Quixoの強解決

田中 智[1,a)]　ボネ フランソワ[1,b)]　ティクシリュー セバスチャン[2,c)]　田村 康将[1,d)]

概要：Quixo は 5 × 5 グリッド上で行われる二人零和確定完全情報ゲームである．ゲームの目標は縦横斜めのいずれか一直線上に自身のマークを揃えることである．本研究では，Value Iteration と Backward Induction を組み合わせたアルゴリズムによりすべての局面状態における勝敗 (引き分けを含む) を明らかにし，Quixo が特に初期状態において引き分けであることを示した．また 5×5 の場合に加え，4×4 および 3×3 の場合について同様に解析を行い，具体的な勝利手順を明らかにした．妥当な時間とメモリ量での計算を可能とする状態表現および勝敗の記録方法，アルゴリズムの高速化について，論文中に説明を加える．

Quixo Is Solved

Satoshi Tanaka[1,a)]　François Bonnet[1,b)]　Sébastien Tixeuil[2,c)]　Yasumasa Tamura[1,d)]

Abstract: Quixo is a two-player game played on a 5 × 5 grid where the players try to align five identical symbols. Specifics of the game require the usage of novel techniques. Using a combination of value iteration and backward induction, we propose the first complete analysis of the game. We describe memory-efficient data structures and algorithmic optimizations that make the game solvable within reasonable time and space constraints. Our main conclusion is that Quixo is a Draw game. The paper also contains the analysis of smaller boards and presents some interesting states extracted from our computations.

1. はじめに

Quixo は 5 × 5 のグリッド上で行われる二人零和確定完全情報ゲームであり，各プレイヤーが交互に行動をとるターン制ゲームである [1], [2]．図 1 に示すように，X および O のマークで示される各プレイヤーの持ち駒を盤上に配置し，いち早く縦・横・斜めのいずれか 1 列 (5 マス) を自身のマークで揃えたプレイヤーが勝者となる．勝利条件は五目並べや〇×ゲーム (Tic-tac-toe) と共通しているが，駒の操作方法が独特であり，1 手で 1 列すべての駒が動く場合もあるため，状態遷移はより複雑化する．

盤面と手番の組をゲームの状態と定義したとき，Quixo では将棋のように同一の状態への遷移が起こり得る．このとき，Quixo には千日手のようなルールが存在しないため，

1 Tokyo Institute of Technology, Tokyo, Japan
2 Sorbonne Université, CNRS, LIP6, Paris, France
a) tanaka.s@coord.c.titech.ac.jp
b) bonnet@c.titech.ac.jp
c) Sebastien.Tixeuil@lip6.fr
d) tamura@c.titech.ac.jp

図 1: ボードとマーク付きキューブで構成される Quixo.

ゲームが有限の手番で終了することが保証されない．したがって，単純な深さ優先探索アルゴリズムなどでは Quixo を解決できない．

本研究は Quixo の強解決を目的とし，すべての状態における勝敗 (引き分けを含む) を計算するアルゴリズムを提案する．一般的な 5 × 5 の Quixo であってもその状態数は膨大 ($1.7 \cdot 10^{12}$ 程度) である．本研究では状態表現の最適化に加え，探索中の勝敗結果の記録方法の工夫，アルゴリズム自体の高速化を行うことで，現実的に妥当な時間およびメモリ量での計算を実現する．

© 2020 Information Processing Society of Japan

本研究の主な貢献は以下の通りである．1) 5×5 の Quixo は初期状態において引き分けであることを明らかにした．2) 5×5 に加え 4×4 および 3×3 の場合について Quixo を強解決した．3) 4×4 および 3×3 の場合について，具体的な先手の最短勝利手順を明らかにした．4) 5×5 までの Quixo を現実的な時間・メモリ量で解決するアルゴリズムを構築した．

2. 関連研究

Nim [3] や Poker [4]，囲碁 [5] などゲームの研究は盛んに行われている．近年では，囲碁や将棋などにおいて，コンピュータが人間のトッププレイヤー以上の棋力を示す事例が報告されている．一方で，Hex は先手プレイヤーの勝利が数学的に証明され [6]，Connect-Four [7] や Gomoku [8] では計算により先手の勝利が示されるなど，ゲームを理論的に解決する研究も行われている．本研究は Quixo について後者のアプローチをとるものである．

三柴らは，グリッドサイズを $n\times n$ ($n\in\mathbb{N}$) に一般化した Quixo について，計算複雑性の観点から EXPTIME 完全なゲームに分類されることを証明している [9]．これは $n>5$ の場合も扱う研究であるが，勝利条件を 5 個以上のマークを 1 列に揃えることとしている．また，特定の状態における具体的な勝敗を明らかにするものではない．

状態数が Quixo と同程度である Connect-Four は 30 年以上前に Allen と Allis によって解決されているが [7]，ゲームの性質が Quixo とは異なる．Connect-Four は，一度配置された駒が移動することのない，有限ゲームである．

3. Quixo

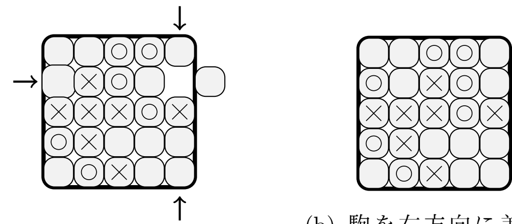
Quixo は 1995 年に Thierry Chapeau によって設計されたゲームであり [1], [2]，アメリカ [10], [11], [12], [13] やフランス [14], [15] で多くの賞を受賞している．Quixo には 4 人プレイヤーのゲームも存在するが，本研究では 2 人プレイヤーのゲームのみを扱う．

