卒業研究報告書

題目

ジャンケン将棋のゲーム AI 開発

指 導 教 員 石水 隆 講師

報告者

19-1-037-0196

竹田侑平

近畿大学理工学部情報学科

令和 5 年 1 月 26 日提出

概要

ジャンケン将棋は，6×6 の盤面とグー，チョキ，パーが二面ずつ描かれたサイコロ状の駒を用いる将棋類の一つである．ジャンケン将棋は，各面にジャンケンの手が描かれた立方体の駒を使用する．自駒と相手駒が隣接しているとき，自駒の上面に描かれている手が相手駒の上面に描かれている手にジャンケンで勝てるときその相手駒を取ることができる．ジャンケン将棋の駒を移動させるときは転がすように回転させるため，駒の上面に描かれている手は移動させるたびに変わる．ジャンケン将棋は駒の種類は一つのみであるが駒を動かすたびに自駒と相手駒の相性が変わるため奥深いゲームとなっている．ジャンケン将棋の知名度は高いとはいえず，公開されているアプリケーションなどはほとんどない．そこで本研究ではジャンケン将棋のゲームアプリケーションを作成し，一人でも対戦できるようジャンケン将棋のゲーム AI を開発する．

本将棋のようなゲーム AI では，局面の評価値を求めそれを比較することで着手選択をする方法が用いられる．本将棋は長年研究されており，局面の評価値を求める手法はある程度確立している．しかし，ジャンケン将棋の場合はそのゲーム性の違いや知名度の低さから，ゲーム AI に必要な局面の評価値の基準が存在しない．そのためその基準を一から求める必要がある．本研究では局面の評価値の求め方を変えながら AI 同士を対戦させることで，適切な評価値を求めていく．

目次

1　序論

1.1 ジャンケン将棋とは

あたまのよくなるゲーム：じゃんけんしょうぎ（以下ジャンケン将棋とする）は，将棋類の一つで，梅田龍一によって考案され，学研の「頭のよくなるゲームシリーズ」として 2010 年に発売されたボードゲームである[2]．対象年齢は 6 歳からで，シンプルなルールであるため，すぐに覚えやすく手軽に楽しめるゲームである．ジャンケン将棋は，各面にジャンケンの手が描かれた立方体の駒を使用する．自駒と相手駒が隣接しているとき，自駒の上面に描かれている手が相手駒の上面に描かれている手にジャンケンで勝てるときその相手駒を取ることができる．ジャンケン将棋の駒を移動させるときは転がすように回転させため，駒の上面に描かれている手は移動させるたびに変わる．ジャンケン将棋は駒の種類は一つのみであるが駒を動かすたびに自駒と相手駒の相性が変わるため奥の深いゲームとなっている．しかし現在は公開されているジャンケン将棋のアプリケーションはほとんどない．そのため本研究で扱うジャンケン将棋とは別のゲームであり，ジャンケン将棋自体の知名度は低いといえる．

1.2 二人零和有限確定完全情報ゲーム

将棋や囲碁に代表されるボードゲームは二人零和有限確定完全情報ゲームに分類されており，世界中に様々なバリエーションが存在する[1]．二人零和有限確定完全情報ゲームとは，以下の条件を満たすゲームである．

- プレイヤーの人数が二人
- プレイヤーの点数の和が 0（得点を奪い合う）
- ゲームの可能な局面の数が有限
- ゲーム内にランダム要素が存在しない
- 全プレイヤー同士で全ての情報が公開されている

二人零和有限確定完全情報ゲームに分類されるボードゲームは，双方最善手を打った場合，先手勝ち，後手勝ち，引き分けのどれになるかはゲーム開始時点で決定しており，理論上，全ての可能な局面を解析することができれば最善の手を打つことができる．

しかし多くのボードゲームでは，可能な局面の総数が極めて大きいため，完全解析を行うことは不可能である．例を挙げれば，可能な局面数はリバーシが 10^{28} 通り，チェスが 10^{5} 通り，将棋が 10^{69} 通り，囲碁が 10^{170} 通り程度あるとされており，現在の計算機の性能を越えている．一方，可能な局面数が少ないゲームでは完全解析されているものもある．連珠は双方最善手を打った場合，47 手で先手が勝つ[6]チェッカーは双方最善手を指すと引き分けとなる[7]．

局面数が大きいゲームについては，ゲーム盤をより小さいサイズに限定した場合の解析も行われている．サイズ 6×6 のリバーシでは，双方最善手を打つと 16 対 20 で後手勝ちとなる[8]．また，サイズ 4×4 の囲碁は双方最善手を打つと持碁(引き分け)[9]，5×5 の囲碁は黒の 25 目勝ちとなる[10]．

ジャンケン将棋も，二人零和有限確定完全情報ゲームの条件を満たすため，理論上はゲーム開始時に勝敗が確定する．なお，ジャンケン将棋には，手番開始時にサイコロを振って一度に指せる手数を決める，という拡張ルールがあり，この拡張ルールを使う場合は確定の条件を満たさないため，勝敗を確定することはできない．

1.3 類似したゲーム

ジャンケン将棋に類似したゲームには，対戦！ジャンケン将棋[4]や小諸じゃんけんしょうぎ[5]がある．「対決！じゃんけん将棋」[4]はスマートフォン用アプリケーションとしてリリースされている．これは駒の種類がグー，チョキ，パーの 3 種類あり，取った駒を自分の駒としてさすことができる．また，駒ごとに進める方向が決まっている．小諸じゃんけんしょうぎ[5]は使用する版や初期配置は本将棋とほとんど同じであり，駒がジャンケンの手になっている．各手番で，「飛，角，金，銀，桂，香」の描かれたサイコロを振って出た駒と同じ動きで自駒を進めるゲームである．

2 つともジャンケンの手の相性で駒を取れるかが決まる点は，ジャンケン将棋と類似しているが，駒やルー

ルが本将棋に近いものである．

1.4 本研究の目的

現在ジャンケン将棋の知名度は低いため，公開されているアプリケーションやゲーム AI は確認できなかった．そこで本研究ではゲーム AI を実装したジャンケン将棋のアプリケーションを作成する．

またゲーム AI を実装するにあたって，局面の評価値を求める必要がある．その評価関数は一から作成することになるが，より強いゲーム AI を作成するために適したものを求めていく．

既存のジャンケン将棋のプログラム[7]があるがこのプログラムは人と人の対戦を目的に作られたものであり，ゲーム AI が無くコンピュータとの対戦はできない．本研究で作成するプログラムはゲーム AI の完成を目指すため，プレイヤーの使いやすさよりも探索の処理を優先している．

1.5 本報告書の構成

本報告書の構成は以下の通りである．

2 節で本研究の対象であるジャンケン将棋について説明する．3 節，4 節では作成したプログラムについてのべる．5 節ではゲーム AI の評価関数について検証とその結果を述べる．6 節では本研究の結果から今後の課題について述べる．

2　ジャンケン将棋

2.1 ジャンケン将棋の概要

ジャンケン将棋は，立方体に描かれたジャンケンの優劣で駒もしくはゴールを取り合う 2 人用ボードゲームである．使用するものは，6×6 のゲーム盤(図 1)，6 つの面にグー，チョキ，パーの 3 種類が 2 つずつ描かれた立方体の駒(図 2)を先手後手それぞれ 4 つの計 8 つ，以上の 2 点である

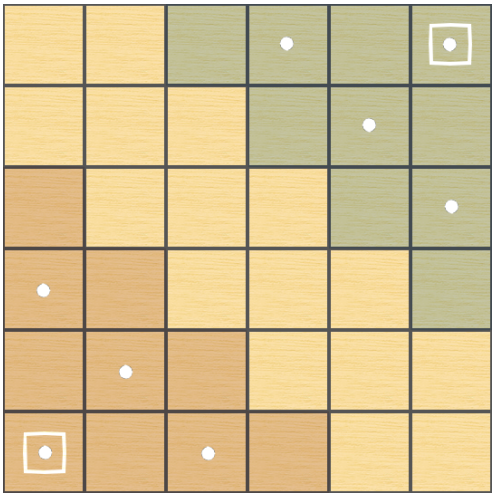

図 1 ゲーム版

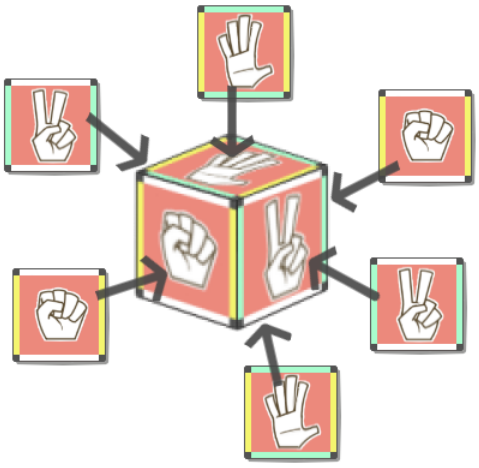

図 2 ジャンケン将棋の駒

2.2 ジャンケン将棋のルール

ジャンケン将棋のルールは以下の通りである．

- 勝利条件
  勝利条件は相手からゴールを奪うか，相手駒を全滅させることである．ゴールの位置は図 1 の盤面の印が付いている対角線上にあるマスの２箇所である．

- 準備
  駒の初期位置は図 1 の盤面の印が付いているますの８箇所である．配置する際，駒の向きをグー，チョキ，パーで自由に決められる．

- 駒の動かし方
  駒は基本回転して上下左右の隣接するマスに移動させる．駒を動かせる回数を行動力といい，各手番で 2 行動力ずつある．この時 1 つの駒に 2 回行動力を使用するか，2 つの駒に 1 回ずつ行動力を使用するかのどちらかを選択できる．ただし，1 つの駒に 2 回行動力を使用する場合，2 回目の移動で元の位置に戻すことはできない．図 4 に駒を動かす様子を示す．

- 駒の取り方
  自駒の上下左右の隣接するマスに相手駒があり，自駒の上面に描かれたジャンケンの手が相手駒の上面に描かれたジャンケンの手に勝っている場合，行動力を 1 使用して回転せずに相手駒を取ることができる．図 3 に駒を取る様子を示す．

- 出戻り禁止エリア
  自分のゴールから３マスの範囲が自分のエリアとなる．図 1 の自駒の初期位置側の色がついているマスが自分の出戻り禁止エリアである．自駒を 1 度このエリアから出すと，その駒をもう自分のエリアには戻すことはできない．

- ふんばりモード
  自駒の数が残り 1 個になると，ふんばりモードという状態になる．この状態では，自分の行動力が 2 から 3 になり，出戻り禁止エリアも自由に移動できるようになる．

