2025.1.4

プロジェクト構造の提案

以下のようにファイルやモジュールを分割すると整理しやすいです：

**go\_types.py**

囲碁で使う基本的な型やデータ構造を定義します。

例: Stone（白石/黒石/空点のEnum）、Player（プレイヤー情報）、GameMode（人対人 or 人対ボット）。

**go\_board.py**

囲碁盤のモデルと関連するロジックを記述します。

例:

碁盤の初期化

石の配置

ルールチェック（陣地、アタリ、コウ、セキなどの判定）

**go\_game.py**

対局全体を管理するクラス/モジュールです。

例:

プレイヤーの管理

ターンの進行

終局判定

**main.py**

プログラムのエントリーポイント。

ここでユーザーインターフェースを統合します（CLI、GUI、またはウェブ）。

**bot.py**

人対ボットの場合のAIロジック。

シンプルなランダムムーブから始め、後でアルゴリズム（例: モンテカルロ木探索）を追加可能。

具体的な取り組み方

ステップ1: 碁盤の初期化と描画

go\_board.pyで、碁盤の状態を管理するクラスを作成。

碁盤をCLI上に表示できる簡単な機能を実装。

ステップ2: 石の配置ロジック

指定した座標に石を置く処理を作成。

配置が合法かを確認するロジック（ルールチェック）を追加。

ステップ3: 対局の進行

ターンごとの進行管理を実装（人対人）。

人対ボットは後で追加。

ステップ4: ゲームモードの切り替え

go\_types.pyにモード選択（人対人 or 人対ボット）用の設定を実装。

モードに応じた動作を変更。

基本クラス例

**go\_types.pyのサンプル**

python

コードをコピーする

from enum import Enum

class Stone(Enum):

EMPTY = 0

BLACK = 1

WHITE = 2

class Player(Enum):

HUMAN = "human"

BOT = "bot"

class GameMode(Enum):

HUMAN\_VS\_HUMAN = "human\_vs\_human"

HUMAN\_VS\_BOT = "human\_vs\_bot"

**go\_board.pyのサンプル**

python

コードをコピーする

class GoBoard:

def \_\_init\_\_(self, size=19):

self.size = size

self.board = [[Stone.EMPTY for \_ in range(size)] for \_ in range(size)]

def place\_stone(self, x, y, stone):

if self.board[y][x] == Stone.EMPTY:

self.board[y][x] = stone

return True

return False

def display\_board(self):

for row in self.board:

print(" ".join(stone.name[0] for stone in row))

**１．以下は、提案した go\_types.py のコードです。**

python

コードをコピーする

from enum import Enum

class Stone(Enum):

"""

囲碁の石の状態を表すEnumクラス。

- EMPTY: 空点

- BLACK: 黒石

- WHITE: 白石

"""

EMPTY = 0

BLACK = 1

WHITE = 2

class Player(Enum):

"""

プレイヤーを表すEnumクラス。

- HUMAN: 人間プレイヤー

- BOT: ボットプレイヤー（将来用）

"""

HUMAN = "human"

BOT = "bot"

class GameMode(Enum):

"""

ゲームモードを表すEnumクラス。

- HUMAN\_VS\_HUMAN: 人対人の対局

- HUMAN\_VS\_BOT: 人対ボットの対局（将来用）

"""

HUMAN\_VS\_HUMAN = "human\_vs\_human"

HUMAN\_VS\_BOT = "human\_vs\_bot"

このコードのポイント

Stone

碁盤の各マスが持つ状態を表現。

デフォルトは EMPTY で、プレイ中に BLACK または WHITE に変化。

Player

プレイヤーが人間かボットかを区別。

現時点では、人間プレイヤーのみ対応。

GameMode

現時点では HUMAN\_VS\_HUMAN のみを使用。

今後、ボットを追加する際に HUMAN\_VS\_BOT を活用。

**2. go\_board.py の実装**

完全版 go\_board.py

python

コードをコピーする

from go\_types import Stone

class GoBoard:

"""