3.1 Quixo のルール

Quixo は 5×5 グリッド上で行われる．それぞれのマスには駒がおかれ，駒には {X,O,blank} の 3 種類の面が存在する．初期状態ではすべての駒が blank となっている．

各プレイヤーが交互に駒を動かし，縦・横・斜めのいずれか 1 列に連続で自分のマーク (X or O) を 5 個揃えると勝ちとなる．両方のマークの列が同時にできてしまった場合，最後のアクションを行ったプレイヤーの負けとなる．

図 2 に駒の動かし方の一例を示す．手番のプレイヤーは，盤の端 (16 マス) に位置する自分のマークか blank の駒を選択する．選択した駒を取り出し，blank の場合は自分のマークに変え，選択したマスを他の駒で詰めるように盤の端から差し込む．



(a) 駒を動かす前の状態．手番は O．差し込む方向の選択肢は，矢印で示す 3 通り．

(b) 駒を右方向に差し込んだ後の状態．手番は X．blank の駒を O に変えて差し込んでいる．

図 2: 例: 上から 2 段目，最右端の駒を動かす場合．

3.2 Quixo における状態と “ゲーム木”

グリッド上の駒の配置と手番の組みを状態と定義する．Quixo では将棋と同様に同じ駒配置で手番のみ異なる状況があり得るため，手番による区別が必要である．駒は $5\times5=25$ 個であり，{X,O,blank} の 3 通りの面を持つ．これに加えて手番が 2 通りあるため，全状態数は $3^{25}\cdot2\approx1.7\cdot10^{12}$ となる．

Quixo では手の選び方により同じ状態に遷移することがあり，一般的な木構造では状態遷移を表現できない．そこで，サイクルを許容した有向グラフとしてゲームの状態遷移を表し，これを Quixo における “ゲーム木” と定義する．このとき，ある状態の子ノードとは，その状態から 1 手で遷移可能なすべての状態である．終了状態 (あるマークが揃った列が存在する状態) からは次の状態に遷移できないため，終了状態に子ノードは存在しない．親ノードは子関係の逆であり，ある状態に対し 1 手前の状態を表す．同じ状態への遷移とは，ある状態の先祖ノードが自身と等しくなるような状態遷移列 (サイクル) を意味する．

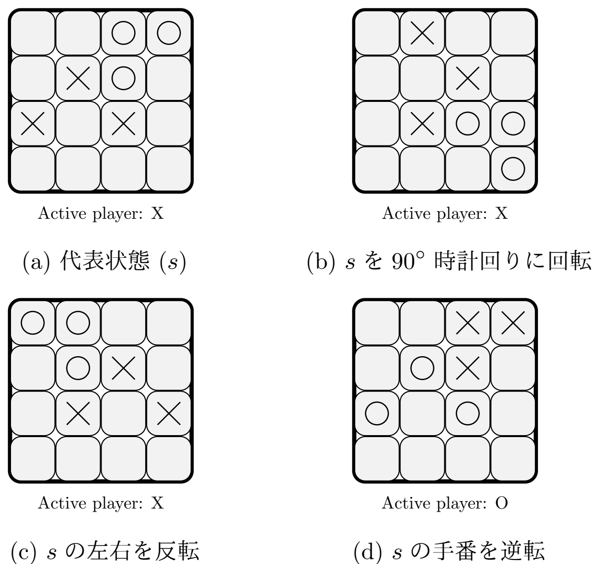
すべての状態には手番プレイヤーにとっての状態価値 (value) があり，これは {Win,Loss,Draw} のいずれかの値をとる．ある状態の value は，少なくとも 1 つの子ノードが Loss の場合に Win，すべての子ノードが Win の場合に Loss，それ以外の場合は Draw となる．

3.3 対称性

囲碁と同様に，Quixo において駒の配置のみを上下左右反転させた状態や 90° 回転させた状態の value は等しくなるため，これらの状態は同じ状態とみなすことができる．同様に，手番を逆転し，盤上の X と O の面を入れ替えた状態の value も同じになり，状態として同一視できる．このような同一視が可能な状態の例を図 3 に示す．

本研究では以上の性質を利用し，手番が X の状態のみを考えることで，計算すべき状態数を 3^{25} に削減する．一方，駒配置を上下左右の反転や回転させた状態は同一視せず，それぞれ別の状態として計算する．これは駒配置の反転や回転を効率的に計算することが難しいためである．

© 2020 Information Processing Society of Japan



(a) 代表状態 (s)　(b) s を 90° 時計回りに回転

(c) s の左右を反転　(d) s の手番を逆転

図 3: value が同一になる状態の例

4. 提案解法

本研究が提案する Quixo の強解決手法は，Value Iteration (4.2.1 項) および Backward Induction (4.2.2 項) から成り立つ．本章ではこれらの手法に先立ち，Quixo を実用的なメモリ量で解決するための状態表現および計算過程の保管方法について説明する．またアルゴリズムを示したのち，実用的な計算時間での解決を実現するに適用した高速化手法について説明する．

4.1 データ構造の最適化

4.1.1 状態表現の最適化

Quixo において各マスは {X, O, blank} の 3 通りの値をとるため，2 ビットで表現可能である．したがって，5×5 グリッドでは 1 状態を $2 \cdot 25 = 50$ ビットで表現可能となり，64 ビット整数 1 つに 1 状態を対応づけることができる．このような方法は一般にビットボード (Bit board) と呼ばれ，オセロやチェスなどにおける状態表現として広く用いられている．具体的な状態表現を図 4 に示す．ここでは簡単のために X および O について 32 ビットごとに分割して管理しているが，パディングは無視しても構わない．

