卒業研究報告書

題目

挟み将棋の AI 作成

指導教員

石水 隆　講師

報告者

21-1-037-0015

今泉 拓土

近畿大学理工学部情報学科

令和 7 年 2 月 2 日提出

概要

将棋は 2 人のプレイヤーがそれぞれ盤の上で交互に駒を動かし，相手の玉将を詰ませることが勝利条件の日本古来のボードゲームである．将棋には本将棋以外にも様々なバリエーションがあり，そのバリエーションによって盤の大きさやコマの動き，勝利条件といったルールが異なるため，それぞれで違った戦略が必要になる．

将棋のバリエーションの 1 つに挟み将棋[1]がある．挟み将棋とは，自分のコマで相手のコマを挟んで取るゲームであり，相手のコマを 5 枚取るか，取ったコマの数に 3 枚差がつくと勝利となる，コマが 1 種類しかない，取ったコマを打つことはできない，という本将棋とは全く異なるルールを用いる．このため，本将棋とは全く異なる戦略が必要となる．

本将棋は多くの AI が作られており，近年ではプロ棋士を凌駕するほどの棋力を持つ強化学習で着手選択を行う AI も存在する[2]．一方，挟み将棋はマイナーなゲームであるため，既存の AI がほとんど存在しない．そこで本研究では，着手選択を機械学習で行う挟み将棋 AI を作成する．

目次

1. 序論

1.1 本研究の背景

将棋は 2 人のプレイヤーがそれぞれ盤の上で交互に駒を動かし，相手の玉将を詰ませることが勝利条件の日本古来のボードゲームである．将棋には本将棋以外にも様々なバリエーションがあり，そのバリエーションによって盤の大きさやコマの動き，勝利条件といったルールが異なるため，それぞれで違った戦略が必要になる．

将棋のバリエーションの一つに挟み将棋がある[1]．挟み将棋とは，自分のコマで相手のコマを挟んで取るゲームであり，相手のコマを 5 枚取るか，取ったコマの数に 3 枚差がつくと勝利となる，コマが 1 種類しかない，取ったコマを打つことはできない，という本将棋とは全く異なるルールを用いる．挟み将棋はルールがシンプルなので，楽しめる層が幅広いといった特徴を持っている．

本将棋は多くの AI が作られており，近年ではプロ棋士を凌駕するほどの棋力を持つ強化学習で着手選択を行う AI も存在する[2]．一方で，挟み将棋には，AI がほとんど存在しない．既存の挟み将棋アプリ[3]，[4]，[5]などがあるが，[5]は対コンピュータ戦ができるがランダム着手であり非常に弱い．また[4]は着手選択方法が明示されていないが mini-max 法による探索を行っていると思われ，機械学習ではない．このように，本将棋においては多くの研究が進められているものの，挟み将棋では探索手法での着手選択が主であり，あまり研究されていないといえる．

1.2 機械学習

機械学習は，データやパターンから規則を学び，新しいデータに対して予測や意思決定を行うアルゴリズムの一群を指し，基本的に多くの機械学習は何らかの損失関数を設定し，その期待値を最小化することを目的とする．本将棋において，現在ではコンピュータ将棋同士が対局を繰り返し，そこから機械学習によって強くするという強化学習の手法が取り入れられており，人間には理解しにくいが強力な手が多数生まれ，人間がコンピュータに勝ちにくくなっているといった例がある[6]．

1.3 本研究の目的

本研究の目的は，機械学習によって着手選択を行う挟み将棋の AI を作成することである．しかし，挟み将棋は本将棋と違い，プロ棋士による棋譜のような学習として利用できるデータが少ない．そこで，ランダム AI との対戦を通じて，各状態と行動のペアに対する Q 値を更新することで，強い AI の作成を目指す．

1.4 本報告書の構成

本報告書の構成は以下の通りである．まず 2 章で本研究の対象である挟み将棋について説明する．3 章ではアルファベータ法について説明する．4 章では機械学習について説明する．5 章では作成したプログラムについて説明する．6 章では作成した AI の対戦結果及び，脆弱性を示す．最後に 7 章にて結論及び今後の課題を述べる．

2 挟み将棋

本章では，本研究の対象である挟み将棋について説明する．

2.1 挟み将棋の概要

挟み将棋とは，自分のコマを動かして相手のコマを挟んで取るゲームであり，相手のコマを 5 枚取るか，取ったコマの数に 3 枚差がつくと勝利となる，コマが 1 種類しかない，取ったコマを打つことはできない，という本将棋とは全く異なるルールを用いる．挟み将棋はルールがシンプルなので，楽しめる層が幅広いといった特徴を持っている．

2.2 挟み将棋のルール

本節では，挟み将棋のルールについて述べる．図 1 に初期配置を示し，以下にルールを示す．

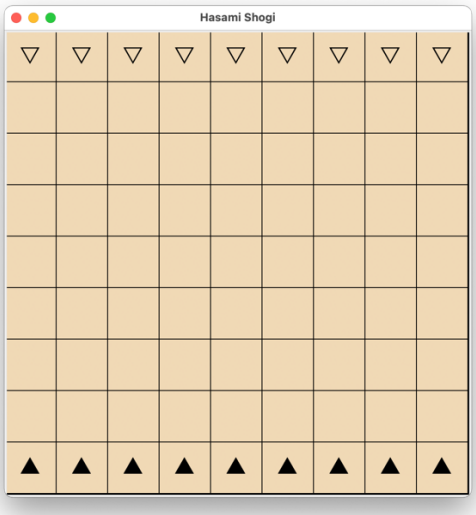

図 1 挟み将棋の初期配置

- コマの種類は 1 種類しかなく，一般的に先手は「歩兵」，後手が「と金」を使う．しかし，本研究では先手を「▲」，後手を「▽」で表している．
- コマは本将棋の飛車のようにタテとヨコに好きなだけ動くことができる．
- コマを交互に動かし，味方のコマ 2 枚で相手のコマをタテかヨコに挟むか，端にいる相手のコマを味方のコマを用いて囲んだ場合にそのコマを取ることができる．以下にその一例を示す．以下の図 2，図 3 は味方のコマ 2 枚で相手のコマを挟む場合で，▽2 五のコマを▲1 五と▲3 五で挟むことによって取った局面である．また，以下の図 4，図 5 は端にいる相手のコマを味方のコマを用いて囲んだ場合で，▽1 一，▽2 一，▽3 一のコマを▲1 二，▲2 二，▲3 二，▲4 一のコマで囲むことで取った局面である．
- 取ったコマは取り捨てであり，持ち駒として使うことはできない．
- どちらかが先に 5 枚を取るか，取ったコマの数の差に 3 枚差がつくと勝敗が決まる．しかし，3 枚差の場合は，相手が直後にコマを取り返し，3 枚差の状態が解消された場合は負けにならない．

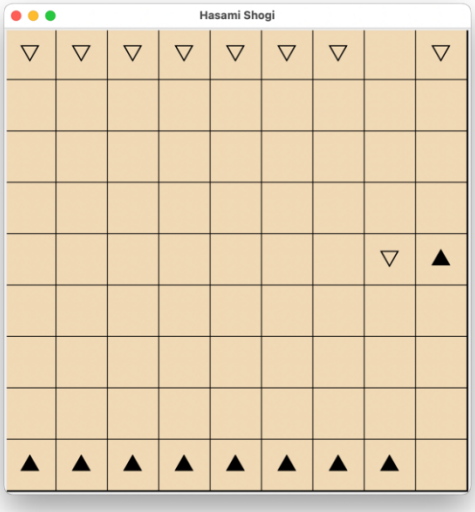

図 2 挟んで取る場合の局面 1

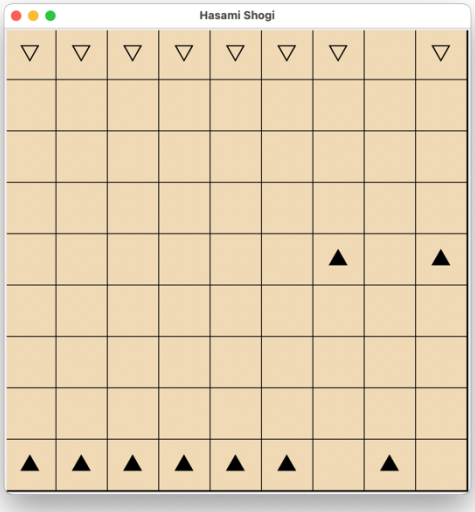

図 3 挟んで取る場合の局面 2

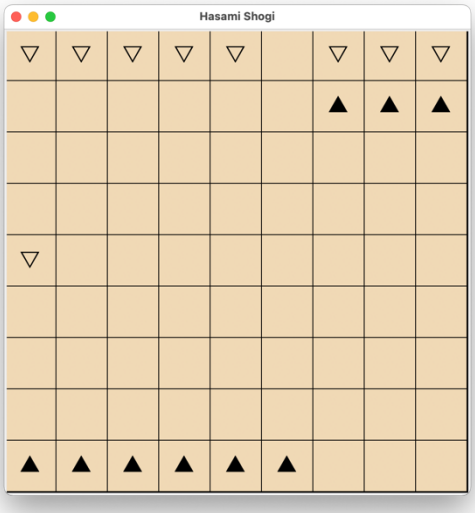

図 4 囲んで取る場合の局面 1

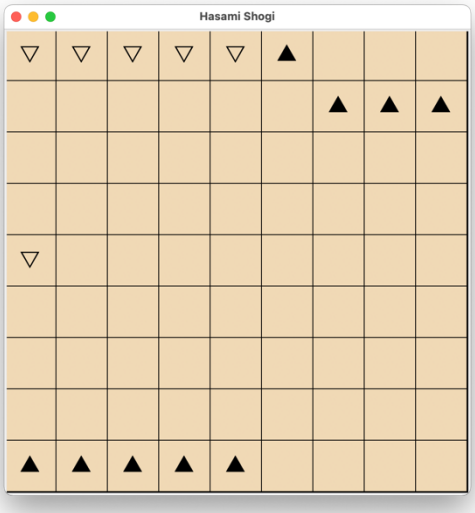

図 5 囲んで取る場合の局面 2

3 着手選択法

将棋をはじめとするボードゲームでは，着手を決定するための様々な手法が知られている．本章では，代表的な着手選択法について述べる．

3.1 アルファベータ法

現在の局面から到達可能な数手先の局面を読み，読んだ局面の中で最も有利な局面に到達できる手を選ぶのが先読みである．先読みを行うことで，コンピュータが有利になる手を選択する方法がいくつかある．中でも二人零和完全情報ゲーム向けのアルゴリズムとして，mini-max 法が知られている．mini-max 法とは，先手番では評価値が最大になる手を選択し，後手番では評価値が最小になる手を選択するアルゴリズムであり，一定の深さまで読んで辿り着いた葉の局面で評価関数を用いて評価し，

最も評価が高かった結果とその手順を得るというのが一般的な動作になっている.

