卒業研究報告書

題目

トレーディングカードゲームにおける公開領域から判断するデッキの予測とそれによるプレイングの変化について

指導教員 石水 隆 講師

報告者

19-1-037-0194

吉田 徹

近畿大学理工学部情報学科

令和６年２月１日提出

概要

「Magic: the Gathering(MtG)」に代表されるトレーディングカードゲーム(TCG)は不完全情報ゲームの１種である.

TCG のプレイングにおける戦略要素の１つとして,相手のデッキ予測がある.これは,場に出ているカードやこれまでに使われたカードなどの情報をもとに相手のデッキ内容を推測し,その推測をもとにどのカードを優先的に使うか,相手のどのカードに対処するべきかを判断するということである.対人戦においては，相手の出したカード相手のデッキを推測し，また相手に自分のデッキを推測されないようにどのカードを出すかが TCG をプレイする面白さの重要な要素である．一方，AI を相手にプレイできる TCG はいくつかあるが，それらの AI の多くはこちらのデッキに関係なく一定の戦略を取り,相手のデッキに合わせて行動を変えることができない．そこでこの判断をエージェントができるようになれば,より人間に近い AI 戦ができるようになるのではと考えた.そこで本研究では，対戦相手のデッキを推定し，相手のデッキに応じた戦略を取る人間らしい動きをする AI の作成を目指す.

目次

1　序論

1.1 本研究の背景

トレーディングカードゲーム(TCG)は,「Magic: the Gathering(MtG)」[7]を初めとし,「ポケモンカードゲーム」[8]や,「遊戯王オフィシャルカードゲーム」[9]などが挙げられる不完全情報ゲームの１種である.

TCG のプレイングにおける戦略要素の１つとして,相手のデッキを予測するというものがある.これは,場に出ているカードやこれまでに使われたカードなどのこれまでに公開された情報をもとに相手のデッキ内容を推測し,その推測をもとにどのカードを優先的に使うか,相手のどのカードに対処するべきかを判断するということである.対人戦においては，相手の出したカード相手のデッキを推測し，また相手に自分のデッキを推測されないようにどのカードを出すかが TCG をプレイする面白さの重要な要素の 1 つである．また,カードゲームによっては３本勝負のルールがあり,一戦一戦の間にサイドボードと呼ばれるカード群とデッキの内容を一部入れ替えることによりチューンナップを行うことができため，相手のデッキに合わせたデッキを構築することがより重要となる.

1.2 TCG に関する既知の結果

TCG は数十年前から幅広くプレイされてきたゲームであり，様々な研究が行われている.

石井と藤田は，TCG におけるサンプリング手法や戦略学習手法を提案している[2][3][4][5]．また，藤井と片寄は石井らの手法に基づく学習機構を作成し，プログラムを作成している[1].また,山田と阿原は MtG においてデッキ作成と対戦の両方を行うエージェントの作成を行なっている[6].

また，TCG は対人戦だけではなく，AI を相手にプレイできるものもいくつかある．代表的なゲームの例として,「遊戯王マスターデュエル」[9]のソロモードや,「デュエル・マスターズプレイス」[10]のストーリーモード,やレジェンドバトルなどが挙げられる.これらはプレイヤーの使用するデッキやプレイングに関わらず一定の戦略を取る.また,「Magic: the Gathering Arena」[11]にも bot 戦が存在するがこれも一定の戦略をとり,また本ゲームでは３本勝負のルールが存在するが,bot 戦では１本勝負しか存在しない.

1.3 本研究の目的

前節で述べた通り，TCG には AI を相手にプレイできるものがいくつかある．しかし既存の AI は，こちらのデッキに関係無く一定の戦略を取り，相手のデッキに合わせて行動を変えることができず，サイドボードを利用して相手のデッキに応じてデッキを組み替えることもできない.

相手のデッキに応じて行動を変えることができれば，より人間に近い行動を取る AI になると考えられる.そこで本研究では，対戦相手のデッキを推定し，相手のデッキに応じた戦略を取る人間らしい動きをする AI の作成を目指す.

1.4 本報告書の構成

本報告書の構成を以下に述べる．まず第 2 章において，本研究の対象である Magic the Gathering (MtG) について説明する．続く第 3 章で本研究の研究内容について述べ，第 4 章で本研究で作成したプログラムについて説明する．第 5 章で結果と考察を述べ，最後に第 6 章で結論と今後の課題を述べる.

2　Magic the Gathering (MtG)

本章では，本研究の対象である Magic the Gathering (MtG)[7] について説明する.

2.1 Magic the Gathering とは

MtG は，魔法使い同士の戦いを描いたトレーディングカードゲームである．プレイヤーは魔法使いとなり，支配する土地から得られるマナと呼ばれるコストを使い，様々なクリーチャーを召喚したり，魔法を使ったりして相手の魔法使いに勝利することを目指す.

