempcube[src_faces[i]][8-j*3];
281             else cube[dst_faces[i]][j*3] = tempcube[src_faces[i]][j*3];
282         }
283     }
284     return;
285 }
286 void rotate_l_prime(char cube[6][9]) {
287     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
    [src_faces[i]][j*3];
304             else if(src_faces[i]==B_FACE_IDX)cube[dst_faces[i]][j*3] =
    tempcube[src_faces[i]][8-j*3];
305             else cube[dst_faces[i]][j*3] = tempcube[src_faces[i]][j*3];
306         }
307     }
308     return;
309 }
310 void rotate_d(char cube[6][9]) {
311     char tempcube[6][9];
312     copycube(cube, tempcube);
313     cube[D_FACE_IDX][0] = tempcube[D_FACE_IDX][6];
314     cube[D_FACE_IDX][1] = tempcube[D_FACE_IDX][3];
315     cube[D_FACE_IDX][2] = tempcube[D_FACE_IDX][0];
316     cube[D_FACE_IDX][3] = tempcube[D_FACE_IDX][7];
317     cube[D_FACE_IDX][5] = tempcube[D_FACE_IDX][1];
318     cube[D_FACE_IDX][6] = tempcube[D_FACE_IDX][8];
319     cube[D_FACE_IDX][7] = tempcube[D_FACE_IDX][5];
320     cube[D_FACE_IDX][8] = tempcube[D_FACE_IDX][2];
321
322
323     int src_faces[] = {L_FACE_IDX, F_FACE_IDX, R_FACE_IDX, B_FACE_IDX};
324     int dst_faces[] = {F_FACE_IDX, R_FACE_IDX, B_FACE_IDX, L_FACE_IDX};
325
326     for (int i = 0; i < 4; ++i) {
327         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()
328             cube[dst_faces[i]][j+6] = tempcube[src_faces[i]][j+6];
329         }
330     }
331     return;
332 }
333 void rotate_d_prime(char cube[6][9]) {
334     char tempcube[6][9];
335     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     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     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                 num=12+i*12+j;
468             }
469         }
470     }
471     for(int i=0;i<6;i++){
472         for(int j=0;j<9;j++){
473             cube[i][j]=tempcube[lut_2t[num][i][j]/9][lut_2t[num][i][j]%9];
474         }
475     }
476 
477     return;
478 }
479 void cubedata_print(Cubedata data[],int indexnum){
480     printcube(data[indexnum].cube);
481     printf("  data[%d],rotation='%s',depth=%d\n",indexnum,data[indexnum].
    rotation,data[indexnum].depth);
482     return;
```

```
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
    ].depth);
487     }
488 }
489 int cubedata_allclear(Cubedata data[],int len){
490     for(int i=0;i<len;i++){
491         resetcube(data[i].cube);
492         for(int j=0;j<40;j++)data[i].rotation[j]='\0';
493         data[i].depth=-1;
494     }
495     return 0;
496 }
497 int cubedata_delete(Cubedata data[],int indexnum){
498     resetcube(data[indexnum].cube);
499     for(int j=0;j<40;j++)data[indexnum].rotation[j]='\0';
500     data[indexnum].depth=-1;
501 
502     return 0;
503 }
504 void cubedata_write(Cubedata data[],int indexnum,Cubedata src){
505     copycube(src.cube,data[indexnum].cube);
506     strcpy(data[indexnum].rotation,src.rotation);
507     data[indexnum].depth=src.depth;
508     return;
509 }
510 Cubedata cubedata_create(const char cube[6][9],char rotation[],char depth){
511     Cubedata data;
512     copycube(cube,data.cube);
513     strcpy(data.rotation,rotation);
514     data.depth=depth;
515     return data;
516 }
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
```

```c
} HashNode;

// 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
static HashNode** hash_table = NULL;



// 以前の visited_states は不要になるので削除します
// static int visited_states[HASH_TABLE_SIZE] = {0}; // ← 削除

// メモリ解放用の関数（プログラム終了時に呼ぶ）
void free_hash_table() {
    if (hash_table == NULL) return;
    for (int i = 0; i < HASH_TABLE_SIZE; i++) {
        HashNode* current = hash_table[i];
        while (current != NULL) {
            HashNode* temp = current;
            current = current->next;
            free(temp);
        }
        hash_table[i] = NULL;
    }
}