一般に，先読みする手数を深くすれば，よりよい手が発見される．しかし，mini-max 法では読まなければならない局面の数は，先読みする手数に対して指数的に増加するため，先読みできる手数には上限が有る．したがって，より深く読むためには，先読みする局面の数を何等かの方法で減らす必要がある.

mini-max 法を改良したアルゴリズムにαβ法というものがある．αβ法とは，mini-max 法による探索を可能な限り減らす手法であり，同じ結果を得ることができるにもかかわらず，同じ時間でほぼ 2 倍の深さまで読めるアルゴリズムである．αβ法ではα，βという範囲を決め，範囲外の値が来たら直ちにその探索を打ち切ることで高速な探索を行っている[7].

3.2 定跡データベース

将棋や囲碁では，過去の多くの対局結果から，特定の局面において優れた手とされる手が存在する．このような過去の経験から各局面で優れた手とされる手が定跡である．将棋や囲碁では，プロ棋士による定跡が確立している．この定跡をデータベースとして保持し，定跡にある局面では定跡通りに指すのが定跡データベースである．しかし，挟み将棋においては本将棋と異なり，プロ棋士による棋譜や優れた手として活用できるデータがほとんど存在しないため，定跡データベースを構築することは困難である.

4　機械学習

前章で述べた通り，ボードゲームにでは着手選択するための様々な手法が存在する．その中で近年主流となっているのが機械学習である．本章では，機械学習について説明し，本研究で作成した挟み将棋 AI の学習手法について述べる.

4.1 機械学習とは

機械学習は，データやパターンから規則を学び，新しいデータに対して予測や意思決定を行うアルゴリズムの一群を指し，基本的に多くの機械学習は何らかの損失関数を設定し，その期待値を最小化することを目的とする[8]．機械学習は教師あり学習，教師なし学習，半教師あり学習，強化学習の主に 4 つに分類することができる．本将棋ではプロ棋士の対戦から得られた膨大なデータを用いることができるが，挟み将棋ではそのようなデータはないため，本研究では強化学習の手法を用いた.

4.2 強化学習とは

強化学習は，エージェントが環境の状態を観測し，行動を選択することで得られる報酬を最大化するように学習が行われるもので，能動的にデータを取得しながら学習を行う．強化学習が目標とするのは報酬を最大化する方策を求めることであり，各遷移で報酬の総和の期待値を更新し，収束するまで続けることで目標を達成しようとするが，遷移確率が既知でないことの方が多く，それを推定する必要がある．これを考慮した学習手法が Q 学習であり，深層学習を用いた Q 学習として Deep Q-network がある.

4.3 Q 学習と Deep Q-network

Q 学習は，状態(s)と行動(a)に対する価値(Q 値)を学習し，それに基づいて最適な行動を選択するアルゴリズムであり，Q 値の更新には次式が用いられる.

$$Q(s,a) \leftarrow \mathrm{Q(s,a)} + \alpha[\mathrm{r} + \gamma \max_{\mathrm{a'}} \mathrm{Q(s',a')} - \mathrm{Q(s,a)}]$$

ここで，$Q(s,a)$は状態 s で行動 a を取る価値(Q 値)，rは即時報酬，γは割引率，αは学習率，$\max_{\mathrm{a'}} \mathrm{Q(s',a')}$は次の状態$\mathrm{s'}$における最良の行動の Q 値を表す.

Deep Q-network(DQN)は，ニューラルネットワークを利用して状態と行動ペアの価値を近似する．DQN は，過去の経験をメモリに保存し，ランダムにサンプリングして学習する再生経験や，Q 値を計算するためにメインネットワークに加え，ターゲットネットワークを用いること，ニューラルネットワーク

に状態空間を入力として与えることで出力として各行動に対応するQ値を得るといった特徴がある.

4.4 本研究におけるDQNの適応

本節では，挟み将棋AI学習におけるDQNの手法について述べる．本研究では，挟み将棋の局面をニューラルネットワークの入力として与え，各行動に対応するQ値を出力するモデルを構築した．具体的なモデル構造としては，入力層，中間層（隠れ層），および出力層の3層から成り，隠れ層には128ユニットを設定した．Q学習に用いるパラメータの設定は，割引率(γ)を0.99とし，長期的な報酬を考慮するようにした．また学習率(α)は0.01とし，安定した学習を行えるように調整した.

学習の流れとしては，ランダムAIとの対戦を通して再生経験バッファに現在の局面，行動，報酬，次の局面の4つの情報を保存し，これらを繰り返し利用することで学習を進めた．再生経験バッファから得られたサンプルを用いて，ニューラルネットワークのパラメータの更新を行う際には，損失関数として平均二乗誤差を採用し，損失関数で計算された誤差に基づき，勾配降下法を使用してニューラルネットワークの重みを更新した．これにより，モデルのQ値の推定精度が向上し，方策の最適化が進んだ.

5　挟み将棋のプログラム

本章では，本研究で作成したプログラムについて述べる.

5.1 挟み将棋プログラムの仕様

本研究では，Pythonを用いて挟み将棋プログラムを作成した．以下の表 1に作成した挟み将棋のPythonプログラムのクラスの一覧を示す.また付録に本研究で作成したPythonプログラムのソースコードを示す．本研究ではPyTorch[9]を用いて機械学習の実装を行った．学習はtrain.pyで行い，その際にexperience_replay.pyクラスに保存した局面の情報を用いて学習を行う．本研究では，まずランダムに手を選択するAI同士の対戦から開始し,十分な量の対戦データを蓄積した後に,対戦を通じて,各状態と行動のペアに対するQ値を更新することで，最適な着手選択を獲得することを目指した．以下の節では各クラスについて述べる.

表 1挟み将棋のPythonプログラムのクラス一覧

ソースコード	説明
Phase.py	挟み将棋の状態を持つ
Human.py	エージェントが人間
HumanGUI.py	HasamishogiGUIでエージェントが人間
Random.py	ランダムエージェント
AlphaBetaAgent.py	探索エージェント
Ai_agent.py	機械学習エージェント
Model.py	機械学習エージェントで使用するモデル
Experience_replay.py	経験バッファ
Train.py	機械学習エージェントの学習を行う
Test.py	テストを行う
HasamishogiGUI.py	GUIを用いてゲームを行う

5.2　Phase.py

Phaseクラスは挟み将棋の状態を持ち，ゲームの進行に用いる．表 2にPhaseクラスのメソッドとその機能を示す．selectメソッドでagentがどのクラスでインスタンス化されているかを判別し，それによってselect2からselect4までのクラスを用いてコマの選択を行う．また，コマを取得する処

理は 3 つのメソッドを組み合わせることで実現しており，囲む処理に関しては getPieces3 のメソッドを再起的に何度か呼び出すことで実現している．

表 2 Phase クラスのメソッドと機能

メソッド	説明
__init__(self, agent1, agent2)	agent を設定し，ゲームの状態を初期化
displayBoard(self)	局面をテキストで表示
selectGUI(self, gui)	HasamishogiGUI によるコマの選択
select(self)	select2 から select4 までのメソッドを用いてコマの選択を行う
select2(self)	agent が Human または Random の場合
select2GUI(self, gui)	agent が HumanGUI の場合
select3(self)	agent が ai_agent の場合
select4(self)	agent が AlphaBetaAgent の場合
checkLegelMove(self, x)	x から動かすことの可能な場所を返す
set(self, x, u)	x から u に選択されたコマを動かす
getPieces(self, x)	コマを取得する
getPieces2(self, x)	囲まれたコマを返す
getPieces3(self, x, checked_list)	getPieces2 から呼び出される
turnChange(self)	ターン変更
isWin(self)	勝敗の判定
reset(self)	ゲームの状態を初期化
board_to_vector(self)	現在の局面を数値ベクトルに変換
get_available_pieces(self)	ターンプレイヤーのコマの位置を返す

5.3 Human.py

Human クラスはエージェントが人間の場合に使用する．表 3 に Human クラスのメソッドとその機能を示す.

表 3 Human クラスのメソッドと機能

メソッド	説明
select1(self, a, b)	動かしたいコマをキーボードで入力
select2(self, a)	配置したい場所をキーボードで入力

5.4 HumanGUI.py

HumanGUI は着手を入力するためのクラスであり，HasamishogiGUI を用いて実行する場合でかつエージェントが人間の場合に使用する．表 4 に HumanGUI クラスのメソッドとその機能を示す.