MtG の勝利条件は相手のライフを 0 にする,相手が山札を 0 枚にする，相手に毒カウンターを 10 カウント分与えるなどいくつかある.このうち山札を 0 枚にすることは余程ゲームが長引くか,山札を破壊することに特化した特殊なデッキを構築しない限り起こりえず，毒カウンターを与えることも毒を与えることに特化した特殊なデッキでしか起こり得ないので,基本的には相手のライフを 0 にすることを目指して戦うことになる.相手のライフを減らす方法はいくつかあるが，最も使われるのが召喚したクリーチャーで相手を直接攻撃することである．攻撃されたプレイヤーはそれを防ぐために攻撃クリーチャーを自分のクリーチャーでブロックすることができる.

MtG には赤,青,緑,白,黒の 5 つの色が存在し,色により強力なクリーチャーがいたり,相手のクリーチャーを破壊できたり,カードを多く引くことができたりなど,得意な行動が異なる.MtG には各色のコストを生み出す土地カードとそのコストを払って使用するカードの 2 種類がある.土地以外のカードを使用するためには先にコストを用意しなければならないため,まず土地をプレイする必要がある.よって,相手のセットした土地を確認することによってある程度デッキに採用されている色が判明する.

2.2 本研究で用いる MtG のルール

MtG には多くの種類のカードがあり，使用されるルールも多岐にわたる．本研究では簡単のために以下のルールのみを用いる.

① プレイヤーは 2 人,初期ライフは 20,デッキはそれぞれ用意した 3 種類のデッキの中からランダムに 1 つを選択する.デッキの内容はそれぞれ土地 16 枚,クリーチャー,ソーサリーの合計が 24 枚の総数 40 枚で構築されている.カードの種類は,各色クリーチャー5 種類,ソーサリー2 種類,各 2 枚ずつと,土地 1 種類各 12 枚ずつ計 130 枚とする.

② プレイヤーはそれぞれライフとマナというステータスを持つ.各カードはカードタイプ,コスト,色の 3 つのステータスを持ち,さらにカードタイプがクリーチャーの場合,パワーとタフネスの 2 つのステータスが追加される.

③ 先攻後攻をランダムに決定し,各プレイヤーのデッキをシャッフルし,順番をランダムにした後,お互いカードを 7 枚引く.この時,手札が全て土地,もしくは全て土地以外のカードだった場合,もう一度手札をデッキに戻してシャッフルし,カードを 7 枚引く.

④ ターンプレイヤーは場のカードを全てアンタップし,カードを 1 枚引く.

⑤ ターンプレイヤーは手札に土地があれば 1 枚アンタップ状態で戦場に出す.その後,自分のアンタップしている土地枚数以下のコストを持ち,なおかつ使用したいカードの色と対応している土地がアンタップ状態の場合,カードのコスト分だけ土地をタップ状態にし,カードを使用することができる.その際,前述した色と対応している土地を最低 1 枚はタップしなければいけない.使用したカードのカードタイプがクリーチャーだった場合,それは戦場に残り,出たターンの間は召喚酔いしている状態となる.ソーサリーだった場合は墓地に行ったのち,カードの効果を適用する.

⑥ ターンプレイヤーは召喚酔いしていないクリーチャーで相手プレイヤー本体もしくは相手のタップ状態のクリーチャー1 体を攻撃することができる.相手プレイヤーに攻撃した場合,相手は攻撃クリーチャーのパワー分ライフにダメージを受ける.クリーチャーに攻撃した場合,相手クリーチャーのタフネスに攻撃クリーチャーのパワー分,攻撃クリーチャーのタフネスに相手クリーチャーのパワー分のダメージをそれぞれ与える.この時,クリーチャーのタフネスが 0 以下になった場合,そのクリーチャーは破壊され墓地に送られる.

⑦ ターンプレイヤーを交代し,④に戻る.以上の行動を繰り返し,どちらかのライフが 0 になる,もしくはデッキが 0 枚になった場合,そのプレイヤーを敗者とし,ゲームを終了する.

例えば，プレイヤー1 とプレイヤー2 が対戦しており，図 1 のようにプレイヤー1 はパワーとタフネスがそれぞれ 3/1, 2/4/, 1/1 の 3 体のクリーチャーa, b, c，プレイヤー2 はパワーとタフネスがそれぞれ 3/2, 1/4 の 2 体のクリーチャーd, e を召喚している場合を考える．プレイヤー1 が a, b, c 3 体を使って攻撃を宣言し，プレイヤー2 が a を d で防御，b を e で防御したとする．このとき，a と d は互いにパワーに等しい 3 点のダメージを相手に与える．a, d のタフネスは共に 3 未満であるため，a, d は共に破壊されてそれぞれのプレイヤーの墓地に送られる．b は e に 2 点，e は b に 1 点のダメージを与えるが，これは共に両者のタフネスより少ないため，b, e は共に生き残る．クリーチャーc はプレイヤー2 のクリーチャーで防御されていないので，c のパワーに等しい 1 点のダメージをプレイヤー2 のライフに受けることになる．

