情報工学実験 1

AJG23055 牧野唯希

2025/05/12

A. 目的

二人零和有限確定完全情報ゲームについて、MINIMAX 法を用いて、「共円ゲーム」と「3 to 15 ゲーム」が、『必ず先手が勝てる』か『必ず後手が勝てる』か『必ず引き分けに持ち込める』のいずれに該当するかを実装し検討する。

具体的なルールは以下の通りである。

a. 共円ゲームのルール

3*3 の正方格子を場としておく。先手プレイヤから交互に一つずつ格子点に石を置く。そのときに同一円周上に四点以上石があるような状態（共円）を作ってしまうと即座に負けとなる。負けなかったプレイヤ，つまり，相手に共円を作らせたプレイヤの勝ちとなる。(実際のゲームでは相手が共円を作ったとしても，指摘がなければゲームは続きますが，今回は考慮しないものとする)

b. 3 to 15 ゲームのルール

最初に 1-9 までの 1 枚ずつのカードを 2 人の真ん中に（以降「場」とします）置く。先手と後手を決めて先手から場のカードを 1 枚ずつ取り自分のカードとする。自分のカードのうちいずれかちょうど 3 枚で合計が 15 になれば勝ち。9 枚ともとり終えたとき(先手が 5 枚，後手は 5 枚)どちらも 15 を作れなければ引き分けとなる。

ただし今回は、0 のカードを追加した 10 枚でゲームを行う。

B. 結果予測

a. 共円ゲームの場合

4 つの点を正方形を作るように設置した場合、それは同一円上に存在すると言える。そのため、9 点全てを置き終わった時に、引き分けるということは存在せず、『必ず先手が勝てる』か『必ず後手が勝てる』のいづれかに該当すると考えられる。

そこで https://yambi.jp/kyouen/を用いて数回試行したところ 5 つ目の点を置いた時点でそれ以上置くことが出来なくなったので、『必ず先手が勝てる』と予想する。

b. 3 to 15 ゲームの場合

友達と数回試行したところ最初に先手がとるカードによって結果が異なると感じた。最初に取る値から最適解があり、最終的には、『必ず先手が勝利する』と予想する。

C. 使用機器

ノートパソコン

使用エディターは Replit（https://replit.com/）を用い、標準搭載されている、エラーチェッカー機能を使用した。

D. 実験方法

各二人零和有限確定完全情報ゲームについて、授業内で与えられた、3 目並べについて MINIMAX 法を行うプログラムをもとに変更を加えて実装した。

a. 共円ゲームの場合

今回行う共円ゲームには、共円が発生する点の組み合わせが全部で 14 通り存在する。それは以下の図 1-1 から図 1-14 の通りである。

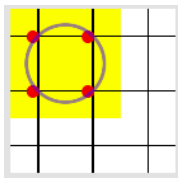

図 1-1

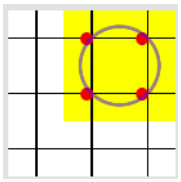

図 1-2

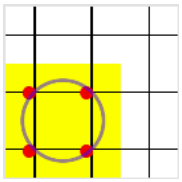

図 1-3

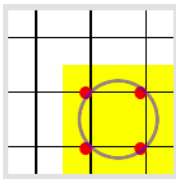

図 1-4

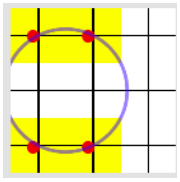

図 1-5

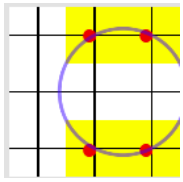

図 1-6

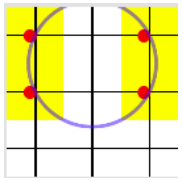

図 1-7

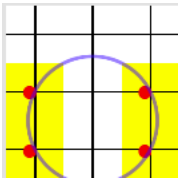

図 1-8

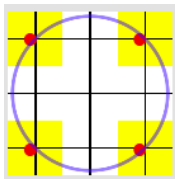

図 1-9

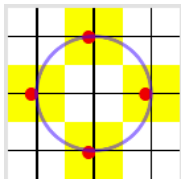

図 1-10

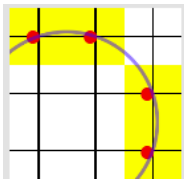

図 1-11

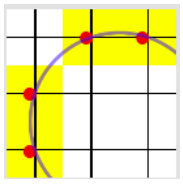

図 1-12

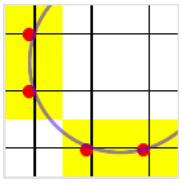

図 1-13

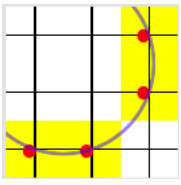

図 1-14

そこで、盤上に点が置かれた状態を 1、何も置かれていない状態を 0 として、盤面の 1 の組み合わせが上記の組み合わせを満たしたときに最後の点を置いた人が負けという方法で MINIMAX 法を実装した。