/**
 * @brief キューブの状態からハッシュ値を計算する (Polynomial rolling hash)
 * @param cube 計算対象のキューブ状態
 * @return unsigned int 計算されたハッシュ値
 */
unsigned int calculate_hash(const char cube[6][9]) {
    unsigned int hash = 0;
    // は一般的に使われる素数31
    const unsigned int prime = 31;

    for (int i = 0; i < 6; i++) {
        for (int j = 0; j < 9; j++) {
            // (hash * prime + value) % size
            // 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%
            hash = (hash * prime + cube[i][j]);
        }
    }
    return hash % HASH_TABLE_SIZE;
}
#define FNV_PRIME_32 16777619
#define FNV_OFFSET_BASIS_32 2166136261U

unsigned int calculate_hash_fnv1a(const char cube[6][9]) {
     // FNV-1で推奨される初期値と素数a (32-版bit)
    unsigned int hash = 0x811c9dc5;
    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
        配列
621  * 3. edgenum: 発見したエッジの総数
622  *
623  * @return int 生成された総ノード数
624  */
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
```

```
    edgenum = 0;
    memset(visited_states, 0, sizeof(visited_states));

    // 1. 初期ノードの準備
    char initial_cube[6][9];
    resetcube(initial_cube);
    Cubedata initial_node = cubedata_create(initial_cube, "", 0);

    // 2. 初期ノードをキューと訪問済みリストに追加
    unsigned int initial_hash = calculate_hash_fnv1a(initial_node.cube);

    // 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)
    visited_states[initial_hash] = tail + 1;

    cubedata_write(maindata, tail, initial_node);
    tail++;

    // 3. 探索ループ キューが空になるまで()
    while (head < tail) {
        // ★ 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
        // これがエッジの始点(from)になる。
        int current_idx = head;
        Cubedata current_node = maindata[current_idx];
        head++; // キューを進める

        // 深さ制限に達したら、このノードからは探索しない
        if (current_node.depth >= BFS_TARGET_DEPTH) {
            continue;
        }

        short child_depth = current_node.depth + 1;

        // 4. 通りの回転を試して子ノードを生成12
        for (int i = 0; i < 12; i++) {
            // 配列が満杯なら探索を中断
            if (tail >= MAX_DATA || edgenum >= MAX_DATA) {
                fprintf(stderr, 警告": または配列が満杯です。探索を中断します。
maindataedge\n");
                return tail;
            }

            char current_move_char = rotationindex[i];

            // 子ノードの状態を生成
            char temp_child_cube[6][9];
            copycube(current_node.cube, temp_child_cube);
            rotate_char(temp_child_cube, &current_move_char, 1);

            // 子ノードのハッシュを計算し、訪問済みかチェック
            unsigned int child_hash = calculate_hash_fnv1a(temp_child_cube);
```

```
680             int existing_node_idx = visited_states[child_hash] - 1; // して
    ベースのインデックスに戻す-10
681
682             // ★ 変更点 2: エッジの始点を「インデックス」で記録
683             edge_from_idx[edgenum] = current_idx;
684
685             if (existing_node_idx >= 0) {
686                 // ケースA: 子ノードは既に訪問済みの場合
687
688                 // ★ 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
689                 edge_to_idx[edgenum] = existing_node_idx;
690                 edgenum++;
691
692                 continue; // この子ノードはキューに追加せず、次の回転へ
693
694             } else {
695                 // ケースB: 子ノードが未訪問の場合
696
697                 // ★ 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
    ス(tail)」で記録
698                 edge_to_idx[edgenum] = tail;
699                 edgenum++;
700
701                 // 新しいノードを訪問済みリストに登録
702                 visited_states[child_hash] = tail + 1;
703
704                 // に格納するために、回転履歴文字列を作成maindata
705                 char child_rotation_history[30];
706                 strcpy(child_rotation_history, current_node.rotation);
707                 int current_len = strlen(child_rotation_history);
708                 if (current_len < sizeof(child_rotation_history) - 1) {
709                     child_rotation_history[current_len] = current_move_char;
710                     child_rotation_history[current_len + 1] = '\0';
711                 }
712
713                 // 新しいノードをキューの末尾に追加
714                 Cubedata child_node = cubedata_create(temp_child_cube,
    child_rotation_history, child_depth);
715                 cubedata_write(maindata, tail, child_node);
716                 tail++; // キューの末尾を更新
717             }
718         }
719     }
720
721     return tail; // 生成された総ノード数を返す
722 }*/
723 /**
724  * @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
725  *
726  * 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。
727  *
```