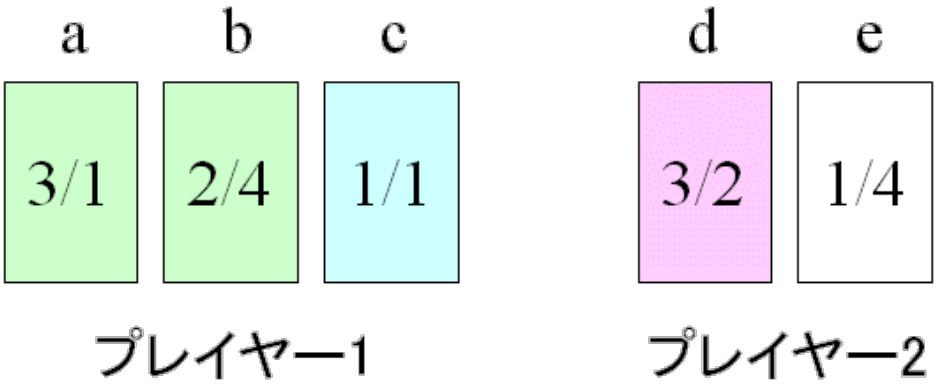

図 1　対戦の例

3　研究内容

前章で述べたように，MtG では相手のデッキを推測し，それに応じた戦略を取る必要がある．しかし既存の AI は相手のデッキに関係無く常に一定の戦略を取り，相手のデッキに応じて戦略を変えることができない．そこで本研究では場に出たカードから相手のデッキを推測し，相手のデッキに応じた戦略を取る MtG の作成を目指す．

本研究では，Java を用いて対戦相手のデッキを推測することができる MtG の AI を作成する．

3.1 本研究で作成する AI の戦略

本研究で作成する AI は，相手が場に出した土地から，相手のデッキで採用されている色を確認する．これにより相手のデッキを推測し,相手のデッキに応じた最適な振る舞いを行うことができると考えられる．

本研究では,対戦相手が使うデッキをそれぞれ赤青白,白黒,赤黒緑の色で構成された 3 種類を用意し,AI が使うデッキは「どの対戦相手にも効果的だが効果量が少ないカード」と「特定の色のカードにしか効果がないが降下量が多いカード」の 2 種類のカードで構成されたものを使用する.これは,色ごとに効果的な対策をする前段階として，色自体を判別して振る舞いを変えることができるかどうか,またそれにより勝率が変わるかどうかを確認するためである.また,対戦相手のデッキには一部同じカードを採用し,すぐにどのデッキタイプか確定できない場合があるようにもする.

4　本研究で作成したプログラム

本研究では Java を用いて相手のデッキを推測する MtG の AI を作成した．付録に本研究で作成したプログラムのソースを示す.

4.1 教師データの記録要素

現時点ではプログラムは未完成である．完成後は文献[6]を参考にニューラルネットワークを用いたエージェントを構築する予定である.また,その際記憶する状態に相手の土地と相手の場に出ているカードの色を含むエージェントと、そうでないエージェントの２つを作成し,相手のデッキごとに使用カードを切り替えられるか,また切り替えられた場合の変化などを確認する予定である.それぞれの記録する予定の内容を表 1 に示す.各状態の内容として

① 自分の手札,各色に土地以外のカード 7 種各 2 枚と土地 12 枚

② 自分のクリーチャー,各色 5 種 2 枚,タップ状態かどうか

③ 対戦相手のクリーチャー,内訳は②と同様

④ 自分の山札の残り,内訳は①と同様

⑤ 自分の土地の状態,各色 12 枚,タップ状態かどうか

⑥ お互いのライフ,20 点,お互い

⑦ プレイしたカードの種類,各色 8 種

⑧ カードをプレイしたかどうか

⑨ アタックしたクリーチャーの種類,各色 5 種

⑩ クリーチャーがアタックしたかどうか

⑪ ブロックしたクリーチャーの種類,各色 5 種

⑫ クリーチャーがブロックしたかどうか

表 1 教師データの記録要素

記録要素	次元数
自分の手札	(7×2+12)×5
自分のクリーチャー	5×5×2×2
対戦相手のクリーチャー	5×5×2×2
自分の山札の残り	(7×2+12)×5
自分の土地の状態	12×5×2
お互いのライフ	20×2
プレイしたカードの種類	40
カードをプレイしたかどうか	1
アタックしたクリーチャーの種類	25