囲碁盤の状態を管理するクラス。

- 碁盤のサイズ（デフォルト19x19）

- 石を置く操作

"""

def \_\_init\_\_(self, size=19):

"""

碁盤を初期化します。

Args:

size (int): 碁盤のサイズ。デフォルトは19。

"""

self.size = size

# 碁盤を2次元リストで初期化

self.board = [[Stone.EMPTY for \_ in range(size)] for \_ in range(size)]

def place\_stone(self, x, y, stone):

"""

指定座標に石を置きます。

Args:

x (int): 碁盤の列インデックス (0～size-1)。

y (int): 碁盤の行インデックス (0～size-1)。

stone (Stone): 置く石の種類 (Stone.BLACK または Stone.WHITE)。

Returns:

bool: 石が正常に置けた場合は True、そうでなければ False。

"""

if 0 <= x < self.size and 0 <= y < self.size: # 範囲チェック

if self.board[y][x] == Stone.EMPTY: # 空いている場合のみ石を置ける

self.board[y][x] = stone

return True

return False

def get\_board(self):

"""

現在の碁盤の状態を返します。

Returns:

list: 2次元リストで碁盤の状態を表します。

"""

return self.board

def display\_board(self):

"""

碁盤をテキスト形式で表示します（デバッグ用）。

"""

for row in self.board:

print(" ".join(stone.name[0] for stone in row))

クラスの概要

\_\_init\_\_

碁盤サイズ（19x19 など）を初期化。

碁盤は Stone.EMPTY で埋められた2次元リスト。

place\_stone

指定した座標に石を置きます。

座標が碁盤の範囲内であり、空いている場合のみ石を置けます。

正常に置けた場合は True を返します。

get\_board

現在の碁盤の状態（Stone の2次元リスト）を返します。

display\_board

碁盤をテキスト形式で表示します（デバッグや開発用）。

**3. main.py の GUI 実装**

以下は tkinter を用いたシンプルな GUI の例です：

python

コードをコピーする

import tkinter as tk

from go\_types import Stone

from go\_board import GoBoard

class GoGameGUI:

def \_\_init\_\_(self, root, board\_size=19):

self.root = root

self.board\_size = board\_size

self.board = GoBoard(size=board\_size)

self.current\_turn = Stone.BLACK

self.cell\_size = 30 # セルの大きさ（ピクセル）

self.canvas = tk.Canvas(root, width=board\_size \* self.cell\_size, height=board\_size \* self.cell\_size, bg="tan")

self.canvas.pack()

self.canvas.bind("<Button-1>", self.handle\_click)

self.draw\_grid()

def draw\_grid(self):

"""碁盤の目を描画"""

for i in range(self.board\_size):

self.canvas.create\_line(self.cell\_size // 2, (i + 0.5) \* self.cell\_size,

self.board\_size \* self.cell\_size - self.cell\_size // 2, (i + 0.5) \* self.cell\_size)

self.canvas.create\_line((i + 0.5) \* self.cell\_size, self.cell\_size // 2,

(i + 0.5) \* self.cell\_size, self.board\_size \* self.cell\_size - self.cell\_size // 2)

def handle\_click(self, event):

"""クリック処理"""

x = int(event.x // self.cell\_size)

y = int(event.y // self.cell\_size)

if self.board.place\_stone(x, y, self.current\_turn):

self.draw\_stone(x, y, self.current\_turn)

self.current\_turn = Stone.BLACK if self.current\_turn == Stone.WHITE else Stone.WHITE

def draw\_stone(self, x, y, stone):

"""石を描画"""

x0 = x \* self.cell\_size + self.cell\_size // 4

y0 = y \* self.cell\_size + self.cell\_size // 4

x1 = (x + 1) \* self.cell\_size - self.cell\_size // 4

y1 = (y + 1) \* self.cell\_size - self.cell\_size // 4

color = "black" if stone == Stone.BLACK else "white"

self.canvas.create\_oval(x0, y0, x1, y1, fill=color)

if \_\_name\_\_ == "\_\_main\_\_":

root = tk.Tk()

root.title("囲碁ゲーム（人対人）")

app = GoGameGUI(root)

root.mainloop()