表 4 HumanGUI クラスのメソッドと機能

メソッド	説明
select1(self, gui)	動かしたいコマをクリックで入力
select2(self, gui)	配置したい場所をクリックで入力

5.5 Random.py

Random クラスはエージェントがランダムの場合に使用する．表 5 に Random クラスのメソッドとその機能を示す．

表 5 Random クラスのメソッドと機能

メソッド	説明
select1(self, board, turn)	局面と手番からランダムに 1 つ動かしたいコマを返す
select2(self, legalMoveList)	選択された位置のコマの合法手の中からランダムに 1 つ配置する場所を返す

5.6 AlphaBetaAgent.py

AlphaBetaAgent クラスはエージェントがアルファベータ法を用いた探索の場合に使用する．表 6 に AlphaBetaAgent クラスのメソッドとその機能を示す．このプログラムの探索部分は，[7]を参考に作成した．ここで作成したアルファベータ法を用いた探索は，ai_agent の検証に用いるために作成したものであり，スコアの計算は単純なものとなっている．スコアの基準は，コマの数を重視しており，次にコマの位置が相手のコマを取りやすい位置であるかを確認している．探索の深度をインスタンス化の際に決定することができ，その深度に応じた探索を行うようになっており，評価が単純であっても探索深度を増すことでかなり強い AI になっている．

表 6 AlphaBetaAgent クラスのメソッドと機能

メソッド	説明
__init__(self, depth)	エージェントの初期化，探索深度を指定
select(self, board, player, is_reach)	与えられた局面から最適な手を探索
alpha_beta(self, board, depth, alpha, beta, player)	アルファベータ探索を行う
get_legal_moves(self, board, player)	局面と手番から合法手を返す
check_legal_moves(self, board, position)	指定された位置のコマの合法手を返す
make_move(self, board, move, player)	指定された手を実行し，局面を更新
capture_line(self, board, position, direction, player)	コマを挟んで取る処理を行う
capture_surrounded(self, board, position, player)	コマを囲んで取る処理を行う
is_terminal(self, board)	勝敗を判定し，ゲームの状態を返す
evaluate_board(self, board, player)	スコアの計算を行う

5.7 Ai_agent.py

Ai_agent クラスはエージェントが本研究で学習を行った機械学習の場合に使用する．表 7 に Ai_agent クラスのメソッドとその機能を示す．select_action メソッドでは，引数で与えられた合法手のペアからその時の局面で最も高い Q 値が選ばれたものを返すようにしている．

表 7 Ai_agent クラスのメソッドと機能

メソッド	説明

__init__(self, model, pair_action_size, epsilon)	エージェントの初期化
select_action(self, state, available_pairs, turn)	合法手のペアから一番Q値の高いペアを返す
get_pair_index(self, pair)	ペアに対応する一意のインデックスを返す
reverse_state(self, state)	局面を反転させる

5.8 Model.py

Model クラスは Ai_agent で使用する機械学習のモデルを保持するクラスである．表 8 に Modeel クラスのメソッドとその機能を示す.

表 8 Model クラスのメソッドと機能

メソッド	説明
__init__(self, state_size, pair_action_size)	モデルの初期化
forward(self, x)	入力された局面の状態からコマと移動先のペアのQ値を計算する
create_model(state_size, pair_action_size)	モデルの作成

5.9 Experience_replay.py

Experience_replay クラスは学習の際に使用する経験バッファを保持するクラスである．表 9 に Experience_replay クラスのメソッドとその機能を示す.

表 9 Experience_replay クラスのメソッドと機能

メソッド	説明
__init__(self, capacity)	経験バッファの初期化，キャパシティの指定
add(self, experience)	経験バッファに経験を追加
sample(self, batch_size)	経験バッファからランダムな経験を返す
__len__(self)	経験バッファのサイズを返す
save(self, filename)	経験バッファの保存を行う
load(self, filename)	経験バッファの読み込みを行う

5.10 Train.py

Train クラスは Ai_agent の学習を行う際に用いる．表 10 に Train クラスのメソッドとその機能を示す．このクラスを用いて，4.4 で説明した通りに学習を行う.

表 10 Train.py クラスのメソッドと機能

メソッド	説明
__init__(self)	トレーニング環境の初期化
train_ai_agent(self, main_model, target_model, memory, batch_size, gamma, optimizer, criterion)	ai_agent の学習を行う
update_target_model(self, main_model,	ターゲットモデルの更新

target_model)	
step1(self, action=None)	ゲームを進める
is_valid_move(self, selected_piece, destination_piece)	選択されたペアが合法手かを確認する
calculate_reward(self, done)	ゲーム終了時の報酬を与える
calculate_proximity_reward(self, position)	動かしたコマの位置が安全かを確認し,報酬を与える
calculate_risk_penalty(self)	現在の局面から次に取られるコマの最大数を求め，報酬を与える
is_safe_position(self, position)	引数で与えられた位置のコマが次の相手の手番で取られる可能性がある場合に,取られる最大のコマの数を返す
save_model(self, main_model, main_model_filename="model.pth")	モデルの保存を行う
plot_training_progress(self, episode_rewards, win_rates)	エピソードごとの報酬の合計値とこれまでの勝率をグラフに描写
train_roop(self)	トレーニングを指定した回数繰り返す

5.11 Test.py

Test クラスはテキストベースでの挟み将棋の実行や，学習させた AI の性能検証をする際に用いる．strat_game1 か start_game2 を実行前に選択することで，2 通りのテストをすることができる．表 11 に Test クラスのメソッドとその機能を示す．

表 11 Test.py クラスのメソッドと機能

メソッド	説明
__init__(self)	テスト環境の初期化
start_game1(self)	テキストベースでの挟み将棋の実行
start_game2(self)	対戦を 100 回繰り返し,最後に先手の勝利数を出力

5.12 HasamishogiGUI.py

HasamishogiGUI クラスは挟み将棋のゲームを行う際に用いる．人間が操作をする場合に，クリックによってコマの選択をすることができるため，直感的な操作を行うことができる．表 12 に HasamishogiGUI クラスのメソッドとその機能を示す．

表 12 HasamishogiGUI.py クラスのメソッドと機能

メソッド	説明
__init__(self, master, phase)	環境の初期化
draw_board(self)	将棋盤とコマを描写
on_click(self, event)	クリック時のイベント処理
wait_for_input(self)	クリックによる入力の待機

6　対戦結果

本章では本研究で作成した AI の性能検証として，ランダム AI，探索 AI，人間と対戦した場合の結果についてについて述べる．

6.1 ランダム AI との対戦

本研究で作成した AI の検証として，まずはランダム AI と先手，後手それぞれ 100 回ずつ対戦を行った．その結果を以下の表 13 に示す．表 13 に示す通り，本 AI の勝率は約 95%となっており，ランダム相手には良好な結果を収めることができた．

表 13 ランダム AI との対戦結果(試行回数 100 回)

先手	後手	先手勝	後手勝
本 AI	ランダム	98	2
ランダム	本 AI	4	96

6.2 探索 AI との対戦

次に，検証用に作成したアルファベータ探索を用いて着手選択を行う AI と先手，後手それぞれ 100 回ずつ対戦を行った．その結果を以下の表 14 に示す．表 14 に示す通り，探索 AI に対しては一度も勝利することができなかった．

表 14 探索 AI との対戦結果(試行回数 100 回)

先手	後手	先手勝	後手勝
本 AI	探索 AI	0	100
探索 AI	本 AI	100	0

6.3 既存の AI との対戦

次に，[4]との対戦を行った．相手のレベルはレベル 1 から 6 まで選択できる内のレベル 3 を選択し，本研究で作成した AI を先手という条件で 10 回対戦を行った．その結果，本研究で作成した AI は一度も勝つことができず，勝率は 0%となってしまった．

6.4 人間との対戦

次に，ランダム相手には良好な成績を納めることができたが，探索 AI や既存の AI には一度も勝つことができなかった原因を探るために，実際に私自身が対戦を行い AI がどのような手を指しているかを確かめた．その際の局面の一部を図 6 および図 7 に示す．図 6 は先手が▲2 六と指した局面であり，次に後手が 3 六にあるコマを動かさなければ一方的に取られてしまう．しかし AI は図 7 に示すとおり，そのコマを逃す動きを選択しなかった．このような手が選択された理由として，学習回数が不十分であったことや，学習の際の相手のレベルが低すぎたことが考えられる．

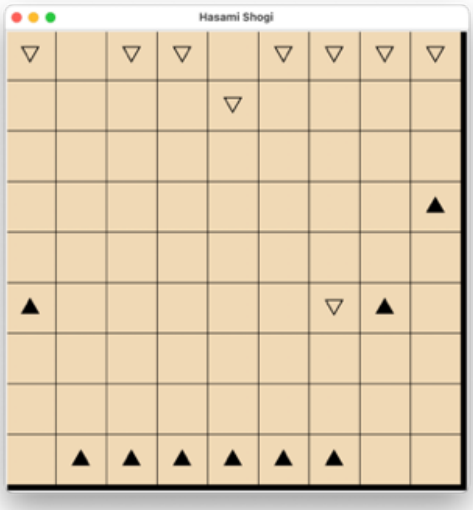

図 6 局面 1

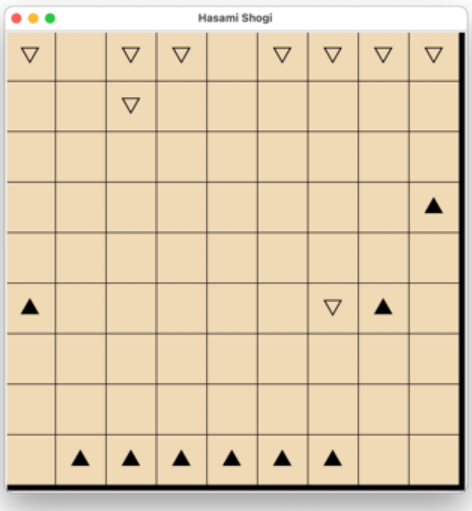

図 7 局面 2

7 結論・今後の課題

本研究では挟み将棋の AI を作成し，対戦を通してその性能を検証した．学習させた AI と対戦した際には，相手のミスを待ち続けるような消極的な戦術や，次に取られてしまうコマを放置するなどの弱点が見られた．これらの戦術はランダムな相手には一定の効果を発揮するものの，人間やアルファベータ法を用いた強い AI との対戦では効果的ではなく，現状では強い AI であるとは言えない．この結果の要因として，GPU を利用できなかったことによる学習時間の制約が考えられる．学習回数が十分でなかったため，AI が有効な戦術を学習しきれなかった可能性が高い．