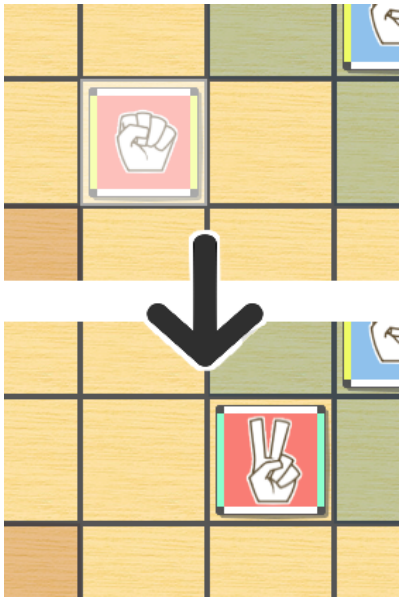

図 4 駒を動かす様子

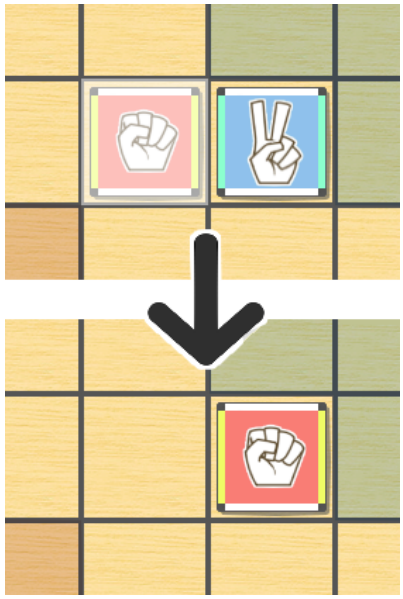

図 3 駒を取る様子

3　ジャンケン将棋プログラム

本章では，本研究で作成したジャンケン将棋プログラムについて述べる．付録に本研究で作成したジャンケン将棋プログラムのソースを示す.

3.1 プログラムの仕様

本節では本研究作成したジャンケン将棋プログラムの仕様について述べる．本研究作成したアプリケーションでは，ゲーム AI で探索処理を何度も行うため，ユーザーインターフェースは CUI で，各手番の行動力は 1 で実装した．プログラムを起動させると，図 5 のように初期配置を設定するテキストが表示される．

```
infonoMacBook-Air-5:jankenshogi Yuhei$  cd /Users/Yuhei/Documents/卒業研究/jankenshogi ; /usr/bin/env /Library/Java/JavaVirtualMachines/jdk
-12.jdk/Contents/Home/bin/java -cp /Users/Yuhei/Documents/卒業研究/jankenshogi/bin JankenShogi
0ターン目
| | | |5| |8|
-------------
| | | | |6| |
-------------
| | | | | |7|
-------------
|2| | | | | |
-------------
| |3| | | | |
-------------
|1| |4| | | |
-------------
1番に設定するダイスの状態をオモテ面、前面の順にジャンケンの手を入力してください（例：グーパー）：グーチョキ
2番に設定するダイスの状態をオモテ面、前面の順にジャンケンの手を入力してください（例：グーパー）：グーパー
3番に設定するダイスの状態をオモテ面、前面の順にジャンケンの手を入力してください（例：グーパー）：パーチョキ
4番に設定するダイスの状態をオモテ面、前面の順にジャンケンの手を入力してください（例：グーパー）：パーグー
5番に設定するダイスの状態をオモテ面、前面の順にジャンケンの手を入力してください（例：グーパー）：チョキグー
6番に設定するダイスの状態をオモテ面、前面の順にジャンケンの手を入力してください（例：グーパー）：チョキパー
7番に設定するダイスの状態をオモテ面、前面の順にジャンケンの手を入力してください（例：グーパー）：g
上記の例と同じ形式で入力してください：
7番に設定するダイスの状態をオモテ面、前面の順にジャンケンの手を入力してください（例：グーパー）：グーチョキ
8番に設定するダイスの状態をオモテ面、前面の順にジャンケンの手を入力してください（例：グーパー）：グーパー
```

図 5 アプリケーション起動後の様子

初期配置の設定が終わると，図 6 のように盤面と駒の状態が表示され，動かす駒と動かす方向を n 湯力できるようになる．動かす駒の番号と，動かす方向の上下左右を（u，d，l，r）の記号で入力すると手番を進めることができる（図 7）.

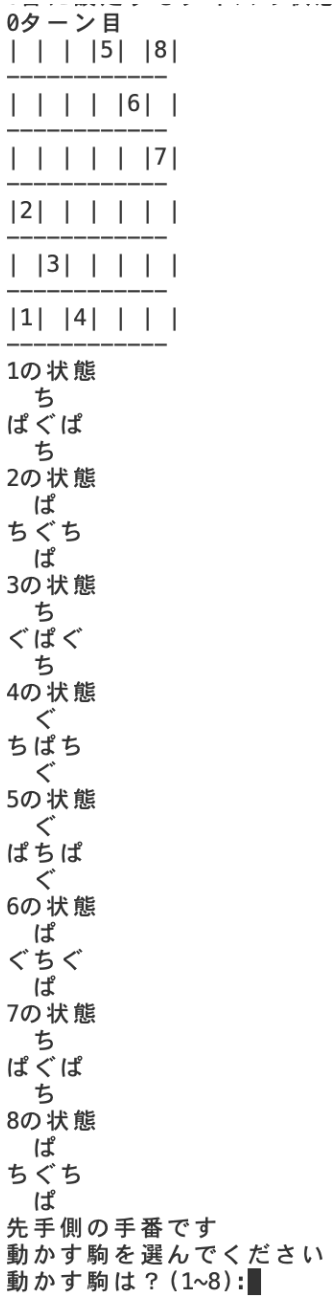

図 6 手番を進める様子(1)

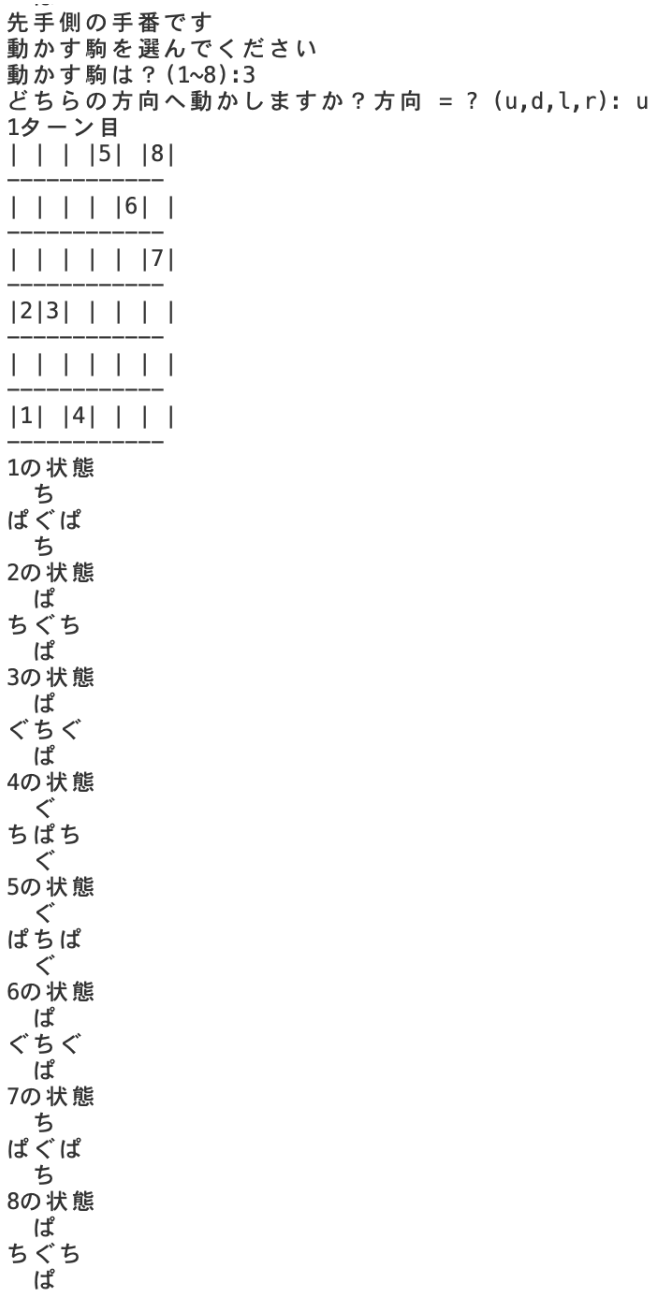

図 7 手番を進める様子(2)

駒の状態は，今後拡張機能として，グーとチョキだけの駒など特殊な駒を追加することができるよう State パターンで実装している．また，本研究で作成するゲーム AI は現在の局面からある程度先の局面を探索し，その局面での評価値を比べることで自分の手を選択する．

3.2 局面の評価値

本節では本研究で作成するゲーム AI で用いる評価値について述べる．
本研究で作成したゲーム AI では以下の要素から評価値を求めた．

- 自駒の目指すゴールまでの距離

- 駒の数
- 駒を動かせる手数

駒とゴールとの距離は近くなるほど評価があげている．駒の数に合わせて評価をあげているが，駒の数が 0 になると負けてしまうため，駒の数が減るほど 1 つあたりの駒の評価が大きくなるようにする．
各要素の係数を$goalValue, diceValue, moveValue$としたときの評価値の計算方法は以下のように求めた．

- (ゴールまでの最大距離-その駒のゴールまでの距離)*goalValue
- 駒の数による係数 k*diceValue
- 駒を動かせる手数*moveValue

表 1 駒の数による係数 k の値

駒の数	0 個	1 個	2 個	3 個	4 個
係数 k	0	10	15	18	20
1 駒あたりの価値	0	10	7.5	6	5

駒の数による係数 k の値は表 1 に示す．また，ゴール時の評価を上げるため，ゴールに到達した局面では$goalValue$を評価値に加える．これらの計算を各駒に対し行い局面の評価値は先手番に有利な要素は正の値，後手番が有利な要素は負の値として合計を求め，その合計値が正か負かでどちらの手番が有利かを判断する．