```
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->
    maindata_idx].cube, sizeof(maindata[0].cube)) == 0) {
741                     printf("DUPLICATE STATE FOUND IN HASH CHAIN! hash_key=%d
    , idx1=%d, idx2=%d\n",
742                            i, p1->maindata_idx, p2->maindata_idx);
743                 }
744                 p2 = p2->next;
745             }
746             p1 = p1->next;
747         }
748     }
749     printf("Hash table check finished.\n");
750 }
751 int bfs_searchall_with_collision_handling() {
752     int head = 0; // キューの先頭 読み込み位置()
753     int tail = 0; // キューの末尾 書き込み位置()
754 
755     // グローバル変数を初期化
756     edgenum = 0;
757     free_hash_table(); // 以前の実行結果が残らないようにクリア
758 
759     // 1. 初期ノードの準備
760     char initial_cube[6][9];
761     resetcube(initial_cube);
762     Cubedata initial_node = cubedata_create(initial_cube, "", 0);
763 
764     // 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
765     // に追加maindata
766     cubedata_write(maindata, tail, initial_node);
767 
768     // ハッシュテーブルに登録
769     unsigned int initial_hash = calculate_hash_fnv1a(initial_node.cube);
770     HashNode* new_hash_node = (HashNode*)malloc(sizeof(HashNode));
771     if (new_hash_node == NULL) {
772         fprintf(stderr, "メモリ確保エラー\n");
773         return 0;
774     }
775     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
    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。
    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,
    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                 // 新しいノードをに追加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,
    child_rotation_history, child_depth);
847                 cubedata_write(maindata, tail, child_node);
848
849                 // 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
850                 HashNode* new_node_for_hash = (HashNode*)malloc(sizeof(
    HashNode));
851                 if (new_node_for_hash == NULL) {
852                     fprintf(stderr, "メモリ確保エラー\n");
853                     return tail;
854                 }
855                 new_node_for_hash->maindata_idx = tail;
856                 new_node_for_hash->next = hash_table[child_hash];
857                 hash_table[child_hash] = new_node_for_hash;
858
859                 // エッジの終点を記録
860                 //edge_to_idx[edgenum] = tail;
861                 edgenum++;
862
863                 // キューの末尾を更新
864                 tail++;
865             }
866             // --- ここまでが最重要部分 ---
867         }
868     }
869     check_hash_table_duplicates();
870     return tail; // 生成された総ノード数を返す
871 }
```

