卒業研究報告書

題目

ジャンケン将棋アプリの開発

指導教員

石水 隆　講師

報告者

14-1-037-0111

今村 栞菜

近畿大学理工学部情報学科

平成 30 年 1 月 30 日提出

概要

将棋・囲碁に代表されるボードゲームは二人零和完全情報ゲームに分類され、様々なバリエーションが存在する。将棋に類するゲームの１つにジャンケン将棋がある。ジャンケン将棋は駒の形や動きは将棋と大きく異なり、本将棋よりも小さな盤面上で、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という特徴がある。また取った駒は取り捨てで本将棋のように持ち駒として打つことはできない。現在、ジャンケン将棋は知名度が低いため競技人口が少なく、アプリケーションも現在まだ公開されているものはない。そこで本研究では、対戦型ジャンケン将棋アプリケーションの作成を行う。

本研究では、PC 上だけではなく Android や iOS などでの動作が可能なように HTML5＋JavaScript でジャンケン将棋のアプリケーション開発を行った。

目次

1　序論

1.1 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^{50} 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超えている。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[2]。また、チェッカーは双方最善手を指すと引き分けとなる[3]。

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[4]。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)[5]、5×5 の囲碁は黒の 25 目勝ちとなる[6]。

将棋については、盤面のサイズや使用する駒の種類を減らしたサイズ 5 五将棋[7]やゴロゴロ将棋[8]などのミニ将棋がある。5 五将棋はサイズ 5×5 の盤と 6 種類の駒、ゴロゴロ将棋はサイズ 5×6 の盤と 4 種類の駒を使用する。これらは本将棋と比べて可能な局面数が少ない。しかしながら現在のところまだこれらは完全解析はされていない。

完全解析されているミニ将棋として、どうぶつしょうぎ[9]やアンパンマンはじめてしょうぎ[10]がある。どうぶつしょうぎはサイズ 3×4 の盤と、ライオン、象、キリン、ひよこの 4 種類の駒を使用し、アンパンマンはじめてしょうはサイズ 3×5 の盤とアンパンマン（バイキンマン）、食パンマン（ホラーマン）、カレーパンマン（ドキンちゃん）の 3 種類の駒を使用する幼児向けのミニ将棋である。どうぶつしょうぎは完全解析により双方最善手を指した場合、78 手で後手が勝つことが判明している[11]。また、アンパンマンはじめてしょうぎは、双方最善手を指すと千日手で引き分けとなることが判明している[12]。

1.2 ゲーム AI の手法

可能な局面数が多いゲームに対して完全解析を行うことは困難である。そのようなゲームに対しては完全な最善手を得ることはできないが、局面の評価値計算、定石データベース、一定手先の先読み、終盤での必勝読みと完全読み、モンテカルロ法などを用いてより有利だと思われる手を選択することができる[13]。

局面の評価値計算とは、局面を判断するための指標である評価値を導出することである。コンピュータ将棋の場合のパラメータは、一般的に、成り駒を含めた駒の価値や、相手の攻め駒と自分の玉との距離などの相対的な位置関係などを数値化したものが用いられている。AI の強さは評価関数の作り方に応じて決まるため、評価関数はできるだけ戦局を適切に評価できるように工夫して作る必要がある。

定石データベースとは、プロ棋士などが指した実践譜をもとに編成したデータベースのことである。このデータベースを使用することでより強い AI になる。しかし、相手があえて定石以外の手を指すなどして、データベースにない局面が出てきたときにはこの手法は使えない。

一定手数の先読みとは、一定手数を先読みすることにより最善の手を打たせることである。たとえば、先読みの深さを 3 とし、局面の分岐数を x とした場合、3^x 個の局面数を評価し、その中から最善の手を打つことができる。一般に先読みする手数が多いほど強い AI となるが、先読み手数の増加に伴い探索時間が指数的に増えるため、適度に枝切りをして探索範囲を減らす工夫をする必要がある。

必勝読みとは、オセロのように勝敗だけでなく石差も問題になるゲームの場合に、勝敗のみを読み

切ることをいい、また石差までを読み切ることを完全読みという。必勝読みのほうが計算時間が少なくて済むため、一般的にまず必勝読みで価値を確定させた上で、残り手数が少なくなると完全読みに切り替えてより点数の高い勝ちを目指すことが多い。将棋では、終盤の読みや詰将棋には完全読みが行われている。

モンテカルロ法とは、乱数を用いたシミュレーションを何度も行うことにより近似解を求める計算手法である。解析的に解くことができない問題でも、十分な回数のシミュレーションを行うことにより、近似的に解を求めることができる。問題によっては他の数値計算手法より簡単に適用できるが、高い精度を得ようとすると計算回数が膨大になってしまうという弱点もある。

昨今注目されている手法にディープラーニング[16]がある。ディープラーニングはニューラルネットワークを利用した機械学習の手法であり、これをゲームに応用することで従来の AI の性能を超える AI が作れる可能性がある。例えば囲碁では、α碁と呼ばれる AI がプロ棋士に勝つなど目覚ましい成績を上げている[17]。

以上の手法を用いることにより、完全解析を行わなくてもある程度の強さのプログラムを作ることが可能であり、ゲームによってはプロに勝つこともできる。

1.3 ジャンケン将棋

ジャンケン将棋[14]は、将棋に類するゲームの 1 つである。ジャンケン将棋は駒の形や動きは将棋と大きく異なり、6×6 の盤面とグー、チョキ、パーが描かれた立方体の駒を使用する。駒を転がして移動し、駒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ボードゲームである。また、取ったサイコロ型の駒は取り捨てで、本将棋のように持ち駒として打つことはできない。

1.4 本報告書の目的

将棋に類するボードゲームは近年に至るまでいくつも考案されており、またそれらに対するアプリケーションも多く作られている。しかしそのうちの一つであるジャンケン将棋は、知名度が低いため競技人口が少なく、アプリケーションも現在まだ公開されているものはない。そこで本研究では、対戦型ジャンケン将棋アプリケーションの作成を行うことを目的とする。

1.5 本報告書の構成

本報告では、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。続く 3 節で、本アプリケーションを作成した際の仕様について述べ、4 節で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。5 節において考察を述べる。

2　ジャンケン将棋

本章では、本研究で作成するアプリケーションのジャンケン将棋について説明する。

2.1 概要

ジャンケン将棋は、熊本県立八代工業高等学校インテリア科教師の梅田龍一氏によって考案され、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。

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。

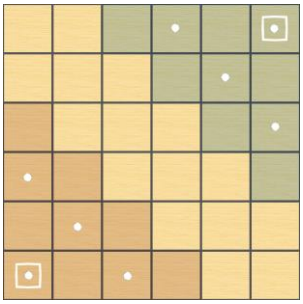

図 1 ゲーム盤

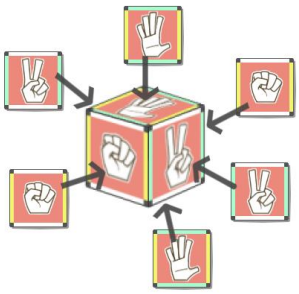

図 2 ジャンケン将棋の駒

2.2 基本的なルール

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、相手駒を全滅させるか、相手のゴールに駒を到達させることである。ゴールの位置は、図 1 のゲーム盤の左下と右上の角にある四角く白で囲んでいるマスの 2 ヶ所である。