3.3 αβ法

本研究では評価値の探索にαβ法を用いる．αβ法は探索アルゴリズムの一つであり，ミニマックス法と基本は同じであるが，αカット，βカットという手法を用いることで余分な探索を中断し，計算量を減らしたアルゴリズムである．まず，ミニマックス法は先手も後手も互いに最善の手を選択することを想定したアルゴリズムである．そのため今回の場合，先手は評価値が最も高い局面，後手は評価値が最も低い局面を選ぶことになる．そして，αカット，βカット，によるαβ法の枝刈りを図 8 に示す．図 8 において，B の値は子の点である E，F のうち最小の値となる．そのため，E の探索が終わりその値が 4 と分かった時点で B の値は 4 以下であることが確定する．その一方で S の値は子の点である A，B のうち最大の値となる．A の値は 5 で確定しており，E の探索により B の値は A の値より大きくならないことが確定した．そして必要がなくなった F の探索を中断する．この流れがαカットである．同様に，D の子の点の探索中に 7 がでてきた時点で，D の値は 7 以上であることが確定する．このため，D の値が C の値より小さくならないことが確定し，これ以上の探索を中断する．これがβカットである．このようにαカットとβカットにより，ミニマックス法から計算量を減らしたものがαβ法である．
ここで，αカットとβカットによる枝刈りを探索の早い段階でできたら，より計算量を減らすことができるとわかる．例えば，図 8 において D の子の点の探索の順番は 7，3，5 で 7 の探索で中断したため 2 回の探索を減らしたことになる．それが探索の順番が 3，7，5 の順であれば 1 回の探索しか減らすことはできない．そのため，探索の順番により，計算量を減らすことができることがわかる．そこで本研究では各手番での探索の順番をゴールに近づく手から探索することで，早い段階でより有効な手を見つけ，計算量の削減を図っている．

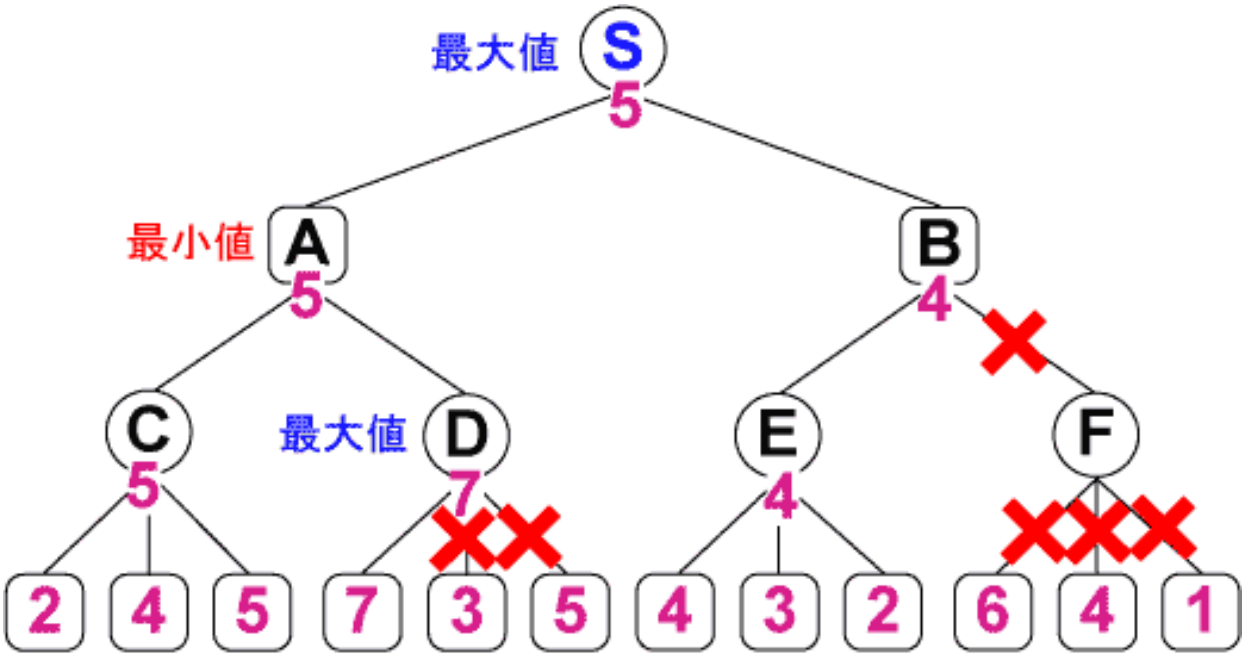

図 8 αβ法の枝刈り

3.4 プログラムの構造

本節では，本研究で作成したジャンケン将棋のプログラムの構造について述べる．付録 1 に本研究で作成したプログラムのソースプログラムを示す．

- Board クラス
- Cgp クラス
- Cpg クラス
- Dice クラス
- DiceState インターフェース
- Gcp クラス
- Gpc クラス
- JankenShogi クラス
- Pcg クラス
- Pgc クラス

次節で各クラスについて述べる．

4 クラス

4.1 Board クラス

Board クラスは盤面の状態を表すクラスである．図 9 に Board クラスのクラス図を示す．

Board		# 盤面の状態を表すクラス
Board ()		# コンストラクタ.
Board (list : ArrayList<Dice>)		# コンストラクタ.
Board (Board preboard)		# コンストラクタ.
showBoard0 ()	: void	# 盤面のみを表示するメソッド.
showBoard ()	: void	# 盤面と駒の状態を表示するメソッド.
showBoard2 ()	: void	# 盤面を壁も含めて数字で表示するメソッド.
move (d:String, type:int)	: boolean	# 駒を動かすメソッド．Type 番の駒を d の方向に動かす.
attack(Dice dice1 , Dice dice2)	: boolean	# 駒の攻撃ができるか判定するメソッド.
nextMove(String d, int type)	: Board	# 駒を動かした時の次の局面を返すメソッド.
value()	: int	# 現在の局面から評価値を計算しそれを返すメソッド
search_main(int depth, int depthMax, int alpha, int beta)	: int	# α β法で探索を行うメソッド.
search()	: int	# α β法での探索を始めるメソッド.
search(int dMax)	: int	# α β法での探索を始めるメソッド.
judge()	: boolean	# 勝敗を判定するメソッド.
nextMoves()	:ArrayList<Board>	# 現在の局面から可能な次の局面をリストで返すメソッド.
isfirst(Dice dice)	: int	# 引数のダイスが先手駒かどうかを判定するメソッド.
bestMove()	: Board	# 現在の局面から最善の局面を返すメソッド.

図 9 Board クラスのクラス図

- Board()
  コンストラクタ.
- Board(ArrayList<Dice> list)
  コンストラクタ．引数のリストの中のダイスに応じて盤面を作成する.
- Board(Board preboard)
  コンストラクタ．今の局面を引数で受け取り，複製する.
- showBoard0(): void
  盤面のみを表示するメソッド.
- showBoard(): void
  盤面と駒の状態を表示するメソッド.
- showBoard2: void
  盤面を壁も含めて数字で表示するメソッド.
- move(Sting d , int type): boolean
  駒を動かすメソッド．Type 番の駒を d の方向に動かす.
- attack(Dice dice1, Dice dice2): boolean
  駒の攻撃ができるか判定するメソッド．dice1 が dice2 を攻撃することができるなら true を返す.
- nextMove(String d, int type): Board
  駒を動かした時の次の局面を返すメソッド.
- value(): int
  現在の局面から評価値を計算しそれを返すメソッド
- search_main(int depth, int depthMax, int alpha, int beta): int

α β法で探索を行うメソッド．深度が最大深度に達するまでこのメソッドを再帰して探索する．最大深度に達するとその局面の評価値を返す．

- search(): int
  α β法での探索を始めるメソッド．
- search(int dMax): int
  α β法での探索を始めるメソッド．
- judge(): boolean
  勝敗を判定するメソッド．勝負がついている場合 true を返す
- nextMoves(): ArrayList<Board>
  現在の局面から可能な次の局面をリストで返すメソッド．
- isfirst(Dice dice):int
  引数のダイスが先手駒かどうかを判定するメソッド．
- bestMove(): Board
  現在の局面から最善の局面を返すメソッド．

4.2 Cgp クラス

Cgp クラスはダイスのオモテ面がチョキ，前面がグーの状態を表すクラスである．図 10 に Cgp クラスのクラス図を示す．

Cgp		# ダイスのオモテ面がチョキ，前面がグーの状態を表すクラス
Cgp(Dice dice)		# コンストラクタ．
showState()	: void	# ダイスの状態を表示するメソッド．
moveUp()	: void	# ダイスの状態遷移を行うメソッド．
moveDown()	: void	# ダイスの状態遷移を行うメソッド．
moveRight()	: void	# ダイスの状態遷移を行うメソッド．
moveLeft()	: void	# ダイスの状態遷移を行うメソッド．
attackDice(Dice dice)	: boolean	# ダイスの攻撃判定を行うメソッド．
getTopString()	: String	# ダイスのオモテ面のゲッター
getName()	: String	# ダイスの状態名のゲッター

図 10 Cgp クラスのクラス図

- Cgp(Dice dice)
  コンストラクタ．
- showState(): void
  ダイスの状態を表示するメソッド．
- moveUp(): void
  ダイスの状態遷移を行うメソッド．
- moveDown(): void
  ダイスの状態遷移を行うメソッド．
- moveRight(): void
  ダイスの状態遷移を行うメソッド．
- moveLeft(): void
  ダイスの状態遷移を行うメソッド．

- attackDice(Dice dice): boolean
  ダイスの攻撃判定を行うメソッド．引数のダイスに対して攻撃できる場合に true を返す
- getTopString(): String
  ダイスのオモテ面のゲッター
- getName(): String
  ダイスの状態名のゲッター

4.3 Cpg クラス

Cpg クラスはダイスのオモテ面がチョキ，前面がパーの状態を表すクラスである．図 11 に Cpg クラスのクラス図を示す．

Cpg		# ダイスのオモテ面がチョキ，前面がパーの状態を表すクラス
Cpg(Dice dice)		# コンストラクタ．
showState()	: void	# ダイスの状態を表示するメソッド．
moveUp()	: void	# ダイスの状態遷移を行うメソッド．
moveDown()	: void	# ダイスの状態遷移を行うメソッド．
moveRight()	: void	# ダイスの状態遷移を行うメソッド．
moveLeft()	: void	# ダイスの状態遷移を行うメソッド．
attackDice(Dice dice)	: boolean	# ダイスの攻撃判定を行うメソッド．
getTopString()	: String	# ダイスのオモテ面のゲッター
getName()	: String	# ダイスの状態名のゲッター

図 11 Cpg クラスのクラス図

- Cpg(Dice dice)
  コンストラクタ．
- showState(): void
  ダイスの状態を表示するメソッド．
- moveUp(): void
  ダイスの状態遷移を行うメソッド．
- moveDown(): void
  ダイスの状態遷移を行うメソッド．
- moveRight(): void
  ダイスの状態遷移を行うメソッド．
- moveLeft(): void
  ダイスの状態遷移を行うメソッド．
- attackDice(Dice dice): boolean
  ダイスの攻撃判定を行うメソッド．引数のダイスに対して攻撃できる場合に true を返す
- getTopString(): String
  ダイスのオモテ面のゲッター
- getName(): String
  ダイスの状態名のゲッター

4.4 Dice クラス

Dice クラスはダイスを表すクラスである．図 12 に Dice クラスのクラス図を示す.