機能のポイント

碁盤描画

tk.Canvas を使い、19x19の碁盤を作成。

石を置くロジック

マウスクリックで座標を取得し、碁盤に石を置く。

石が置けたら黒/白のターンを切り替える。

GUIの利便性

シンプルなレイアウトで動作し、視覚的に囲碁がプレイできる。

次の目標

石の合法性チェック（自殺手、コウの判定など）を追加。

終局の判定や地の数え上げ。

このコードを試しつつ、次に進めたい機能を決めていきましょう！

**go\_board.pyを書き換えたあとmain.pyを実行すると碁盤が表示され、交互に黒白とうつことができるようになりました。次は、（１）囲んだ後，石をとる（２）囲まれた中には、石がうてない（自殺手の禁止）（３）（２）の例外で、コウがある、というルールをいれたいです。go\_board.pyの書き換えで良いですか？**

ChatGPT:

ChatGPT

はい、これらのルール（石の取り、自殺手の禁止、コウの判定）を実現するには、go\_board.py を拡張してルールを実装する必要があります。具体的には以下のような手順で進めます。

必要なルールの拡張

囲みルール（石の取り）

石が完全に囲まれた場合、その石を取り除きます。

自己の連結したグループが「呼吸点（自由点）」を持っているかどうかを確認するロジックが必要です。

自殺手の禁止

石を置いた結果、自分のグループが呼吸点を持たない場合、その手を無効とします。

自殺手は例外を持つ（コウのルールを考慮）。

コウの判定

同じ状態に戻る手を禁止します（無限ループ防止）。

一手前の碁盤状態を保存しておき、比較する必要があります。

go\_board.py の拡張案

以下に新しい go\_board.py を示します。追加する主なメソッドは以下の通りです：

get\_liberties: 指定の石（またはグループ）の呼吸点を確認。

remove\_dead\_stones: 囲まれた石を取り除く。

is\_suicidal\_move: 自殺手かどうかを確認。

is\_ko\_violation: コウの状態かどうかを確認。

改訂版 go\_board.py

python

コードをコピーする

from go\_types import Stone

class GoBoard:

def \_\_init\_\_(self, size=19):

self.size = size

self.board = [[Stone.EMPTY for \_ in range(size)] for \_ in range(size)]

self.previous\_board = None # コウ判定のために前の状態を保存

def place\_stone(self, x, y, stone):

if not (0 <= x < self.size and 0 <= y < self.size):

return False # 範囲外

if self.board[y][x] != Stone.EMPTY:

return False # 既に石がある

# 一時的に石を置いて呼吸点や自殺手を確認

self.board[y][x] = stone

if self.is\_suicidal\_move(x, y, stone):

self.board[y][x] = Stone.EMPTY # 無効な手なので戻す

return False

# 石を置いた後、相手の石を取り除く

self.remove\_dead\_stones(Stone.BLACK if stone == Stone.WHITE else Stone.WHITE)

# コウのチェック

if self.is\_ko\_violation():

self.board[y][x] = Stone.EMPTY # 無効な手なので戻す

return False

# 前の状態を保存（コウ用）

self.previous\_board = [row[:] for row in self.board]

return True

def get\_liberties(self, x, y):

"""

指定した石（またはグループ）の呼吸点を取得。

"""

stone = self.board[y][x]

visited = set()

liberties = set()

def dfs(cx, cy):

if (cx, cy) in visited:

return

visited.add((cx, cy))

for nx, ny in self.get\_neighbors(cx, cy):

if self.board[ny][nx] == Stone.EMPTY:

liberties.add((nx, ny))

elif self.board[ny][nx] == stone:

dfs(nx, ny)

dfs(x, y)

return liberties

def remove\_dead\_stones(self, stone):

"""