- 準備

  駒の初期配置位置は図 1 の白の点が描かれてある 8 ヶ所のマスである。配置時の駒の向きはグー、チョキ、パーのどの面を上にしてもよい。

- 駒の動かし方

  駒は基本回転して上下左右 1 マスを移動する。駒を動かせる数のことを行動力という。各手番では 2 行動力を使用できる。このとき 1 つの駒に対して 2 行動力を使用するか 2 つの駒を 1 行動力ずつ使用するかのどちらかを選ぶことができる。ただし、同じ駒を動かして元の場所に戻ることはできない。図 4 に移動の様子を示す。

- 駒の取り方

  自駒の上下左右の 1 マスに相手駒があり、自駒の上面のジャンケンの手がその相手駒の上面のジャンケンの手に勝っていた場合、行動力を 1 使い回転せずに取ることができる。図 3 に攻撃の様子を示す。

- 出戻り禁止エリア

  自分のゴールから 3 マス離れたところまでが自分のエリアとなる。図 1 の自分のゴールがある色の付いた全てのマスがこの出戻り禁止エリアである。一度自駒がここから出てしまうと、もうその駒は自分のエリアには戻れなくなる。

- ふんばりモード

自駒が最後の 1 つになると、ふんばりモードと呼ばれる状態になる。このとき自分の手番の行動力が 2 から 3 になる。また、本来なら出戻り禁止エリアから出るともう戻れないが、このモードの時は自由に行動できるようになる。

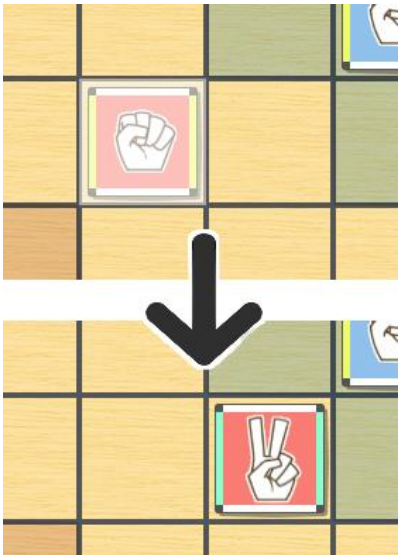

図 4 移動の様子

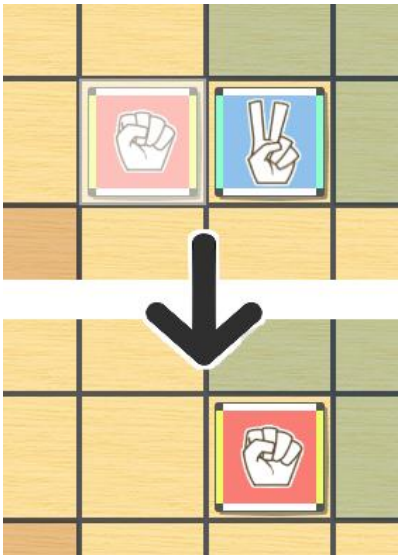

図 3 攻撃の様子

3　アプリケーションの仕様

本章では、本研究で作成したジャンケン将棋アプリケーションについて述べる。

3.1 HTML5 について

HTML5 とは、Web サイトを構築する際に使用するマークアップ言語の 5 回目の大幅改修が行われたバージョンである[15]。従来のゲーム開発では対応機種がなにかを考えなくてはならなかった。しかし HTML5 はブラウザにアクセスできれば、どの端末であっても同じプログラムが動き、マルチデバイス・マルチプラットフォームを実現する技術の一つとして注目されている。

本研究では PC 上だけではなく Android や iOS などでの動作が可能なように、HTML5＋JavaScript でジャンケン将棋のアプリケーション開発を行った。

3.2 ゲームの流れ

本研究で作成したゲームの流れを説明する。

- 初期配置位置の設定

  本アプリケーションは実行された時点では自駒のプレイヤーの初期配置入力待ち状態になっている。手番になったプレイヤーは以下の状態を繰り返し、初期配置位置を決めていく。図 5 にゲーム実行時の初期配置位置設定中の画面を示す。

図 5 初期配置位置設定

1. 配置する駒を選択
   手番になったプレイヤーの駒は最初に入力待ち状態になっている。手番になっている側の駒の選択をクリックまたはタッチにて行う。この時図 5 の右下の紺色の配置駒リストの中にある白の四角ボタンをクリックまたはタッチすると、状態 2 へ移行する。駒が選択されたらその駒の色を白に変更し、状態 3 へ移行する。
2. 駒を回転する
   図 5 の右下にある紺色の配置駒リストの白の四角ボタンに対応する駒が 90°回転する。また、回転した駒を選択中の場合は、自動的に選択駒も回転後のものに更新する。その後状態 1 または状態 3 に移行する。
3. 選択した駒を配置するマスを選択
   選択した駒の移動先マスをクリックまたはタッチにて選択できる。選択されたマスが初期配置可能マスであれば配置を行う。この時、配置不可能なマスを選択しても駒は反応しない。またこの時、状態 1 の時点で選択された駒以外の配置駒を選択すると、新たに選択した駒が選択状態となり、前に選択していた駒は選択状態が解除される。
4. 移動した後
   この状態ではプレイヤーは操作を行わない。
   盤面に 8 つの駒が配置されていれば、初期配置位置設定を終了し、ゲーム開始を行う。
   初期配置位置設定が完了していなければ、選択されていた駒の色が元の色に戻り選択状態も解除される。手番になったプレイヤーが 4 つの駒の配置を終えた場合はプレイヤーが相手に移り、それ以外の場合はプレイヤーをそのままに状態 1 へ移行する。

また、プレイヤーが配置完了するまでは何度でも自駒の配置を変更することができる。

- ゲーム開始

初期配置位置設定を終えた時点でゲームが開始される。開始時点では自駒のプレイヤーの入力待ち状態になっており、手番になったプレイヤーは以下の状態を繰り返し、ゲームを進めていく。図 6 にゲーム実行様子を示す。

1. 移動する駒を選択

手番になったプレイヤーの駒は最初に入力待ち状態になっている。手番になっている側の駒の選択をクリックまたはタッチにて行う。その駒があるマスの色を白に変更し、状態 2 に移行する。

2. 選択した駒を移動する先を選択

選択した駒の移動先マスをクリックまたはタッチにて選択できる。選択されたマスが移動できるマスの場合、回転移動をして状態 3 に移行する。また、選択されたマスに攻撃できる駒があればゲームから取り除き、回転をせずに移動をして状態 3 へ移行する。

この時動けないマスを選択しても駒は反応せず、状態 2 から移行しない。またこの時、状態 1 の時点で選択された駒以外の自駒を選択すると新たに選択した駒が選択状態となり、前に選択していた駒は選択状態が解除される。

3. 移動した後

この状態ではプレイヤーは操作を行わない。

ゲームの終了条件を満たしていればゲーム上にリセットボタンとともにどちらのプレイヤーが勝利したかが表示される。表示されているリセットボタンを押すとページを更新し、ゲームを改めて初期配置位置設定の状態 1 から始めることができる。図 8 にゴールに到達した場合の終了画面を、図 7 に相手の駒を全滅させた場合の終了画面を示す。

ゲームの終了条件を満たしていなければ、選択されていた駒の色が元の色に戻り選択状態も解除される。プレイヤーが行動力を全て使い終えた場合はプレイヤーが相手に移り、使い終わっていない場合はプレイヤーをそのままに状態 1 へ移行する。

図 6 ゲーム実行様子

図 8 ゴール到達時の終了画面

図 7 相手全滅時の終了画面

また初期配置位置設定、ゲーム開始や状態に関わらず駒を選択すると図 6 の右上に選択した駒の立体図が表示されるようになっている。“前”と書いてある面が上下に回転移動した際の表示絵柄であり、“横”と書いてある面が左右に回転移動した際の表示絵柄となっている。