Dice		# ダイスを表すクラス
Dice(int x,　int y)		# コンストラクタ.
Dice(int x,　int y,　DiceState state)		# コンストラクタ.
setState(DiceState state)	:void	# ダイスの状態のセッター
getState()	: DiceState	# ダイスの状態のゲッター.
showState()	: void	# ダイスの状態を表示するメソッド.
moveUp()	: void	# ダイスの状態遷移を行うメソッド.
moveDown()	: void	# ダイスの状態遷移を行うメソッド.
moveRight()	: void	# ダイスの状態遷移を行うメソッド.
moveLeft()	: void	# ダイスの状態遷移を行うメソッド.
attackDice(Dice dice)	: boolean	# ダイスの攻撃判定を行うメソッド.
clone()	: Object	# ダイスの複製を行うメソッド

図 12 Dice クラスのクラス図

- Dice(int x,　int y)
  コンストラクタ．駒の初期状態は Gcp になる.
- Dice(int x,　int y,　DiceState state)
  コンストラクタ．引数の状態が駒の初期状態になる.
- setState(DiceState state):void
  ダイスの状態のセッター.
- getState(): DiceState
  ダイスの状態のゲッター.
- showState(): void
  ダイスの状態を表示するメソッド.
- moveUp(): void
  ダイスの状態遷移を行うメソッド.
- moveDown(): void
  ダイスの状態遷移を行うメソッド.
- moveRight(): void
  ダイスの状態遷移を行うメソッド.
- moveLeft(): void
  ダイスの状態遷移を行うメソッド.
- attackDice(Dice dice): boolean
  ダイスの攻撃判定を行うメソッド．引数のダイスに対して攻撃できる場合に true を返す.
- clone(): Object
  ダイスの複製を行うメソッド.

4.5 DiceState インターフェース

DiceState インターフェースはダイスの状態を表すインターフェースである．図 13 に DiceState インターフェースのクラス図を示す.

DiceState		# ダイスの状態を表すインターフェース
showState()	: void	# ダイスの状態を表示するメソッド.
moveUp()	: void	# ダイスの状態遷移を行うメソッド.
moveDown()	: void	# ダイスの状態遷移を行うメソッド.
moveRight()	: void	# ダイスの状態遷移を行うメソッド.
moveLeft()	: void	# ダイスの状態遷移を行うメソッド.
attackDice(Dice dice)	: boolean	# ダイスの攻撃判定を行うメソッド.
getTopString()	: String	# ダイスのオモテ面のゲッター
getName()	: String	# ダイスの状態名のゲッター

図 13 DiceState インターフェースのクラス図

- showState(): void
  ダイスの状態を表示するメソッド.
- moveUp(): void
  ダイスの状態遷移を行うメソッド.
- moveDown(): void
  ダイスの状態遷移を行うメソッド.
- moveRight(): void
  ダイスの状態遷移を行うメソッド.
- moveLeft(): void
  ダイスの状態遷移を行うメソッド.
- attackDice(Dice dice): boolean
  ダイスの攻撃判定を行うメソッド．引数のダイスに対して攻撃できる場合に true を返す.
- getTopString(): String
  ダイスのオモテ面のゲッター
- getName(): String
  ダイスの状態名のゲッター

4.6 Gcp クラス

Gcp クラスはダイスのオモテ面がグー，前面がチョキの状態を表すクラスである．図 14 図 13 に Gcp クラスのクラス図を示す.

Gcp		# ダイスのオモテ面がグー，前面がチョキの状態を表すクラス
Gcp(Dice dice)		# コンストラクタ.
showState()	: void	# ダイスの状態を表示するメソッド.
moveUp()	: void	# ダイスの状態遷移を行うメソッド.
moveDown()	: void	# ダイスの状態遷移を行うメソッド.
moveRight()	: void	# ダイスの状態遷移を行うメソッド.
moveLeft()	: void	# ダイスの状態遷移を行うメソッド.
attackDice(Dice dice)	: boolean	# ダイスの攻撃判定を行うメソッド.
getTopString()	: String	# ダイスのオモテ面のゲッター
getName()	: String	# ダイスの状態名のゲッター

図 14 Gcp クラスのクラス図

- Gcp(Dice dice)
  コンストラクタ.
- showState(): void
  ダイスの状態を表示するメソッド.
- moveUp(): void
  ダイスの状態遷移を行うメソッド.
- moveDown(): void
  ダイスの状態遷移を行うメソッド.
- moveRight(): void
  ダイスの状態遷移を行うメソッド.
- moveLeft(): void
  ダイスの状態遷移を行うメソッド.
- attackDice(Dice dice): boolean
  ダイスの攻撃判定を行うメソッド．引数のダイスに対して攻撃できる場合に true を返す
- getTopString(): String
  ダイスのオモテ面のゲッター
- getName(): String
  ダイスの状態名のゲッター

4.7 Gpc クラス

Gpc クラスはダイスのオモテ面がグー，前面がパーの状態を表すクラスである．図 15 に Gpc クラスのクラス図を示す.

Gpc		# ダイスのオモテ面がグー，前面がパーの状態を表すクラス
Gpc(Dice dice)		# コンストラクタ.
showState()	: void	# ダイスの状態を表示するメソッド.
moveUp()	: void	# ダイスの状態遷移を行うメソッド.
moveDown()	: void	# ダイスの状態遷移を行うメソッド.
moveRight()	: void	# ダイスの状態遷移を行うメソッド.
moveLeft()	: void	# ダイスの状態遷移を行うメソッド.
attackDice(Dice dice)	: boolean	# ダイスの攻撃判定を行うメソッド.
getTopString()	: String	# ダイスのオモテ面のゲッター
getName()	: String	# ダイスの状態名のゲッター

図 15 Gpc クラスのクラス図

- Gpc(Dice dice)
  コンストラクタ.
- showState(): void
  ダイスの状態を表示するメソッド.
- moveUp(): void
  ダイスの状態遷移を行うメソッド.
- moveDown(): void
  ダイスの状態遷移を行うメソッド.

- moveRight(): void
  ダイスの状態遷移を行うメソッド.
- moveLeft(): void
  ダイスの状態遷移を行うメソッド.
- attackDice(Dice dice): boolean
  ダイスの攻撃判定を行うメソッド．引数のダイスに対して攻撃できる場合に true を返す
- getTopString(): String
  ダイスのオモテ面のゲッター
- getName(): String
  ダイスの状態名のゲッター

4.8 JankenShogi クラス

JankenShogi クラスはジャンケン将棋の対戦を行うクラスである．図 16 に JankenShogi クラスのクラス図を示す.

JankenShogi		# ジャンケン将棋の対戦を行うクラス.
JankenShogi()		# コンストラクタ.
makeBoard(int n)	:void	# 盤面の生成を行うメソッド.
makeBoard()	: void	# 盤面の生成を行うメソッド.
setMakeBoard()	: void	# 盤面の生成を行うメソッド.
selectDice()	: int	# 動かす駒をプレイヤーに選択させるメソッド
selectDirection ()	: String	# 駒を動かす方向をプレイヤーに選択させるメソッド
hand ()	: Board	# プレイヤーの手番を行うメソッド.
comPlay ()	: Board	# コンピュータ側の手番を行うメソッド.
judge()	: boolean	# 勝利判定を行うメソッド.
set(int diceType, int x, int y	: void	# 駒を動かすメソッド.
main(String[] args)	: void	# メインメソッド

図 16 JankenShogi クラスのクラス図

- JankenShogi()
  コンストラクタ.
- makeBoard(int n): void
  盤面の生成を行うメソッド．引数の数の駒を用意する.
- makeBoard(): void
  盤面の生成を行うメソッド.
- setMakeBoard(): void
  盤面の生成を行うメソッド．入力に応じて駒の状態を決め，セットする.
- selectDice(): int
  動かす駒をプレイヤーに選択させるメソッド
- selectDirection(): String
  駒を動かす方向をプレイヤーに選択させるメソッド
- hand(): Board
  プレイヤーの手番を行うメソッド.

- comPlay(): Board
  コンピュータ側の手番を行うメソッド.
- judge(): boolean
  勝利判定を行うメソッド.
- set(int diceType,　int x,　int y):void
  駒を動かすメソッド.
- main(String[] args): void
  メインメソッド.

4.9 Pcg クラス

Pcg クラスはダイスのオモテ面がパー，前面がチョキの状態を表すクラスである．図 17 に Pcg クラスのクラス図を示す.

Pcg		# ダイスのオモテ面がパー，前面がチョキの状態を表すクラス
Pcg(Dice dice)		# コンストラクタ.
showState()	: void	# ダイスの状態を表示するメソッド.
moveUp()	: void	# ダイスの状態遷移を行うメソッド.
moveDown()	: void	# ダイスの状態遷移を行うメソッド.
moveRight()	: void	# ダイスの状態遷移を行うメソッド.
moveLeft()	: void	# ダイスの状態遷移を行うメソッド.
attackDice(Dice dice)	: boolean	# ダイスの攻撃判定を行うメソッド.
getTopString()	: String	# ダイスのオモテ面のゲッター
getName()	: String	# ダイスの状態名のゲッター

図 17 Pcg クラスのクラス図

- Pcg(Dice dice)
  コンストラクタ.
- showState(): void
  ダイスの状態を表示するメソッド.
- moveUp(): void
  ダイスの状態遷移を行うメソッド.
- moveDown(): void
  ダイスの状態遷移を行うメソッド.
- moveRight(): void
  ダイスの状態遷移を行うメソッド.
- moveLeft(): void
  ダイスの状態遷移を行うメソッド.
- attackDice(Dice dice): boolean
  ダイスの攻撃判定を行うメソッド．引数のダイスに対して攻撃できる場合に true を返す
- getTopString(): String
  ダイスのオモテ面のゲッター
- getName(): String
  ダイスの状態名のゲッター