```
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、失敗した場合は0-1
877 int write_cubedata_to_file(const Cubedata data[], int num_elements, const
        char *filename) {
878     FILE *outfile;
879
880     // ファイルをバイナリ書き込みモードで開く ("wb")
881     outfile = fopen(filename, "wb");
882     if (outfile == NULL) {
883         perror("Error opening file for writing");
884         return -1; // ファイルオープン失敗
885     }
886
887     // を使用して配列全体を一度に書き込むfwrite
888     // 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)
889     // 戻り値は実際に書き込まれた要素数
890     size_t items_written = fwrite(data, sizeof(Cubedata), num_elements,
        outfile);
891
892     if (items_written != num_elements) {
893         fprintf(stderr, "Error writing data to file. Wrote %zu of %d items.\
        n", items_written, num_elements);
894         fclose(outfile);
895         return -1; // 書き込みエラー
896     }
897
898     // ファイルを閉じる
899     if (fclose(outfile) != 0) {
900         perror("Error closing file");
901         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()
902     }
903
904     printf("Successfully wrote %d Cubedata elements to %s\n", num_elements,
        filename);
905     return 0; // 成功
906 }
907
908 // 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()
909 int read_cubedata_from_file(Cubedata data[], int num_elements_to_read, const
         char *filename) {
910     FILE *infile;
911
912     // ファイルをバイナリ読み込みモードで開く ("rb")
913     infile = fopen(filename, "rb");
914     if (infile == NULL) {
915         perror("Error opening file for reading");
916         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,
        infile);
921
922     if (items_read != num_elements_to_read) {
923         // ファイルの終端に達した場合やエラーの場合がある
924         if (feof(infile)) {
925             fprintf(stderr, "Error reading data: unexpected end of file.
        Read %zu of %d items.\n", items_read, num_elements_to_read);
926         } else if (ferror(infile)) {
927             perror("Error reading data from file");
928         }
929         fclose(infile);
930         return -1;
931     }
932
933     // ファイルを閉じる
934     if (fclose(infile) != 0) {
935         perror("Error closing file after reading");
936         // データは読み込めて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
937     }
938
939     printf("Successfully read %zu Cubedata elements from %s\n", items_read,
        filename);
940     return 0; // 成功
941 }
942
943 /*
944 int dotter(Cubedata data[],int elements, const char *filename){
945
946     char nodename[elements][30];
947
948     // generate node
949     strcpy(nodename[0],"e");
950     for(int i=1;i<elements;i++){
951         strcpy(nodename[i],data[i].rotation);
952     }
953
954     FILE *outfile;
955
956     // ファイルをバイナリ書き込みモードで開く ("wb")
957     outfile = fopen(filename, "w");
958     if (outfile == NULL) {
959         perror("Error opening file for writing");
960         return -1; // ファイルオープン失敗
961     }
962
963     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
        ",nodename[i]);
968         if(strlen(nodename[i])==2)fprintf(outfile, "  %s [fillcolor=blue];\n
        ",nodename[i]);
969         if(strlen(nodename[i])==3)fprintf(outfile, "  %s [fillcolor=
        lightblue];\n",nodename[i]);
970         if(strlen(nodename[i])==0)break;
971     }
972
973     for(int i=1; i<elements;i++){
974         fprintf(outfile, "  %s -> %s;\n",edgefrom[i],edgeto[i]);
975         if(strlen(edgefrom[i])==0&&strlen(edgeto[i])==0)break;
976     }
977     fprintf(outfile, "}\n");
978     fclose(outfile);
979     printf("Successfully wrote %d Cubedata elements to %s\n", elements,
        filename);
980     printf("[DOTTER] Success.");
981     return 0;
982 }
983 */
984 /*
985 int dotter(Cubedata data[], int node_count, int edge_count, const char *
        filename){
986     FILE *outfile;
987
988     outfile = fopen(filename, "w");
989     if (outfile == NULL) {
990         perror("Error opening file for writing");
991         return -1;
992     }
993
994     fprintf(outfile, "graph G {\n");
995     fprintf(outfile, "  graph [overlap=scale, splines=true];\n");
996     fprintf(outfile, "  node [style=filled, shape=circle, fixedsize=true,
        width=0.6];\n");
997
998     // --- ノード定義 ---
999     // 初期ノード "e" 空文字列の代わりに()
1000    fprintf(outfile, "  e [label=\"\", fillcolor=gold];\n");
1001
1002    // 深さごとのノード
1003    for(int i = 1; i < node_count; i++){
1004        const char* color;
1005        switch(data[i].depth) {
1006            case 1: color = "lightcoral"; break;
1007            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].
    rotation, data[i].rotation, color);
1013        }
1014
1015        // --- エッジ定義 ---
1016        for(int i = 0; i < edge_count; i++){
1017            // ノード名が空文字列の場合は "e" として扱う
1018            const char* from_node = (strlen(edgefrom[i]) == 0) ? "e" : edgefrom[
    i];
1019            const char* to_node = (strlen(edgeto[i]) == 0) ? "e" : edgeto[i];
1020            fprintf(outfile, "  %s -- %s;\n", from_node, to_node);
1021        }
1022
1023        fprintf(outfile, "}\n");
1024        fclose(outfile);
1025        printf("Successfully wrote %d nodes and %d edges to %s\n", node_count,
    edge_count, filename);
1026        printf("[DOTTER] Success.\n");
1027        return 0;
1028    }
1029    */
1030    // 各ノードが持つ隣接ノードのリスト
1031    /*
1032    typedef struct AdjacencyList {
1033        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
1034        int neighbor_count;
1035    } AdjacencyList;
1036
1037    // 全ノード分の隣接リスト
1038    AdjacencyList adj[MAX_DATA];
1039
1040    void build_adjacency_list(int node_count, int edge_count) {
1041        // 初期化
1042        for (int i = 0; i < node_count; i++) {
1043            adj[i].neighbor_count = 0;
1044        }
1045
1046        // エッジ情報インデックス()から直接リストを構築
1047        for (int i = 0; i < edge_count; i++) {
1048            int from_idx = edge_from_idx[i];
1049            int to_idx = edge_to_idx[i];
1050
1051            // from -> to を追加（重複チェック付き）
1052            char exists = 0;
1053            for (int k = 0; k < adj[from_idx].neighbor_count; k++) {
1054                if (adj[from_idx].neighbors[k] == to_idx) {
```