4　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プログラムのソースを示す。

本研究で作成したジャンケン将棋プログラムは、game.js、board.js、hand.js の 3 つのスクリプトからなる。

4.1 game.js

game.js は、クリックイベントや描画の呼び出しを行う、ジャンケン将棋プログラムの中心となるスクリプトである。game.js のメソッドを以下に挙げる。

- initGame()

  initGame()はゲームを開始処理を行うメソッドである。マウスイベントの登録を行い、盤面の初期化を行った後表示をする。

- setEvents()

  setEvents()はクリックやマウスを動かした際のイベントの関連付けを行うメソッドである。スマホでの操作時はクリックではなくタッチに紐付けている。

- mouseMove()

  mouseMove()はマウスが動いた時の処理を行うメソッドである。現在マウスが選択している場所の座標を保存する。

- mouseClick()

  mouseClick()はマウスをクリックした時の処理を行うメソッドである。現在マウスが選択している場所の座標を保存し、以下の処理を行った後盤面の表示を行う。

  - 初期配置駒リストをクリックした時

    初期配置設定が未完了のみ、駒をクリックすると配置駒を選択し、駒の隣の白の四角ボタンをクリックすると駒を回転させる。

  - 盤面内をクリックした時

    - 初期配置が未完了時

      配置駒を選択している状態で初期配置位置をクリックすると、プレイヤーが選択したマスに選択した駒を置く。入力不可能な場合は処理を行わない。また、それぞれ手番ごとに 4 つの初期配置駒を置き終わるまでは何度でも置き直せる。

    - ゲーム開始時

      自駒をクリックするとその駒を選択する。この時選択された駒以外の自駒をクリックすると新たに駒を選択し直す。

      自駒を選択している状態で移動可能な位置をクリックすると、プレイヤーが選択したマスに選択した駒を置く。入力不可能な場合は処理を行わない。

      プレイヤーが行動力を使い終わると手番を交代する。また、ゲームの終了条件を満たしていればゲーム上にリセットボタンとともにゲームの勝敗を表示する。

- getMousePosition()

  getMousePosition()はマウスの座標を取得するメソッドである。

- hitTest()

  hitTest()はマウスがある場所（初期配置駒リストか盤面）とマスの位置の登録を行うメソッドである。

- reset()

  reset()はゲームのリセットを行うメソッドである。

- stateCopy()

  盤面のコピーを行うメソッドである。

4.2 board.js

board.js は画面描画を担当するスクリプトである。board.js のメソッドを以下に挙げる。

- showBoard()

  showBoard()はキャンバスを描画するメソッドである。初回呼び出し時のみ盤面、駒、マス選択、立体駒、初期配置駒リストの 5 つのオフスクリーンキャンバスを設定する。また、マス選択、立体駒はこの関数が呼び出されるごとに設定し直し、初期配置駒リストと盤面内の駒は変化している場合のみ設定し直す。その後キャンバスに描画するが、初期配置駒リストは初期配置設定未完了時のみ表示するようにする。

  また、画面左上に現在の状況に応じて先手か後手か、何手目かの表示を行う。

- drawBoard()

  drawBoard()は盤面をオフスクリーンキャンバスに描画するメソッドである。初回呼び出し時のみ盤面キャンバスを作成する。

- drawPieceAll()

  drawPieceAll()は全ての駒をオフスクリーンキャンバスに描画するメソッドである。初回呼び出し時のみ駒キャンバスを作成する。盤面の状態を確認し、駒が置かれているマスの場所と駒の種類を引数にして drawPiece()を呼び出す。

- drawPiece()

  drawPiece()は指定されたマスに駒を 1 つオフスクリーンに描画するメソッドである。

- drawSelect()

  drawSelect()はマス選択（色変更）をオフスクリーンに描画する。また初期配置設定時のみ、初期配置可能マスにも色変更を行うメソッドである。クリックごとに盤面の状態を確認し、マウスの座標が盤面のマス目内の自駒もしくは初期配置駒リスト内の駒の上にあった時、そのマス目の色変更を行う。

- drawDice()

  drawDice()は選択した駒の立体図をオフスクリーンに描画する。初回呼び出し時のみ立体駒キャンバスを作成するメソッドである。クリックごとに盤面の状態を確認し、マウスの座標が盤面のマス目内の自駒もしくは初期配置駒リスト内の駒の上にあった時、その駒の立体図を描画する。

- drawSet()

  drawSet()は初期配置駒リストをオフスクリーンに描画する。初回呼び出し時のみ初期配置駒リストキャンバスを作成するメソッドである。クリックごとにリストの状態を確認し、駒が置かれているマスと駒の種類を引数にして drawPiece()を呼び出す。

- stateCopy()

  stateCopy()は盤面のコピーを行うメソッドである。

4.3 hand.js

hand.js は盤面の状態を判断するスクリプトである。hand.js のメソッドを以下に挙げる。

- isSet()

  isSet()は盤面にいくつ置かれているかを返すメソッドである。初期配置設定時は盤面にある全ての駒数、ゲーム開始時はその手番が持つ駒数を返す。

- canSet()

  canSet()は初期配置設定時のみ、選択したマスが初期配置可能マスであるかどうかの判定を行うメソッドである。

- canSelect()

  canSelect()は盤面の選択したマスに自駒が置かれているか（駒を選択できるか）どうかの判定を行うメソッドである。

- canPut()

  canPut()は選択したマスに選択した駒が置けるかどうかの判定を行うメソッドである。

- getDirection()
  getDirection()は指定した駒から見た選択したマスの方向を返すメソッドである。
- inArea()
  inArea()は指定したマスが自エリア内かどうかの判定を行うメソッドである。
- rollPiece()
  rollPiece()は指定した駒を選択したマスへ転がした場合の駒の種類を返すメソッドである。
- attack()
  attack()は指定した駒で相手駒を攻撃した場合の勝利判定を返すメソッドである。
- setMap()
  setMap()は初期配置設定時のみ、初期配置可能マスに駒を置いた後の盤面の状態を返すメソッドである。
- putMap()
  putMap()は選択したマスに駒を移動した後の盤面の状態を返すメソッドである。
- isEnd()
  isEnd()はゲームの勝敗判定を行うメソッドである。

5 考察

本研究の目的であるジャンケン将棋アプリケーションの開発は、基本ルールの要件を満たしており、平面上では駒の側面の手が見えないことの対策として選択した駒の立体表示機能をつけるなど視覚的に分かりやすくなっており、操作性については問題ないと評価する。しかし、駒の面の端が側面の手に対応した色になっているとはいえ、選択した駒の立体図が分かるだけでは慣れなければ一目で盤面の様子を理解できない人もでてくると考えられる。ジャンケン将棋の駒は転がして移動するため、上点を変更できるようにするなどの視認性の向上を行うことが考えられる。

また、対人戦アプリケーションを開発したが、ネットワークを介したアプリケーションの実装ができておらず、現状一つの画面上でしか対人戦が行えない。そのため Servlet を用いて違うデバイス同士で対戦ができるようにすることが必要である。

6 結論・今後の課題

本研究では、対人戦ジャンケン将棋アプリケーションを開発した。本アプリケーションでは 2 人のプレイヤーが対戦を行える。平面上では駒の側面の手が見えないことの対策として選択した駒の立体表示機能をつけるなど視覚的に分かりやすくなっているが、真上からの盤面の様子しかなく、更なる視認性の向上を行いプレイしやすい環境を構築することが改善点として挙げられる。また、対人戦アプリケーションを開発したが、現状ひとつの画面上でしかプレイすることができず、離れた場所での対戦ができない。そのため Servlet を用いて違うデバイス同士で対戦ができるようにすることが必要である。また今後の課題としては、対人戦だけでなく CPU との対戦ができるようにジャンケン将棋の AI を作成することが考えられる。例を挙げると既存の将棋やチェスの AI には 1.2 節で述べた定石データベースの利用、先読みによる盤面の評価値計算、終盤での詰み読みといった手法が利用されている。ジャンケン将棋でも同様に、これらの手法を用いることによって人間とまともに戦える CPU が作成できるのではないかと考えられる。