指定した石のグループで囲まれているものを取り除く。

"""

to\_remove = []

for y in range(self.size):

for x in range(self.size):

if self.board[y][x] == stone and not self.get\_liberties(x, y):

to\_remove.append((x, y))

for x, y in to\_remove:

self.board[y][x] = Stone.EMPTY

def is\_suicidal\_move(self, x, y, stone):

"""

自殺手かどうかを判定。

"""

liberties = self.get\_liberties(x, y)

if liberties:

return False # 呼吸点があるなら自殺手ではない

# 石を置いた後、敵の石を取れる場合も自殺手ではない

opponent = Stone.BLACK if stone == Stone.WHITE else Stone.WHITE

for nx, ny in self.get\_neighbors(x, y):

if self.board[ny][nx] == opponent and not self.get\_liberties(nx, ny):

return False

return True

def is\_ko\_violation(self):

"""

コウの判定：現在の碁盤状態が前の状態と同じなら違反。

"""

if not self.previous\_board:

return False

return self.board == self.previous\_board

def get\_neighbors(self, x, y):

"""

指定座標の隣接点を取得。

"""

neighbors = []

if x > 0:

neighbors.append((x - 1, y))

if x < self.size - 1:

neighbors.append((x + 1, y))

if y > 0:

neighbors.append((x, y - 1))

if y < self.size - 1:

neighbors.append((x, y + 1))

return neighbors

def display\_board(self):

for row in self.board:

print(" ".join(stone.name[0] for stone in row))

追加した機能の説明

get\_liberties

石またはグループの呼吸点を再帰的に探索します。

remove\_dead\_stones

囲まれて呼吸点がなくなった石を取り除きます。

is\_suicidal\_move

石を置いた際、自分のグループが呼吸点を持たない場合に無効とします。

敵の石を取れる場合は自殺手ではないとみなします。

is\_ko\_violation

現在の碁盤状態が前の状態と同じなら、その手を無効にします。

get\_neighbors

指定した座標の上下左右の隣接点を返します。

**囲んでも、画面から囲まれた石がきえません**

ChatGPT:

ChatGPT

囲まれた石が画面から消えない原因として考えられるのは、以下の可能性です：

remove\_dead\_stones メソッドが正しく呼び出されていない

place\_stone メソッド内で石を取り除く処理が適切に動作しているか確認します。

GUI が碁盤の状態を正しく反映していない

石が削除された後、main.py の描画部分で碁盤の状態を再描画する必要があります。

修正案

1. go\_board.py の確認

以下のコードに問題がないか確認します。

python

コードをコピーする

def remove\_dead\_stones(self, stone):

"""

指定した石のグループで囲まれているものを取り除く。

"""

to\_remove = []

for y in range(self.size):

for x in range(self.size):

if self.board[y][x] == stone and not self.get\_liberties(x, y):

to\_remove.append((x, y))

for x, y in to\_remove:

self.board[y][x] = Stone.EMPTY

ここで get\_liberties が正しく動作し、囲まれた石の座標が to\_remove に収集されていることを確認してください。

2. main.py での再描画処理

囲まれた石が削除された後、GUI を再描画する必要があります。以下を追加してください。

修正箇所: handle\_click メソッド

python

コードをコピーする

def handle\_click(self, event):

"""クリック処理"""

x = int(event.x // self.cell\_size)

y = int(event.y // self.cell\_size)

if self.board.place\_stone(x, y, self.current\_turn):

self.redraw\_board() # すべての石を再描画

self.current\_turn = Stone.BLACK if self.current\_turn == Stone.WHITE else Stone.WHITE

追加: redraw\_board メソッド

python

コードをコピーする

def redraw\_board(self):

"""碁盤全体を再描画"""

self.canvas.delete("all") # 現在の描画をクリア

self.draw\_grid() # 碁盤の目を再描画

for y in range(self.board\_size):

for x in range(self.board\_size):

stone = self.board.board[y][x]

if stone != Stone.EMPTY:

self.draw\_stone(x, y, stone)