```
                exists = 1;
                break;
            }
        }
        if (!exists && adj[from_idx].neighbor_count < 12) {
            adj[from_idx].neighbors[adj[from_idx].neighbor_count++] = to_idx
    ;
        }

        // to -> from を追加（重複チェック付き）
        exists = 0;
        for (int k = 0; k < adj[to_idx].neighbor_count; k++) {
            if (adj[to_idx].neighbors[k] == from_idx) {
                exists = 1;
                break;
            }
        }
        if (!exists && adj[to_idx].neighbor_count < 12) {
            adj[to_idx].neighbors[adj[to_idx].neighbor_count++] = from_idx;
        }
    }
    printf("Adjacency list has been built successfully.\n");
}

// とのにおけるインデックスを引数とするcube1cube2maindata
int is_distance_at_least_3_improved(int idx1, int idx2) {

    // 距離 0 のチェック
    if (idx1 == idx2) return 0; // false

    // 距離 1 のチェック
    // の隣接ノードにが含まれているか調べるidx1idx2
    for (int i = 0; i < adj[idx1].neighbor_count; i++) {
        if (adj[idx1].neighbors[i] == idx2) {
            return 0; // false, 距離1
        }
    }

    // 距離 2 のチェック
    // 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
    べるidx2
    for (int i = 0; i < adj[idx1].neighbor_count; i++) {
        int neighbor1_idx = adj[idx1].neighbors[i];

        // の隣接ノードを調べるneighbor1
        for (int j = 0; j < adj[neighbor1_idx].neighbor_count; j++) {
            int neighbor2_idx = adj[neighbor1_idx].neighbors[j];
            if (neighbor2_idx == idx2) {
                return 0; // false, 距離2
            }
```

```
        }
    }

    // 上記のいずれにも当てはまらなければ、距離は以上3
    return 1; // true
}*/

// greedy() が生成するデータ
int codeword_indices[MAX_DATA]; // 見つけた符号語のにおけるインデックスを格納maindata
int codeword_count = 0;         // 見つけた符号語の総数

int reverse_rotation(char in[]) {

    // --- ^^e2^^91^^a0 安全性のためのチェック ---
    if (in == NULL) {
        return -1;
    }

    // --- ^^e2^^91^^a1 文字列の反転処理（前回と同じ） ---
    int length = strlen(in);
    if (length == 0) {
        return 0; // 空文字列なら何もしない
    }

    int start = 0;
    int end = length - 1;
    char temp;

    while (start < end) {
        temp = in[start];
        in[start] = in[end];
        in[end] = temp;

        start++;
        end--;
    }

    // --- ^^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） ---
    for (int i = 0; i < length; i++) {
        // 現在の文字を取得
        char c = in[i];

        // もし大文字なら小文字に変換
        if (isupper(c)) {
            in[i] = tolower(c);
        }
        // もし小文字なら大文字に変換
        else if (islower(c)) {
            in[i] = toupper(c);
        }
```

```
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
    では許容
1199            head++;
1200
1201            if (current_node.depth >= BFS_TARGET_DEPTH) {
```

```
            continue;
        }

        short child_depth = current_node.depth + 1;

        // 4. 通りの回転を試して子ノードを生成12
        for (int i = 0; i < 2; i++) {
            if (tail >= MAX_DATA || edgenum >= MAX_DATA) {
                fprintf(stderr, "警告: または配列が満杯です。探索を中断します。maindataedge\n");
                return tail;
            }

            char current_move_char = rotationindex[i];

            char temp_child_cube[6][9];
            copycube(current_node.cube, temp_child_cube);
            rotate_char(temp_child_cube, &current_move_char, 1);

            // --- ここからが最重要部分 ---

            // 4a. 子ノードのハッシュを計算し、訪問済みかチェック
            unsigned int child_hash = calculate_hash_fnv1a(temp_child_cube);