具体的には、以下のような関数などを用いた。

関数名	内容
check 関数	共円が出来たかの判定。
evaluate 関数	共円が出来た状態のスコアを-1、それ以外を 0 とする
available_moves 関数	盤上で値が 0（点が置かれていない）の点を返す
minimax 関数	ミニマックス法を行い探索を行う。

minimax 関数で判明したスコアに応じた場所に点を置き続け、共円が出来るまで繰り返すことで、検証した。

それぞれの関数は以下の通りである。

- available_moves 関数

```
1. def available_moves(board):
2.     """利用可能な手の一覧を返す関数"""
3.     moves = []
4.     for i in range(9):
5.         if board[i] == 0:
6.             moves.append(i)
7.     return moves
8.
```

- check 関数

```
1. def check(board, turn):
2.     """ゲーム盤面を評価する関数"""
3.     for a, b, c, d in kyouen:
4.
5.         if board[a] == board[b] == board[c] == board[d] == 1:
6.             return True
7.     return False
8.
```

- evaluate 関数

```
1. def evaluate(board, turn):
2.     if check(board, turn):
3.         return -1  # 直前に置いた手番の負け
4.     if not available_moves(board):
5.         return 0   # 引き分け
6.     return None    # ゲーム続行
7.
```

- minimax 関数

```
1. def minimax(board, depth, Maximizing, turn):
```

```
    """
    ミニマックス法で次の手を決定する
    board: 現在の盤面
    depth: 探索の深さ (完全に探索したい場合は 9 とする)
    Maximizing: True ならoを置く手番、False ならxを置く手番
    """
    # 終端状態(勝敗がついているまたは引き分け)の場合は評価値を返す
    result = evaluate(board, turn)
    if result is not None:
        return result


    for move in available_moves(board):
        board[move] = 1 # 点を置く
        score = minimax(board, depth - 1, True, turn + 1) # 深さ 9 で完全に探索
        board[move] = 0 # 点を戻す
        if score == -1: # 相手が負ける＝自分の勝ち
            return 1 # 直前に置いた手番の勝ち

    return -1 # どこに置いても相手を負けにできない＝自分の負け

```

また、これらの関数を play_minimax 関数を使って呼び出した。

```
def play_minimax():
    """ミニマックス法でゲームを実行する"""
    board = [0] * 9
    turn = 0

    while True:
        print(board[0], "|", board[1], "|", board[2])
        print("--+--+--")
        print(board[3], "|", board[4], "|", board[5])
        print("--+--+--")
        print(board[6], "|", board[7], "|", board[8])

        moves = available_moves(board)
        if not moves:
            print("引き分けです")
            break

        best_move = None
        for move in moves:
            board[move] = 1
            score = minimax(board, 9, False, turn + 1)  # 深さ 9 で完全に探索
            board[move] = 0
            if score == 1:
                best_move = move
                break

        if best_move is None:
            best_move = moves[0]

```

```
        board[best_move] = 1 # 石を置く

        if turn % 2 == 0:
            print("player0: ", best_move)
            if check(board, turn):
                print("共円ができたので、後手が必ず勝つ")
                break
        else:
            print("player1: ", best_move)
            if check(board, turn):
                print("共円ができたので、先手が必ず勝つ")
                break

        turn += 1

```

なお、共円の組み合わせとしては、最初に説明した 14 通りであり、配列としては以下の通りである。

```
kyouen = [[0,1,3,4],[1,2,4,5],[3,4,6,7],
          [4,5,7,8],[0,1,6,7],[1,2,7,8],
          [0,2,3,5],[3,5,6,8],[0,2,6,8],
          [1,3,5,7],[0,1,5,8],[1,2,3,6],
          [0,3,7,8],[2,5,6,7]]

```

b. 3 to 15 ゲームの場合

3 to 15 ゲームは魔法陣上での三目並べと全く等しいゲームと考えることが出来る。しかし、今回のゲームは 0 のカードが追加されるため、そのまま実装することはできない。そこで、いくつかの変更を加えた。

まず、15 が出来る組み合わせとして、魔法陣の縦横斜めの組み合わせのほかに、[0, 7, 8]、[0, 6, 9]を追加した。

また、探索の深さを 11 に変更した。

使用した関数は以下の通りである。

関数名	内容
evaluate 関数	先手がそろったら 1、後手がそろったら-1、引き分けは 0 を返す
available_moves 関数	まだ指定されていないカードを返す
minimax 関数	ミニマックス法を行い探索を行う。