この処理では、碁盤全体を消去し、GoBoard の状態に基づいて石を再描画します。

修正後の main.py

修正後の main.py は以下のようになります：

python

コードをコピーする

import tkinter as tk

from go\_types import Stone

from go\_board import GoBoard

class GoGameGUI:

def \_\_init\_\_(self, root, board\_size=19):

self.root = root

self.board\_size = board\_size

self.board = GoBoard(size=board\_size)

self.current\_turn = Stone.BLACK

self.cell\_size = 30 # セルの大きさ（ピクセル）

self.canvas = tk.Canvas(root, width=board\_size \* self.cell\_size, height=board\_size \* self.cell\_size, bg="tan")

self.canvas.pack()

self.canvas.bind("<Button-1>", self.handle\_click)

self.draw\_grid()

def draw\_grid(self):

"""碁盤の目を描画"""

for i in range(self.board\_size):

self.canvas.create\_line(self.cell\_size // 2, (i + 0.5) \* self.cell\_size,

self.board\_size \* self.cell\_size - self.cell\_size // 2, (i + 0.5) \* self.cell\_size)

self.canvas.create\_line((i + 0.5) \* self.cell\_size, self.cell\_size // 2,

(i + 0.5) \* self.cell\_size, self.board\_size \* self.cell\_size - self.cell\_size // 2)

def handle\_click(self, event):

"""クリック処理"""

x = int(event.x // self.cell\_size)

y = int(event.y // self.cell\_size)

if self.board.place\_stone(x, y, self.current\_turn):

self.redraw\_board() # 碁盤を再描画

self.current\_turn = Stone.BLACK if self.current\_turn == Stone.WHITE else Stone.WHITE

def redraw\_board(self):

"""碁盤全体を再描画"""

self.canvas.delete("all") # 現在の描画をクリア

self.draw\_grid() # 碁盤の目を再描画

for y in range(self.board\_size):

for x in range(self.board\_size):

stone = self.board.board[y][x]

if stone != Stone.EMPTY:

self.draw\_stone(x, y, stone)

def draw\_stone(self, x, y, stone):

"""石を描画"""

x0 = x \* self.cell\_size + self.cell\_size // 4

y0 = y \* self.cell\_size + self.cell\_size // 4

x1 = (x + 1) \* self.cell\_size - self.cell\_size // 4

y1 = (y + 1) \* self.cell\_size - self.cell\_size // 4

color = "black" if stone == Stone.BLACK else "white"

self.canvas.create\_oval(x0, y0, x1, y1, fill=color)

if \_\_name\_\_ == "\_\_main\_\_":

root = tk.Tk()

root.title("囲碁ゲーム（人対人）")

app = GoGameGUI(root)

root.mainloop()

**うまくいきました。次に欲張りですが、囲んで取った石は、自分の「ごけ」にいれておくことができます（終局の際、相手の「地」にいれる）。この「ごけ」に相当するものを、碁盤の外に描画し、取ったあとその中に取った数の相手の石をいれたものを表示せねばなりません。どうしたらいいでよう？**

ChatGPT:

ChatGPT

囲んだ石を「ごけ」に入れ、それを碁盤の外に描画する仕組みを追加する方法をご案内します。この機能を実現するために、以下の手順を取ります。

目標

取った石を「ごけ」として管理

各プレイヤーの取った石の数をカウントし、保持します。

「ごけ」を碁盤の外に描画

碁盤の隣に「ごけ」を表示し、取られた石の数を更新します。

ゲーム進行中に「ごけ」を更新

石を取った際、自動的に「ごけ」に反映させます。

修正手順

1. GoBoard クラスに「ごけ」の管理を追加

go\_board.py に以下の変更を加えます。

python

コードをコピーする

class GoBoard:

def \_\_init\_\_(self, size=19):

self.size = size

self.board = [[Stone.EMPTY for \_ in range(size)] for \_ in range(size)]

self.previous\_board = None # コウ判定用

self.captured\_stones = {Stone.BLACK: 0, Stone.WHITE: 0} # 取った石の数を管理

def remove\_dead\_stones(self, stone):

"""