今後の課題として，ランダムな相手だけでなく強い相手にも通用する手を選択できるようにすることが挙げられる．そのためには，より強い相手との対戦による学習を取り入れ，十分な学習回数を重ねて AI の強化を図る必要がある．また，アルファベータ法を用いた強い AI との対戦を通じて，AI の強さや戦術をより客観的に検証することが今後の重要な取り組みとなるだろう．

謝辞

本研究を行うにあたり，石水隆講師に中間報告書やレジュメ，卒業研究報告書の添削などの指導を受けました．ここに感謝の意を表します．

参考文献

[1] はさみ将棋｜将棋の基礎知識｜日本将棋連盟, https://www.shogi.or.jp/knowledge/hasami_shogi/

[2] 大槻知史, 三宅陽一郎:最強囲碁 AI アルファ碁解体新書, 翔泳社(2017)

[3] Ultima Atchtect Inc.:挟み将棋(2021)
https://play.google.com/store/apps/details?id=jp.co.ultimaarchitect.android.hasamishogi.free&hl=ja

[4] Kotake Studio LLC:ねこはさみ～はさみ将棋(2021)https://apps.apple.com/jp/app/ねこはさみ-はさみ将棋/id1542437319

[5] Kazutaka Sato:挟み将棋-AI(2022)https://apps.apple.com/jp/app/はさみ将棋-ai/id1608744238 号, pp. 31-38 (2018).

[6] 松原仁:機械の学習と人間の学習―ゲーム情報学を題材として., 情報処理学会論文誌教育とコンピュータ(TCE), 2017, 3.1, 1-6.

[7] 池泰弘:Java 将棋のアルゴリズム, 工学社(2016)

[8] 鈴木 大慈:機械学習の概要, 応用数理, 2018, 28, 1, 32-37

[9] PyTorch, https://pytorch.org/

付録

以下に本研究で作成したプログラムのソースを示す.

Phase クラス

```
from HumanGUI import HumanGUI
from Ai_agent import Ai_agent
from AlphaBetaAgent import AlphaBetaAgent
import numpy as np

class Phase():

    # 盤面の初期状態
    initial_board = [99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99,
                     99, -1, -1, -1, -1, -1, -1, -1, -1, -1, 99,
                     99,  0,  0,  0,  0,  0,  0,  0,  0,  0, 99,
                     99,  0,  0,  0,  0,  0,  0,  0,  0,  0, 99,
                     99,  0,  0,  0,  0,  0,  0,  0,  0,  0, 99,
                     99,  0,  0,  0,  0,  0,  0,  0,  0,  0, 99,
                     99,  0,  0,  0,  0,  0,  0,  0,  0,  0, 99,
                     99,  0,  0,  0,  0,  0,  0,  0,  0,  0, 99,
                     99,  0,  0,  0,  0,  0,  0,  0,  0,  0, 99,
                     99,  1,  1,  1,  1,  1,  1,  1,  1,  1, 99,
                     99, 99, 99, 99, 99, 99, 99, 99, 99, 99, 99]

    gameBoard = []

    board_size = 9

    moveList = [-1, 1, -11, 11]

    turn = 1

    isReach = 0

    def _init_(self, agent1, agent2):
        """ゲームの状態の初期化"""
        self.agent1 = agent1
        self.agent2 = agent2
        self.reset()

    def displayBoard(self):
        """局面をテキストで表示"""
        kansuji_map = ["一", "二", "三", "四", "五", "六", "七", "八", "九"]
        print(" ９ ８ ７ ６ ５ ４ ３ ２ １ ")
        for i in range(0, len(self.gameBoard)):
            if self.gameBoard[i] != 99:
                if i % 11 == 9:  # 列の最後
```

```
                        if self.gameBoard[i] == 1:
                            print("|歩|" + kansuji_map[i // 11 - 1])
                        elif self.gameBoard[i] == -1:
                            print("|と|" + kansuji_map[i // 11 - 1])
                        else:
                            print("|  |" + kansuji_map[i // 11 - 1])
                    else:  # 列の途中
                        if self.gameBoard[i] == 1:
                            print("|歩", end="")
                        elif self.gameBoard[i] == -1:
                            print("|と", end="")
                        else:
                            print("|  ", end="")
                if i % 11 == 9 and i < 110:  # 横線を追加
                    print("+ー+ー+ー+ー+ー+ー+ー+ー+ー+")

    def selectGUI(self, gui):
        """HasamishogiGUI によるコマの選択"""
        if self.turn == 1:
            if isinstance(self.agent1, HumanGUI):
                self.select2GUI(gui)
            else:
                self.select()
        else:
            if isinstance(self.agent2, HumanGUI):
                self.select2GUI(gui)
            else:
                self.select()

    def select(self):
        """コマの選択を行う、どのクラスでインスタンス化されたかを判別し次のメソッドへ"""
        if self.turn == 1:
            if isinstance(self.agent1, Ai_agent):
                self.select3()
            elif isinstance(self.agent1, AlphaBetaAgent):
                self.select4()
            else:
                self.select2()
        else:
            if isinstance(self.agent2, Ai_agent):
                self.select3()
            elif isinstance(self.agent2, AlphaBetaAgent):
                self.select4()
            else:
                self.select2()

    def select2(self):
```

```
        """agent が Human または Random の場合"""
        if self.turn == 1:
            z   = self.agent1.select1(self.gameBoard, self.turn)
        else:
            z = self.agent2.select1(self.gameBoard, self.turn)

        if self.turn == self.gameBoard[z]:
            legalMoveList = self.checkLegalMove(z)

            if len(legalMoveList) > 0:
                if self.turn == 1:
                    u = self.agent1.select2(legalMoveList)
                else:
                    u = self.agent2.select2(legalMoveList)
                if u in legalMoveList:
                    self.set(z, u)
                    self.getPieces(u)
                else:
                    print("その位置に動かすことはできません")
                    self.select2()
            else:
                print("選択された位置のコマに合法手がありません")
                self.select2()
        else:
            print("選択された位置にあなたのコマはありません")
            self.select2()

    def select2GUI(self, gui):
        """agent が HumanGUI の場合"""
        if self.turn == 1:
            z   = self.agent1.select1(gui)
        else:
            z = self.agent2.select1(gui)

        if self.turn == self.gameBoard[z]:
            legalMoveList = self.checkLegalMove(z)

            if len(legalMoveList) > 0:
                if self.turn == 1:
                    u = self.agent1.select2(gui)
                else:
                    u = self.agent2.select2(gui)
                if u in legalMoveList:
                    self.set(z, u)
                    self.getPieces(u)
                else:
                    print("その位置に動かすことはできません")
```

```
                    self.select2GUI(gui)
            else:
                print("選択された位置のコマに合法手がありません")
                self.select2GUI(gui)
        else:
            print("選択された位置にあなたのコマはありません")
            self.select2GUI(gui)

    def select3(self):
        """agent が ai_agent の場合"""
        available_pieces = self.get_available_pieces()
        legalMoveLists = [self.checkLegalMove(piece) for piece in available_pieces]
        # available_pairs を作成（駒と目的地のペアのリスト）
        available_pairs = []
        for piece in available_pieces:
            for destination in legalMoveLists[available_pieces.index(piece)]:
                available_pairs.append((piece, destination))
        if self.turn == 1:
            action = self.agent1.select_action(self.board_to_vector(), available_pairs, self.turn)  #
ai_agent を利用して行動を選択
        else:
            action = self.agent2.select_action(self.board_to_vector(), available_pairs, self.turn)  #
ai_agent を利用して行動を選択

        selected_piece, destination_piece = action

        if action not in available_pairs:
            print(f"{action}は合法でない手のため、再探索します")
            self.select3()
        else:
            # 合法手の場合、盤面を更新
            self.set(selected_piece, destination_piece)
            self.getPieces(destination_piece)

    def select4(self):
        """agent が AlphaBetaAgent の場合"""
        if self.turn == 1:
            selected_piece, destination_piece  = self.agent1.select(self.gameBoard, self.turn,
self.isReach)
        else:
            selected_piece, destination_piece = self.agent2.select(self.gameBoard, self.turn,
self.isReach)
        self.set(selected_piece, destination_piece)
        self.getPieces(destination_piece)

    def checkLegalMove(self, x):
        """x から動かすことの可能な場所を返す"""
```

```
        list = []
        for number in self.moveList:
            i = number
            while(self.gameBoard[x + i] == 0):
                list.append(x + i)
                i += number
        return list

    def set(self, x, u):
        """選択されたコマを動かす"""
        self.gameBoard[x] = 0
        self.gameBoard[u] = self.turn

    def getPieces(self, x):
        """
        コマを取る
        :return: 取ったコマの数
        """
        captured_pieces = 0  # 駒を取った数をカウントする変数
        for destination in self.moveList:
            temp_list = []
            i = destination
            while self.gameBoard[x + i] == -self.turn:
                temp_list.append(x + i)
                i += destination
                if self.gameBoard[x + i] == self.turn:
                    for place in temp_list:
                        self.gameBoard[place] = 0
                        print(place)
                        captured_pieces += 1  # 駒を取るたびにカウントする
                    break
            else:
                temp_list = []

        get_result = self.getPieces2(x)
        if get_result != []:
            for place in get_result:
                self.gameBoard[place] = 0
                print(place)
                captured_pieces += 1
        if(captured_pieces != 0):
            print(x)
            self.displayBoard()
        return captured_pieces  # 駒を取った数を返す

    def getPieces2(self, x):
        """囲まれたコマを返す"""
```

```
        list = []
        for move in self.moveList:
            if self.gameBoard[x + move] == -self.turn:
                list.append(x + move)
        list2 = []
        for place in list:
            list2 += self.getPieces3(place , [])
        return list2

    def getPieces3(self, x, checked_list):
        """getPieces2 用のメソッド"""
        # 現在の位置をチェック済みリストに追加
        checked_list.append(x)

        # すべての移動方向を確認
        for move in self.moveList:
            next_position = x + move

            # 盤面外に出ないようにチェック
            if next_position < 0 or next_position >= len(self.gameBoard):
                continue

            # 隣接しているマスが敵の駒なら、再帰的にチェック
            if self.gameBoard[next_position] == -self.turn:
                # まだチェックしていない位置であれば再帰的に関数を呼び出す
                if next_position not in checked_list:
                    result = self.getPieces3(next_position, checked_list)
                    if result == []:
                        return []  # 空のマスにたどり着いたら[]を返す

            # 隣接しているマスが空マスなら即座に[]を返す
            if self.gameBoard[next_position] == 0:
                return []

        # すべての隣接マスが敵の駒で囲まれている場合はリストを返す
        return checked_list

    def turnChange(self):
        """ターン変更"""
        self.turn = - self.turn

    def isWin(self):
        """
        勝敗の判定
        :return: ゲームの終了状態, 勝者
        """
        isWin = 0
```

```
            for board in self.gameBoard:
                if(board == -1):
                    isWin += 1
            if(isWin <= 4):
                return True, 1
            isWin = 0
            for board in self.gameBoard:
                if(board == 1):
                    isWin += 1
            if(isWin <= 4):
                return True, -1
            isWin = 0
            for board in self.gameBoard:
                if(board != 99):
                    isWin += board
            if(isWin >= 3):
                if(self.isReach == 1):
                    return True, 1
                self.isReach = 1
            elif(isWin <= -3):
                if(self.isReach == -1):
                    return True, -1
                self.isReach = -1
            else:
                self.isReach = 0
            return False, 0

        def reset(self):
            """ゲームの状態を初期化する"""
            self.gameBoard = self.initial_board.copy()  # 盤面を初期状態にリセット
            self.turn = 1  # 初期のターンは「歩」
            self.isReach = 0  # リーチ状態のリセット

        def board_to_vector(self):
            """現在の盤面を数値ベクトルに変換"""
            return np.array([x if x != 99 else 0 for x in self.gameBoard])  # 境界の 99 を 0 に

        def get_available_pieces(self):
            """自分のコマを取得"""
            available_pieces = []
            for index, place in enumerate(self.gameBoard):
                if place == self.turn:
                    available_pieces.append(index)
            return available_pieces
```