4.10　Pgc クラス

Pgc クラスはダイスのオモテ面がパー，前面がグーの状態を表すクラスである．図 18 に Pgc クラスのクラス図を示す．

Pgc		# ダイスのオモテ面がパー，前面がグーの状態を表すクラス．
Pgc(Dice dice)		# コンストラクタ．
showState()	: void	# ダイスの状態を表示するメソッド．
moveUp()	: void	# ダイスの状態遷移を行うメソッド．
moveDown()	: void	# ダイスの状態遷移を行うメソッド．
moveRight()	: void	# ダイスの状態遷移を行うメソッド．
moveLeft()	: void	# ダイスの状態遷移を行うメソッド．
attackDice(Dice dice)	: boolean	# ダイスの攻撃判定を行うメソッド．
getTopString()	: String	# ダイスのオモテ面のゲッター
getName()	: String	# ダイスの状態名のゲッター

図 18 Pgc クラスのクラス図

- Pgc(Dice dice)
  コンストラクタ．
- showState(): void
  ダイスの状態を表示するメソッド．
- moveUp(): void
  ダイスの状態遷移を行うメソッド．
- moveDown(): void
  ダイスの状態遷移を行うメソッド．
- moveRight(): void
  ダイスの状態遷移を行うメソッド．
- moveLeft(): void
  ダイスの状態遷移を行うメソッド．
- attackDice(Dice dice): boolean
  ダイスの攻撃判定を行うメソッド．引数のダイスに対して攻撃できる場合に true を返す
- getTopString(): String
  ダイスのオモテ面のゲッター
- getName(): String
  ダイスの状態名のゲッター

5　評価値の検証

本章では本研究で作成したプログラムが着手選択ための評価値の計算方法およびその評価値の妥当性の検証について述べる．

5.1 評価方法の優先度

3.2 節で述べたように本研究で作成したプログラムは局面の評価値を求めて着手選択を行う．評価値の計算には複数の要素を用いているが，どの要素をより優先すべきか不明である．そこで以下では，その優先順位を検証していく．それぞれの要素の優先順位を係数を入れ替えて評価値を計算する複数のプログラム同士を対戦させたところ次のことがわかった．

- 駒を動かせる手数を他の二つよりも優先すると，壁側に駒を進めなくなり，ゴールに近づいても到達しなくなる．
- ゴールまでの距離と駒の数に関しては，優先度を入れ替えてもどちらも正常にゲームを完了できた．

以上のことから評価値を求める際の各要素の優先度は「ゴールまでの距離，駒の数＞駒を動かせる手数」であることがわかった．

残り 2 つの要素については次節で検証する．

5.2 検証

本研究ではより強いゲーム AI の作成を目指すためジャンケン将棋のゲーム性を理解する必要がある．そのため先手後手のどちらが有利か，2 つある勝利条件のどちらがより勝利に関わるかを検証していく．また，駒の数によってそれらがどう変わるかも検証していく．

ジャンケン将棋のゴールへの到達を優先した評価関数と，相手の駒を取ることを優先した評価関数を用意する．この 2 つの評価関数を持つ最大深度 4 のゲーム AI を先手後手入れ替え，駒の数を 1~4 個の場合でそれぞれ 50 回ずつ対戦させた時の結果を表 2 と 表 3 に示す．

ここで勝率pの勝負をN回行った時の標準偏差sは

$$s = \sqrt{N * p * (1 - p)}$$

となる．$p = 0．5$と仮定すると$N = 100$の場合

$$s = \sqrt{100 * 0．5 * 0．5} = 5$$

となる．統計学において 95%信頼区間となるのは平均からの差が$s * 1．96$となる区間である．これを求めると 40~60 勝となる．同様に 200 回，400 回勝負を行なった場合の 95%信頼区間を求めると，それぞれ，86~114 勝，180~220 勝となる．これを踏まえて表 2 の合計勝利数を見ると，駒の数が 4 個の時に信頼区間を超えた勝利数となっているため，先手後手関わらずゴール優先 AI が統計上有利であったことがわかる．また，表 3 を見ると，駒の数が 1 個，2 個の時に信頼区間を超えた勝利数となっているため，先手が有利であることがわかる．また合計勝利数から，駒の数が 1~4 個の場合だと先手が有利になるゲームだとわかる．

表 2 ゴール優先 AI の攻撃優勢 AI に対する勝利数（各 50 戦）

駒の数	1 個	2 個	3 個	4 個
先手	50	36	30	34
後手	0	22	28	32
合計勝利数	50	58	58	66

表 3 先手側の勝利数（各 100 戦）

駒の数	1 個	2 個	3 個	4 個	合計勝利数
勝利数	100	64	52	52	268

6　結論・今後の課題

本研究では Java を用いてジャンケン将棋のアプリケーションを作成し，ゲーム AI を実装したことで対戦できるようになった．また，ジャンケン将棋は駒の数が少ないと先手が有利で，駒の数が増えるとゴールによる勝利の方が重要になるというゲーム性がわかった．

今後の課題としては実際に UI を GUI での実装や，ゲーム AI をより強くするために機械学習による評価関数の最適化などが考えられる．

7　謝辞

本研究を作成するにあたり，担当教員の石水隆講師には大変お世話になりました．ここに感謝の意を表します．

参考文献

[1] 松田道弘：世界のゲーム辞典, 東京堂出版（1989).
[2] あたまのよくなるゲーム　じゃんけんしょうぎ, 学研教育出版, (2010)
[3] https://hon.gakken.jp/book/1575033700
[4] 今村栞菜：ジャンケン将棋アプリの開発，近畿大学理工学部情報学科 2017 年度卒業報告（2018）
[5] G.Gear.inc： 対戦！じゃんけん将棋，Google Play (2022)
[6] https://play.google.com/store/apps/details?id=air.jp.globalgear.jan&hl=ja&gl=US
[7] 上原淳一 : 小諸じゃんけんしょうぎ, ボドゲ―マ, https://bodoge.hoobby.net/games/komoro-janken-shogi
[8] Janos Wagner and Istvan Virag，Solving renju：ICGA Journal，Vol.24， No.1， pp.30-35 (2001)
[9] http://www.sze.hu/~gtakacs/download/wagnervirag_2001.pdf
[10] Jonathan Schaeffer，Neil Burch， Yngvi Bjorsson， Akihiro Kishimoto， Martin Muller， Robert Lake， Paul Lu, and Steve Suphen，Checkers is solved： Science Vol.317， No,5844， pp.1518-1522 (2007) http://www.sciencemag.org/content/317/5844/1518.full.pdf
[11] Jonathan Schaeffer, Neil Burch，Yngvi Bjorsson，Akihiro Kishimoto，Martin Muller，Robert Lake，Paul Lu，and Steve Suphen，Checkers is solved： Science Vol.317，No,5844，pp.1518-1522 (2007) http://www.sciencemag.org/content/317/5844/1518.full.pdf
[12] 清慎一, 川嶋俊, 探索プログラムによる四路盤囲碁の解, 研究報告ゲーム情報学(GI), Vol. 2000-GI-004, pp.69-76, 情報処理学会, (2000), http://id.nii.ac.jp/1001/00058633/
[13] Eric C.D. van der Welf, H.Jaap van den Herik, and Jos W.H.M.Uiterwijk：Solving Go on Small Boards, ICGA Journal, Vol.26, No.2, pp.92-107 (2003).
[14] https://www.researchgate.net/publication/2925531_Solving_Go_On_Small_Boards/link/0fcfd511dfb651482c000000/download
[15] 田中哲郎 :「どうぶつしょうぎ」の完全解析，情報処理学会研究報告， Vol.2009-GI-22 No.3, pp.1-8 (2009) http://id.nii.ac.jp/1001/00062415/
[16] 塩田好，石水隆，山本博史 :「アンパンマンはじめてしょうぎ」の完全解析，2013 年度 情報処理 学会関西支部 支部大会 講演論文集，(2013) http://id.nii.ac.jp/1001/00096792/
[17] Joel Feinstein：Amenor Wins World 6x6 Championships!, Forty billion noted under the tree (July 1993), pp.6-8, British Othello Federation's newsletter, (1993)
[18] オセロゲーム開発 ～アルファベータ法（alpha-beta search）～， https://uguisu.skr.jp/othello/alpha-beta.html

付録A ソースプログラム

本研究で作成したプログラムのソースファイルを以下に示す.