囲まれた石を取り除き、取った石を「ごけ」に追加する。

"""

to\_remove = []

for y in range(self.size):

for x in range(self.size):

if self.board[y][x] == stone and not self.get\_liberties(x, y):

to\_remove.append((x, y))

for x, y in to\_remove:

self.board[y][x] = Stone.EMPTY

self.captured\_stones[stone] += len(to\_remove) # 取った石をカウント

2. main.py に「ごけ」の描画を追加

以下の修正を加えて、碁盤の右側に「ごけ」を表示します。

修正箇所: 初期化に「ごけ」用の要素を追加

python

コードをコピーする

class GoGameGUI:

def \_\_init\_\_(self, root, board\_size=19):

self.root = root

self.board\_size = board\_size

self.board = GoBoard(size=board\_size)

self.current\_turn = Stone.BLACK

self.cell\_size = 30 # セルの大きさ（ピクセル）

# キャンバスの設定

canvas\_width = board\_size \* self.cell\_size + 100 # 右側に「ごけ」のスペースを追加

canvas\_height = board\_size \* self.cell\_size

self.canvas = tk.Canvas(root, width=canvas\_width, height=canvas\_height, bg="tan")

self.canvas.pack()

self.canvas.bind("<Button-1>", self.handle\_click)

self.draw\_grid()

self.draw\_goke() # 「ごけ」を描画

修正箇所: 「ごけ」の描画

python

コードをコピーする

def draw\_goke(self):

"""取った石（ごけ）を描画"""

self.canvas.create\_text(self.board\_size \* self.cell\_size + 20, 50, text="ごけ", font=("Arial", 14), anchor="w")

self.update\_goke\_display()

def update\_goke\_display(self):

"""「ごけ」の石の数を更新"""

# 黒のごけ

self.canvas.create\_text(self.board\_size \* self.cell\_size + 20, 100,

text=f"黒: {self.board.captured\_stones[Stone.WHITE]}",

font=("Arial", 12), anchor="w")

# 白のごけ

self.canvas.create\_text(self.board\_size \* self.cell\_size + 20, 130,

text=f"白: {self.board.captured\_stones[Stone.BLACK]}",

font=("Arial", 12), anchor="w")

修正箇所: 再描画時に「ごけ」を更新

python

コードをコピーする

def redraw\_board(self):

"""碁盤全体を再描画"""

self.canvas.delete("all") # 現在の描画をクリア

self.draw\_grid() # 碁盤の目を再描画

self.draw\_goke() # 「ごけ」を再描画

for y in range(self.board\_size):

for x in range(self.board\_size):

stone = self.board.board[y][x]

if stone != Stone.EMPTY:

self.draw\_stone(x, y, stone)

完成した main.py

以下は、上記変更を反映した完成版の main.py です。

python

コードをコピーする

import tkinter as tk

from go\_types import Stone

from go\_board import GoBoard

class GoGameGUI:

def \_\_init\_\_(self, root, board\_size=19):

self.root = root

self.board\_size = board\_size

self.board = GoBoard(size=board\_size)

self.current\_turn = Stone.BLACK

self.cell\_size = 30 # セルの大きさ（ピクセル）

# キャンバスの設定

canvas\_width = board\_size \* self.cell\_size + 100 # 右側に「ごけ」のスペースを追加

canvas\_height = board\_size \* self.cell\_size

self.canvas = tk.Canvas(root, width=canvas\_width, height=canvas\_height, bg="tan")

self.canvas.pack()

self.canvas.bind("<Button-1>", self.handle\_click)

self.draw\_grid()

self.draw\_goke() # 「ごけ」を描画

def draw\_grid(self):

"""碁盤の目を描画"""

for i in range(self.board\_size):

self.canvas.create\_line(self.cell\_size // 2, (i + 0.5) \* self.cell\_size,

self.board\_size \* self.cell\_size - self.cell\_size // 2, (i + 0.5) \* self.cell\_size)

self.canvas.create\_line((i + 0.5) \* self.cell\_size, self.cell\_size // 2,

(i + 0.5) \* self.cell\_size, self.board\_size \* self.cell\_size - self.cell\_size // 2)

def handle\_click(self, event):