Human クラス

```
class Human():
```

```
    def select1(self, a, b):
        """動かしたいコマの選択"""
        while(True):
            print("動かしたい駒の座標をスペース区切りで入力(横,縦の順に数字で入力)", end=": ")
            try:
                x,y = (int(x) for x in input().split())
                break
            except ValueError:
                print("2 つの数字をスペース区切りで入力してください")
        return (10 - x) + y * 11

    def select2(self, a):
        """コマを配置したい位置の選択"""
        while(True):
                print("駒を配置したい座標をスペース区切りで入力(横,縦の順に数字で入力)", end=":
"")
                try:
                    s,t = (int(s) for s in input().split())
                    break
                except ValueError:
                    print("2 つの数字をスペース区切りで入力してください")
        return (10 - s) + t * 11
```

HumanGUI クラス

```
class HumanGUI():
    def select1(self, gui):
        """動かしたいコマを選択"""
        print("動かしたい駒の場所をクリックで入力", end=": ")
        x, y = map(int, gui.wait_for_input().split())
        return (10 - x) + y * 11

    def select2(self, gui):
        """コマを配置したい位置の選択"""
        print("駒を配置したい座標をクリックで入力", end=": ")
        s,t = map(int, gui.wait_for_input().split())
        return (10 - s) + t * 11
```

Random クラス

```
import random

class Random():

    def select1(self, board, turn):
        """動かすコマを自分のコマの中からランダムで 1 つ返す"""
        list = []
        i = 0
        while(i < len(board)):
```

```
                if(board[i] == turn):
                    list.append(i)
                i += 1
            if(len(list) - 1 == 0):
                z = list[0]
                return z
            else:
                z = list[random.randint(0, len(list) - 1)]
                return z


    def select2(self, legalMoveList):
        """配置する座標を合法手からランダムで 1 つ返す"""
        if(len(legalMoveList) - 1 == 0):
            u = legalMoveList[0]
            return u
        else:
            u = legalMoveList[random.randint(0, len(legalMoveList) - 1)]
            return u
```

AlphaBetaAgent クラス

```
from copy import deepcopy

class AlphaBetaAgent:
    def _init_(self, depth):
        """
        アルファベータ探索を行うエージェントを初期化する。
        :param depth: 探索の深さ
        """
        self.depth = depth  # 探索の深さ
        self.move_list = [-1, 1, -11, 11]
        self.is_reach = 0

    def select(self, board, player, is_reach):
        """
        与えられた局面(board)から最適な手を探索する。
        :param board: 現在の盤面の状態 (リスト形式)
        :param player: 自分の手番 (1 または-1)
        :return: (動かしたい駒の位置, 動かしたい場所)
        """
        self.is_reach = is_reach  # 初期リーチ状態をコピー



        best_move = None
        best_value = float('-inf') if player == 1 else float('inf')

        for move in self.get_legal_moves(board, player):
```

```
                new_board = self.make_move(board, move, player)
                value = self.alpha_beta(new_board, self.depth - 1, float('-inf'), float('inf'), -player)
                if (player == 1 and value > best_value) or (player == -1 and value < best_value):
                    best_value = value
                    best_move = move

        return best_move

    def alpha_beta(self, board, depth, alpha, beta, player):
        """
        アルファベータ探索のメインロジック。
        :param board: 現在の盤面の状態
        :param depth: 残りの探索深さ
        :param alpha: 現在のアルファ値（最大化側の最良値）
        :param beta: 現在のベータ値（最小化側の最良値）
        :param player: 現在の手番（1 または-1）
        :return: 評価値
        """
        terminal = self.is_terminal(board)

        if terminal == 1:
            return 999
        elif terminal == -1:
            return -999
        elif depth == 0:
            return self.evaluate_board(board, player)

        if player == 1:  # 自分の手番（最大化）
            max_eval = float('-inf')
            for move in self.get_legal_moves(board, player):
                new_board = self.make_move(board, move, player)
                eval = self.alpha_beta(new_board, depth - 1, alpha, beta, -player)
                max_eval = max(max_eval, eval)
                alpha = max(alpha, eval)
                if beta <= alpha:  # 枝刈り
                    break
            return max_eval
        else:  # 相手の手番（最小化）
            min_eval = float('inf')
            for move in self.get_legal_moves(board, player):
                new_board = self.make_move(board, move, player)
                eval = self.alpha_beta(new_board, depth - 1, alpha, beta, -player)
                min_eval = min(min_eval, eval)
                beta = min(beta, eval)
                if beta <= alpha:  # 枝刈り
                    break
            return min_eval
```

```
def get_legal_moves(self, board, player):
    """
    指定された盤面と手番に基づいて合法手をリストとして返す。
    :param board: 現在の盤面の状態
    :param player: 現在の手番 (1 または-1)
    :return: 合法手 [(駒の位置, 移動先), ...] のリスト
    """
    legal_moves = []
    for idx, piece in enumerate(board):
        if piece == player:  # 自分の駒のみ対象
            moves = self.check_legal_moves(board, idx)
            legal_moves.extend([(idx, move) for move in moves])
    return legal_moves

def check_legal_moves(self, board, position):
    """
    指定された位置にある駒の合法手を取得する。
    :param board: 現在の盤面の状態(リスト)
    :param position: 駒の位置
    :return: 移動可能な場所のリスト
    """
    legal_moves = []
    move_list = [-1, 1, -11, 11]  # 左、右、上、下への移動方向(11x11 の盤面を仮定)

    for direction in move_list:
        offset = direction
        while board[position + offset] == 0:
            legal_moves.append(position + offset)
            offset += direction

    return legal_moves


def make_move(self, board, move, player):
    """
    指定された手を実行し、盤面を更新する。
    :param board: 現在の盤面の状態 (リスト)
    :param move: 実行する手 (駒の位置, 移動先) のタプル
    :param player: 現在の手番 (1 または-1)
    :return: 更新後の盤面
    """
    local_board = deepcopy(board)
    start, end = move  # 駒の現在位置と移動先

    # 駒を移動させる
    local_board[end] = player
```

```
        local_board[start] = 0

        # 駒の取得処理（直線方向と囲みの両方を考慮）
        for direction in self.move_list:  # self.move_list は[-1, 1, -11, 11]
            self.capture_line(local_board, end, direction, player)

        surrounded_pieces = self.capture_surrounded(local_board, end, player)

        # 囲まれた駒を盤面から削除
        for pos in surrounded_pieces:
            local_board[pos] = 0

        return local_board

    def capture_line(self, board, position, direction, player):
        """
        直線方向で挟み取った駒を取得する。
        :param board: 現在の盤面の状態（リスト）
        :param position: 現在の位置
        :param direction: チェックする方向
        :param player: 現在の手番（1 または-1）
        """
        captured_positions = []
        offset = direction
        while board[position + offset] == -player:  # 相手の駒をチェック
            captured_positions.append(position + offset)
            offset += direction
            if board[position + offset] == player:  # 自分の駒で挟んでいる場合
                for pos in captured_positions:
                    board[pos] = 0  # 駒を取得
                return

    def capture_surrounded(self, board, position, player):
        """
        囲まれた駒を取得する。
        :param board: 現在の盤面の状態（リスト）
        :param position: 駒の位置
        :param player: 現在の手番（1 または-1）
        :return: 囲まれている駒のリスト
        """
        surrounded_positions = []

        def dfs(pos, visited):
            # DFS による探索
            visited.add(pos)
            for direction in self.move_list:
                next_pos = pos + direction
```

```
                if next_pos < 0 or next_pos >= len(board):  # 盤面外チェック
                    continue
                if next_pos in visited:  # すでに訪問済み
                    continue
                if board[next_pos] == 0:  # 空マスに接続している場合、囲まれていない
                    return []
                if board[next_pos] == -player:  # 相手の駒ならさらに探索
                    result = dfs(next_pos, visited)
                    if result == []:
                        return []  # 囲まれていない場合
            return visited

        # 周囲の相手の駒をチェック
        for direction in self.move_list:
            adjacent_pos = position + direction
            if board[adjacent_pos] == -player:
                visited = set()
                result = dfs(adjacent_pos, visited)
                if result:  # 囲まれている場合
                    surrounded_positions.extend(result)

        return list(set(surrounded_positions))  # 重複を削除して返す


    def is_terminal(self, board):
        """
        勝敗を判定し、ゲームの状態を返す。
        :param board: 現在の盤面の状態 (リスト)
        :param is_reach: 現在の到達状態 (1, -1, 0 のいずれか)
        :return: ゲームの状態:
                - 1: 先手の勝利
                - -1: 後手の勝利
                - 0: ゲーム継続
        """
        # 1. 相手駒が 4 つ以下であるかをチェック(先手の勝利)
        if board.count(-1) <= 4:
            return 1

        # 2. 自駒が 4 つ以下であるかをチェック(後手の勝利)
        if board.count(1) <= 4:
            return -1

        # 3. ボードの合計値をチェックして「到達」状態かを判定
        total = sum(piece for piece in board if piece != 99)  # 99 は壁と仮定
        if total >= 3:  # 先手の到達条件
            if self.is_reach == 1:
                return 1  # 先手の勝利
```

```
                self.is_reach = 1
            elif total <= -3:  # 後手の到達条件
                if self.is_reach == -1:
                    return -1  # 後手の勝利
                self.is_reach = -1
            else:
                self.is_reach = 0  # 到達状態リセット

            return 0  # ゲーム継続

        def evaluate_board(self, board, player):
            """スコアを返す"""
            score = 0

            # 駒の数による基本スコア
            my_pieces = board.count(player)
            opponent_pieces = board.count(-player)
            score += (my_pieces - opponent_pieces) * 10

            # 中央部のボーナス
            central_bonus = 0
            for idx, piece in enumerate(board):
                if piece == player:
                    row, col = divmod(idx, 11)
                    central_bonus += max(0, 2 - max(abs(row - 5), abs(col - 5)))  # 最大 2 点
            score += central_bonus

            # キャプチャ可能性のボーナス
            capture_bonus = 0
            for idx, piece in enumerate(board):
                if piece == player:
                    capture_bonus += min(2, len(self.check_legal_moves(board, idx)))  # 最大 2 点
            score += capture_bonus

            # 移動範囲の広さボーナス
            mobility_bonus = sum(len(self.check_legal_moves(board, idx)) for idx, piece in enumerate(board) if
        piece == player)
            score += mobility_bonus * 0.2  # 重みを 0.2 に調整

            return score * player
```