このような状態表現の採用により，事前に用意した適当なビットパターンとビット演算 (shift, and, or, not) を使用することで，状態遷移および終了判定を高速に計算することができる．例として図 5 に示す状態 `s` とビットパターン `A`，`B`，`C` を考える．このとき，`s` は図 2a に示す状態を表現し，`A` は最上段に位置する X の駒の位置を取得するビットパターン，`B` および `C` は図 2 に示す状態遷移操作を実現するためのビットパターンである．これらのビット列を使用することで，状態 `s` について次に示す操作をビット演算で表現できる．

- X と O の取り替え: `s << 32 | s >> 32`
- 駒の存在判定: `s & A != 0`
- 終了状態判定: `s & A == A`
- 状態遷移:
  左から右へ:`(((s & B) << 1) & B) | (s & ~B) | C`
  右から左へ:`(((s & B) >> 1) & B) | (s & ~B) | C`
  上から下へ:`(((s & B) >> 5) & B) | (s & ~B) | C`
  下から上へ:`(((s & B) << 5) & B) | (s & ~B) | C`
  選択駒のマークに依存せず同様の演算を適用可能．

`A`，`B`，`C` は適用する操作に応じて異なるパターンが用いられるが，最低限必要な基底パターンはそれほど多くはない．状態遷移操作は，次の状態への (前方) 遷移だけではなく，前の状態への (後方) 遷移も同様に計算可能である．

4.1.2 結果の保存方法の最適化

ゲームを強解決するため，すべての状態の value を保存する必要がある．value は {Win,Loss,Draw} のいずれかの値をとるため 2 ビットで保存可能である．状態を保存するためには 50 ビット必要であるため，状態とその value をセットで保存する場合，1 状態に対して少なくとも 52 ビットが必要となる．1 状態とその value を 64 ビット整数 (8 バイト) で保存する場合，全状態とその value を保存するには $8 \times 3^{25} \approx 6.8 \times 10^{12} \approx 6.2\,\mathrm{TB}$ の記憶領域が必要となり，これは現実的に妥当なメモリ量ではない．

そこで，状態は保存せずに，value だけを保存する場合を考える．value を 2 ビット (2/8 バイト) で保存すると，全状態の value を保存するためには $2/8 \times 3^{25} \approx 2.1 \times 10^{11} \approx 197\,\mathrm{GB}$ 程度必要となる．ただし，これはすべての value を常に RAM 上に展開しておく場合である．本研究において使用する Backward Induction (4.2.2 項) では，ある状態の value を一部の関連する状態の value のみから決定可能である．したがって，状態空間を分割し，必要な状態の value のみを RAM 上に展開することで，計算時の (アクティブ) メモリ量を削減する．Quixo における状態遷移は，X あるいは O の数が 1 だけ増えるケースあるいは共に増えないケースに分割できるため，本研究では X および O の数によって状態を分割することで状態遷移に関連する状態数を削減するアプローチをとる．同様の手法は，Schaeffer らによるチェッカーの解決において用いられている [16]．説明を簡略化するため，これ以降 X が x 個，O が o 個存在する状態の集合を $\mathcal{C}_{x,o}$ と表記する．ただし，手番は常に X である (3.3 節) ことに注意する．

value のみを保存する場合，状態と value の対応関係を別に管理する必要が生じる．状態はその表現方法から決められた順序での列挙が自明に可能であるため，value を対応する配列に保存し，インデックスから状態への全単射を作成することでこの問題に対処する．ただし，本研究では {X, O} の数により状態空間を分割しているため，$\mathcal{C}_{x,o}$ に含まれるすべての状態から整数 $\left\{0, \ldots, \binom{25}{x} \cdot \binom{25-x}{o} - 1\right\}$

© 2020 Information Processing Society of Japan

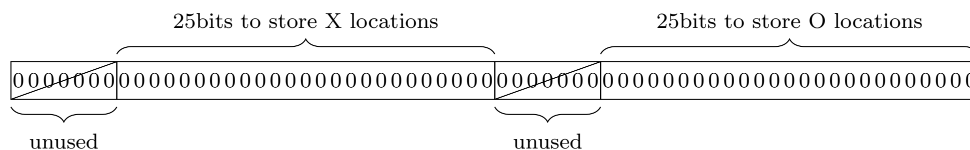


図 4: 64 ビット整数を用いた状態表現 (右が最下位ビット)．1 から 25 ビットが実際の盤の右下から左上の O に対応し，O が存在するとき 1 に，存在しないとき 0 になる．同様に 33 から 57 ビットが X の位置に対応する．

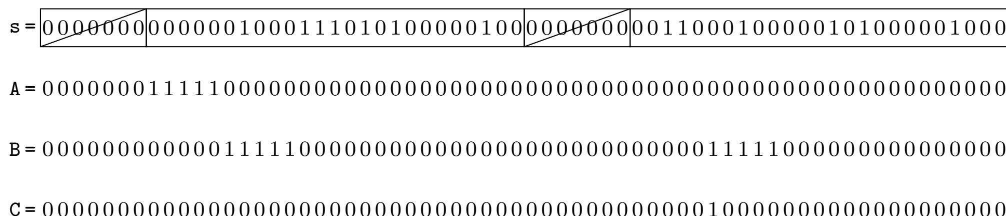


図 5: 図 2a の状態と終了状態判定や状態遷移に使用する `A` および `B`，`C`．`A` は X が最上段にて列をなすかを判定するために使用する (`s & A == A`)．`B` と `C` は図 2a から図 2b への状態遷移を計算する (`(((s & B) >> 1) & B) | (s & ~B) | C`)．