            // 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
            int existing_node_idx = -1;
            HashNode* current_chain = hash_table[child_hash];
            while (current_chain != NULL) {
                // ハッシュ値が同じでも、実際のデータが同じかで比較memcmp
                if (memcmp(maindata[current_chain->maindata_idx].cube, temp_child_cube, sizeof(temp_child_cube)) == 0) {
                    existing_node_idx = current_chain->maindata_idx;
                    break;
                }
                current_chain = current_chain->next;
            }

            // 4c. エッジを記録

            if (existing_node_idx >= 0) {
                // ケースA: 子ノードは既に訪問済みの場合
                edgenum++;

            } else {
                // ケースB: 子ノードが未訪問の場合

                // 新しいノードをに追加maindata
                // 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
                // 必要なら後で復元する

```

```
                char child_rotation_history[40];
                strcpy(child_rotation_history, current_node.rotation);
                int current_len = strlen(child_rotation_history);
                if (current_len < sizeof(child_rotation_history) - 1) {
                    child_rotation_history[current_len] = current_move_char;
                    child_rotation_history[current_len + 1] = '\0';
                }

                Cubedata child_node = cubedata_create(temp_child_cube,
    child_rotation_history, child_depth);
                cubedata_write(maindata, tail, child_node);

                // 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
                HashNode* new_node_for_hash = (HashNode*)malloc(sizeof(
    HashNode));
                if (new_node_for_hash == NULL) {
                    fprintf(stderr, "メモリ確保エラー\n");
                    return tail;
                }
                //printf("[BFS-UF]%s\n",child_rotation_history);
                new_node_for_hash->maindata_idx = tail;
                new_node_for_hash->next = hash_table[child_hash];
                hash_table[child_hash] = new_node_for_hash;

                // エッジの終点を記録
                edgenum++;

                // キューの末尾を更新
                tail++;
            }
            // --- ここまでが最重要部分 ---
        }
    }
    check_hash_table_duplicates();
    return tail; // 生成された総ノード数を返す
}
int koukanshi(char cube[6][9], char rotation1[],char rotation2[]){
    char temp_rotation[30];
    rotate_char(cube,rotation1,strlen(rotation1));
    rotate_char(cube,rotation2,strlen(rotation2));

    strcpy(temp_rotation,rotation1);
    reverse_rotation(temp_rotation);
    rotate_char(cube,temp_rotation,strlen(temp_rotation));
    strcpy(temp_rotation,rotation2);
    reverse_rotation(temp_rotation);
    rotate_char(cube,temp_rotation,strlen(temp_rotation));
    return 0;
}
typedef struct {
```

```
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まで全て記録する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,
      inverse_rotation);
1317          rotate_char(tempcube.cube,temp_rotation,strlen(temp_rotation));
1318          cubedata_write(maindata,j,tempcube);
1319
1320          strcpy(maindata[j].rotation,temp_rotation);
1321
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];
```

```
                rotate_char_lut(tempcompare[count],rotation);
                count++;
            }
        }
        is_duplicate=0;
        //#pragma omp parallel for schedule(dynamic,8)
        for(int j=0;j<codeword_count;j++){
            if(is_duplicate==1)continue;
            if(maindata[i].rotation[1]==maindata[codeword_indices[j]].
    rotation[1])continue;

            //printf("[K-Search] Compared %s & %s\n",maindata[i].rotation,
    maindata[j].rotation);
            for(int k=0;k<count;k++){
                if(is_duplicate==1)continue;
                if(memcmp(tempcompare[k], maindata[codeword_indices[j]].cube
    , sizeof(tempcompare[k]))==0){
                    //#pragma omp atomic write
                    is_duplicate=1;
                    break;
                }

            }

        }
        if(is_duplicate==0){
            codeword_indices[codeword_count]=i;
            //printf("[K-Search] Codeword append:%d",i);
            codeword_count++;
            if (codeword_count % 1000 == 0) {
                double elapsed = (double)(clock() - start_time) /
    CLOCKS_PER_SEC;
                printf("%d,%.3f\n", codeword_count, elapsed);
            }
        }
    }
    // 3.1: 生成された状態同士を全て検証する必要はない。とのように、先頭の文字が異なるもの同士の組み
    合わせと、原点の組み合わせを調べれば良い。AAAABAAA