クリーチャーがアタックしたかどうか	1
ブロックしたクリーチャーの種類	25
クリーチャーをブロックしたかどうか	1
合計次元数	713

4.2 プログラムの詳細

本節では，本研究で作成したプログラムの詳細を述べる．

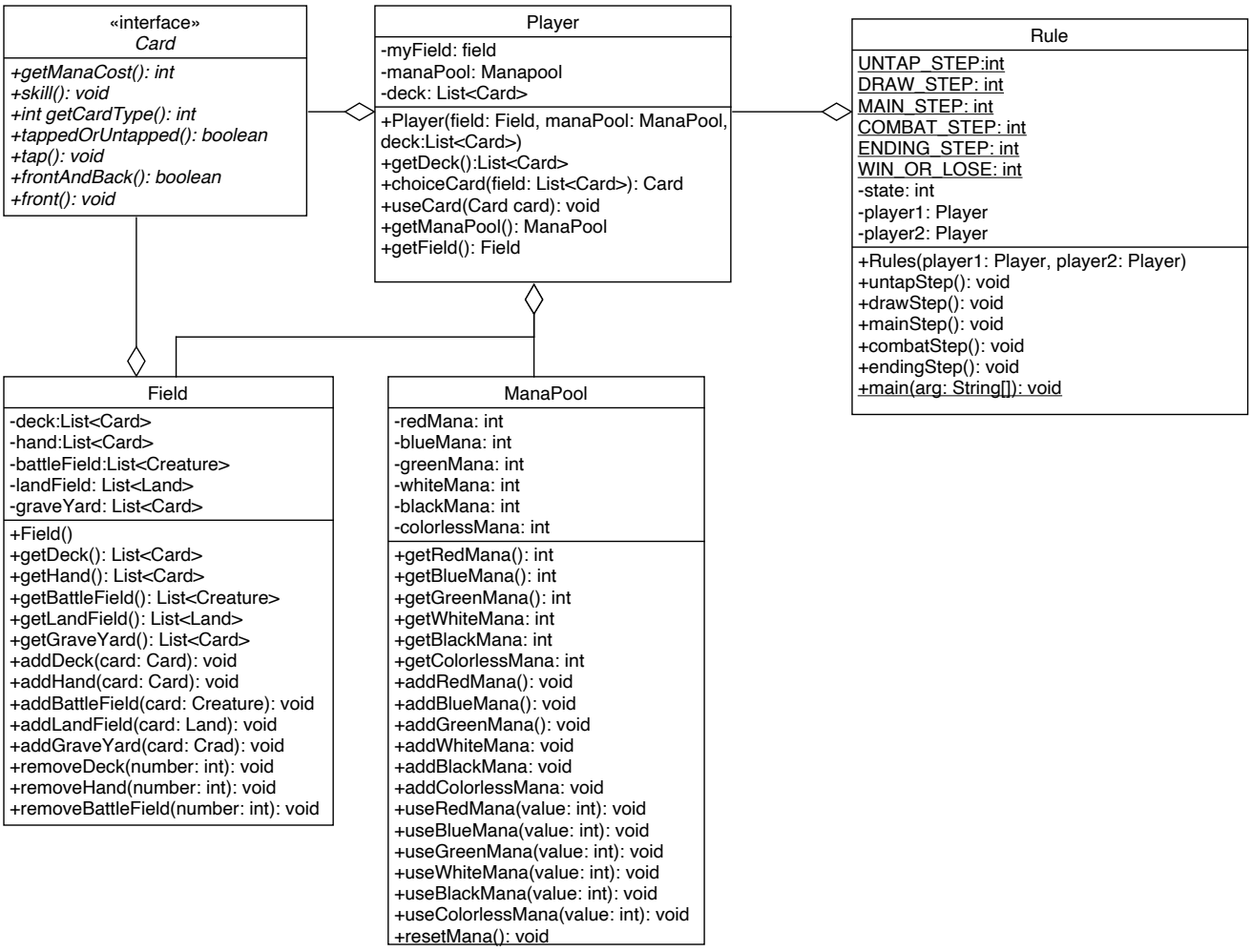

図２：クラス図１

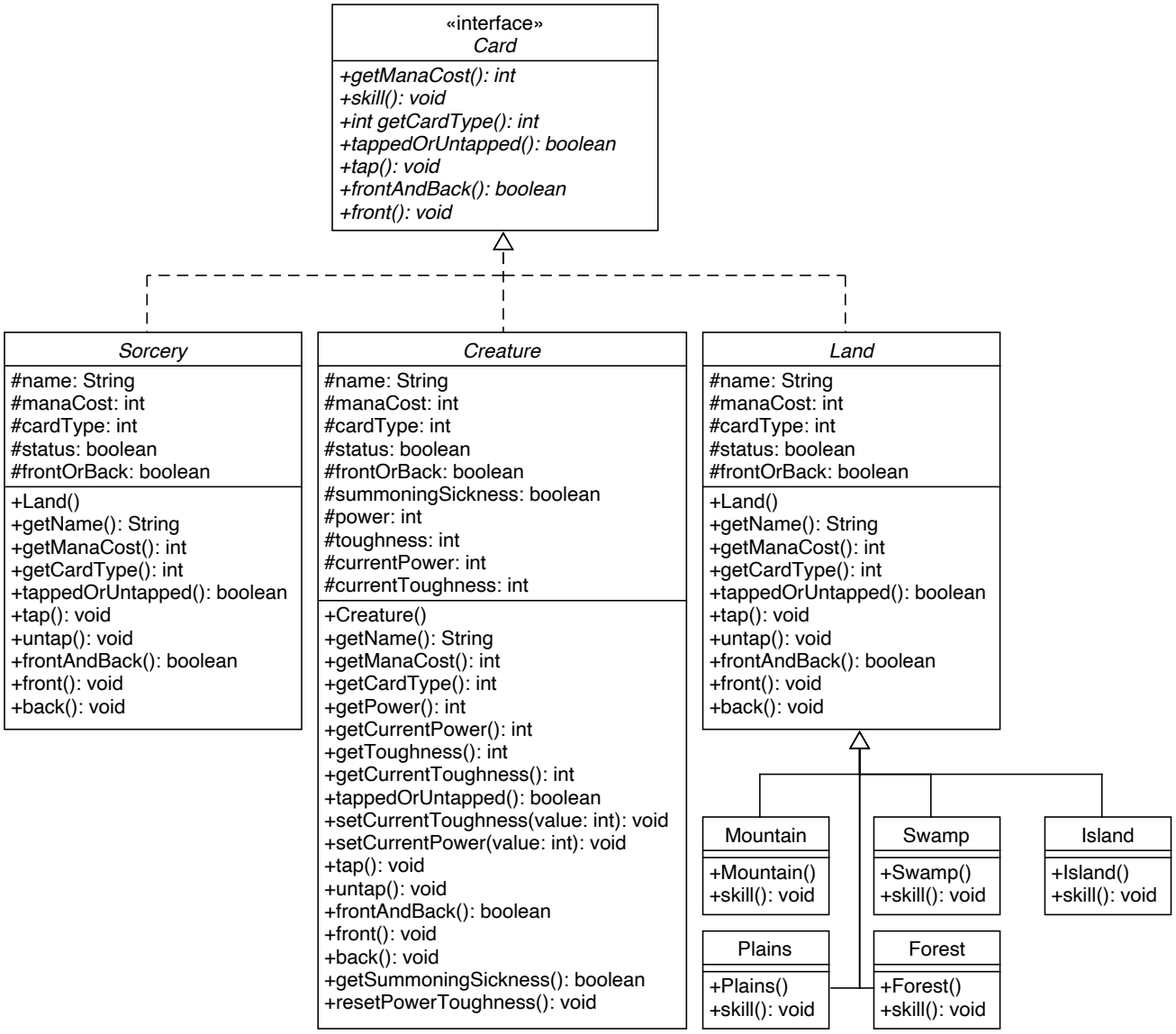

図 3：クラス図その 2

以下に各クラスの説明を述べる

- Card: カード全般のインターフェース
  - メソッド
    - getManaCost(), getName(), getCardType(), tappedOrUntapped(), frontAndBack(): フィールドのゲッター
    - skill(): 特殊能力
    - tap,untap(): タップとアンタップの処理
    - front, back(): カードの表裏切替

- Sorcery, Land: ソーサリーと土地の抽象クラス,メソッド概要は同じなのでまとめる
  - フィールド
    - name: String 型,カード名
    - manaCost: int 型, カードの点数で見たマナコスト. 土地は 0
    - cardType: int カードタイプ,土地は 0,ソーサリーは 2

    - status: boolean 型, カードの位相
    - frontOrBack: boolean 型, カードの表裏
  - メソッド
    - Land(), Sorcery: コンストラクタ
    - getName(), getManaCost(), getCardType(), tappedOrUntapped(), frontAndBack(): フィールドのゲッター
    - tap(), untap(): タップとアンタップの処理
    - front(), back(): カードの表裏切替

- Creature: クリーチャーの土地クラス,Sorcery, Land と同様のものに関しては省略
  - フィールド
  - メソッド

- Mountain, Island, Forest, Plains, Swamp: 基本土地クラス群
  - メソッド
    - skill: それぞれの土地により色マナを出す.順に赤,青,緑,白,黒.