謝辞

本研究を作成するにあたり、指導教員の石水隆講師から、丁寧かつ熱心なご指導を賜りました。 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（1986）

[2] Janos Wagner and Istvan Virag、Solving renju : ICGA Journal、Vol.24、 No.1、 pp.30-35 (2001) http://www.sze.hu/~gtakacs/download/wagnervirag_2001.pdf

[3] 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

[4] 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

[5] 清慎一、川嶋俊：探索プログラムによる四路盤囲碁の解、情報処理学会研究報告、GI 2000(98)、pp.69--76 (2000) https://ipsj.ixsq.nii.ac.jp/ej/?action=repository_uri&item_id=58632

[6] 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)

[7] 日本 5 五将棋連盟 http://www.geocities.co.jp/Playtown-Spade/8662/

[8] 「ごろごろどうぶつしょうぎ」発売開始！、 お知らせ、 日本将棋連盟、2012 年 11 月 26 日 (2012) http://www.shogi.or.jp/topics/2012/11/post-652.html

[9] 北尾まどか、藤田麻衣子：どうぶつしょうぎねっと（2010） http://dobutsushogi.net/

[10] アンパンマンはじめて将棋、セガトイズ （2012) http://www.segatoys.co.jp/anpan/product/popup/_legacy/learn/06.htm

[11] 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/

[12] 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/

[13] 小谷善行、岸本章宏、柴原一友、鈴木豪：ゲーム計算メカニズム —将棋・囲碁・オセロ・チェスのプログラムはどう動く—、コロナ社（2010）

[14] 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） https://hon.gakken.jp/book/1575033700

[15] Mark Pilgrim：入門 HTML5、オライリー・ジャパン（2011）

[16] 藤田一弥、高原歩夢 ：実装ディープラーニング、オーム社(2016)

[17] 伊藤毅志、村松正和：ディープラーニングを用いたコンピュータ囲碁～ Alpha Go の技術と展望～、 情報処理、Vol.57、No.4, pp.335-337、情報処理学会（2016） http://id.nii.ac.jp/1001/00158059/

付録 ソースプログラム

以下に本研究で作成したプログラムのソース及び HTML コードを示す。

- Game.js

```
Game = Game;
Game.reset = reset;

function Game(){}

var square = 6; //マス目
var context; //2D コンテキスト
var evented = false; //マウスイベントを登録したかどうか
var point = { //マウスの座標
    x: 0,
    y: 0
}
var finish = false; //ゲームが終了しているか
var state = {} //ゲーム盤
var init_state = { //開始時の盤面
    board: [0, 0, 0, 0, 0, 0,
            0, 0, 0, 0, 0, 0,
            0, 0, 0, 0, 0, 0,
            0, 0, 0, 0, 0, 0,
            0, 0, 0, 0, 0, 0,
            0, 0, 0, 0, 0, 0,
            ],
    list: [1, 3, 5, 0, 0, 0], //初期配置駒リスト
    isSet: 0, //盤面の配置コマ数
    turn: 1,   //手番、先手が 1、後手が-1
    count: 0, //手番何回目かのカウント
    extra: 0, //ふんばりモード時の追加手番のカウント
    flag: false, //駒を選択した状態か
    target: null, //選択中の駒があるマス
    startCell: null, //1 手目で移動した駒がもともとあったマス目
    startPiece: null, //1 手目で移動した駒のマス目（移動後）
    change: 0, //初期配置駒リストを描画し直すか
    update: 0, //盤面を描画し直すか
    selected: { //どのマスが選択されているか
        name: "", //マウスのある場所
        value: 0 //マウスが選択しているマス
    }
}

/*
 * ゲーム開始処理
 */
window.onload = function(){
```

```
	var context = document.getElementById("board").getContext("2d");
	initGame(context);
}

/*
 * ゲームを開始する
 * obj：2D コンテキスト
 */
function initGame(obj){
	context = obj;
	state = stateCopy(init_state); //盤面の初期化
	if(!evented){
		evented = true;
		setEvents();
	}
	var winner = document.getElementById("winner");
	winner.style.display = "none"; //勝敗結果の非表示
	var reset = document.getElementById("reset");
	reset.style.display = "none"; //リセットボタンの非表示

	Board.showBoard(context, state); //盤面の表示
}

/*
 * マウスイベント連携
 */
function setEvents(e){
	var touch;
	if("ontouchatart" in window) touch = true; //スマホでの操作時
	else touch = false; //パソコンによる操作時

	if(touch){
		context.canvas.addEventListener("touchstart", mouseClick);
	}
	else{
		context.canvas.addEventListener("mousemove", mouseMove);
		context.canvas.addEventListener("mouseup", mouseClick);
	}
}

/*
 * マウスが動いた時の処理
 */
function mouseMove(e){
	getMousePosition(e);
	state.selected = hitTest(point.x, point.y); //選択中の座標を保存
}
```

```
/*
 * マウスがクリックした時の処理を行う
 */
function mouseClick(e){
    var selected = hitTest(point.x, point.y); //マウスの座標の取得
    var c; //何手目か
    var t = 1; //手番

    if(finish) return; //ゲーム終了時は処理を行わない

        if(selected.name == "setSize"){ //初期配置駒リスト内にマウス座標がある場合
         if(state.isSet < 8){ //初期配置設定未完了時
            if(selected.value < 3){ //配置駒を選択する
                state.flag = true;
                state.target = selected.value;
            }
            else{ //ボタンを押したとき、対応する駒を回転する
                if(state.list[selected.value % 3] % 2 == 0){
                    state.list[selected.value % 3] =
                        (Math.abs(state.list[selected.value % 3]) - 1) * state.turn;
                }
                else{
                    state.list[selected.value % 3] =
                        (Math.abs(state.list[selected.value % 3]) + 1) * state.turn;
                }
            }
                state.change += 1; //初期配置コマリストを更新
                Board.showBoard(context, state);
        }
    }

        if(selected.name == "boardSize"){ //盤面内にマウス座標がある場合
         if(state.isSet < 8){ //初期配置設定未完了時
            if(state.flag){ //自駒選択時
                if(Hand.canSet(selected.value, state.turn)){ //プレイヤーの指した手を盤面に反映
                    state.board = Hand.setMap(state.board, state.turn,
                                              selected.value, state.target, state.list);
                    state.isSet = Hand.isSet(state.board, 0); //現在盤面にある駒の数を登録
                    state.update += 1; //盤面の状態を更新
                    state.flag = false; //駒の選択状態を解除
                    state.target = null;

                    if(state.isSet != 0 && state.isSet % 4 == 0){ //先手の初期配置設定完了時
                        state.turn = -1 * state.turn; //手番交代
                        state.list = [-1, -3, -5, 0, 0, 0]; //駒リストを変更
                        state.change += 1; //初期配置駒リストを更新
                    }
```

```
                    var t = document.getElementById("teban");
                    if(state.isSet == 4) t.innerHTML =
"初期配置設定中:　後手番（駒を白いエリアに配置してください）<br>";
                    else if(state.isSet == 8){ //初期配置設定完了時
                        t.innerHTML = "先手番　1 手目<br>";
                        var c = document.getElementById("configuration");
                        c.style.display = "none";
                        var g = document.getElementById("guide");
                        g.style.display = "table-cell";
                    }

                    Board.showBoard(context, state); //画面を描画する
                }
            }
            return;
        }

        if(state.flag){ //盤上の駒を選択している場合
            //選択したマスに駒を配置する
            if(Hand.canPut(state.board, state.turn, selected.value, state.target,
        state.startCell, state.startPiece)){
                //プレイヤーの指した手を盤面に反映
                state.board = Hand.putMap(state.board, state.turn,
                                                selected.value, state.target);
                state.update += 1; //盤面の状態を更新
                state.flag = false; //駒の選択状態を解除

                //ふんばりモードの 3 手目
                if(state.extra %2 == 1){
                    state.count += 1;
                    state.extra += 1;
                }
                //ふんばりモードの 2 手目
                else if(Hand.isSet(state.board, state.turn) == 1 && state.count%2 == 1){
                    state.extra += 1;
                }
                else state.count += 1; //手番を進める

                //1 手目で行動した駒の位置を登録(2 手目で元の位置に戻るの防止)
                if(state.count%2 == 1) {
                    state.startCell = state.target;
                    state.startPiece = selected.value;
                }
                else {
                    state.startCell = null;
                    state.startPiece = null;
                }
```

```
                state.target = null; //マスの選択状態を解除
                Board.showBoard(context, state); //画面を描画する

                //表示用の現在の手数を登録
                if(state.extra %2 == 1) c = (state.count%2)+2;
                else c = (state.count%2)+1;

                //表示用の手番を登録
                if((!state.flag && state.count%4 == 0) ||
                        (!state.flag && state.count%4 == 2)) t = state.turn * -1; //手番交代時
                else t = state.turn;

                var teban = document.getElementById("teban");
                if(t > 0) teban.innerHTML = "先手番　"+ c +"手目<br>";
                else teban.innerHTML = "後手番　" + c + "手目<br>";


                if(Hand.isEnd(state.board) != 0) { //対戦終了判定
                    //勝敗結果の表示
                    var w = document.getElementById("winner");
                    if(Hand.isEnd(state.board) > 0) w.innerHTML = "先手の勝利！<br>";
                    else w.innerHTML = "後手の勝利！<br>";
                    w.style.display = "inline";
                    //リセットボタンの表示
                    var r = document.getElementById("reset");
                    r.style.display = "inline";
                    finish = true;
                }

                if(state.count %  2 == 0) state.turn = -1 * state.turn; //手番交代
            }

            else if(Hand.canSelect(state.board, selected.value, state.turn)){ //自駒を選択し直す
                state.target = selected.value;
                Board.showBoard(context, state);
            }
        }

        else{ //盤上の駒を選択する
            if(Hand.canSelect(state.board, selected.value, state.turn)){
                state.flag = true;
                state.target = selected.value;
                Board.showBoard(context, state);
            }
        }
    }
}
```

```
/*
 * マウス座標を取得する
*/
function getMousePosition(e){
	var rect = e.target.getBoundingClientRect();
	point.x = e.clientX - rect.left;
	point.y = e.clientY - rect.top;
}

/*
 * マウスの座標の設定
 * x： マウスの x 座標
 * y： マウスの y 座標
 */
function hitTest(x, y){
	var objectBd = [Board.boardSize];
	var objectSet = [Board.setSize];
	var selected = {
		name: "",
		value: 0
	}

	for(var i = 0; i < objectBd.length; i++){
		//ボード内にマウス座標がある場合
		if(objectBd[i].w >= x && objectBd[i].x <= x && objectBd[i].h >= y && objectBd[i].y <= y){
			selected.name = "boardSize";
			break;
		}
	}

	for(var j = 0; j < objectSet.length; j++){
		//初期選択リスト内にマウス座標がある場合
		if((objectSet[j].w + objectSet[j].x) >= x && objectSet[j].x + 30 <= x &&
		   (objectSet[j].h + objectSet[j].y) >= y && objectSet[j].y <= y){
			selected.name = "setSize";
			break;
		}
	}

	//selected の設定
	if(selected.name === "boardSize"){
		selected.name = "boardSize";
		selected.value = Math.floor(y / Board.cellSize) * square + Math.floor(x / Board.cellSize);
	}
	else if(selected.name === "setSize"){
		selected.name = "setSize";
		selected.value = Math.floor((y - Board.setSize.y) / Board.cellSize) +
			Math.floor((x - Board.setSize.x- 10) / Board.cellSize) * 3;
```

```
    }

    return selected; //選択されたマスを返す
}

/*
 * ゲームのリセット(初期開始処理)
 */
function reset(){
    location.reload();
}

/*
 * 盤面のコピー
 */
function stateCopy(obj){
    return JSON.parse(JSON.stringify(obj));
}
```