への全単射を作成する必要がある．

以下，状態からインデックスを計算する方法を示す．64 ビットの状態表現から 25 ビットずつの S_X と S_O を得る．このとき，S_X は 25 マスのなかで X が存在する場所を示し，S_O は X が存在するマスを除いた $25-x$ マスにおいて O が存在する場所を示す．つまり S_O は S_X において 1 が立つ桁を詰めたものになる．図 6 に S_X と S_O の一例を示す．S_X, S_O を用いて，$\mathcal{C}_{x,o}$ に含まれる状態からインデックスへの変換は以下のように定義される．

$$ord(S_X) \cdot \binom{25-x}{o} + ord(S_O), \qquad (1)$$

ここで，$ord(\cdot)$ は同じハミング重みを持つ状態に対し順序を定義する関数である．$ord(S_X)$ が $\binom{25}{x}$ 通り，$ord(S_O)$ が $\binom{25-x}{o}$ 通りの出力を持つよう構成することで，式 (1) は状態からインデックスへの全単射を与える式となる．

インデックスから状態を求めるとき，その状態の X と O の数を知る必要がある．そこで，駒の数 (ハミング重み) を返す関数 $pop(\cdot)$ を定義し，$pop(\cdot)$ と式 (1) の逆の計算をすることで，インデックスから状態を計算する．$pop(\cdot)$ 関数と $ord(\cdot)$ 関数の例を表 1 に示す．

状態とインデックスの相互変換について実行速度を向上させるため，$pop(\cdot)$ と $ord(\cdot)$ の計算をあらかじめ行い，テーブルとして保持する．$pop(\cdot)$ の上限は 25，$ord(\cdot)$ の上限は $\binom{25}{12} \leq 2^{32}$ となるため，双方とも 32 ビット整数で構成可能であり，$pop(\cdot)$ と $ord(\cdot)$ への変換テーブルを保持するために合計で $2^{25} \times 32 \times 2 = 256\,\mathrm{MB}$ 使用する．また，$pop(\cdot)$ と $ord(\cdot)$ からの逆変換を行うために，$pop(y)$ と $ord(y)$ の値のペアに対する y の値を保存する．$y < 2^{25}$ であるため, y は 32 ビット整数でも十分に保存可能であるが，状態表現に合わせてここでも 64 ビット整数で保存する．$pop(\cdot)$ と $ord(\cdot)$ の値のペアは 2^{25} 通りあるため，$pop(y)$ と $ord(y)$ から y への逆変換テーブルを保持するために合

表 1: x に対する $pop(x)$ と $ord(x)$ の例．$pop(x)$ は 1 の数，$ord(x)$ は $pop(x)$ が等しい数の集合における順序づけ関数．

x	$pop(x)$	$ord(x)$	x	$pop(x)$	$ord(x)$
00000000	0	0	00001100	2	5
00000001	1	0	00001101	3	2
00000010	1	1	00001110	3	3
00000011	2	0	00001111	4	0
00000100	1	2	00010000	1	4
00000101	2	1	00010001	2	6
00000110	2	2	00010010	2	7
00000111	3	0	00010011	3	4
00001000	1	3	00010100	2	8
00001001	2	3	00010101	3	5
00001010	2	4	00010110	3	6
00001011	3	1	00010111	4	1

計で $2^{25} \times 64 = 256\,\mathrm{MB}$ 使用する．以上より，$pop(\cdot)$ と $ord(\cdot)$ への変換および逆変換の結果を保持するために合計で 512 MB のメモリを使用する．

4.2 基本的な探索アルゴリズム

Quixo の状態数は 1.7×10^{12} 程度となるため，それぞれの状態の value を計算する Value Iteration (4.2.1 項) と状態を分割して計算する Backward Induction (4.2.2 項) を使用して解析する．このため，Value Iteration ではメモリのみを使用して計算可能になる．

本研究と類似したアプローチとして，後退解析 (Retrograde Analysis) [17] がある．田中は後退解析を用いることで，状態数が 9.7×10^7 程度のゲームにおいて，すべての計算過程を RAM 上に展開して後退解析を行った [18]．また Romein らは，状態数が 2.04×10^{11} 程度のゲームに対し，RAM だけでなく補助記憶装置を併用する分散環境にて後退解析を行った [19]．後退解析では終了状態 (勝敗が決定

© 2020 Information Processing Society of Japan

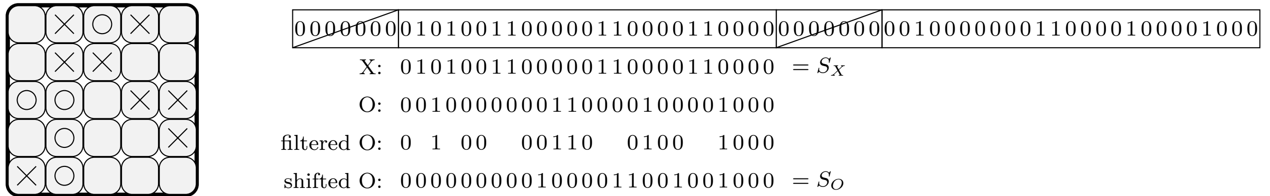


図 6: $\mathcal{C}_{8,5}$ に含まれる状態とその 64 ビット表現および S_X と S_O．S_X と S_O はインデックスを計算するために使用される．