- Field: フィールドのクラス,山札やクリーチャー,土地,墓地の内容を記憶する
  - フィールド
    - deck: List<Card>型, デッキ置き場
    - hand: List<Card>型, 手札
    - battleField: List<Creature>型バトル場
    - landField: List<Land>型, 土地置き場
    - graveyard: List<Card>型, 墓地
  - メソッド
    - getDeck(), getHand(), getBattleField(), getLandField(), getGraveYard(): フィールドのゲッター
    - addDeck(Card card), addHand(Card card), addBattleField(Creature Card), addLandField(Land Card), addFraveYard(Card card): フィールドにカードを追加する
    - removeDeck(int number), removeHand(int number), removeBattleField(int number): フィールドからカードを取り除く

- ManaPool: 各色のマナの量を記憶する
  - フィールド
    - redMana, blueMana, greenMana, blackMana, whiteMana, colorlessMana: int 型, 各色のマナを記憶する.colorlessMana は無色マナを表す
  - メソッド
    - getRedMana(): int 型,フィールドのゲッター.他の全てのフィールドに関してもそれぞれ実装
    - addRedMana(): void 型,フィールドの数を 1 増やす,同様に他の全てのフィールドにもそれぞれ実装
    - usedRedMana(int value): boolean 型, マナを使う場合,マナプールより使いたいマナが多ければ false,少なければその分だけマナを減らし,true を返す.各色に実装
- Player
  - フィールド
    - myField: Field 型, 自分の場
    - manaPool: ManaPool 型, 自分のマナプール
    - deck: List<Card>型, 自分のデッキ
  - メソッド