- Board.js

```
Board = Board;
Board.showBoard = showBoard;
Board.boardSize = boardSize;
Board.cellSize = cellSize;
Board.setSize = setSize;

var square = 6; //マス目
var canvasSize = { //canvas の大きさ
    x: 0,
    y: 0,
    w: 660,
    h: 486
};
var boardSize = { //盤面の大きさ
    x: 0,
    y: 0,
    w: 486,
    h: 486
};
var diceSize = { //立体ダイスの大きさ
    x: 496,
    y: 0,
    w: 162,
    h: 135
};
```

```
var setSize = { //初期配置する駒の大きさ
	x: 496,
	y: 243,
	w: 162,
	h: 243
};
var area1 = {
	a: 18,
	b: 25,
	c: 30,
	d: 32
};
var area2 = {
	a: 3,
	b: 5,
	c: 10,
	d: 17
};
var cellSize = boardSize.w / square | 0; //マスの大きさ

var preUpdate = -1; //盤面の変化確認
var preChange = -1; //初期配置駒リストの変化確認
var complete = 0; //初期配置の駒数

var plates = { //盤面、駒、選択マスの canvas
	canvasBoard: null,
	canvasPieaces: null,
	canvasSelect: null
};
var dices = { //立体駒の canvas
	canvasDice : null
};
var sets = { //初期配置する駒の canvas
	canvasSet : null
};

function Board(){}

/*
 * 盤面を表示する
 * context：2D コンテキスト
 * state： 盤面の様子
 */
function showBoard(context, state){
	//初期設定
	if(preUpdate < 0){
		plates.canvasBoard = drawBoard(state);
		plates.canvasPieaces = drawPieceAll(state);
```

```
        plates.canvasSelect = drawSelect(state);
        dices.canvasDice = drawDice(state);
        sets.canvasSet = drawSet(state);
        Board.boardSize = boardSize; //ボードのサイズ設定
        Board.cellSize = cellSize; //マスのサイズ設定
        Board.setSize = setSize; //初期選択駒リストのサイズ設定
}
else{
        //初期配置駒リストが変化している場合、駒を回転させる
        if(state.change != preChange){
                sets.canvasSet = drawSet(state);
                complete = Hand.isSet(state.board);
        }
        //盤面が変化している場合、駒を置き直す
        if(state.update != preUpdate){
                plates.canvasPieaces = drawPieceAll(state);
        }
        plates.canvasSelect = drawSelect(state);
        dices.canvasDice = drawDice(state);
}

//先手後手の表示
var i = state.count;
var count; //何手目か
var teban = 1; //手番

//何手目かの表示
if((!state.flag && i%2 == 0) || i == 0) count = 1;
else if(!state.flag && i%2 == 1) count = 2;
else count = (i%2)+1;

//先手か後手かの表示
if(i == 0) teban = state.turn;
else if((!state.flag && i%4 == 0) ||
        (!state.flag && i%4 == 2)) teban = state.turn * -1;
else teban = state.turn;

context.clearRect(0, 0, canvasSize.w, canvasSize.h);
setTimeout(function(){
        if(complete < 8){ //初期配置未完了時のみ描画
                context.drawImage(sets.canvasSet, setSize.x, setSize.y,
                                setSize.w, setSize.h);
        }
        context.drawImage(plates.canvasBoard, 0, 0, boardSize.w, boardSize.h);
        context.drawImage(plates.canvasPieaces, 0, 0, boardSize.w, boardSize.h);
        context.drawImage(plates.canvasSelect, 0, 0, canvasSize.w, canvasSize.h);
        context.drawImage(dices.canvasDice, diceSize.x, 0, diceSize.w, diceSize.h);
        preUpdate = state.update; //今の状態を記憶
```

```
            preChange = state.change;
        }, 40);
}

/*
 * 盤面を描く
 * state：盤面の様子
 */
function drawBoard(state){
        //初回呼び出し時のみ盤面キャンバスを作成
        if(!plates.canvasBoard){
                plates.canvasBoard = document.createElement("canvas");
                plates.canvasBoard.width = boardSize.w;
                plates.canvasBoard.height = boardSize.h;
        }

        var context = plates.canvasBoard.getContext("2d");
        context.clearRect(0, 0, canvasSize.w, canvasSize.h);

        //盤面を描画する
        var plateImage = new Image();
        plateImage.src = "img/plate.png";
        for(var i = 0; i < 1; i++){
                plateImage.onload = function(){
                        context.drawImage(plateImage, 0, 0, boardSize.w, boardSize.h);
                }
        }
        return plates.canvasBoard;
}

/*
 * 全ての駒を設置する
 * state：盤面の様子
 */
function drawPieceAll(state){
        //初回呼び出し時のみ盤面キャンバスを作成
        if(!plates.canvasPieaces){
                plates.canvasPieaces = document.createElement("canvas");
                plates.canvasPieaces.width = boardSize.w;
                plates.canvasPieaces.height = boardSize.h;
        }

        var context = plates.canvasPieaces.getContext("2d");
        context.clearRect(0, 0, boardSize.w, boardSize.h);
```

```
	//指定したマスに駒を設置
	for(var x = 0; x < square; x++){
		for(var y = 0; y < square; y++){
			if(state.board[y * square + x] != 0){
				drawPiece(context, x * cellSize, y * cellSize,
						state.board[y * square + x]);
			}
		}
	}

	return plates.canvasPieaces;
}

/*
 * 駒を描画する
 * context：2D コンテキスト
 * x： 駒のある x 座標
 * y： 駒のある y 座標
 * value： 駒の種類
 *            P
 * GPC = CGC
 *            P
 */
function drawPiece(context, x, y, value){
	var pieceImage = new Image();
	switch(value){
	case 1: //先手 GPC
	case 11:
		pieceImage.src = "img/piece/p_GPC1.png";
		break;

	case 2: //先手 GCP
	case 12:
		pieceImage.src = "img/piece/p_GCP1.png";
		break;

	case 3: //先手 PGC
	case 13:
		pieceImage.src = "img/piece/p_PGC1.png";
		break;

	case 4: //先手 PCG
	case 14:
		pieceImage.src = "img/piece/p_PCG1.png";
		break;

	case 5: //先手 CPG
	case 15:
```

```
        pieceImage.src = "img/piece/p_CPG1.png";
        break;

    case 6: //先手 CGP
    case 16:
        pieceImage.src = "img/piece/p_CGP1.png";
        break;

    case -1: //後手 GPC
    case -11:
        pieceImage.src = "img/piece/p_GPC2.png";
        break;

    case -2: //後手 GCP
    case -12:
        pieceImage.src = "img/piece/p_GCP2.png";
        break;

    case -3: //後手 PGC
    case -13:
        pieceImage.src = "img/piece/p_PGC2.png";
        break;

    case -4: //後手 PCG
    case -14:
        pieceImage.src = "img/piece/p_PCG2.png";