Algorithm 1 Value Iteration (VI)

```
 1: for all states s do
 2:     if there is a line of Xs in s then
 3:         outcome[s] ← Win
 4:     else if there is a line of Os in s then
 5:         outcome[s] ← Loss
 6:     else
 7:         outcome[s] ← Draw
 8: repeat
 9:     for all states s such that outcome[s] = Draw do
10:         if at least one child of s is Loss then
11:             outcome[s] ← Win
12:         else if all children of s are Win then
13:             outcome[s] ← Loss
14: until no update in the last iteration
```

Algorithm 2 Backward Induction using VI internally

```
1: for n = 25 to 0 do
2:     for x = 0 to ⌈n/2⌉ do
3:         o ← n − x
4:         if n < 25 then
5:             Load outcomes of classes C_{x,o+1} and C_{o,x+1}
6:         Compute outcomes of classes C_{x,o} and C_{o,x} using VI
7:         Save outcomes of classes C_{x,o} and C_{o,x}
8:         Unload all outcomes
```

した状態) から探索を開始しその親となる状態を順に解析するのに対し，本研究における Backward Induction ではマークの数が多い状態から探索を開始し，マークの数が 1 つ少ない状態を順に解析する．

4.2.1 Value Iteration

ミニマックス法が初期状態から探索を開始するのに対し，Value Iteration では終了状態，すなわち勝敗が決定した状態の value を決定し，value が決定していない状態を 3.2 節の定義に従って更新する．value の更新がなくなるまで更新を繰り返すことで，すべての状態の value が決定される．

アルゴリズムを Algorithm 1 に示す．1-7 行ではマークが揃っている状態の value を Win または Loss に決定し，マークが揃っていない状態の value は Draw としている．手番が X となる状態のみを考慮しているため， X が揃っているかを先に確かめる．これにより，両方のマークが同時に揃ったとき，最後に手番だったプレイヤーが負けるというルールを満たす．8-14 行では結果が収束するまで更新を繰り返す．最終的に Win または Loss の条件を満たさない状態の value は Draw となる．

4.2.2 Backward Induction（後退帰納法）

Quixo では 1 手で増えるマークの数は 1 か 0 である．本研究では手番のマークは X と仮定しているため増えるマークは常に X となる．つまり $\mathcal{C}_{x,o}$ の子は $\mathcal{C}_{o,x} \bigcup \mathcal{C}_{o,x+1}$ となる．すなわち，$\mathcal{C}_{x,o+1} \bigcup \mathcal{C}_{o,x+1}$ の状態の value のみから，$\mathcal{C}_{o,x} \bigcup \mathcal{C}_{o,x}$ の状態の value が計算可能となる．また，$x + o = 25$ となる状態からはマークが増えることはない．そこで，$x + o = 25$ となる状態集合から value を求めることで，状態集合を分割して計算する．

メモリに保存する状態は $\mathcal{C}_{x,o} \bigcup \mathcal{C}_{o,x} \bigcup \mathcal{C}_{x,o+1} \bigcup \mathcal{C}_{o,x+1}$ となり，$x = 8, o = 8$ のとき最大となる．ただし，実際の計算では $x = o$ のとき，$\mathcal{C}_{x,o}$ と $\mathcal{C}_{o,x}$ は同じ集合になるため，区別せずに計算可能である．$\mathcal{C}_{8,8}$ の状態数は $\binom{25}{8} * \binom{25-8}{8} \approx 2.6 \times 10^{10}$ となる．1 状態の value を保存するために 2 ビット使用するため約 6.1 GB のメモリが必要となる．同じサイズの集合が四つあるため合計で最大約 24.4 GB のメモリが必要になる．これに加えて 0.5 GB 使用する状態とインデックスの変換表 (4.1.2 節) を保存する必要がある．本研究では 32 GB の RAM を使用したため十分計算可能となる．

Value Iteration に加えて Backward Induction を使用したアルゴリズムを Algorithm 2 に示す．これによりすべての状態を解析可能となるが，現実的な時間で計算可能にするために 4.3 節に示す高速化を行う．

4.3 高速化された探索アルゴリズム

4.3.1 親の更新

状態の value が Win になるとき少なくとも 1 つの子が Loss である．言い換えれば Loss の親は必ず Win となるため，Loss の状態が見つかったときに親の value を即座に Win に更新してもよい．この手法は Romein ら [19] も使用している．これにより Algorithm 1 の 10-11 行目を省略でき，子の探索は 12-13 行のすべての子が Win かを調べる部分のみになる．このとき，子の中に少なくとも 1 つでも Win でない状態があれば探索を即座に終了してよいため，計算速度の大幅な向上を見込める．$\mathcal{C}_{x,o}$ の終了状態からの更新部分のアルゴリズムを Algorithm 3 に示す．

© 2020 Information Processing Society of Japan

Algorithm 3 Update outcomes of $\mathcal{C}_{x,o}$ using terminal states

1: **for all** states $s \in \mathcal{C}_{x,o}$ **do**
2: **if** there is a line of Xs in s **then**
3: outcome[s] $\leftarrow Win$
4: **else if** there is a line of Os in s **then**
5: outcome[s] $\leftarrow Loss$
6: **for all** parents $p \in \mathcal{C}_{o,x}$ of s **do**
7: outcome[p] $\leftarrow Win$

Algorithm 4 Update values of $\mathcal{C}_{x,o}$ using inductively-computed values of $\mathcal{C}_{o,x+1}$

1: **for all** states $c \in \mathcal{C}_{o,x+1}$ **do**
2: **if** outcome[c] $= Loss$ **then**
3: **for all** parents $s \in \mathcal{C}_{x,o}$ of c **do**
4: outcome[s] $\leftarrow Win$
5: **else if** outcome[c] $= Draw$ **then**
6: **for all** parents $s \in \mathcal{C}_{x,o}$ of c s.t. outcome[s] $= Draw$ **do**
7: outcome[s] $\leftarrow WinOrDraw$