- getField(), getManaPool(), getDeck(): フィールドのゲッター
- setField(): フィールドのセッター
- draw(): boolean 型, カードを引く処理,引ければ true,引けなければ false を返す
- choiceCard(List<Card> field): Card 型,カードリストの中からカードを選んで返す
- useCard(Card card): void 型,カードの使用処理.カードタイプが土地なら landField に,クリーチャーなら battleField に,ソーサリーなら効果処理後に graveYard に送る

- Rules: ルール全般のクラス,またシミュレーションの実行もこのクラスで行う
  - フィールド
    - player1: Player 型, プレイヤー1 人目
    - player2: Player 型, プレイヤー2 人目
  - メソッド
    - untapStep(): void 型, アンタップの処理を行う
    - drawStep(): void 型, ドローの処理を行う
    - mainStep(): void 型, メインステップの処理,土地を置く,呪文を唱える等の処理を行う
    - combatStep(): void 型, 先頭ステップの処理,ターンプレイヤーの攻撃クリーチャーの選択や、非ターンプレイヤーの防御クリーチャーの選択,またそれに伴う破壊などの処理を行う
    - endingStep(): void 型, ターンプレイヤーの交代,クリーチャーのステータスのリセットを行う
    - main(String[] argss): void 型, シミュレーションの実行を行う

5 結果と考察

現時点ではゲームのプログラムは未完成である．カード,プレイヤー,ルールのクラスの内,戦闘タイミングのルール以外の実装は完了しているが,戦闘ルールや個別のカードの実装がまだできていない.

6 結論・今後の課題

本研究では相手の色を推測する MtG の AI を作成した。この AI を用いることで，相手のデッキの色を推測でき，相手の色に応じた行動を取ることができるようになる.

今後の課題としてはまず未完成のプログラムとエージェントの構築を完了させることである.次に,色ごとの対策ができれば,カードごとの対策ができるようにしたい.その時その時で大会で特に使用されているデッキを大会結果や統計サイトなどから取得し,対戦させることでどのカードを出されると勝率が下がるか,逆に自分がどのカードを出せば勝率が上がるかなどを推測することでより効果的なカードの使用法を割り出せるエージェントを構築できると考える.

また,今回作成したプログラムではカードタイプが 3 種類しか実装できていない.本来 MtG では 7 種類のカードタイプが存在するため,そのカードらも戦略に組み込みたい

謝辞

指導を受けた教員や，本研究を完成するにあたって支援を受けた研究室の諸氏に対しお礼の言葉を，独立したページに記述する．詳しくは指導教員に尋ねること．

参考文献

[1] 藤井叙人，片寄晴弘：戦略型トレーディングカードゲームのための戦略獲得手法，情報処理学会論文誌 藤田 肇，石井 信：マルチエージェントカードゲームのための強化学習法の改良，電子情報通信学会技術研究報告，Vol.102, No.731, pp.167–172 (2003).
https://www.ieice.org/publications/ken/summary.php?contribution_id=KJ00001015516&society_cd=ISS&ken_id=NC&year=2003&presen_date=2003/3/12&schedule_id=AN10091178_102(731)&lang=jp&expandable=3

[2] Ishii, S. and Fujita, H.: A Reinforcement Learning Scheme for a PartiallyObservable Multi-Agent Game, Machine Learning, Vol.59, pp.31–54 (2005).
https://link.springer.com/article/10.1007/s10994-005-0461-8

[3] Fujita, H. and Ishii, S.: Model-Based Reinforcement Learning for Partially Observable Games with Sampling-Based State Estimation, Neural Computation, Vol.19, pp.3051–3087 (2007).
https://ieeexplore.ieee.org/document/1245625

[4] 山田豊大，阿原一志：トレーディングカードゲームにおけるデッキ作成とエージェント構築を目標としたニューラルネットを用いた学習モデルの検討，The 23rd Game Programming Workshop, pp.128-132, 情報処理学会 (2018), http://id.nii.ac.jp/1001/00191976/