        break;

    case -5: //後手 CPG
    case -15:
        pieceImage.src = "img/piece/p_CPG2.png";
        break;

    case -6: //後手 CGP
    case -16:
        pieceImage.src = "img/piece/p_CGP2.png";
        break;

    case -10:
        pieceImage.src = "img/scissors2.png";

    default:
        break;
    }
```

```
    for(var i = 0; i < 1; i++){
        pieceImage.onload = function(){ //指定されたマスに駒を描き込む
            context.drawImage(pieceImage, x, y);
        }
    }

    return context;
}

/*
 * 選択したマスの色変更を行う
 * state：盤面の様子
 */
function drawSelect(state){
    //初回呼び出し時のみキャンバスを作成
    if(!plates.canvasSelect){
        plates.canvasSelect = document.createElement("canvas");
        plates.canvasSelect.width = canvasSize.w;
        plates.canvasSelect.height = canvasSize.h;
    }
    //x、y は選択マス、x1~x4、y1~y4 は初期配置可能エリア
    var x, y, x1, y1, x2, y2, x3, y3, x4, y4 = null;
    var area = {};
    if(state.isSet < 8){ //初期配置設定時
        if(state.turn > 0) area = area1; //先手
        else area = area2; //後手

        x = setSize.x + 10;
        y = (state.target * cellSize) + setSize.y;

        x1 = (area.a % square | 0) * cellSize;
        y1 = (area.a / square | 0) * cellSize;
        x2 = (area.b % square | 0) * cellSize;
        y2 = (area.b / square | 0) * cellSize;
        x3 = (area.c % square | 0) * cellSize;
        y3 = (area.c / square | 0) * cellSize;
        x4 = (area.d % square | 0) * cellSize;
        y4 = (area.d / square | 0) * cellSize;
    }else{ //ゲーム時
        x = (state.target % square | 0) * cellSize;
        y = (state.target / square | 0) * cellSize;
    }

    var context = plates.canvasSelect.getContext("2d");
    context.clearRect(0, 0, canvasSize.w, canvasSize.h);
    context.globalAlpha = 0.5;
    context.fillStyle = "#FFFFFF";
    context.lineWidth = 1;
```

```
	context.beginPath();
	context.fillRect(x, y, cellSize, cellSize);
	if(!state.flag) context.clearRect(0, 0, canvasSize.w, canvasSize.h);
	context.fillRect(x1, y1, cellSize, cellSize);
	context.fillRect(x2, y2, cellSize, cellSize);
	context.fillRect(x3, y3, cellSize, cellSize);
	context.fillRect(x4, y4, cellSize, cellSize);

	return plates.canvasSelect;
}

/*
 * 選択した駒の立体図を表示する
 * state：盤面の様子
 */
function drawDice(state){
	//初回呼び出し時のみ盤面キャンバスを作成
	if(!dices.canvasDice){
		dices.canvasDice = document.createElement("canvas");
		dices.canvasDice.width = diceSize.w;
		dices.canvasDice.height = diceSize.h;
	}

	var context = dices.canvasDice.getContext("2d");
	context.clearRect(0, 0, diceSize.w, diceSize.h);

	if(state.isSet < 8){
		var value = state.list[state.target];
	}else{
		var value = state.board[state.target];
	}
	var boardImage = new Image();
	boardImage.src = "img/dice/plate.png";
	var diceImage = new Image();

	switch(value){
	case 1: //先手 GPC
	case 11:
		diceImage.src = "img/dice/GPC1.png";
		break;

	case 2: //先手 GCP
	case 12:
		diceImage.src = "img/dice/GCP1.png";
		break;

	case 3: //先手 PGC
	case 13:
```

```
        diceImage.src = "img/dice/PGC1.png";
        break;

case 4: //先手 PCG
case 14:
        diceImage.src = "img//dice/PCG1.png";
        break;

case 5: //先手 CPG
case 15:
        diceImage.src = "img/dice/CPG1.png";
        break;

case 6: //先手 CGP
case 16:
        diceImage.src = "img/dice/CGP1.png";
        break;

case -1: //後手 GPC
case -11:
        diceImage.src = "img/dice/GPC2.png";
        break;

case -2: //後手 GCP
case -12:
        diceImage.src = "img/dice/GCP2.png";
        break;

case -3: //後手 PGC
case -13:
        diceImage.src = "img/dice/PGC2.png";
        break;

case -4: //後手 PCG
case -14:
        diceImage.src = "img/dice/PCG2.png";
        break;

case -5: //後手 CPG
case -15:
        diceImage.src = "img/dice/CPG2.png";
        break;

case -6: //後手 CGP
case -16:
        diceImage.src = "img/dice/CGP2.png";
        break;
```

```
        case -10:
            pieceImage.src = "img/scissors2.png";
            break;

        default:
            break;
        }
        context.fillStyle = "#66675F";
        context.fillRect(0, 0, diceSize.w, diceSize.h);
        context.clearRect(5, 5, diceSize.w-10, diceSize.h-10);

        for(var i = 0; i < 1; i++){
            boardImage.onload = function(){ //指定されたマスにプレートを描き込む
                context.drawImage(boardImage, 0, 60);
                if(!state.flag) context.clearRect(0, 0, diceSize.w, diceSize.h);
            }
            diceImage.onload = function(){ //指定されたマスに駒を描き込む
                context.drawImage(diceImage, 34, -9);
                if(!state.flag) context.clearRect(0, 0, diceSize.w, diceSize.h);
            }
        }

        return dices.canvasDice;
}

/*
 * 初期配置駒リストを表示する
 * state：盤面の様子
 */
function drawSet(state){
        //初回呼び出し時のみ初期配置駒リストキャンバスを作成
        if(!sets.canvasSet){
            sets.canvasSet = document.createElement("canvas");
            sets.canvasSet.width = setSize.w;
            sets.canvasSet.height = setSize.h;
        }

        var context = sets.canvasSet.getContext("2d");
        context.clearRect(0, 0, setSize.w, setSize.h);

        context.fillStyle = "#66675F";
        context.fillRect(0, 0, setSize.w, setSize.h);

        //指定したマスに駒を設置
        for(var y = 0; y < 3; y++){
            drawPiece(context, 10, y * cellSize, state.list[y]);
            context.clearRect((cellSize*3/2) - 10, (y * cellSize) + 30.5, 20, 20);
        }
```

```
	if(state.isSet >= 8) complete = true;
	return sets.canvasSet;

}

/*
 * 状態変化表のコピー
 */
function stateCopy(obj){
	return JSON.parse(JSON.stringify(obj));
}
```