"""クリック処理"""

x = int(event.x // self.cell\_size)

y = int(event.y // self.cell\_size)

if self.board.place\_stone(x, y, self.current\_turn):

self.redraw\_board() # 碁盤を再描画

self.current\_turn = Stone.BLACK if self.current\_turn == Stone.WHITE else Stone.WHITE

def redraw\_board(self):

"""碁盤全体を再描画"""

self.canvas.delete("all") # 現在の描画をクリア

self.draw\_grid() # 碁盤の目を再描画

self.draw\_goke() # 「ごけ」を再描画

for y in range(self.board\_size):

for x in range(self.board\_size):

stone = self.board.board[y][x]

if stone != Stone.EMPTY:

self.draw\_stone(x, y, stone)

def draw\_stone(self, x, y, stone):

"""石を描画"""

x0 = x \* self.cell\_size + self.cell\_size // 4

y0 = y \* self.cell\_size + self.cell\_size // 4

x1 = (x + 1) \* self.cell\_size - self.cell\_size // 4

y1 = (y + 1) \* self.cell\_size - self.cell\_size // 4

color = "black" if stone == Stone.BLACK else "white"

self.canvas.create\_oval(x0, y0, x1, y1, fill=color)

def draw\_goke(self):

"""取った石（ごけ）を描画"""

self.canvas.create\_text(self.board\_size \* self.cell\_size + 20, 50, text="ごけ", font=("Arial", 14), anchor="w")

self.update\_goke\_display()

def update\_goke\_display(self):

"""「ごけ」の石の数を更新"""

# 黒のごけ

self.canvas.create\_text(self.board\_size \* self.cell\_size + 20, 100,

text=f"黒: {self.board.captured\_stones[Stone.WHITE]}",

font=("Arial", 12), anchor="w")

# 白のごけ

self.canvas.create\_text(self.board\_size \* self.cell\_size + 20, 130,

text=f"白: {self.board.captured\_stones[Stone.BLACK]}",

font=("Arial", 12), anchor="w")

if \_\_name\_\_ == "\_\_main\_\_":

root = tk.Tk()

root.title("囲碁ゲーム（人対人）")

app = GoGameGUI(root)

root.mainloop()

このプログラムの動作

石を囲むと、囲まれた石が消え、それが「ごけ」に追加されます。

「ごけ」の数は碁盤の右側にリアルタイムで表示されます。

**今後の課題**

1. 終局の判定

さもないと、空いている自分の「地」にどんどん、自分でうっていってしまう。

1. 人　vs　ボット　そしてボットを強くする

既存のオープンソース実装の概要

Leela Zero

言語: 主にC++で実装されています。

特徴:

強力な性能を持つ囲碁AI。

モデルの学習済みデータを利用可能。

Pythonと連携するために「外部プロセス」や「通信」を使用できます。

MiniGo

言語: Pythonを使用しています。

特徴:

Googleの研究者によって開発された簡易版AlphaGo/AlphaZero。

TensorFlowを利用しているため、Pythonで直接操作可能。

小規模環境でも利用可能。

**MiniGoに興味ありますね。今回つっくった囲碁ゲームの続きでやっていけますか？**

ChatGPT:

ChatGPT

はい、今回作成した囲碁ゲームに MiniGo を統合するのは十分可能です！MiniGoはPythonで書かれており、既存の学習済みモデルを活用すればスムーズに統合できます。以下に具体的なステップを示します。