4.3.2 WinOrDraw の使用

value は 2 ビットで表現されており，最終的には {Win,Loss,Draw} のいずれかの値に決定される．2 ビットでは最大 4 つの value を表現できるため，計算の途中に新たな value として WinOrDraw を導入する．WinOrDraw は最終的に Win または Draw となる状態であることを意味し，言い換えると Loss にならない状態である．WinOrDraw に更新されるのは，自身が Draw (子に Loss を含まない) であり，子の中に少なくとも一つの Draw が存在するときである．

すべての状態を一度に計算する後退解析では，状態が Draw であるかは更新が収束するまでわからない．しかし本研究では状態を分割し，マークの数が多い状態から解析しているため，マークの数が多い状態については Draw であるかが決定している．具体的には，$\mathcal{C}_{x,o}$ の解析をしているとき，$\mathcal{C}_{o,x+1}$ の状態の解析は終了している．

Value Iteration の更新を確認するためには状態のすべての子を確認する必要がある (Algorithm 1 の 10 行および 12 行)．子状態にはすでに value が解析されている状態と，まだ解析されていない状態が存在する．具体的には $\mathcal{C}_{x,o}$ の解析をしているとき，子は $\mathcal{C}_{o,x}$ または $\mathcal{C}_{o,x+1}$ となる．$\mathcal{C}_{o,x}$ の中にはまだ解析されていない状態が含まれるが，$\mathcal{C}_{o,x+1}$ のすべての状態は既に解析されている．

WinOrDraw を導入することで，Algorithm 4 に示す $\mathcal{C}_{o,x+1}$ からの更新操作を一度行えば，その後 $\mathcal{C}_{o,x+1}$ の探索をする必要がないことを以下に示す．$s \in \mathcal{C}_{x,o}$ の $\mathcal{C}_{o,x+1}$ に含まれる子について

- 少なくとも一つの Loss となる状態があるとき，s は即座に Win となる．
- すべての状態が Win であるとき，s は Draw に初期化される．つまり s の子で $\mathcal{C}_{o,x}$ に含まれる状態の value のみによって s が {Win,Loss,Draw} のいずれの値になるか決定される．
- 少なくとも一つの Draw となる状態が存在し Loss となる状態が存在しないとき，s は WinOrDraw となる．s の子で $\mathcal{C}_{o,x}$ に含まれる状態の中で一つでも Loss となる状態があれば s は Win となり，そうでなければ Draw となる．

Algorithm 4 の 6 行目において親の value が Draw であることを確認しているのは，すでに Win である親が WinOrDraw になることを避けるためである．Algorithm 3 の 6 行目では，すべての親ノードが Draw または Win であることが保証されているため，このような確認処理が不要である．

4.2.2 項では一度に 4 つの状態集合をメモリに保存する必要があったが，WinOrDraw の導入により一度に最大 2 つの状態集合を保存するだけで計算が可能となる．具体的には，$\mathcal{C}_{x,o}$ と $\mathcal{C}_{o,x}$ の解析を行うとき，$\mathcal{C}_{o,x+1}$ から $\mathcal{C}_{x,o}$ の更新を行い，結果をストレージに保存する．次に $\mathcal{C}_{x,o+1}$ から $\mathcal{C}_{o,x}$ の更新を行い，結果をストレージに保存する．最後に $\mathcal{C}_{x,o}$ と $\mathcal{C}_{o,x}$ の解析を行えば，一度にメモリに展開する状態集合は最大 2 つとなる．ただし，本研究ではストレージに保存する時間ロスを避けるため，4 つの状態集合をメモリに保存する方法を採用している．

4.3.3 提案アルゴリズムの全体像

本研究で Quixo を解決するために使用した完全なアルゴリズムを Algorithm 5 に示す．Value Iteration および Backward Induction に加え，高速化戦略として親の更新，WinOrDraw を採用している．

提案アルゴリズムには計算を並列化可能な部分が存在し，また並列化の方法も複数考えられる．誌面の都合上ここでは割愛するが，arXiv に登録したプレプリント版 [20] (英文) では，並列計算についても考察を加えている．

4.4 最短勝利手数

Algorithm 5 によりすべての状態の value が計算可能となり，これにより 5×5 以下の Quixo の強解決を行う準備ができた．しかし，たとえ常に自分にとって Win となる状態に遷移してたとしても，サイクルに入るような手を選び続けるとゲームは終了しない．したがって，勝利が確定した盤面から実際の勝利状態に到達する具体的な状態遷移列を抽出するためには，確実にゲームが終了する状態遷移列を選択する戦略が必要となる．

サイクルで遷移し続ける問題を解決する簡単な方法として，自分にとって Win となる候補手の中からランダムに次の 1 手を選択する方法がある．すべての Win となる状態からは自分が勝利となってゲームが終了する状態までのパ