[5] MAGIC THE GATHERING 日本公式ウェブサイト, https://mtg-jp.com/

[6] ポケモンカードゲーム トレーナーズウェブサイト，https://www.pokemon-card.com/

[7] 遊戯王オフィシャルサイト，https://yu-gi-oh.jp/

[8] 遊戯王マスターデュエル 公式ウェブサイト, https://www.konami.com/yugioh/masterduel/jp/ja/

[9] DUEL MASTERS PLAY'S 公式ウェブサイト, https://dmps.takaratomy.co.jp/

[10] MTG ARENA MAGIC THE GATHERING 日本公式ウェブサイト,https://mtg-jp.com/mtgarena/

付録

本研究で作成したプログラムを以下に示す.

```
public interface Card {

   public int getManaCost() ; // マナコスト取得
   public void skill(); // 特殊能力
   public String getName(); //
   カード名取得 public int getCardType(); //
   カードタイプ取得 0→土地、1→クリーチャー、2→ソーサリー
   public boolean tappedOrUntapped(); //タップされているかどうか、1 でアンタップ、0 でタップ
   public void tap(); // タップ処理
   public void untap(); // アンタップ処理
   public boolean frontAndBack(); // カードの表裏の状態、1 で表、0 で裏
   public void front(); //表向きにする public void back(); // 裏向きにする
}

public abstract class Creature implements Card {

        protected String name; // カード名
        protected int manaCost; // 点数で見たマナコスト
        protected int cardType; // カードタイプ（クリーチャーは 1)
        protected boolean status; // カードの位相
        protected boolean frontOrBack; // カードの表裏
        protected boolean summoningSickness; // 召喚酔いしてるかどうか　0 でしている
        protected int power; //クリーチャーのパワー
        protected int toughness; // クリーチャーのタフネス
        protected int currentPower;
        protected int currentToughness;

        /**
         * クリーチャー全般のコンストラクタ
         * カードタイプは土地(1)、マナコストはカード毎
         * 初期状態はアンタップ状態で裏向き
         */
        public Creature() {
                this.cardType = 1;
                this.status = true;
                this.name = "";
                this.frontOrBack = false;
                this.summoningSickness = false;
        }

        public String getName() {
                return name;
        }
```

```
public int getManaCost() {
	return manaCost;
}

public int getCardType() {
	return cardType;
}

public int getPower() {
	return power;
}

public int getCurrentPower() {
	return currentPower;
}

public int getToughness() {
	return toughness;
}

public int getCurrentToughness() {
	return currentToughness;
}


public boolean tappedOrUntapped() {
	return status;
}

public void setCurrentToughness(int value) {
	currentToughness = value;
}

public void setCurrentPower(int value) {
	currentPower = value;
}

public void tap() {
	status = false;
}

public void untap() {
	status = true;
}

public boolean frontAndBack() {
	return frontOrBack;
}
```

```
	public void front() {
		frontOrBack = true;
	}

	public void back() {
		frontOrBack = false;
	}

	public boolean getSummoningSickness() {
		return summoningSickness;
	}

	public void resetPowerToughness() {
		currentPower = power;
		currentToughness = toughness;
	}



}

public abstract class Land  implements Card{
	protected String name; // カード名
	protected int manaCost; // 点数で見たマナコスト
	protected int cardType; // カードタイプ（土地は 0）
	protected boolean status; // カードの位相
	protected boolean frontOrBack; // カードの表裏

	/**
	 * 土地全般のコンストラクタ
	 * カードタイプは土地(0)、マナコストは 0
	 * 初期状態はアンタップ状態で裏向き
	 */
	public Land() {
		this.cardType = 0;
		this.manaCost = 0;
		this.status = true;
		this.name = "";
		this.frontOrBack = false;
	}

	public String getName() {
		return name;
	}

	public int getManaCost() {
		return manaCost;
```

```
	}

	public int getCardType() {
		return cardType;
	}

	public boolean tappedOrUntapped() {
		return status;
	}

	public void tap() {
		status = false;
	}

	public void untap() {
		status = true;
	}

	public boolean frontAndBack() {
		return frontOrBack;
	}

	public void front() {
		frontOrBack = true;
	}

	public void back() {
		frontOrBack = false;
	}


}

public class Mountain extends Land {

	public Mountain() {
		super();
		name = "山";
	}

	public void skill() {

	}
}

public class Island extends Land {

	public Island() {
```

```
		super();
		name = "海";
	}

	public void skill() {

	}
}

public class Forest extends Land {
	public Forest() {
		super();
		name = "森";
	}

	public void skill() {

	}
}

public class Plains extends Land {

	public Plains() {
		super();
		name = "平地";
	}

	public void skill() {

	}
}

public class Swamp extends Land {

	public Swamp() {
		super();
		name = "沼";
	}

	public void skill() {

	}
}

public class Field {

	private List<Card> deck;
	private List<Card> hand;
```

```
private List<Creature> battleField;
private List<Land> landField;
private List<Card> graveYard;

public Field() {
	this.deck = new ArrayList<Card>();
	this.hand = new ArrayList<Card>();
	this.battleField = new ArrayList<Creature>();
	this.landField = new ArrayList<Land>();
	this.graveYard = new ArrayList<Card>();
}

public List<Card> getDeck(){
	return deck;
}

public List<Card> getHand(){
	return hand;
}

public List<Creature> getBattleField(){
	return battleField;
}

public List<Land> getLandField(){
	return landField;
}

public List<Card> getGraveYard(){
	return graveYard;
}