1. MiniGoの準備

MiniGoのコードを取得し、学習済みモデルをダウンロードします。

(1) MiniGoリポジトリのクローン

MiniGoの公式GitHubリポジトリをクローンします。

bash

コードをコピーする

git clone https://github.com/tensorflow/minigo.git

cd minigo

(2) 必要な依存ライブラリをインストール

Python環境で必要なライブラリをインストールします（TensorFlowなど）。

bash

コードをコピーする

pip install -r requirements.txt

(3) 学習済みモデルをダウンロード

MiniGoの学習済みモデルを公式サイトまたはGitHubからダウンロードします。

ダウンロードURL: https://cloud.google.com/tpu/docs/tutorials/minigo

(4) モデルの場所を確認

ダウンロードしたモデルのパスを確認しておきます。

2. 囲碁ゲームへの統合

以下は、MiniGoをBotとして導入する具体的な方法です。

(1) MiniGoのモデルをロード

学習済みモデルをロードし、碁盤の状態を入力して次の手を予測します。

サンプルコード: モデルロードと推論

python

コードをコピーする

import tensorflow as tf

import numpy as np

class MiniGoBot:

def \_\_init\_\_(self, model\_path):

self.model = tf.keras.models.load\_model(model\_path)

def predict\_move(self, board\_state, current\_turn):

"""

次の手を予測します。

Args:

board\_state (2D list): 現在の碁盤の状態 (19x19)。

current\_turn (Stone): 現在の手番 (Stone.BLACK or Stone.WHITE)。

Returns:

tuple: 次の手の座標 (x, y)。

"""

# 碁盤の状態をTensorFlowの入力形式に変換

input\_tensor = self.prepare\_input(board\_state, current\_turn)

# モデルで予測

prediction = self.model.predict(input\_tensor)

move\_index = np.argmax(prediction[0]) # 最もスコアの高い手を取得

# 碁盤の座標に変換

x, y = divmod(move\_index, 19)

return x, y

def prepare\_input(self, board\_state, current\_turn):

"""

モデル入力形式に変換します。

"""

input\_array = np.zeros((19, 19, 2), dtype=np.float32)

for y in range(19):

for x in range(19):

if board\_state[y][x] == Stone.BLACK:

input\_array[y][x][0] = 1

elif board\_state[y][x] == Stone.WHITE:

input\_array[y][x][1] = 1

if current\_turn == Stone.BLACK:

input\_array = np.expand\_dims(input\_array, axis=0)

return input\_array

(2) MiniGoBotを統合

上記の MiniGoBot クラスを、既存の囲碁ゲームの「Bot」として組み込みます。

main.py の修正例

python

コードをコピーする

# MiniGoBotをインポート

from minigo\_bot import MiniGoBot

class GoGameGUI:

def \_\_init\_\_(self, root, board\_size=19):

...

self.bot = MiniGoBot("path/to/your/minigo/model") # Botの初期化

def handle\_click(self, event):

"""クリック処理"""

x = int(event.x // self.cell\_size)

y = int(event.y // self.cell\_size)

if self.board.place\_stone(x, y, self.current\_turn):

self.redraw\_board() # 碁盤を再描画

self.current\_turn = Stone.BLACK if self.current\_turn == Stone.WHITE else Stone.WHITE

# Botのターン

if self.current\_turn == Stone.WHITE: # 例: 白がBotの場合

bot\_move = self.bot.predict\_move(self.board.board, self.current\_turn)

if self.board.place\_stone(bot\_move[0], bot\_move[1], self.current\_turn):

self.redraw\_board()

self.current\_turn = Stone.BLACK

3. 必要な注意点

(1) 碁盤の状態の変換

MiniGoのモデルが期待する入力フォーマットに碁盤の状態を変換する必要があります。

prepare\_input メソッドを適切に設計してください。

(2) 手の合法性チェック

MiniGoが予測した手が合法であることを確認する処理を追加します。

GoBoard クラスに既にある place\_stone メソッドを活用できます。

(3) モデルの重さ

MiniGoのモデルは比較的軽量ですが、高性能なGPUがあるとスムーズに動作します。

ローカル環境で問題がある場合は、Google Colabなどを利用して実験することも検討してください。

4. まとめ

MiniGoのBotを統合することで、現行の囲碁ゲームに強力なAI対戦機能を追加できます。

MiniGoの学習済みモデルを利用することで、学習プロセスを省略して即座に活用可能です。

必要に応じて、モデルフォーマットや碁盤状態の変換をカスタマイズできます。

この方針で進める際、具体的な実装で問題があればいつでもご相談ください！