それ以外にも小さな変更を多数加え、詳細はプログラムのコメントアウトに表示し

た。

- available_moves 関数

```
def available_moves(board):
    """利用可能な手の一覧を返す関数"""
    moves = []
    for i in range(10):  # 0 から 9 までの数字を使用
        if board[i] != "o" and board[i] != "×":
            moves.append(i)
    return moves

```

- evaluate 関数

```
def evaluate(board):
    """ゲーム盤面を評価する関数"""
    # 横、縦、斜めのラインを確認
    for a, b, c in winlines:
        # oが 3 つ揃っている場合および、0,6,9 と 0,7,8 が揃っている場合
        if board[a] == board[b] == board[c] == "o":
            return 1  # oの勝ち
        # ×が 3 つ揃っている場合および、0,6,9 と 0,7,8 が揃っている場合
        elif board[a] == board[b] == board[c] == "×":
            return -1  # ×の勝ち

    # 空きマスがない場合 (引き分け)
    if not available_moves(board):
        return 0

    # それ以外の場合はゲームが続いている
    return None

```

- minimax 関数

```
def minimax(board, depth, Maximizing):
    """
    ミニマックス法で次の手を決定する
    board: 現在の盤面
    depth: 探索の深さ (完全に探索したい場合は 9 とする)
    Maximizing: True ならoを置く手番、False ならxを置く手番
    """
    # 終端状態(勝敗がついているまたは引き分け)の場合は評価値を返す
    score = evaluate(board)
    # print(score)
    if score is not None:
        return score

    # 最大化プレイヤー(oを置く手番)
```

```
    if Maximizing:
        best_score = -200  # 初期値を十分小さい値に設定
        for move in available_moves(board):
            board[move] = "o"
            score = minimax(board, depth - 1, False)
            if score == 1:
                board[move] = move  # ロジックを戻す
                return 1
            if depth <= 7:
                score = score + 1
            elif depth <= 5:
                score = score + 2
            board[move] = move  # ロジックを戻す
            best_score = max(best_score, score)
        # print(best_score)
        return best_score

    # 最小化プレイヤー(xを置く手番)
    else:
        best_score = 200  # 初期値を十分大きい値に設定
        for move in available_moves(board):
            board[move] = "x"
            score = minimax(board, depth - 1, True)
            if score == -1:
                board[move] = move  # ロジックを戻す
                return -1
            if depth <= 7:
                score = score - 1
            elif depth <= 5:
                score = score - 2
            board[move] = move  # ロジックを戻す
            best_score = min(best_score, score)
        # print(best_score)
        return best_score

```

これらの関数を先ほどと同様 play_minimax 関数を使って呼び出した。

```
def play_minimax():
    """ミニマックス法でゲームを実行する"""
    board = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
    turn = 0
    while True:
        print("----------")
        print(board[2], "|", board[9], "|", board[4])
        print("--+---+---")
        print(board[7], "|", board[5], "|", board[3])
        print("--+---+---")
        print(board[6], "|", board[1], "|", board[8])
        print("----------\n")
        print(board[0], "|", board[6], "|", board[9])
        print("----------")
        print(board[0], "|", board[7], "|", board[8])

        if turn % 2 == 0:  # oの手番
            moves = available_moves(board)
```

```
            if not moves:
                print("引き分けです")
                print("よって必ず引き分ける")
                break
            best_move = None
            best_score = -2
            for move in moves:
                board[move] = "o"
                score = minimax(board, 11, False)
                board[move] = move
                if score > best_score:
                    best_score = score
                    best_move = move
            board[best_move] = "o"
            print(f"Player0 の手: {best_move}")
            print(f"Player0 の手の評価値: {best_score}")
        else:  # ×の手番
            moves = available_moves(board)
            if not moves:
                print("引き分けです")
                print("よって必ず引き分ける")
                break
            best_move = None
            best_score = 2
            for move in moves:
                board[move] = "×"
                score = minimax(board, 11, True)
                board[move] = move
                if score < best_score:
                    best_score = score
                    best_move = move
            board[best_move] = "×"
            print(f"Player1 の手: {best_move}")
            print(f"Player1 の手の評価値: {best_score}")

        turn += 1

        # 勝敗判定
        winner = evaluate(board)
        if winner == 1:
            print("Player0 の勝ちです")
            print("よって必ず先手が勝利する")
            break
        elif winner == -1:
            print("Player1 の勝ちです")
            print("よって必ず後手が勝利する")
            break

```

なお、15 が出来る組み合わせは以下の通りとした。

```
winlines = [[1,5,9],[1,6,8],[2,6,7],
            [2,5,8],[2,4,9],[3,4,8],
```

```
3.             [3,5,7],[4,5,6],[0,6,9],
4.             [0,7,8]]
5.
```