public void addDeck(Card card) {
	card.back();
	deck.add(card);
}

public void addHand(Card card) {
	card.front();
	hand.add(card);
}

public void addBattleField(Creature card) {
	card.front();
	battleField.add(card);

}
```

```
	public void addLandField(Land card) {
		card.front();
		landField.add(card);
	}

	public void addGraveYard(Card card) {
		card.front();
	}

	public void removeDeck(int number) {
		deck.remove(number);
	}

	public void removeHand(int number) {
		hand.remove(number);
	}

	public void removeBattleField(int number) {
		hand.remove(number);
	}
}

public class ManaPool {
	private int redMana;
	private int blueMana;
	private int greenMana;
	private int blackMana;
	private int whiteMana;
	private int colorlessMana;

	public ManaPool() {
		this.redMana = 0;
		this.blueMana = 0;
		this.greenMana = 0;
		this.whiteMana = 0;
		this.blackMana = 0;
		this.colorlessMana = 0;
	}


	/**
	 * 以下フィールドのゲッター
	 */
	public int getRedMana() {
		return redMana;
	}

	public int getBlueMana() {
```

```
		return blueMana;
}

public int getGreenMana() {
		return greenMana;
}

public int getBlackMana() {
		return blackMana;

}

public int getWhiteMana() {
		return whiteMana;
}

public int getColorlessMana() {
		return colorlessMana;
}

/**
 * フィールドの値を増やす
 */
public void addRedMana() {
		redMana += 1;
}

public void addBlueMana() {
		blueMana += 1;
}

public void addGreenMana() {
		greenMana += 1;
}

public void addBlackMana() {
		blackMana += 1;
}

public void addWhiteMana() {
		whiteMana += 1;
}

public void addColorlessMana() {
		colorlessMana +=1;
}

/**
```

```
	 * マナを使う場合、マナプールより使いたいマナが多ければ false,使いたいマナが少なければその分だけマナを減らし、true を返す
	 * @param value
	 * @return
	 */
	public boolean useRedMana(int value) {
		if(value <= redMana) {
			redMana -= value;
			return true;
		} else {
			return false;
		}
	}

	public boolean useBluedMana(int value) {
		if(value <= blueMana) {
			blueMana -= value;
			return true;
		} else {
			return false;
		}
	}

	public boolean useGreenMana(int value) {
		if(value <= greenMana) {
			greenMana -= value;
			return true;
		} else {
			return false;
		}
	}

	public boolean useBlackMana(int value) {
		if(value <= blackMana) {
			blackMana -= value;
			return true;
		} else {
			return false;
		}
	}

	public boolean useWhiteMana(int value) {
		if(value <= whiteMana) {
			whiteMana -= value;
			return true;
		} else {
			return false;
		}
```

```
	}

	/**
	 * マナプールのリセット
	 */
	public void resetMana() {
		redMana = 0;
		blueMana = 0;
		greenMana = 0;
		whiteMana = 0;
		blackMana = 0;
		colorlessMana = 0;
	}

}

public class Player {
	private Field myField;
	private ManaPool manaPool;
	private List<Card> deck;

	public Player(Field field, ManaPool manaPool, List<Card> deck) {
		this.myField = field;
		this.manaPool = manaPool;
		this.deck = deck;
	}

	public List<Card> getDeck(){
		return deck;
	}

	public boolean draw() {
		if(myField.getDeck().size() > 0) {
			Card card = myField.getDeck().getFirst();
			myField.addHand(card);
			myField.getDeck().removeFirst();
			return true;
		} else {
			return false;
		}
	}

	public Card choiceCard(List<Card> field) {
		return null;
	}

	public void useCard(Card card) {
```

```
	}

	public ManaPool getManaPool() {
		return manaPool;
	}

	public Field getField() {
		return myField;
	}
}

public class Rules {

	final static int UNTAP_STEP = 0;
	final static int DRAW_STEP = 1;
	final static int MAIN_STEP = 2;
	final static int COMBAT_STEP = 3;
	final static int ENDING_STEP = 4;
	final static int WIN_OR_LOSE = 5;

	int state = UNTAP_STEP;


	private Player player1;
	private Player player2;

	public Rules(Player player1, Player player2) {
		this.player1 = player1;
		this.player2 = player2;
	}


	public void untapStep() {
		for(int i = 1; i <= player1.getField().getBattleField().size(); i++) {
			player1.getField().getBattleField().get(i).untap();
		}
		for(int i = 1; i <= player1.getField().getLandField().size(); i++) {
			player1.getField().getLandField().get(i).untap();
		}
		for(int i = 1; i <= player2.getField().getBattleField().size(); i++) {
			player2.getField().getBattleField().get(i).untap();
		}
		for(int i = 1; i <= player2.getField().getLandField().size(); i++) {
			player2.getField().getLandField().get(i).untap();
		}
		state = DRAW_STEP;
	}
```

```
	public void drawStep() {
		if(player1.draw()) {
			state = MAIN_STEP;
		} else {
			state = WIN_OR_LOSE;
		}
	}

	public void mainStep() {

	}

	public void combatStep() {

	}

	public void endingStep() {
		for(int i=1; i<= player1.getField().getBattleField().size(); i++) {
			player1.getField().getBattleField().get(i).resetPowerToughness();
		}
		for(int i=1; i<= player2.getField().getBattleField().size(); i++) {
			player2.getField().getBattleField().get(i).resetPowerToughness();
		}

		Player player = player1;
		player1 = player2;
		player2 = player;
		state = UNTAP_STEP;
	}
}
```