An Agent For Playing A Customizable Card Game

JOSEPH F. GRECO, Carnegie Mellon University

I developed an agent to can play a simplified version of the Star Trek Customizable Card Game (STCCG) collectible deck-building card game. This required the creation of an environment providing a game-state and the listing of all possible (rule-following) actions that the agent can carry out in that state, with their appropriate effects. An abstracted, simplified version of the game was necessary to limit the unique actions possible. After building enough of the environment, an agent was developed, heuristics were implemented to improve the agent's efficiency, and changes were made in the implementation to better support the agent's evaluation function.

Categories and Subject Descriptors: **I.2.1 [ARTIFICIAL INTELLIGENCE – Applications and Expert Systems]**: Games

General Terms: Design, Algorithms

Additional Key Words and Phrases: Game modeling, game abstraction, search, game trees, alpha beta search, iterative deepening

ACM Reference Format:

Joseph F. Greco, 2013. An Agent For Playing A Customizable Card Game. *Unpublished.* DOI: None

1. INTRODUCTION

The Star Trek Customizable Card Game (STCCG) is a collectible deck-building card game based on elements from the Star Trek series of television shows and movies. After the successful first edition (1994-2006) of the game grew increasingly complex, a streamlined second edition (2E) was released and expanded from 2002 to 2007 [1]. An official online version of the games was available, but was essentially a player versus player chatroom "sandbox" environment that did not attempt to enforce game rules.

A similar-in-concept game is the popular Magic: The Gathering. Some work has been done with comparing gameplay strategies' effectiveness in 2009[2]. The experimental design seemed an appropriate model for use with a game such as STCCG 2E (specifically, the setup of the test environment and decision separation, acknowledging incomplete knowledge).

The goal of this project was to create an agent to play a simplified version of the STCCG 2E. This required the creation of an environment providing a game-state and possible (rule-following) actions and effects that the agent can manipulate. A simplified version is necessary to limit the unique actions possible. After the creation of the environment, an agent was developed to manipulate the environment intelligently.

2. GAME OVERVIEW

In the game, each player has three groupings of cards:

- A set of 5 mission cards representing locations in the universe. These locations are either planet missions or space missions, solved to score points, or headquarters missions that are used for reporting compatible cards from hand. These are placed on the table at the beginning of the game.
- A draw deck of at least 35 cards of the following card types:
    - o personnel – a character who has skills and attributes to solve missions

- ships – a starship capable of moving personnel and equipment
- equipment – a piece of equipment such as a tricorder, typically supplementing the skills/attributes of some personnel present
- event – plays during the player's turn to have some effect on the universe
- interrupt – plays in response to almost any action of any player, typically affecting that action

- A dilemma pile of at least 20 dilemmas. The player uses these to attempt to stymie attempts by another to solve missions.

Players draw a hand of cards from their draw deck, then start taking alternating turns of the following structure:

1. Play and Draw Cards: seven counters allow playing the (varying-cost) cards from hand or drawing additional cards from the deck
2. Execute Orders: beaming personnel and equipment between locations/ships, moving ships between missions, attempting missions, miscellaneous special actions on cards
    a. When missions are attempted, the opponent draws and chooses dilemmas to be encountered by the attempters, ideally requiring skills and attributes not present.
3. Discard Excess Cards: discard (choose any) cards over seven from hand.

The game ends when draw decks have been exhausted, and victory is determined based on missions solved and points scored.

3. EXISTING WORK

The Continuing Committee, an unofficial group of Star Trek enthusiasts, emerged after the discontinuation of the games, providing a community for organized play as well as future "virtual expansions" of the games. Various online methods of play (non-rule-enforcing) have been developed and maintained by the group, as well as a database of card metadata. This database of card information is already somewhat machine-readable; with additional modifications it was useful as a starting point for providing the data necessary to populate the environment I created. Modifications were made to database file to be imported to support all the information and in the format needed. The collection of card images was also useful for the debugging server I developed.