Ai_agent クラス

```
import numpy as np
import torch
import random

class Ai_agent:
```

```
    def __init__(self, model, pair_action_size, epsilon):
        """
        エージェントの初期化
        :param model: 訓練済みまたは新しいモデル
        :param pair_action_size: 駒と目的地ペアの選択肢数
        :param epsilon:  ε -greedy 戦略の ε（ランダム行動の確率）
        """
        self.model = model
        self.pair_action_size = pair_action_size
        self.epsilon = epsilon    #  ε -greedy 戦略用の値

    def select_action(self, state, available_pairs, turn):
        """
        :param state: 現在の盤面の状態
        :param available_pairs: 駒と目的地の合法なペアリスト（例：[(piece1, dest1), (piece2, dest2), ...]）
        :return: 選択された駒と移動先
        """
        if turn == -1:   # 後手の場合、state を反転
            state = self.reverse_state(state)
            available_pairs = [(120 - piece, 120 - dest) for piece, dest in available_pairs]

        if np.random.rand() <= self.epsilon:
            # ランダム行動
            selected_pair = random.choice(available_pairs)
        else:
            # 状態をテンソルに変換
            state_tensor = torch.tensor(state, dtype=torch.float32).unsqueeze(0)  # バッチ次元を追加
            with torch.no_grad():  # 勾配計算を無効化して予測
                pair_logits = self.model(state_tensor).squeeze(0)
            pair_logits = pair_logits.numpy()

            # 合法なペアのみ評価
            legal_q_values = [(pair, pair_logits[self.get_pair_index(pair)]) for pair in available_pairs]
            selected_pair = max(legal_q_values, key=lambda x: x[1])[0]

        # 後手の場合、選択したペアを元の視点に戻す
        if turn == -1:
            selected_pair = (120 - selected_pair[0], 120 - selected_pair[1])

        return selected_pair

    def get_pair_index(self, pair):
        """
        ペア（駒，目的地）のインデックスを取得する関数。
        :param pair: (駒のインデックス，目的地のインデックス) のタプル
        :return: ペアに対応する一意のインデックス
        """
```

```
            piece, destination = pair
            return piece * 121 + destination

        def reverse_state(self, state):
            """
            後手視点用に state を反転させる。
            - 配列を逆順にする。
            - 1 と-1 を反転。
            :param state: 現在の盤面の状態
            :return: 反転した state
            """
            reversed_state = state[::-1]  # 配列を逆順に
            return np.array([-x if x != 99 else 0 for x in reversed_state])  # 1 と-1 を反転
```

Model クラス

```
import torch
import torch.nn as nn

class Model(nn.Module):
    def __init__(self, state_size, pair_action_size):
        """モデルの初期化"""
        super(Model, self).__init__()
        self.fc1 = nn.Linear(state_size, 128)
        self.fc2 = nn.Linear(128, 128)
        self.fc3 = nn.Linear(128, 128)

        self.dropout = nn.Dropout(0.3)

        # 駒と移動先のペア用の出力ヘッド
        self.pair_head = nn.Linear(128, pair_action_size)

    def forward(self, x):
        """入力された局面の状態からコマと移動先のペアの Q 値を計算する"""
        x = torch.relu(self.fc1(x))
        x = self.dropout(torch.relu(self.fc2(x)))
        x = torch.relu(self.fc3(x))

        # 駒と移動先ペアの Q 値
        pair_logits = self.pair_head(x)

        return pair_logits

def create_model(state_size, pair_action_size):
    """
    駒と移動先ペアを選択するモデルを作成するためのファクトリ関数
    :param state_size: 盤面の状態のサイズ（入力の次元数）
    :param pair_action_size: 駒と移動先のペアの総数
```

```
    :return: モデルのインスタンス
    """
    model = Model(state_size, pair_action_size)
    return model
```

Experience_replay クラス

```
from collections import deque
import pickle
import random

class ExperienceReplay:
    def __init__(self, capacity):
        """経験バッファの初期化, キャパシティの設定を行う"""
        self.memory = deque(maxlen=capacity)

    def add(self, experience):
        """経験バッファに経験を追加"""
        self.memory.append(experience)

    def sample(self, batch_size):
        """経験バッファからランダムな経験を返す"""
        return random.sample(self.memory, batch_size)

    def __len__(self):
        """経験バッファのサイズを返す"""
        return len(self.memory)

    def save(self, filename):
        """経験バッファの保存を行う"""
        with open(filename, 'wb') as f:
            pickle.dump(self.memory, f)

    def load(self, filename):
        """経験バッファの読み込みを行う"""
        with open(filename, 'rb') as f:
            self.memory = pickle.load(f)
```