- Board クラス

```
import java.util.ArrayList;
import java.util.Random;

public class Board {
    int board[][];
    final int size=6;
    // ダイスの位置情報[y,x]

    ArrayList<Dice> list;
    int num; // 駒の数
    int turn=1;
    int turnNum=0;
    int depthMax=3;
    int depthValue=1; // 探索深度による評価への影響度
    int boardValue;
    int comType;
    private static Random random = new Random();

    // コンストラクタ
    public Board(){
        list=new ArrayList<>();
        list.add(new Dice(3,4));
        list.add(new Dice(4,3));
        num=2;

        // 盤面作成
        board=new int[size+2][size+2];

        for(int i=0;i<size+2;i++){
            for(int j=0;j<size+2;j++){
                board[i][j]=0;
                for(int n=0;n<list.size();n++){
                    if(i==0||i==size+1||j==0||j==size+1){
                    board[i][j]=-1;
                    }else if(i==list.get(n).y&&j==list.get(n).x){
                        board[i][j]=n+1;
                    }
                }
            }
        }
    }
```

```
    // 引数ありのコンストラクタ（駒の数を指定した場合）
    public Board(ArrayList<Dice> list){
        this.list=new ArrayList<>();
        for(Dice dice:list){
            this.list.add(dice);
        }
        num=list.size();

        // 盤面作成
        board=new int[size+2][size+2];

        for(int i=0;i<size+2;i++){
            for(int j=0;j<size+2;j++){
                board[i][j]=0;
                for(int n=0;n<list.size();n++){
                    if(i==0||i==size+1||j==0||j==size+1){
                    board[i][j]=-1;
                    }else if(i==list.get(n).y&&j==list.get(n).x){
                        board[i][j]=n+1;
                    }
                }
            }
        }

    }

    // 今の盤面を複製する
    public Board(Board preBoard){
        list=new ArrayList<>();
        board=new int[preBoard.size+2][preBoard.size+2];
        turn=preBoard.turn;
        turnNum=preBoard.turnNum;
        num=preBoard.num;
        comType=preBoard.comType;

        for(int i=0;i<size+2;i++){
            for(int j=0;j<size+2;j++){
                board[i][j]=preBoard.board[i][j];
            }
        }

        for(Dice d:preBoard.list){
            Dice dice=new Dice(d.x, d.y, d.getState());
            list.add(dice);
        }
    }

    // board の表示(壁あり)
```

```
    public void showBoard2(){
        for(int i=0;i<size+2;i++){
            System.out.print("|");
            for(int j=0;j<size+2;j++){
                if(board[i][j]==-1){
                    System.out.print("*");
                }else if(board[i][j]==0){
                    System.out.print(" ");
                }else{
                    System.out.print(board[i][j]);
                }
                System.out.print("|");
            }
            System.out.println();
            for(int k=0;k<(size+2)*2;k++){
                System.out.print("-");
            }
            System.out.println();
        }
    }

    // boardの表示(壁なし)
    public void showBoard(){
        System.out.println(turnNum+"ターン目");
        for(int i=1;i<size+1;i++){
            System.out.print("|");
            for(int j=1;j<size+1;j++){
                if(board[i][j]!=-1){
                    if(board[i][j]==0){
                        System.out.print(" ");
                    }else{
                        System.out.print(board[i][j]);
                    }
                    System.out.print("|");
                }
            }
            System.out.println();
            for(int k=0;k<(size)*2;k++){
                System.out.print("-");
            }
            System.out.println();
        }

        for(int i=0;i<num;i++){
            System.out.println((i+1)+"の状態");
            list.get(i).showState();
        }
```

```
        //System.out.println("この盤面の評価"+this.value());
        //System.out.println("この盤面から予測される評価"+boardValue);
    }

        // board の表示(壁なし,ダイスの状態なし)
        public void showBoard0(){
            System.out.println(turnNum+"ターン目");
            for(int i=1;i<size+1;i++){
                System.out.print("|");
                for(int j=1;j<size+1;j++){
                    if(board[i][j]!=-1){
                        if(board[i][j]==0){
                            System.out.print(" ");
                        }else{
                            System.out.print(board[i][j]);
                        }
                        System.out.print("|");
                    }
                }
                System.out.println();
                for(int k=0;k<(size)*2;k++){
                    System.out.print("-");
                }
                System.out.println();
            }
        }

    // 動く方向 d,扱う駒のタイプ type(リスト番号)
    public boolean move(String d,int type){
        boolean flag=true;
        // 移動後の座標[x,y]
        int x=0,y=0;

        x=list.get(type).x;
        y=list.get(type).y;

        // 決着した盤面では動かせない（nextMove が動かないように
        if(this.judge()){
            return false;
        }
        switch(d){
            case"u":
            y=y-1;
            break;
            case"d":
            y=y+1;
            break;
            case"l":
```

```
            x=x-1;
            break;
            case"r":
            x=x+1;
            break;
            default:
            System.out.println("移動キーを選択してください");
            break;
        }
        // 移動先に駒があれば board 内の attack()を呼び出す.
        if(board[y][x]==0){
            board[list.get(type).y][list.get(type).x]=0;
            board[y][x]=type+1;
            turn*=-1;
            turnNum+=1;

            switch(d){
                case"u":
                list.get(type).moveUp();
                break;
                case"d":
                list.get(type).moveDown();
                break;
                case"l":
                list.get(type).moveLeft();
                break;
                case"r":
                list.get(type).moveRight();
                break;
                default:
                System.out.println("移動キーを選択してください");
                break;
            }
        }
        else if(board[y][x]==-1){
            //System.out.println("移動先が空きマスではありません");
            flag=false;
        }else{
            // 移動する駒と移動先の駒が敵同士
            if(isfirst(list.get(type))!=isfirst(list.get(board[y][x]-1))){
                if(this.attack(list.get(type), list.get(board[y][x]-1))==false){
                    //System.out.println("移動先の駒に攻撃できません");
                    flag=false;
                }else{
                    turn*=-1;
                    turnNum+=1;
                }
            }else{
```

```
            flag=false;
            }
        }
        return flag;
    }

    // attack メソッド  dice1 が dice2 を攻撃する
    public boolean attack(Dice dice1,Dice dice2){
        if(dice1.attackDice(dice2)){
            board[dice1.y][dice1.x]=0;
            dice1.x=dice2.x;
            dice1.y=dice2.y;
            board[dice2.y][dice2.x]=list.indexOf(dice1)+1;
            dice2.x=0;
            dice2.y=0;
        }
        return dice1.attackDice(dice2);
    }


    // 駒を動かした時のボードを返す
    public Board nextMove(String d,int type){
        Board nextBoard=new Board(this);
        if(nextBoard.move(d, type)){
            return nextBoard;
        }
        return null;
    }


    // 現在のボードの評価値を返す.
    public int value(){
        int value=0;
        int count1=0; // 先手の駒の数
        int count2=0; // 後手の駒の数
        int moveValue=10; // 駒の動ける数の評価の重さ
        int goalValue=10000; // ゴールからの距離の評価の重さ
        int diceValue=100; //コマの数の評価の重さ


        if(comType==0){
            goalValue=10000;
            diceValue=100;
        }else if(comType==1){
            goalValue=100;
            diceValue=10000;
        }
```

```
        for(int i=1;i<size+1;i++){
            for(int j=1;j<size+1;j++){
                if(0<board[i][j]&&board[i][j]<=num/2){
                    count1+=1;
                }else if(num/2<board[i][j]&&board[i][j]<=num){
                    count2+=1;
                }
            }
        }

        // 駒を動かせる数
        if(turn==1){
            value+=this.nextMoves().size()*moveValue;
            turn*=-1;
            value-=this.nextMoves().size()*moveValue;
            turn*=-1;
        }else if(turn==-1){
            value-=this.nextMoves().size()*moveValue;
            turn*=-1;
            value+=this.nextMoves().size()*moveValue;
            turn*=-1;
        }

        // 駒の数に応じて評価値を増減する
        switch(count1){
            case 0:
            value+=0;
            break;
            case 1:
            value+=10*diceValue;
            break;
            case 2:
            value+=15*diceValue;
            break;
            case 3:
            value+=18*diceValue;
            break;
            case 4:
            value+=20*diceValue;
            break;
        }
        switch(count2){
            case 0:
            value-=0;
            break;
            case 1:
            value-=10*diceValue;
            break;
```

```
            case 2:
            value-=15*diceValue;
            break;
            case 3:
            value-=18*diceValue;
            break;
            case 4:
            value-=20*diceValue;
            break;
        }


        // ゴールからの距離
        for(int k=0;k<num;k++){
            int distance=0;
            // 駒が取られていない時　(0,0)じゃない時
            if(list.get(k).x!=0&&list.get(k).y!=0){
                if(k<num/2){
                    distance+=(size-list.get(k).x);
                    distance+=(list.get(k).y-1);
                    value+=((size-1)*2-distance)*goalValue;
                    if(list.get(k).x==size&&list.get(k).y==1){
                        value+=goalValue*50;
                    }
                }else{
                    distance+=(list.get(k).x-1);
                    distance+=(size-list.get(k).y);
                    value-=((size-1)*2-distance)*goalValue;
                    if(list.get(k).x==1&&list.get(k).y==size){
                        value-=goalValue*50;
                    }
                }
            }
        }

        return value;
    }

    // 評価値探索の再帰部分
    public int search_main(int depth,int depthMax,int alpha,int beta){
        int e=0;
        int dValue=0;
        if(depth==depthMax){
            // 評価値を返す
            return this.value();
        }
        else{
            if(this.judge()){
```

```
            if(turn==1){
                return this.value();
            }else{
                return this.value();
            }
        }
        if(depth==0){
            if(turn==1){
                alpha=this.value();
            }else if(turn==-1){
                beta=this.value();
            }
        }

        if(turn==1){
            dValue=depthValue*-1;
        }
        ArrayList<Board> newMoves=this.nextMoves();
        for(Board b:newMoves){
            e=b.search_main(depth+1, depthMax, alpha, beta)+(depth+1)*dValue;

            if(turn==1){
                // 先手なら
                if(e>=beta){
                    return e;
                }else if(e>alpha){
                    alpha=e;
                }
            }else if(turn==-1){
                // 後手なら
                if(e<=alpha){
                    return e;
                }else if(e<beta){
                    beta=e;
                }
            }
        }
    }

    if(turn==1){
        return alpha;
    }else if(turn==-1){
        return beta;
    }
    return 0;
}

// 探索の開始（深度の指定なし）
```

```
    public int search(){
        int depth=0;
        int alpha=Integer.MIN_VALUE;
        int beta=Integer.MAX_VALUE;
        return search_main(depth,depthMax,alpha,beta);
    }

    // 探索の開始（深度の指定あり）
    public int search(int dMax){
        int depth=0;
        int alpha=Integer.MIN_VALUE;
        int beta=Integer.MAX_VALUE;
        return search_main(depth,dMax,alpha,beta);
    }

    // 勝利判定
    public boolean judge(){
        boolean isWin=false;
        int count=0; // 自分が倒した駒の数
        int goal=0; // ゴールにたどり着いた駒の数
        if(turn==1){
            for(int i=num/2;i<num;i++){ //自分の番に相手の駒を全て取ったか
                if(list.get(i).x==0&&list.get(i).y==0)count=count+1;
            }
            for(int i=0;i<num/2;i++){ //自分の番に相手ゴールに駒が届いたか
                if(list.get(i).x==size&&list.get(i).y==1)goal=goal+1;
            }
        }else{
            for(int i=0;i<num/2;i++){ //相手の番に自分の駒を全て取ったか
                if(list.get(i).x==0&&list.get(i).y==0)count=count+1;
            }
            for(int i=num/2;i<num;i++){ //相手の番に自分ゴールに駒が届いたか
                if(list.get(i).x==1&&list.get(i).y==size)goal=goal+1;
            }
        }
        if(count==num)isWin=true;
        if(goal>0)isWin=true;
        return isWin;
    }

    // 次の手を進めた盤面の集合を返す
    public ArrayList<Board> nextMoves(){
        ArrayList<Board> newBoards=new ArrayList<>();

        if(turn==1){
            for(int i=0;i<num/2;i++){
                if(list.get(i).x!=0&&list.get(i).y!=0){
                    newBoards.add(this.nextMove("u", i));
```

```
                newBoards.add(this.nextMove("r", i));
                newBoards.add(this.nextMove("d", i));
                newBoards.add(this.nextMove("l", i));
            }
        }
    }else{
        for(int i=num/2;i<num;i++){
            if(list.get(i).x!=0&&list.get(i).y!=0){
                newBoards.add(this.nextMove("d", i));
                newBoards.add(this.nextMove("l", i));
                newBoards.add(this.nextMove("u", i));
                newBoards.add(this.nextMove("r", i));
            }
        }
    }

    for(int i=0;i<newBoards.size();i++){
        if(newBoards.get(i)==null){
            newBoards.remove(i);
            i--;
        }
    }
    return newBoards;
}

// ダイスが先手駒か後手駒かを返す
public int isfirst(Dice dice){
    int result=0;
    if(0<board[dice.y][dice.x]&&board[dice.y][dice.x]<=num/2){
        result=1;
    }else if(num/2<board[dice.y][dice.x]&&board[dice.y][dice.x]<=num){
        result=-1;
    }
    return result;
}

// 最適の盤面を返す
public Board bestMove(){
    ArrayList<Board> newBoards=new ArrayList<>();
    newBoards=this.nextMoves();
    ArrayList<Board> bestBoards=new ArrayList<>();
    //boardValue=this.search();
    //bestBoards.add(this);


    if(turn==1){
        boardValue=Integer.MIN_VALUE;
        for(Board b :newBoards){
```

```
                int v=b.search();
                if(v>boardValue){
                    boardValue=v;
                    b.boardValue=boardValue;
                    bestBoards.clear();
                    bestBoards.add(b);
                }else if(v==boardValue){
                    b.boardValue=boardValue;
                    bestBoards.add(b);
                }
            }
        }else if(turn==-1){
            boardValue=Integer.MAX_VALUE;
            for(Board b:newBoards){
                int v=b.search();
                if(v<boardValue){
                    boardValue=v;
                    b.boardValue=boardValue;
                    bestBoards.clear();
                    bestBoards.add(b);
                }else if(v==boardValue){
                    b.boardValue=boardValue;
                    bestBoards.add(b);
                }
            }
        }

        int r = random.nextInt(bestBoards.size());
        for(Board b:bestBoards){
            //b.showBoard();
        }
        return bestBoards.get(r);
    }
}
```