© 2020 Information Processing Society of Japan

Algorithm 5 Complete algorithm used to solve Quixo

1: **for** $n = 25$ **to** 0 **do**
2: **for** $x = 0$ **to** $\lceil n/2 \rceil$ **do**
3: $o \leftarrow n - x$
4: **for all** states $s \in \mathcal{C}_{x,o} \bigcup \mathcal{C}_{o,x}$ **do**
5: outcome$[s] \leftarrow Draw$
6: Update outcomes of $\mathcal{C}_{x,o}$ using terminal states (Algo 3)
7: Update outcomes of $\mathcal{C}_{o,x}$ using terminal states (Algo 3)
8: **if** $n < 25$ **then**
9: Update outcomes of $\mathcal{C}_{x,o}$ (Algo 4)
10: Update outcomes of $\mathcal{C}_{o,x}$ (Algo 4)
11: **repeat**
12: **for all** states $s \in \mathcal{C}_{x,o} \bigcup \mathcal{C}_{o,x}$ such that outcome$[s] = Draw$ **do**
13: **if** all children $c \in \mathcal{C}_{x,o} \bigcup \mathcal{C}_{o,x}$ of s are Win **then**
14: outcome$[s] \leftarrow Loss$
15: **for all** parents $p \in \mathcal{C}_{x,o} \bigcup \mathcal{C}_{o,x}$ of s **do**
16: outcome$[p] \leftarrow Win$
17: **until** no update in the last iteration
18: **for all** states $s \in \mathcal{C}_{x,o} \bigcup \mathcal{C}_{o,x}$ such that outcome$[s] = WinOrDraw$ **do**
19: outcome$[s] \leftarrow Draw$

Algorithm 6 Value Iteration with steps to Win or Loss

1: **for all** states s **do**
2: **if** there is a line of Xs in s **then**
3: outcome$[s] \leftarrow Win$
4: step$[s] \leftarrow 0$
5: **else if** there is a line of Os in s **then**
6: outcome$[s] \leftarrow Loss$
7: step$[s] \leftarrow 0$
8: **else**
9: outcome$[s] \leftarrow Draw$
10: $i \leftarrow 1$
11: **repeat**
12: **for all** states s such that outcome$[s] = Draw$ **do**
13: **if** at least one child of s exists whose outcome$[c]$ is Loss and step$[c]$ is $i - 1$ **then**
14: outcome$[s] \leftarrow Win$
15: step$[s] \leftarrow i$
16: **else if** all children of s are Win **then**
17: outcome$[s] \leftarrow Loss$
18: step$[s] \leftarrow$ max(children steps) $+ 1$
19: $i \leftarrow i + 1$
20: **until** no update in last iteration

スが必ず存在するため，十分長いステップ数 n までにある確率 p で勝利することができる．$n \to \infty$ とすると $p \to 1$ となり，いつかは勝利することができる．

ランダムに行動を選択する戦略はステップ数に関して最適な戦略ではない．そこで Win 状態から最短手数で勝利する状態遷移列を見つけることとする．このとき負ける側の最善行動は，最長手数で負ける手の選択となる．最短勝利手数を計算するため，最終状態までのステップ数を保存する変数 (`step`) を導入する．`step` は以下に定義される．

- 終了状態において，`step` は 0,
- Win 状態において，`step` は Loss となる子の `step` の最小値+1,
- Loss 状態において，`step` は Win となる子の `step` の最大値+1.

最短勝利手数と最長敗北手数を求めるには，基本的な探索アルゴリズムの Value Iteration 部分に `step` 変数の更新を加えれば良い．変更後の Value Iteration 部分のアルゴリズムを Algorithm 6 に示す．10 行目の i はステップ数を表す．i より大きなステップになる可能性があるとき Win の更新は行わない．こうすることで Win に更新されるノードの子に Loss となるノードが他に存在しても，その Loss となるノードの `step` は i 以上であることが保証され，Win となるノードは正しい `step` に更新される．

Algorithm 6 に 4.3 節で説明した高速化を加えることも可能である．一方で，最短勝利手数の計算はメモリサイズに問題がある．終了状態までのステップを 8 ビットで表現できると仮定しても，$\mathcal{C}_{8,8}$ の状態集合の value と step を保存するために約 $6.1 + 6.1 \times 4 = 30.5$ GB 必要となり (4.2.2 項)，これを複数 RAM 上に展開し続けることは難しい．

表 2: それぞれのサイズにおける初期状態の最初のプレイヤーの value と終了状態までのステップ数および計算時間.

	value	steps	computation time
3×3	Win	7	~ 0.1 s
4×4	Win	21	13.4 s
5×5	Draw	∞	$\sim 19\,500$ min

5. 結果

提案解法を用いて，Quixo のすべての状態における {Win, Loss, Draw} を計算する．誌面の都合上ここでは重要な結果のみを示すが，arXiv 上のプレプリント版 [20] にてより詳細な解析結果を報告している．

本研究では，典型的な 5×5 の Quixo だけでなく，4×4 および 3×3 についても強解決を行った．このとき，計算には Core i9 9960X CPU，32 GB RAM を搭載した Ubuntu 18.04LTS サーバを使用した．

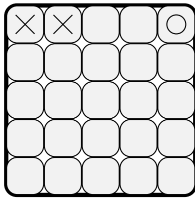
それぞれのボードサイズに対する提案解法による計算結果を表 2 に示す．表中の value は初期状態が先手プレイヤーにとって {Win, Loss, Draw} のいずれであるかを表し，step は最適なプレイヤーを仮定した時にゲーム終了までに要する最小手数，computation time はシングルスレッドで計算を行った際の計算時間を表す．最適なプレイヤーとは，Win 状態では最短手数で勝つ手，Loss 状態では最長手数で負ける手，Draw 状態では Draw 状態をキープする手を選択するプレイヤーを意味する．

表 2 より，典型的な 5×5 の Quixo は初期状態において Draw であることを明らかにした．したがって，プレイヤーが互いに最適な戦略をとれば Draw 状態のループとなりゲームが終了しないため，step は ∞ となる．5×5 の

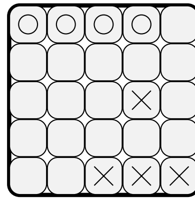
© 2020 Information Processing Society of Japan

表 3: 勝ち，負け，引き分けの状態数 (5×5)

勝ち	負け	引き分け
441,815,157,309	279,746,227,956	125,727,224,178



(a) X の勝ちとなる状態の中でもっとも駒が少ないものの一つ.