4. GAME ENVIRONMENT & ABSTRACTION

I picked Java to develop this project in due to my familiarity with the language and development environment, as well as the portability of the code for future purposes.

The code is divided into several groupings of classes:

4.1 Constants

Constants – these are used throughout the game code to consistently represent symbols and classifications.

4.2 Card database

These classes used to create a database of the unique game cards with their attributes and information

- Card – implementing common card characteristics, such as id, game text, title, set identification and number, rarity, type, and uniqueness. Various String outputs are implemented for debugging purposes, as well as an equals method.
- Dilemma – representing a dilemma card, extending Card with mission type (planet/space) and cost information.
- Equipment – representing an equipment card, extending Card with keywords (relevant for some cards) and cost information.
- Event – representing an event card, extending Card with keywords (relevant for some cards) and cost information.
- Interrupt – representing an interrupt card, extending Card with keywords (relevant for some cards). No cost information is needed as interrupts do not have a numeric cost.
- Mission – representing a mission card, extending Card with the mission type, skills needed to solve, keywords, affiliations able to attempt, span, quadrant, and, most importantly, points. A method in this is called at initiation to place the skills read in from the data file to table form.
- Personnel – representing a personnel card, extending Card with the card subtitle (lore text but relevant to gameplay), cost, skills, keywords, affiliation, species, standard values (integrity, cunning, and strength), and special icons.
- Ship – representing a ship card, extending Card with the card subtitle (lore text but relevant to gameplay), cost, keywords, affiliation, ship classification, standard values (range, weapons and shields), and special icons.

4.3 Cards in play

These wrappers representing cards actually in play (including groupings) with any attribute modifiers or state information, referencing the templates to save space. Only some card types were implemented with matching wrappers:

- CardInPlay – implements common card characteristics, with a link to the template from the card database, the owner, and number the card is in the player's deck. Common methods to be overridden and called by the below classes include methods for performing start and end of turn actions, comparing and cloning cards, and methods for debugging.
- Pile – a class containing an ArrayList of CardInPlay cards, with methods for taking cards, peeking at cards, randomizing cards, as well as gathering information aggregated over the cards.
- MissionInPlay – extending CardInPlay, implements some of the below interfaces as well as piles for cards of ships in orbit or personnel on the surface of the planet, if applicable.
- EquipmentInPlay, PersonnelInPlay, ShipInPlay, extending CardInPlay and implementing some of the interfaces below.

4.4 Card interfaces

These are implemented by cards in play, assisting with the organized determination of possible actions, as rather than referencing particular card types, casting can be used for convenience with certainty that necessary methods are implemented.

- Beamable – methods a card must implement if it is able to be transported from a location to another, such as being given a location and determining if it is suitable, as well as actually beaming.
- Costs – methods a card must implement if counters are used on play, such as getting the cost and uniqueness (as uniqueness limits the playability of a card).

- Moveable – typically ships, methods a card must implement for moving, such as determining if a ships is staffed for movement, remaining range for movement, and actually moving.
- Oxygen – methods a card must implement if it is a location that cards can be beamed to. This includes receiving or departing a personnel, as well as performing aggregate information gathering for use by action determination.
- Reportable – methods a card must implement if it can be played from hand to the headquarters card, such as reporting the card or discarding it.
- Stoppable – methods a card must implement if it can be stopped during a turn, such as stopping, unstopping, and getting stopped status.

4.5 Game classes

These objects represent the game state, modeling players (including the decks, table cards, and hand), as well as logic for determining legal actions based on the game state.

- Action – a container for an action, including an action description "method," optional cards affected, and optional pile relevant.
- GameState – the main game container class, handling start and end of game, turn logic, and the game loop. Supports console, GUI, and server debug modes.
- GameWrapper – sets up the card database and reads in player information to create the game state.
- Main – simple class to create a GameWrapper.
- Player – maintains all the piles of the player (draw deck, hand, discard pile, missions, etc) as well as legal action possibilities generation as well as action interpretation.