Train クラス

```
from Phase import Phase
from Ai_agent import Ai_agent
from Model import create_model  # ニューラルネットワークを定義したファイル
from Experience_replay import ExperienceReplay  # 経験再生を定義したファイル
import numpy as np
import torch
import torch.nn as nn
import matplotlib.pyplot as plt
```

```
class Train():
    def _init_(self):
        """トレーニング環境の初期化"""
        # パラメータの設定
        self.state_size = 121  # 11x11 の盤面をフラットにしたサイズ
        self.pair_action_size = 121 * 121  # 駒の選択肢数 * 目的地の選択肢数
        self.gamma = 0.99  # 割引率
        self.batch_size = 256  # バッチサイズ
        self.step = 0
        self.target_update_interval = 100  # ターゲットネットワークの更新間隔
        self.epsilon = 0.1  #  ε -greedy 戦略の ε

        # モデルの作成
        # agent1 用のモデルの作成
        self.main_model_agent1 = create_model(self.state_size, self.pair_action_size)
        try:
            self.main_model_agent1.load_state_dict(torch.load("main_model.pth", weights_only=True))
            self.main_model_agent1.train()
            print("モデルを main_model.pth からロードしました")
        except FileNotFoundError:
            print("agent1 を新しいモデルで初期化しました")
        self.target_model_agent1 = create_model(self.state_size, self.pair_action_size)
        self.update_target_model(self.main_model_agent1, self.target_model_agent1)  # 初期化時にターゲットネットワークをメインネットワークで更新

        # agent2 用のモデルの作成
        main_model_agent2 = create_model(self.state_size, self.pair_action_size)

        # Optimizer と Loss Function の設定
        self.learning_rate = 0.001
        self.optimizer_agent1 = torch.optim.Adam(self.main_model_agent1.parameters(), lr=self.learning_rate)  # agent1 用の optimizer
        try:
            self.optimizer_agent1.load_state_dict(torch.load('optimizer_agent1.pth', weights_only=True))# 保存されているファイルがあればロード
            print("Optimizer_agent1 loaded successfully.")
        except FileNotFoundError:
            print("No optimizer_agent1 file found, starting fresh.")

        self.criterion = nn.MSELoss()  # 損失関数

        # 経験再生バッファの作成
        self.memory = ExperienceReplay(capacity=500000)
        try:
            self.memory.load('experience_replay.pkl')  # 保存されているファイルがあればロード
            print("Experience replay loaded successfully.")
        except FileNotFoundError:
```

```
            print("No experience replay file found, starting fresh.")

        #epsilon の調整
        self.epsilon_start = 1.00   # 初期の epsilon 値(完全にランダム)
        self.epsilon_min = 0.01        # 探索をほぼ停止する最小 epsilon 値
        self.decay_rate = 0.9999        # エピソードごとに epsilon を減少させる割合

        # エージェントのインスタンス化
        self.agent1 = Ai_agent(self.main_model_agent1, self.pair_action_size, epsilon=self.epsilon_start)  #
agent1: epsilon=epsilon_start
        self.agent2 = Ai_agent(main_model_agent2, self.pair_action_size, epsilon=1.0)  # agent2:
epsilon=1.0

        # 環境の初期化
        self.phase = Phase(self.agent1, self.agent2)  # ゲーム環境を初期化

        # トレーニングデータの記録用リスト
        self.episode_rewards = []  # 各エピソードの総報酬を記録するリスト
        self.win_rates = []  # 勝率を記録するリスト
        self.max_win_rate = 0
        self.total_episodes = 0
        self.win_count = 0

def train_ai_agent(self, main_model, target_model, memory, batch_size, gamma, optimizer, criterion):
        """Ai_agent の学習を行う"""
        if len(memory) < batch_size:
            return

        # サンプルを経験再生バッファから取得
        minibatch = memory.sample(batch_size)

        states, actions, rewards, next_states, dones = zip(*minibatch)

        # Numpy 配列を Tensor に変換
        states = torch.tensor(np.array(states, dtype=np.float32))
        actions = torch.tensor(np.array(actions, dtype=np.int64))  # 駒の位置と目的地
        rewards = torch.tensor(np.array(rewards, dtype=np.float32))
        next_states = torch.tensor(np.array(next_states, dtype=np.float32))
        dones = torch.tensor(np.array(dones, dtype=np.float32))

        # 現在の状態の Q 値を計算
        pair_logits = main_model(states)

        # actions を (piece_action, destination_action) の形から 1 次元のインデックスに変換
        piece_action = actions[:, 0]  # 駒の選択肢インデックス
        destination_action = actions[:, 1]  # 移動先の選択肢インデックス
```

```
        # 11x11 の盤面に基づいてインデックスを変換(行列の形式に合わせる)
        destination_index = (destination_action // 11) * 11 + (destination_action % 11)
        one_dim_action_index = piece_action * 121 + destination_index

        # `pair_logits` から `q_values` を取得
        q_values = pair_logits.gather(1, one_dim_action_index.unsqueeze(1)).squeeze(1)

        # 次の状態のターゲット Q 値を計算
        with torch.no_grad():
            next_pair_logits = target_model(next_states)
            next_q_values = next_pair_logits.max(1)[0]  # max(1)で次の状態の最大 Q 値を取得
            target_q_values = rewards + (gamma * next_q_values * (1 - dones))

        # 損失を計算
        loss = criterion(q_values, target_q_values)

        # モデルの最適化
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    def update_target_model(self, main_model, target_model):
        """ターゲットモデルの更新を行う"""
        target_model.load_state_dict(main_model.state_dict())

    def step1(self, action=None):
        """ゲームを進める"""
        if action is None:
            return self.phase.board_to_vector(), -1, False, 0

        selected_piece, destination = action  # action から駒の位置と目的地を取得

        # 駒の位置が有効かどうかを確認
        if selected_piece not in self.phase.get_available_pieces():
            # 無効な駒選択
            reward = -2  # 無効なアクションに対するペナルティ
            next_state = self.phase.board_to_vector()
            done = False
            return next_state, reward, done, 0

        # 駒を選択した位置から目的地までの移動が有効か確認
        if not self.is_valid_move(selected_piece, destination):
            reward = -1  # 無効なアクションに対するペナルティ
            next_state = self.phase.board_to_vector()
            done = False
            return next_state, reward, done, 0
```

```
        # 盤面の更新
        self.phase.set(selected_piece, destination)

        # 駒を取った数を取得
        captured_pieces = self.phase.getPieces(destination)

        # 駒を取った場合の報酬
        reward = captured_pieces * 5 if captured_pieces > 0 else 0

        # 駒の場所による報酬
        reward += self.calculate_proximity_reward(destination)

        reward += self.calculate_risk_penalty()

        # ゲームが終了したかどうかを確認
        done, which_win = self.phase.isWin()

        # ゲーム終了時の追加報酬を計算
        reward += self.calculate_reward(done)

        # 次の状態を取得
        next_state = self.phase.board_to_vector()

        if not done:
            # ターンの変更
            self.phase.turnChange()

        return next_state, reward, done, which_win

    def is_valid_move(self, selected_piece, destination_piece):
        """選択されたペアが合法かどうかを確認する"""
        legalMoveList = self.phase.checkLegalMove(selected_piece)
        if len(legalMoveList) > 0:
            if destination_piece in legalMoveList:
                return destination_piece
            else:
                #その位置に動かすことはできない場合
                return None
        else:
            #選択された位置のコマに合法手がない場合
            return None

    def calculate_reward(self, done):
        """
        勝利時の報酬
        :param done: ゲーム終了状態かどうか
        :return: 報酬
```

```
        """
        if done:
            my_pieces = [index for index, place in enumerate(self.phase.gameBoard) if place ==
self.phase.turn]
            opponent_pieces = [index for index, place in enumerate(self.phase.gameBoard) if place == -
self.phase.turn]

            my_piece_count = len(my_pieces)
            opponent_piece_count = len(opponent_pieces)
            return 10 + 2 * (my_piece_count - opponent_piece_count)
        else:
            return 0  # ゲームが終了していない場合

    def calculate_proximity_reward(self, position):
        """動かしたコマの位置が安全かを判定し，報酬を与える"""
        if self.is_safe_position(position) == 0:
            for move in self.phase.moveList:
                if self.phase.gameBoard[position + move] == -self.phase.turn:
                    return 0.001
            return 0
        else:
            return -1

    def calculate_risk_penalty(self):
        """現在の局面から次に取られるコマの最大数を求め，報酬を与える"""
        # 自分と相手の駒の数をカウント
        my_pieces = [index for index, place in enumerate(self.phase.gameBoard) if place ==
self.phase.turn]
        opponent_pieces = [index for index, place in enumerate(self.phase.gameBoard) if place == -
self.phase.turn]

        my_piece_count = len(my_pieces)
        opponent_piece_count = len(opponent_pieces)

        # ペナルティの基準値を駒数で調整
        if my_piece_count > opponent_piece_count:
            penalty_per_piece = -1.75
        elif my_piece_count == opponent_piece_count:
            penalty_per_piece = -2
        else:  # 相手の駒が多い場合
            penalty_per_piece = -2.25

        # 全ての自分の駒に対してリスクを評価
        max_taken = max(self.is_safe_position(my_piece) for my_piece in my_pieces)

        # ペナルティの合計を計算
        total_penalty = penalty_per_piece * max_taken
```

```
        return total_penalty

    def is_safe_position(self, position):
        """
        指定された駒の位置が次の相手の手番で取られる場合に、取られる最大の自分の駒の数を返す。
        安全な場合は 0 を返す。
        """
        opponent_pieces = [index for index, place in enumerate(self.phase.gameBoard) if place == -self.phase.turn]
        legalMoveLists = [self.phase.checkLegalMove(piece) for piece in opponent_pieces]
        flattened_list = set(move for sublist in legalMoveLists for move in sublist)

        max_taken_pieces = 0  # 取られる駒の最大数を追跡

        for move in self.phase.moveList:
            search_position = position + move
            captured_count = 0

            # 自分の駒が連続する間、カウントしながら探索
            while self.phase.gameBoard[search_position] == self.phase.turn:
                captured_count += 1
                search_position += move

            # 最後が相手の駒で、かつ合法な移動であれば取られる
            if self.phase.gameBoard[search_position] == -self.phase.turn:
                search_position2 = position - move
                reverse_captured_count = 0

                # 逆方向にも自分の駒を探索
                while self.phase.gameBoard[search_position2] == self.phase.turn:
                    reverse_captured_count += 1
                    search_position2 -= move

                # 逆方向の終点が相手の合法手の中にあれば取られる
                if search_position2 in flattened_list:
                    total_captured = captured_count + reverse_captured_count + 1  # 中心の駒も含める
                    max_taken_pieces = max(max_taken_pieces, total_captured)

        return max_taken_pieces

    def save_model(self, main_model, main_model_filename="model.pth"):
        """モデルの保存を行う"""
        torch.save(main_model.state_dict(), main_model_filename)
        print(f"Main_Model saved to {main_model_filename}")
```

```
def plot_training_progress(self, episode_rewards, win_rates):
    """エピソード毎の報酬と，勝率をグラフに描写"""
    plt.figure(figsize=(12, 5))

    # 報酬の推移をプロット
    plt.subplot(1, 2, 1)
    plt.plot(episode_rewards, label='Total Reward per Episode')
    plt.xlabel('Episode')
    plt.ylabel('Total Reward')
    plt.title('Reward Over Episodes')
    plt.legend()

    # 勝率の推移をプロット
    plt.subplot(1, 2, 2)
    plt.plot(win_rates, label='Win Rate', color='green')
    plt.xlabel('Episode')
    plt.ylabel('Win Rate')
    plt.title('Win Rate Over Episodes')
    plt.legend()

    plt.tight_layout()
    plt.show()

def train_roop(self):
    """トレーニングを指定した回数行う"""
    for episode in range(self.total_episodes, 10000):  # エピソードの数
        self.phase.reset()  # 盤面と状態をリセットする
        state = self.phase.board_to_vector()  # 初期状態の盤面を取得
        done = False
        total_reward = 0  # 各エピソードの総報酬を初期化
        while not done:
            #phase.displayBoard()
            # available_pieces とそれぞれに対応する合法な移動リストを取得
            available_pieces = self.phase.get_available_pieces()
            legalMoveLists = [self.phase.checkLegalMove(piece) for piece in available_pieces]

            # available_pairs を作成（駒と目的地のペアのリスト）
            available_pairs = []
            for piece in available_pieces:
                for destination in legalMoveLists[available_pieces.index(piece)]:
                    available_pairs.append((piece, destination))

            if self.phase.turn == 1:
                # available_pairs を select_action に渡す
                piece_action, destination_action = self.agent1.select_action(state, available_pairs,
self.phase.turn)
```

```
                    action = (piece_action, destination_action)

                    # 環境でアクションを実行し、次の状態と報酬を取得
                    next_state, reward, done, which_win = self.step1(action)  # `step`メソッドでゲームを進める
                    total_reward += reward  # 報酬を加算

                    # エージェントの経験をバッファに追加し、学習
                    self.memory.add((state, action, reward, next_state, done))
                else:
                    piece_action, destination_action = self.agent2.select_action(state, available_pairs, self.phase.turn)

                    action = (piece_action, destination_action)

                    # 環境でアクションを実行し、次の状態と報酬を取得
                    next_state, reward, done, which_win = self.step1(action)  # `step`メソッドでゲームを進める
                    reward = reward * -1
                    total_reward += reward  # 報酬を加算

                    # エージェントの経験をバッファに追加
                    self.memory.add((state, action, reward, next_state, done))

                # 状態を更新
                state = next_state

                self.train_ai_agent(self.main_model_agent1, self.target_model_agent1, self.memory, self.batch_size, self.gamma, self.optimizer_agent1, self.criterion)

                self.step += 1

                if self.step % 500 == 0:  # 500 ステップごとにターゲットネットワークを更新
                    self.update_target_model(self.main_model_agent1, self.target_model_agent1)

            self.agent1.epsilon = max(self.epsilon_min, self.agent1.epsilon * self.decay_rate)

            # エピソードの結果を記録
            self.episode_rewards.append(total_reward)  # 総報酬を記録

            if which_win == 1:  # agent1 が勝利した場合
                self.win_count += 1

            self.total_episodes += 1
            self.win_rate = self.win_count / self.total_episodes
            self.win_rates.append(self.win_rate)
```

```
                print(f"Episode {episode + 1} completed.")

                # 1000 エピソードごとにモデルの保存
                if (episode + 1) % 1000 == 0:
                    self.save_model(self.main_model_agent1, "main_model_" + str(episode + 1) + ".pth")
                    torch.save(self.optimizer_agent1.state_dict(), 'optimizer_agent1.pth')
                    self.memory.save('experience_replay.pkl')  # バッファの内容を保存する
                    print(self.agent1.epsilon, self.win_rate, self.win_count)

            #トレーニング進捗をプロット
            self.plot_training_progress(self.episode_rewards, self.win_rates)

if __name__ == "__main__":
    train1 = Train()
    train1.train_roop()
```