- Hand.js

```
Hand = Hand;
Hand.isSet = isSet;
Hand.canSet = canSet;
Hand.canPut = canPut;
Hand.canSelect = canSelect;
Hand.setMap = setMap;
Hand.putMap = putMap;
Hand.isEnd = isEnd;

var square = 6; //マス目
var allCell = square * square; //全てのマスの合計

function Hand(){}

/*
 * 盤面に駒がいくつ置かれているかを返す
 * board: 盤面の様子
 * turn: 手番　1 が先手、-1 が後手、0 が盤面の合計駒数
 */
function isSet(board, turn){
	var piece1 = 0; //先手駒
	var piece2 = 0; //後手駒
	for(var x = 0; x < square; x++){
		for(var y = 0; y < square; y++){
			if(board[y * square + x] > 0) piece1++;
			else if(board[y * square + x] < 0)  piece2++;
		}
	}
	if(turn > 0) return piece1;
	else if(turn < 0) return piece2;
	else return (piece1 + piece2);
}

/*
```

```
 * 選択したマスが初期配置可能エリアであるかどうかの判定
 * number： 選択しているマス
 * turn： 手番　先手が 1、後手が-1
 */
function canSet(number, turn){
    if(turn > 0){
        //先手の初期配置可能エリア
        if(number == square*(square-1) || number == (square*(square-2)+1) ||
            number == square*(square-3) || number == (square*(square-1)+2)){
             return true;
        }
    }
    else{
        //後手の初期配置可能エリア
        if(number == (square-1) || number == (square*(square-4)-2) ||
            number == (square-3) || number == (square*(square-3)-1)){
             return true;
        }
    }
    return false;
}

/*
 * 盤面の選択したマスに自駒が置かれているかどうかの判定
 * board： 盤面の様子
 * number： 選択しているマス
 * turn： 手番　先手が 1、後手が-1
 */
function canSelect(board, number, turn){
    var x = (number % square) | 0; //駒のある行
    var y = (number / square) | 0; //駒のある列

    if (board[y* square + x] * turn > 0) { //確認したマスに自分の駒が置かれていた場合
        return true;
    }
    return false;
}




/*
 * 駒を置けるかどうかの判定
 * board： 盤面の様子
 * turn： 手番　先手が 1、後手が-1
 * number： 置く予定のマス
 * target： 現在選択している駒のマス
```

```
 * startCell：1 手目で行動する駒がもともとあった位置
 * startPiece：1 手目で行動する駒が移動した後の位置
 */
function canPut(board, turn, number, target, startCell, startPiece) {
    var x = (number % square) | 0; //置く予定のマスの行
    var y = (number / square) | 0; //置く予定のマスの列
    var targetX = (target % square) | 0; //選択中の駒のある行
    var targetY = (target / square) | 0; //選択中の駒のある列
    var gapX = Math.abs(x - targetX);
    var gapY = Math.abs(y - targetY);

    if((gapX + gapY) == 1){ //現在選択している駒の前後左右 1 マスを指定している場合
        //指定している駒が出戻り禁止駒で、自エリア内を選択している場合
        if(10 < Math.abs(board[target])){
            if(inArea(board, number, target)) return false;
        }

        if(board[number] * turn > 0) return false; //置く予定のマスに自駒を置いている場合

        if(startPiece == target){
            if(startCell == number) return false; //元々あった位置に駒を動かそうとしている場合
        }

        if(board[number] == 0) return true; //空のマスの場合

        //相手駒が置いてある場合
        if(attack(board[number], board[target])) return true; //自駒勝利時
    }
    return false;
}

/*
 * 指定した駒から見た選択マスの方向を返す
 * number： 駒を置く予定のマス
 * target： 現在選択している駒のマス
 */
function getDirection(number, target){
    var x = (number % square) - (target % square); //行の差
    var y = Math.floor(target / square) - Math.floor(number / square); //列の差
    var direction;

    switch(x){
        case 1:
        if(y == 0) direction = "RIGHT";
        else direction = "ERROR";
        break;

        case -1:
```

```
            if(y == 0) direction = "LEFT";
            else direction = "ERROR";
            break;

            case 0:
            if(y == 1) direction = "UP";
            else if(y == -1) direction = "DOWN";
            else direction = "ERROR";
            break;

            default: direction = "ERROR";
            break;
        }
        return direction;
}

/*
 * 指定したマスが自エリア内かどうかの判定
 * board： 盤面の様子
 * number： 駒を置く予定のマス
 * target： 現在指定している駒のマス
 */
function inArea(board, number, target){
        var x = (number % square) | 0; //置く予定のマスの行
        var y = (number / square) | 0; //置く予定のマスの列

        if(0 < board[target]){ //自駒
                if((x + Math.abs(y-square+1)) < 4) return true; //移動先が自エリア内
        }
        else if(board[target] < 0){ //相手駒
                if((Math.abs(x-square+1) + y) < 4) return true; //移動先が相手エリア内
        }

        return false;
}




/*
 * 指定した駒を指定した場所へ転がした場合の駒の種類を返す
 * board： 盤面の様子
 * target： 指定している駒のマス
 * turn： 手番
 * direction： 転がす方向
```

```
 */
function rollPiece(board, target, turn, direction){
	var value = Math.abs(board[target]); //指定している駒の種類
	var rollValue = 0; //転がした後の駒の種類

	switch(direction){
	case "UP":
	case "DOWN":
		switch(value){
		case  1: //GPC(1) → PGC(3)
		case 11:
			rollValue = (value+2)*turn;
			break;

		case  2: //GCP(2) → CGP(6)
		case 12:
			rollValue = (value+4)*turn;
			break;

		case  3: //PGC(3) → GPC(1)
		case 13:
			rollValue = (value-2)*turn;
			break;

		case  4: //PCG(4) → CPG(5)
		case 14:
			rollValue = (value+1)*turn;
			break;

		case  5: //CPG(5) → PCG(4)
		case 15:
			rollValue = (value-1)*turn;
			break;

		case  6: //CGP(6) → GCP(2)
		case 16:
			rollValue = (value-4)*turn;
			break;

		default: break;
		}
		break;

	case "LEFT":
	case "RIGHT":
		switch(value){
		case  1: //GPC(1) → CPG(5)
		case 11:
```

```
                rollValue = (value+4)*turn;
                break;

            case  2: //GCP(2) → PCG(4)
            case 12:
                rollValue = (value+2)*turn;
                break;

            case  3: //PGC(3) → CGP(6)
            case 13:
                rollValue = (value+3)*turn;
                break;

            case  4: //PCG(4) → GCP(2)
            case 14:
                rollValue = (value-2)*turn;
                break;

            case  5: //CPG(5) → GPC(1)
            case 15:
                rollValue = (value-4)*turn;
                break;

            case  6: //CPG(6) → GPC(3)
            case 16:
                rollValue = (value-3)*turn;
                break;

            default: break;
            }
            break;

        case "ERROR":
            rollValue = -10;
            break;
        }

        return rollValue;
}



/*
 * 指定した駒で攻撃した場合の勝利判定を返す
 * numValue： 攻撃先の駒の種類
 * tarValue： 指定した駒の種類
 */
function attack(numValue, tarValue){
```

```
if(numValue == 0 || tarValue == 0){ //自駒もしくは相手駒が空白の場合
	return false;
}

if(numValue * tarValue > 0){ //攻撃先が自駒だった場合
	return false;
}

switch(Math.abs(tarValue)){
case  1: //グー
case  2:
case 11:
case 12:
	switch(Math.abs(numValue)){
	case  5: //チョキ
	case  6:
	case 15:
	case 16:
		return true;

	default: return false;
	}

case  3: //パー
case  4:
case 13:
case 14:
	switch(Math.abs(numValue)){
	case  1: //グー
	case  2:
	case 11:
	case 12:
		return true;

	default: return false;
	}

case  5: //チョキ
case  6:
case 15:
case 16:
	switch(Math.abs(numValue)){
	case  3: //パー
	case  4:
	case 13:
	case 14:
		return true;
```

```
            default: return false;
            }
        default: return false;
        }
}

/*
 * 開始前のみ、初期配置に駒を置いた後の盤面を返す
 * board： 盤面の様子
 * turn： 手番
 * number： 駒を置く予定のマス
 * target： 現在指定している駒のマス
 * list： 初期配置駒リスト
 */
function setMap(board, turn, number, target, list){
    var boardCopy = board.concat(); //盤面のコピー
    boardCopy[number] = list[target]; //駒の移動を行う
    return boardCopy;
}

/*
 * 石を置いた後の盤面を返す
 * board： 盤面の様子
 * turn： 手番
 * number： 駒を置く予定のマス
 * target: 現在指定している駒のマス
 */
function putMap(board, turn, number, target) {
    var piece = 0; //指定したマスに置く予定の駒
    var boardCopy = board.concat(); //盤面のコピー
    var direction = getDirection(number, target); //移動する方向

    if(boardCopy[number] == 0){ //置くマスが空の場合
        piece = rollPiece(board, target, turn, direction);
    }
    //置くマスに相手駒がある場合
    else if(((-square) <= boardCopy[number] && boardCopy[number] <= square) ||
            ((-square-10) <= boardCopy[number] && boardCopy[number] <= -11) ||
            (11 <= boardCopy[number] && boardCopy[number] <= (square+10))){
        piece = boardCopy[target];
    }

    //自エリアから出戻り禁止エリアに出る時の処理
    if(1 <= Math.abs(piece) && Math.abs(piece) <= square){
        if(!inArea(board, number, target)) piece = (Math.abs(piece)+10) * turn;
    }
    boardCopy[number] = piece; //駒の移動を行う
    boardCopy[target] = 0;
```

```
    return boardCopy;
}

/*
 * ゲームの勝利判定
 * board：盤面の様子
 */
function isEnd(board) {
    if(board[square-1] > 0) return 1; //自駒がゴールにたどり着いた時
    else if(board[square*(square-1)] < 0) return -1; //相手駒がゴールにたどり着いた時

    var flag = 0;
    for (var i = 0; i < allCell; i++) { //駒の有無による勝利判定
        if (flag == 0) {
            if(board[i] > 0) flag = 1;
            else if(board[i] < 0) flag = -1;
        }
        else if(flag * board[i] < 0){ //自駒、相手駒両方が盤面にある時
            return 0;
        }
    }

    if(flag > 0) return 1; //自駒のみ
    else if(flag < 0) return -1; //相手駒のみ

}
```