(b) x と o の数が同一で，X の負けとなる状態の中でもっとも駒が少ないものの一つ.

図 7: 5×5 Quixo の特徴的な状態．手番は X.

計算には約 19500 分 (約 325 時間) を要しており，これは 4×4 および 3×3 と比較して明らかに長い．アルゴリズムの一部を並列化し，32 スレッドの並列計算をした場合には，これを約 1900 分 (約 32 時間) まで短縮できた.

4×4 および 3×3 については，初期状態が先手にとって Win であることを明らかにした．このとき，4.4 節に示す方法により最短勝利手数を求めることができ，それぞれ表記の通りとなった．また，初期状態から先手プレイヤーが最短手数で勝利するための具体的な手順も明らかにしているが，本紙では割愛する.

5×5 Quixo の詳細解析

計算したすべての状態のうち，勝ち，負け，引き分けとなる状態数を表 3 に示す．初期状態は引き分けだが，引き分けの状態数の割合は 15% 以下で，勝敗が決まっている状態の方が多いことがわかる.

図 7 に初期状態に近く特徴的な状態を示す．図 7a は勝敗が決定している状態のなかでもっともマークの数が少ない状態の一つである．この状態に到達するまでに O は自分のマークを増やさない行動をしている．さらに，$\mathcal{C}_{2,1}$ に含まれるすべての状態で X が勝利することがわかった．一方，図 7b はお互いに自分のマークを増やし続けて，最短で先手が負ける状態である.

6. 結論

本研究ではで二人プレイの Quixo の到達不可能な状態を含むすべての状態の結果を解明し，初期状態は引き分けとなることを示した．また，初期状態に近く勝敗が決定している状態などいくつかの特徴的な状態を例示した．さらに，4×4 のサイズでは先手が最短 21 ステップで勝利し，3×3 のサイズでは先手が 7 ステップで勝利することを示した.

本研究では 5×5 以下のサイズにおける Quixo を強解決したが，$n \times n$ のサイズに一般化された Quixo における $n > 5$ の結果は明らかではない．n 個のマークの列を揃えることは n が大きくなるにつれて難しくなると考えられるが，その証明は今後の課題である．また，$n \times n$ のサイズで 5 個のマークを揃えることが可能か否かも明らかではない.

Quixo の結果を解明したが，人間が最適な戦略でプレイすることは依然として難しい．5×5 のサイズでは結果が引き分けとなるが，すべての状態における最適な戦略を記憶することはおそらく不可能である．人間が理解できる戦略を発見することも今後の課題である.

参考文献

[1] Quixo page on Gigamic website, `https://en.gigamic.com/game/quixo`. Accessed: 2020-10-11.

[2] Quixo page on BoardGameGeek website, `https://boardgamegeek.com/boardgame/3190/quixo`. Accessed: 2020-10-11.

[3] Bouton, C. L.: Nim, a game with a complete mathematical theory, *The Annals of Mathematics*, Vol. 3, No. 1/4, pp. 35–39 (1901).

[4] Bowling, M., Burch, N., Johanson, M. and Tammelin, O.: Heads-up limit hold'em poker is solved, *Science*, Vol. 347, No. 6218, pp. 145–149 (online), DOI: 10.1126/science.1259433 (2015).

[5] van der Werf, E. C. and Winands, M. H.: Solving Go for rectangular boards, *Icga Journal*, Vol. 32, No. 2, pp. 77–88 (2009).

[6] Gardner, M.: *The Scientific American Book of Mathematical Puzzles and Diversions*, Simon and Schuster (1959).

[7] Allis, L. V.: A Knowledge-Based Approach of Connect-Four., *ICGA Journal*, Vol. 11, No. 4, p. 165 (1988).

[8] Allis, L. V., van den Herik, H. J. and Huntjens, M. P. H.: Go-Moku Solved by New Search Techniques, *Computational Intelligence*, Vol. 12, No. 1, pp. 7–23 (1996).

[9] Mishiba, S. and Takenaga, Y.: QUIXO is EXPTIME-complete, *Information Processing Letters*, Vol. 162, p. 105995 (2020).

[10] Mensa Select Top 5 Best Games (1995). in USA.

[11] Games Magazine "Games 100 Selection" (1995). in USA.

[12] Games Magazine "Best New Strategy Game" (1995). in USA.

[13] Parent's Choice Gold Award (1995). in USA.

[14] As d'Or Festival International des Jeux (1995). in Cannes, France.

[15] Oscar du Jouet-Toy Oscar (1995). in Paris, France.

[16] Schaeffer, J., Burch, N., Björnsson, Y., Kishimoto, A., Müller, M., Lake, R., Lu, P. and Sutphen, S.: Checkers is solved, *science*, Vol. 317, No. 5844, pp. 1518–1522 (2007).

[17] Thompson, K.: Retrograde analysis of certain endgames., *J. Int. Comput. Games Assoc.*, Vol. 9, No. 3, pp. 131–139 (1986).

[18] 田中哲朗：ボードゲーム「シンペイ」の完全解析，情報処理学会論文誌，Vol. 48, No. 11, pp. 3470–3476 (2007).

[19] Romein, J. W. and Bal, H. E.: Solving awari with parallel retrograde analysis, *Computer*, Vol. 36, No. 10, pp. 26–33 (2003).

[20] Tanaka, S., Bonnet, F., Tixeuil, S. and Tamura, Y.: Quixo Is Solved (2020).

© 2020 Information Processing Society of Japan