E. 実験結果

a. 共円ゲームの場合

```
 1. 0 | 0 | 0
 2. --+--+--
 3. 0 | 0 | 0
 4. --+--+--
 5. 0 | 0 | 0
 6. player0:  0
 7. 1 | 0 | 0
 8. --+--+--
 9. 0 | 0 | 0
10. --+--+--
11. 0 | 0 | 0
12. player1:  1
13. 1 | 1 | 0
14. --+--+--
15. 0 | 0 | 0
16. --+--+--
17. 0 | 0 | 0
18. player0:  3
19. 1 | 1 | 0
20. --+--+--
21. 1 | 0 | 0
22. --+--+--
23. 0 | 0 | 0
24. player1:  2
25. 1 | 1 | 1
26. --+--+--
27. 1 | 0 | 0
28. --+--+--
29. 0 | 0 | 0
30. player0:  7
31. 1 | 1 | 1
32. --+--+--
33. 1 | 0 | 0
34. --+--+--
35. 0 | 1 | 0
36. player1:  4
37. 共円ができたので、先手が必ず勝つ
```

よってお互いに最適手を指した場合、予想していた通り、『必ず先手が勝てる』とわかった。

b. 3 to 15 ゲームの場合

```
 1. ----------
```

```
 2 | 9 | 4
--+---+---
7 | 5 | 3
--+---+---
6 | 1 | 8
----------

0 | 6 | 9
----------
0 | 7 | 8
Player0 の手: 2
Player0 の手の評価値: 0
----------
○ | 9 | 4
--+---+---
7 | 5 | 3
--+---+---
6 | 1 | 8
----------

0 | 6 | 9
----------
0 | 7 | 8
Player1 の手: 0
Player1 の手の評価値: 1
----------
○ | 9 | 4
--+---+---
7 | 5 | 3
--+---+---
6 | 1 | 8
----------

× | 6 | 9
----------
× | 7 | 8
Player0 の手: 4
Player0 の手の評価値: 1
----------
○ | 9 | ○
--+---+---
7 | 5 | 3
--+---+---
6 | 1 | 8
----------

× | 6 | 9
----------
× | 7 | 8
Player1 の手: 1
Player1 の手の評価値: 1
----------
○ | 9 | ○
--+---+---
7 | 5 | 3
--+---+---
6 | × | 8
```

```
----------

× | 6 | 9
----------
× | 7 | 8
Player0 の手: 3
Player0 の手の評価値: 1
----------
○ | 9 | ○
--+---+---
7 | 5 | ○
--+---+---
6 | × | 8
----------

× | 6 | 9
----------
× | 7 | 8
Player1 の手: 5
Player1 の手の評価値: 1
----------
○ | 9 | ○
--+---+---
7 | × | ○
--+---+---
6 | × | 8
----------

× | 6 | 9
----------
× | 7 | 8
Player0 の手: 8
Player0 の手の評価値: 1
Player0 の勝ちです
よって必ず先手が勝利する

```

よって、3 to 15 ゲームでは、お互い最適手を指した場合、『必ず先手が勝利する』と分かった。

F. 検討考察

まず、共円ゲームについて検討考察する。

共円ゲームは同一円上に 4 つの点を並べてしまうと負けというルールであった。マス数は 3×3 の 9 マスのため、共円が出来るパターンについて人力で数えることが出来た。また、最終局面を全て予想できたので、ミニマックス法を適応させて勝敗を予測することが出来た。そこで、4×4 の共円ゲームだとどうなるだろうと数えてみたところ、100 パターン

以上存在することが分かり、別の方法が必要だと思った。例えば、座標上の 4 点が同一円上に存在するときの条件である。「円周角の定理の逆」や「方べきの定理の逆」、「複素数平面上の定理 ($\frac{(\beta-\gamma)(\alpha-\delta)}{(\alpha-\gamma)(\beta-\delta)}$が実数)」を用いる必要がありそうだ。今回は時間がなくてそこまで手を回すことはできなかったが、また挑戦してみたいと思う。

次に、3 to 15 ゲームについて検討考察する。

3 to 15 ゲームでは 1〜9 の 9 枚で行った場合、『必ず引き分けに持ち込める』ことが知られている。この検証方法として、3*3 のマス目に魔法陣の配置で 1〜9 の番号を配置して、三目並べを行うことで、ミニマックス法を用いて、三目並べと同じプログラムで検証することが出来る。しかし、0 のカードが加わったとこにより、縦・横・斜めといった並べ方は使用できなくなった。そこで、今回は視覚的に理解できるように 0,6,9 と 0,7,8 の並びを追加させた。ただ、コンビネーションを使って組み合わせを計算することで、全パターンを用意しなくても、判別することが出来そうだと感じた。

また、両方のゲームに言えるが、より早く決着をつけるようにプログラムを改良したほうが自然な試合に見えると感じた。