    printf("[K-Search] %d/%d codewords found.",codeword_count,node_count);
    return 0;
}
int kyoyaku_search_hash(char kyoya[]){
    clock_t start_time = clock();
    int word_count=0;
    //1: とからのみ生成される状態をまで全て記録するrotation1rotation2DEPTH
    int node_count = bfs_uf();
    //build_adjacency_list(node_count, edgenum);
    //2: で生成された状態を全て1で共役するkyoyaku
    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            snprintf(temp_rotation,40,"%s%s%s",kyoya,maindata[j].rotation,
    inverse_rotation);
1398            rotate_char(tempcube.cube,temp_rotation,strlen(temp_rotation));
1399            cubedata_write(maindata,j,tempcube);
1400            strcpy(maindata[j].rotation,temp_rotation);
1401
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),
    item);
1442    }
1443
1444
1445    // --- メインの探索ループ (O(N^2)からO(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,
    node_count);
1448
1449        forbidden_item *found_item = NULL;
1450        // 候補が禁止リストi(forbidden_set)に入っているか高速に検索
1451        HASH_FIND(hh, forbidden_set, maindata[i].cube, sizeof(maindata[i].
    cube), found_item);
1452
1453        if (found_item == NULL) {
1454
1455            // 見つからなかった場合 => 新しい符号語として採用！
1456            codeword_indices[codeword_count] = i;
1457            codeword_count++;
1458            if (codeword_count % 10000 == 0) {
1459                double elapsed = (double)(clock() - start_time) /
    CLOCKS_PER_SEC;
1460                printf("%d,%.3f\n", codeword_count, elapsed);
1461            }
1462            // 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
1463            // 上の原点の近傍を計算したコードとほぼ同じ()
1464            count = 1;
1465            copycube(maindata[i].cube, tempcompare[0]);
1466            // 距離1
1467            for (int j = 0; j < 12; j++) {
1468                copycube(maindata[i].cube, tempcompare[count]);
1469                rotate_char(tempcompare[count], &rotationindex[j], 1);
1470                count++;
1471            }
1472            // 距離2
1473            for (int j = 0; j < 12; j++) {
1474                for (int k = 0; k < 12; k++) {
1475                    if (j == k + 6 || k == j + 6) continue;
1476
1477                    copycube(maindata[i].cube, tempcompare[count]);
1478                    rotation[0] = rotationindex[j]; rotation[1] =
    rotationindex[k];
1479                    rotate_char_lut(tempcompare[count], rotation);
1480                    count++;
1481                }
1482            }
1483
1484            // 新しい禁止状態をハッシュテーブルに追加
1485            for (int k = 0; k < count; k++) {
```

```
                    // 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）
                    HASH_FIND(hh, forbidden_set, tempcompare[k], sizeof(
    tempcompare[k]), found_item);
                    if (found_item == NULL) {
                        forbidden_item *item = malloc(sizeof(forbidden_item));
                        memcpy(item->cube_state, tempcompare[k], sizeof(item->
    cube_state));
                        HASH_ADD(hh, forbidden_set, cube_state, sizeof(item->
    cube_state), item);
                    }
                }
            }
        }
    }
    printf("[K-Search] %d/%d codewords found.", codeword_count, node_count);
    // --- 最後に、ハッシュテーブルで確保したメモリを全て解放 ---
    forbidden_item *current_item, *tmp;
    HASH_ITER(hh, forbidden_set, current_item, tmp) {
        HASH_DEL(forbidden_set, current_item); // テーブルからエントリを削除
        free(current_item);                    // メモリを解放
    }


    return 0;
}
int main(){

    if (hash_table == NULL) {
        hash_table = calloc(HASH_TABLE_SIZE, sizeof(HashNode*));
        if (hash_table == NULL) {
            fprintf(stderr, "Failed to allocate hash_table\n");
            return 1;
        }
    }
    maindata = malloc(sizeof(Cubedata) * MAX_DATA);
    kyoyaku_search_hash("R");
    return 0;
}
/*
int main(){
    printf("Starting BFS search for depth %d...\n", BFS_TARGET_DEPTH);

    // 1. でノードとエッジをインデックスで生成BFS
    int node_count = bfs_uf();

    printf("BFS finished. Found %d nodes and %d edges.\n", node_count,
    edgenum);

    printf("Checking for duplicates...\n");
    //isuucheckall(node_count);
    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
1558  */
```

Listing 1　ルービックキューブ探索プログラム