- Main.html

```
<!DOCTYPE html>
<html>
        <head>
                <meta charset="utf-8">
                <title>ジャンケン将棋</title>
                <link rel="stylesheet" href="css/main.css" type="text/css" media="all">
                <script src="js/board.js"></script>
                <script src="js/game.js"></script>
                <script src="js/hand.js"></script>
        </head>
        <body>
                <h1>ジャンケン将棋</h1>
                <span id="teban">初期配置設定中:　先手番 (駒を白いエリアに配置してください)
<br></span>
                <canvas id="board" width="850" height="540">
                        Canvas が利用できるブラウザを使用してください。
                </canvas><br>
                <span id="winner"></span>
                  <input id="reset" type="button" value="リセット" onclick="Game.reset()">
```

```
                <span id="area">駒の端の色は、それぞれ以下の側面の絵柄と対応しています。
<br>
                  <span id="rock">           </span>  グー
                  <span id="paper">            </span>  パー
                  <span id="sissors">            </span>  チョキ
                </span>
                <span id="configuration">駒の回転: ■をクリック</span>
                <span id="guide">
                        駒の動かし方<br>
                    ・相手のゴールを踏むか、<br>
                      相手の駒を全滅させたら<br>
                      勝ち<br>
                    ・駒は１人２回ずつ動かす<br>
                    ・通常は駒を回転して移動<br>
                      攻撃時はスライドで移動<br>
                      (勝敗はジャンケンに基準)<br>
                    ・一度自分のエリアを出た<br>
                      駒は、再び中に戻ること<br>
                      が出来ない<br>
                    ・自駒が残り１個になると<br>
                      手番が３回に増え、自分<br>
                      のエリアに戻ることがで<br>
                      きるようになる<br>
                </span>
        </body>
```