- Cgp クラス

```
public class Cgp implements DiceState{

    Dice dice;
    String top ="ち";
    String name="cgp";
    public Cgp(Dice dice){
        this.dice=dice;
    }
```

```
    // ダイスの状態（表面が何かとか）を表示
    public void showState(){
        System.out.println("  ぐ  ");
        System.out.println("ぱちぱ");
        System.out.println("  ぐ  ");
    }
    // ダイスの状態遷移
    public void moveUp(){
        dice.setState(dice.gcp);
    }
    public void moveDown(){
        dice.setState(dice.gcp);
    }
    public void moveRight(){
        dice.setState(dice.pgc);
    }
    public void moveLeft(){
        dice.setState(dice.pgc);
    }
    // ダイスの攻撃判定
    public boolean attackDice(Dice dice){
        boolean isAttack=false;
        if(dice.state.getTopString()=="ぱ"){
            isAttack=true;
        }
        return isAttack;
    }

    // ダイスのオモテ面を返すメソッド
    public  String getTopString(){
        return top;
    }

    // ダイスの状態名を返すメソッド
    public  String getName(){
        return name;
    }
}
```

- Cpg クラス

```
public class Cpg implements DiceState{
```

```
    Dice dice;
    String top="ち";
    String name="cpg";
    public Cpg(Dice dice){
        this.dice=dice;
    }

    // ダイスの状態（表面が何かとか）を表示
    public void showState(){
        System.out.println("  ぱ  ");
        System.out.println("ぐちぐ");
        System.out.println("  ぱ  ");
    }
    // ダイスの状態遷移
    public void moveUp(){
        dice.setState(dice.pcg);
    }
    public void moveDown(){
        dice.setState(dice.pcg);
    }
    public void moveRight(){
        dice.setState(dice.gpc);
    }
    public void moveLeft(){
        dice.setState(dice.gpc);
    }
    // ダイスの攻撃判定
    public boolean attackDice(Dice dice){
        boolean isAttack=false;
        if(dice.state.getTopString()=="ぱ"){
            isAttack=true;
        }
        return isAttack;
    }

    // ダイスのオモテ面を返すメソッド
    public  String getTopString(){
        return top;
    }

    // ダイスの状態名を返すメソッド
    public  String getName(){
        return name;
    }
}
```

- Dice クラス

```
public class Dice implements Cloneable{
    DiceState gcp;
    DiceState gpc;
    DiceState cpg;
    DiceState cgp;
    DiceState pcg;
    DiceState pgc;

    DiceState state;


    int x,y=0;

    // コンストラクタ
    public Dice(int x,int y){
        gcp = new Gcp(this);
        gpc = new Gpc(this);
        cpg = new Cpg(this);
        cgp = new Cgp(this);
        pcg = new Pcg(this);
        pgc = new Pgc(this);

        state=gcp;
        this.x=x;
        this.y=y;
    }

    // ダイス複製用のコンストラクタ
    public Dice(int x,int y,DiceState state){
        gcp = new Gcp(this);
        gpc = new Gpc(this);
        cpg = new Cpg(this);
        cgp = new Cgp(this);
        pcg = new Pcg(this);
        pgc = new Pgc(this);

        switch(state.getName()){
            case "gcp":
            this.state=gcp;
            break;
            case "gpc":
            this.state=gpc;
            break;
            case "cpg":
            this.state=cpg;
            break;
```

```
            case "cgp":
            this.state=cgp;
            break;
            case "pcg":
            this.state=pcg;
            break;
            case "pgc":
            this.state=pgc;
            break;

        }
        this.x=x;
        this.y=y;
    }

    //  状態のセッター
    public void setState(DiceState state){
        this.state=state;
    }

    // 状態のゲッター
    public DiceState getState(){
        return this.state;
    }


    // ダイスの状態（表面が何かとか）を表示
    public void showState(){
        state.showState();
    }

    // ダイスの状態遷移
    public void moveUp(){
        state.moveUp();
        y--;
        //System.out.println("ダイスの状態("+this.state+")");
    }
    public void moveDown(){
        state.moveDown();
        y++;
    }
    public void moveRight(){
        state.moveRight();
        x++;
    }
    public void moveLeft(){
        state.moveLeft();
        x--;
```

```
    }

    // ダイスの攻撃判定
    public boolean attackDice(Dice dice){
        return state.attackDice(dice);
    }

    // ダイスの複製
    public Object clone(){
        try{
            Dice dice =(Dice)super.clone();
            dice.state=this.state;
            return dice;
        }catch(CloneNotSupportedException ex){return null;}
    }
}
```

- DiceState インターフェース

```
public interface DiceState {
    // ダイスの状態（表面が何かとか）を表示
    public abstract void showState();
    // ダイスの状態遷移
    public abstract void moveUp();
    public abstract void moveDown();
    public abstract void moveRight();
    public abstract void moveLeft();
    // ダイスの攻撃判定
    public abstract boolean attackDice(Dice dice);
    // ダイスのオモテ面を返すメソッド
    public abstract String getTopString();
    // ダイスの状態名を返すメソッド
    public abstract String getName();
}
```

- Gcp クラス

```
 public class Gcp implements DiceState{

    Dice dice;
    String top="ぐ";
    String name="gcp";
    public Gcp(Dice dice){
        this.dice=dice;
    }
```

```
    // ダイスの状態（表面が何かとか）を表示
    public void showState(){
        System.out.println("  ち  ");
        System.out.println("ぱぐぱ");
        System.out.println("  ち  ");
    }
    // ダイスの状態遷移
    public void moveUp(){
        dice.setState(dice.cgp);
    }
    public void moveDown(){
        dice.setState(dice.cgp);
    }
    public void moveRight(){
        dice.setState(dice.pcg);
    }
    public void moveLeft(){
        dice.setState(dice.pcg);
    }
    // ダイスの攻撃判定
    public boolean attackDice(Dice dice){
        boolean isAttack=false;
        if(dice.state.getTopString()=="ち"){
            isAttack=true;
        }
        return isAttack;
    }

    // ダイスのオモテ面を返すメソッド
    public  String getTopString(){
        return top;
    }

    // ダイスの状態名を返すメソッド
    public  String getName(){
        return name;
    }
}
```

- Gpc クラス

```
public class Gpc implements DiceState{

    Dice dice;
    String top="ぐ";
    String name="gpc";
    public Gpc(Dice dice){
```

```
        this.dice=dice;
    }

    // ダイスの状態（表面が何かとか）を表示
    public void showState(){
        System.out.println("  ぱ  ");
        System.out.println("ちぐち");
        System.out.println("  ぱ  ");
    }
    // ダイスの状態遷移
    public void moveUp(){
        dice.setState(dice.pgc);
    }
    public void moveDown(){
        dice.setState(dice.pgc);
    }
    public void moveRight(){
        dice.setState(dice.cpg);
    }
    public void moveLeft(){
        dice.setState(dice.cpg);
    }
    // ダイスの攻撃判定
    public boolean attackDice(Dice dice){
        boolean isAttack=false;
        if(dice.state.getTopString()=="ち"){
            isAttack=true;
        }
        return isAttack;
    }

    // ダイスのオモテ面を返すメソッド
    public  String getTopString(){
        return top;
    }

    // ダイスの状態名を返すメソッド
    public  String getName(){
        return name;
    }
}
```

- JankenShogi クラス

```
 import java.util.Scanner;
import java.util.ArrayList;