Test クラス

```
from Phase import Phase
from Human import Human
from Random import Random
from AlphaBetaAgent import AlphaBetaAgent
from Ai_agent import Ai_agent
from Model import create_model
import torch

class Test():
    def __init__(self):
        """テスト環境の初期化"""
        # パラメータの設定
        state_size = 121  # 11x11 の盤面をフラットにしたサイズ
        pair_action_size = 121 * 121

        # モデルの作成
        # agent1 用のモデルの作成
        main_model = create_model(state_size, pair_action_size)
        try:
            main_model.load_state_dict(torch.load("main_model.pth", weights_only=True))
            main_model.eval()
            print(f"モデルを main_model.pth からロードしました")
        except FileNotFoundError:
            print("新しいモデルで初期化しました")

        self.phase = Phase(Ai_agent(main_model, pair_action_size, epsilon=0), Random())

    def start_game1(self):
        """1 回のみ、局面をテキストベースで表示"""
        while(True):
```

```
                self.phase.displayBoard()
                self.phase.select()
                x,y = self.phase.isWin()
                if(x == True):
                    print(y)
                    break
                self.phase.turnChange()

    def start_game2(self):
        """100 回繰り返し、最後に先手の勝った数を表示"""
        win_count = 0
        for episode in range(1):   # エピソードの数
            self.phase.reset()   # 盤面と状態をリセットする
            done = False
            while not done:
                self.phase.select()
                # ゲームが終了したかどうかを確認
                done, which_win = self.phase.isWin()

                if not done:
                    # ターンの変更
                    self.phase.turnChange()

            if which_win == 1:   # agent1 が勝利した場合
                win_count += 1

            print(f"Episode {episode + 1} completed.")
        print("total_win_agent1 =", win_count)

if __name__ == "__main__":
    game = Test()
    game.start_game2()
```

HasamishogiGUI クラス

```
import tkinter as tk
from Phase import Phase
from Human import Human
from AlphaBetaAgent import AlphaBetaAgent
from HumanGUI import HumanGUI
from Random import Random
from Ai_agent import Ai_agent
from Model import create_model
import torch

class HasamiShogiGUI:
    def __init__(self, master, phase):
        """環境の初期化"""
```

```
        self.master = master
        self.phase = phase
        self.selected_piece = None
        self.input_value = None
        self.wait_for_input_flag = False

        # ウィンドウタイトルとサイズ設定
        self.master.title("Hasami Shogi")
        self.master.geometry("545x545")

        # 将棋盤の描画領域を作成
        self.board_canvas = tk.Canvas(self.master, width=600, height=600, bg="black")
        self.board_canvas.pack()
        self.draw_board()

    def draw_board(self):
        """将棋盤と駒を描画"""
        self.board_canvas.delete("all")  # 前の状態をクリア
        cell_size = 60

        for i in range(9):
            for j in range(9):
                x1, y1 = j * cell_size, i * cell_size
                x2, y2 = x1 + cell_size, y1 + cell_size

                # 市松模様の色を設定
                fill_color = "#F0D9B5"# if (i + j) % 2 == 0 else "#B58863"
                self.board_canvas.create_rectangle(x1, y1, x2, y2, fill=fill_color, outline="black")

                # 駒を描画
                piece = self.phase.gameBoard[(i + 1) * 11 + (j + 1)]
                if piece == 1:
                    self.board_canvas.create_text((x1 + x2) // 2, (y1 + y2) // 2, text="▲",
    fill="black", font=("Helvetica", 24))
                elif piece == -1:
                    self.board_canvas.create_text((x1 + x2) // 2, (y1 + y2) // 2, text="▽",
    fill="black", font=("Helvetica", 24))

        # クリックイベントを設定
        self.board_canvas.bind("<Button-1>", self.on_click)

    def on_click(self, event):
        """クリック時のイベント処理"""
        cell_size = 60
        col = event.x // cell_size + 1
        row = event.y // cell_size + 1
```

```
            if self.wait_for_input_flag:  # 入力待機中のみ動作
                self.input_value = f"{10 - col} {row}"  # 入力フォーマットに変換
                self.wait_for_input_flag = False  # 待機フラグを解除
                print(f"Clicked position: {self.input_value}")
            else:
                print("Input is not being requested.")

        def wait_for_input(self):
            """クリックによる入力を待機する"""
            self.wait_for_input_flag = True
            self.input_value = None

            print("Waiting for input via click...")
            while self.wait_for_input_flag:
                self.master.update()  # Tkinter のイベントループを処理

            return self.input_value  # ユーザーがクリックした値を返す


if __name__ == "__main__":
    """ゲームの実行"""
    root = tk.Tk()

    # モデルの作成
    # agent1 用のモデルの作成
    main_model = create_model(state_size=121, pair_action_size=121*121)
    try:
        main_model.load_state_dict(torch.load("main_model.pth", weights_only=True))
        main_model.eval()
        print(f"モデルを main_model.pth からロードしました")
    except FileNotFoundError:
        print("新しいモデルで初期化しました")

    phase = Phase(HumanGUI(), Ai_agent(main_model, pair_action_size=121*121, epsilon=0))
    gui = HasamiShogiGUI(root, phase)

    while True:
        gui.draw_board() #盤面を表示
        phase.selectGUI(gui)  # GUI を使った選択処理
        win, winner = phase.isWin()
        if win:
            gui.draw_board()  # ゲーム終了時の盤面を表示
            gui.master.update_idletasks()
            gui.master.update()
            print(f"Game over! Winner: {'Player 1' if winner == 1 else 'Player 2'}")
            break
        phase.turnChange()
```

```
print("Press the close button on the window to exit.")
gui.master.mainloop()
```