```

```java
public class JankenShogi {
    int turn = 1; // 手番（1が先手-1が後手）
    Board board;  // ゲーム盤
    int num; // 駒の数
    int com=0; // 手番をAIが操作するか
    Scanner keyBoardScanner = new Scanner(System.in);

    // コンストラクタ
    public JankenShogi(){
        //this.makeBoard();
        this.setMakeBoard();
    }



    // 駒の数を指定した盤面作成
    private void makeBoard(int n){
        ArrayList<Dice> list=new ArrayList<>();

        lavel:while (true) { // 適切な方向が選択されるまでループ
            switch (n){
                case 1:
                list.add(new Dice(1,6));
                list.add(new Dice(6,1));
                board=new Board(list);
                num=1;
                break lavel;
                case 2:
                list.add(new Dice(1, 6));
                list.add(new Dice(2, 5));
                list.add(new Dice(5, 2));
                list.add(new Dice(6, 1));
                num=2;
                board=new Board(list);
                break lavel;
                case 3:
                list.add(new Dice(1, 6));
                list.add(new Dice(1, 4));
                list.add(new Dice(3, 6));
                list.add(new Dice(4, 1));
                list.add(new Dice(6, 3));
                list.add(new Dice(6, 1));
                board=new Board(list);
                num=3;
                break lavel;
                case 4:
                list.add(new Dice(1, 6));
                list.add(new Dice(1, 4));
```

```
            list.add(new Dice(2, 5));
            list.add(new Dice(3, 6));
            list.add(new Dice(4, 1));
            list.add(new Dice(5, 2));
            list.add(new Dice(6, 3));
            list.add(new Dice(6, 1));
            board=new Board(list);
            num=4;
            break lavel;
            default:
            System.out.println("1~4 の数字の中で選択してください：");
            continue;
        }
    }
}

// 駒の数を聞いて盤面作成
private void makeBoard(){
    ArrayList<Dice> list=new ArrayList<>();
    lavel:while (true) { // 適切な方向が選択されるまでループ
    System.out.print ("駒の個数を 1~4 個のうち選択してください：");
    int n=keyBoardScanner.nextInt();
        switch (n){
            case 1:
            list.add(new Dice(3,4));
            list.add(new Dice(4,3));
            board=new Board(list);
            num=1;
            break lavel;
            case 2:
            list.add(new Dice(1, 6));
            list.add(new Dice(2, 5));
            list.add(new Dice(5, 2));
            list.add(new Dice(6, 1));
            num=2;
            board=new Board(list);
            break lavel;
            case 3:
            list.add(new Dice(1, 6));
            list.add(new Dice(1, 4));
            list.add(new Dice(3, 6));
            list.add(new Dice(4, 1));
            list.add(new Dice(6, 3));
            list.add(new Dice(6, 1));
            board=new Board(list);
            num=3;
            break lavel;
            case 4:
```

```
                list.add(new Dice(1, 6));
                list.add(new Dice(1, 4));
                list.add(new Dice(2, 5));
                list.add(new Dice(3, 6));
                list.add(new Dice(4, 1));
                list.add(new Dice(5, 2));
                list.add(new Dice(6, 3));
                list.add(new Dice(6, 1));
                board=new Board(list);
                num=4;
                break lavel;
                default:
                System.out.println("1~4 の数字の中で選択してください：");
                continue;
            }
        }
    }


    // 入力に応じて状態を指定するタイプの盤面作成
    private void setMakeBoard(){
        ArrayList<Dice> list=new ArrayList<>();
        ArrayList<DiceState> stateList=new ArrayList<>();
        list.add(new Dice(1, 6));
        list.add(new Dice(1, 4));
        list.add(new Dice(2, 5));
        list.add(new Dice(3, 6));
        list.add(new Dice(4, 1));
        list.add(new Dice(5, 2));
        list.add(new Dice(6, 3));
        list.add(new Dice(6, 1));
        Board fakeBoard=new Board(list);
        list.clear();
        fakeBoard.showBoard0();
        for(int i=1;i<9;i++){
            Dice fakDice=new Dice(0, 0);
            lavel:while (true) { // 適切な状態が入力されるまでループ
            System.out.print  (i+"番に設定するダイスの状態をオモテ面，前面の順にジャンケンの手を入力してください（例：グーパー）：");
            String n=keyBoardScanner.next();
                switch (n){
                    case "チョキグー":
                    DiceState cgp=new Cgp(fakDice);
                    stateList.add(cgp);
                    break lavel;
                    case "チョキパー":
                    DiceState cpg=new Cpg(fakDice);
                    stateList.add(cpg);
```

```
                break lavel;
                case "グーチョキ":
                DiceState gcp=new Gcp(fakDice);
                stateList.add(gcp);
                break lavel;
                case "グーパー":
                DiceState gpc=new Gpc(fakDice);
                stateList.add(gpc);
                break lavel;
                case "パーチョキ":
                DiceState pcg=new Pcg(fakDice);
                stateList.add(pcg);
                break lavel;
                case "パーグー":
                DiceState pgc=new Pgc(fakDice);
                stateList.add(pgc);
                break lavel;
                default:
                System.out.println("上記の例と同じ形式で入力してください：");
                continue;
            }
        }
    }
    list.add(new Dice(1, 6,stateList.get(0)));
    list.add(new Dice(1, 4,stateList.get(1)));
    list.add(new Dice(2, 5,stateList.get(2)));
    list.add(new Dice(3, 6,stateList.get(3)));
    list.add(new Dice(4, 1,stateList.get(4)));
    list.add(new Dice(5, 2,stateList.get(5)));
    list.add(new Dice(6, 3,stateList.get(6)));
    list.add(new Dice(6, 1,stateList.get(7)));
    board=new Board(list);
    num=4;
}


// 動かす駒を選ぶ
public int selectDice(){
    String inputString;
    int diseType=0;
    System.out.println ("動かす駒を選んでください");
    while (true) { // 適切な駒が選択されるまでループ
        while (true) { // 適切な数字が選択されるまでループ
            System.out.print ("動かす駒は？(1~"+num*2+"):");
            inputString = keyBoardScanner.next();
            try {
                diseType = Integer.parseInt (inputString);
            } catch (NumberFormatException e) { // 整数値以外が入力された場合
```

```
                System.out.println ("1～"+num*2+"を入力してください");
                continue;
                }
                if (diseType<1 || num*2<diseType) {
                    System.out.println ("1～"+num*2+"を入力してください");
                    continue;
                }
                break;
            }
            if(diseType>num&&turn==1){
                System.out.println ("敵の駒です");
                continue;
            } else if(diseType<=num&&turn==-1){
                System.out.println ("敵の駒です");
                continue;
            }else break;
        }
        return diseType;
    }

    // 動かす方向を選ぶ
    public String selectDirection(){
        String d="";
        String inputString;
        System.out.print ("どちらの方向へ動かしますか？");
        while (true) { // 適切な方向が選択されるまでループ
            System.out.print ("方向 = ? (u,d,l,r): ");
            inputString = keyBoardScanner.next();
            if  (inputString.equals  ("u")||inputString.equals  ("d")||inputString.equals
("l")||inputString.equals ("r")) {
                break;
            }else {
                System.out.println ("u,d,l,r のどれかを入力してください");
                continue;
            }
        }
        d=inputString;
        return d;
    }

    // プレイヤーの手番
    public Board hand(){
        String turnName="";
        if(turn==1){
            turnName="先手側";
        }else turnName="後手番";
        System.out.println (turnName + "の手番です");
        while(true){
```

```
            int diceType=selectDice()-1;
            if(board.move(selectDirection(), diceType)){
                break;
            }
        }
        board.showBoard();
        if(judge()){
            System.out.println (turnName + "の勝ち！");
            System.exit(0);
        }
        return board;
    }

    // CPUの手番
    public Board comPlay(){
        board=board.bestMove();

        if(judge()){
            System.out.println ( board.comType+"で勝ち！");
            System.out.println ( "com"+turn+"の勝ち！");
            board.showBoard();
            System.exit(0);
        }
        return board;
    }

    // 勝利判定メソッド
    public boolean judge(){
        boolean isWin=false;
        int count=0; // 自分が倒した駒の数
        int goal=0; // ゴールにたどり着いた駒の数
        if(turn==1){
            for(int i=num;i<num*2;i++){ //自分の番に相手の駒を全て取ったか
                if(board.list.get(i).x==0&&board.list.get(i).y==0)count=count+1;
            }
            for(int i=0;i<num;i++){ //自分の番に相手ゴールに駒が届いたか
                if(board.list.get(i).x==board.size&&board.list.get(i).y==1)goal=goal+1;
            }
        }else{
            for(int i=0;i<num;i++){ //相手の番に自分の駒を全て取ったか
                if(board.list.get(i).x==0&&board.list.get(i).y==0)count=count+1;
            }
            for(int i=num;i<num*2;i++){ //相手の番に自分ゴールに駒が届いたか
                if(board.list.get(i).x==1&&board.list.get(i).y==board.size)goal=goal+1;
            }
        }
        if(count==num)isWin=true;
        if(goal>0)isWin=true;
```

```
        return isWin;
    }

    // 好きに駒を移動させる
    public void set(int diseType,int x,int y){
        board.board[y][x]=diseType;
        board.board[board.list.get(diseType-1).y][board.list.get(diseType-1).x]=0;
        board.list.get(diseType-1).x=x;
        board.list.get(diseType-1).y=y;
    }

    public static void main(String[] args) throws Exception {


        JankenShogi jan = new JankenShogi();
        jan.board.showBoard();

        while (true) {

            jan.hand();
            jan.turn=jan.turn*(-1);

            // com=0 はゴール重視
            jan.board.comType=0;
            jan.board=jan.comPlay();
            jan.board.showBoard();
            jan.turn*=-1;

            // com=1 は攻撃重視
            /*jan.board.comType=1;
            jan.board=jan.comPlay();
            jan.board.showBoard();
            jan.turn*=-1;*/
         }
    }
}
```

- Pcg クラス

```
public class Pcg implements DiceState{

    Dice dice;
    String top="ぱ";
    String name="pcg";
    public Pcg(Dice dice){
        this.dice=dice;
    }
```

```
    // ダイスの状態（表面が何かとか）を表示
    public void showState(){
        System.out.println("  ち  ");
        System.out.println("ぐぱぐ");
        System.out.println("  ち  ");
    }
    // ダイスの状態遷移
    public void moveUp(){
        dice.setState(dice.cpg);
    }
    public void moveDown(){
        dice.setState(dice.cpg);
    }
    public void moveRight(){
        dice.setState(dice.gcp);
    }
    public void moveLeft(){
        dice.setState(dice.gcp);
    }
    // ダイスの攻撃判定
    public boolean attackDice(Dice dice){
        boolean isAttack=false;
        if(dice.state.getTopString()=="ぐ"){
            isAttack=true;
        }
        return isAttack;
    }

    // ダイスのオモテ面を返すメソッド
    public  String getTopString(){
        return top;
    }

    // ダイスの状態名を返すメソッド
    public  String getName(){
        return name;
    }
}
```

- Pgc クラス

```
public class Pgc implements DiceState{

    Dice dice;
    String top="ぱ";
    String name="pgc";
```

```
    public Pgc(Dice dice){
        this.dice=dice;
    }

    // ダイスの状態（表面が何かとか）を表示
    public void showState(){
        System.out.println("  ぐ  ");
        System.out.println("ちぱち");
        System.out.println("  ぐ  ");
    }

    // ダイスの状態遷移
    public void moveUp(){
        dice.setState(dice.gpc);
    }
    public void moveDown(){
        dice.setState(dice.gpc);
    }
    public void moveRight(){
        dice.setState(dice.cgp);
    }
    public void moveLeft(){
        dice.setState(dice.cgp);
    }
    // ダイスの攻撃判定
    public boolean attackDice(Dice dice){
        boolean isAttack=false;
        if(dice.state.getTopString()=="ぐ"){
            isAttack=true;
        }
        return isAttack;
    }

    // ダイスのオモテ面を返すメソッド
    public  String getTopString(){
        return top;
    }


    // ダイスの状態名を返すメソッド
    public  String getName(){
        return name;
    }
}
```