4.6 Search program

These classes are used to represent the game and do the game tree searching.

- AlphaBetaSearch – code from the AIMA website implementing alpha beta search as described in the textbook
- CutoffAlphaBetaSearch – based on code from AlphaBetaSearch, modified based on the textbook-described needed changes for iterative deepening.
- Game – code from the AIMA website of the interface needed to be implemented for the representation of a game used for search.
- MinmaxSearch – code from the AIMA website implementing min max search as described in the textbook
- Searcher – invoked as a thread from GameState, used to call CutoffAlphaBetaSearch to highest depth possible until stopped.
- STGame – an interface implementing Game, reading GameState to provide the information needed for the search, including the evaluation function.

4.7 Server

The server was used for debugging graphically, allowing for action possibilities to be verified for correctness by manually examining card information

- Servlet – interfaces with the GameWrapper to support manipulating the game environment through a web browser.
- JSP & Images – used for rendering the game environment.

5. IMPLEMENTATION

Starting from the online card database, modifications were made to support reading this file through the OpenCSV library, as well as to clean up and insert new fields as needed. Creating the classes to handle the card database mostly mapped to those fields, though as implementation went on the fields were revised to store additional data.

Work on the classes for the card wrappers and interfaces was mostly simultaneous. As methods were determined to be needed, the interfaces they would belong in and supporting methods in the card wrappers were created. The creation of a Pile class also assisted in allowing for methods to be performed across multiple cards at once. Only the standard parts of Missions, Ships, and Personnel were implemented, with special skills and abilities unimplemented, to keep the game simple and devote more time to search.

The game classes were challenging to implement, as the game logic is complicated and actions varied. Development of a console-based interface, allowing selection from a generated list of legal actions based on state, was completed first. However, debugging the game logic was difficult as state printouts were long and difficult to read.

A GUI was created to better present the state of the game for logic debugging. However, this was also problematic, as verification required knowing and comparing card details. A simple Java Tomcat servlet was developed, delivering the complete state as HTML and allowing for action selection and game control from the page.

6. SEARCH

Building on code from the AIMA site, an interface between the gamestate for using the Minimax search was created. This interface implemented methods to provide a model of the game:

- `STATE getInitialState();`
- `PLAYER[] getPlayers();`
- `PLAYER getPlayer(STATE state);`
- `List<ACTION> getActions(STATE state);`
- `STATE getResult(STATE state, ACTION action);`
- `boolean isTerminal(STATE state);`
- `double getUtility(STATE state, PLAYER player);`
- `boolean isCutoff(STATE state, int depth);`
- `double getEstimatedUtility(STATE state, PLAYER player);`

These methods allowed for the search to access all the information needed to build the trees for search for the game. Getting the current player enabled the multiple moves by a single player in series to be supported.

Alpha Beta Pruning search was then substituted, and finally a custom version of Alpha Beta Search using iterative deepening with a time limit (5 seconds was selected) was developed. A move ordering heuristic was implemented to allow Alpha Beta Search to work more efficiently, as well as additional actions such as "beam all" eligible personnel from one location to another, allowing for multiple similar actions to be played in one move, essentially getting deeper in the prior search trees faster.

The evaluation function for the search went through multiple iterations, before a simple function incorporating domain-specific knowledge, prioritizing points, personnel/staffed ships with personnel meeting missions requirements at unsolved missions, staffed ships at the headquarters, and finally personnel (using the cost metric as an estimate of value) at the headquarters:

```
public double getEstimatedUtility(GameState state, Integer player) {
        // most importantly, if terminal
        if (isTerminal(state)) {
                return getUtility(state, player); // simply returns points
        }

        // start a counter
        double tempUtility = 0;
        // get the current player
        Player p = state.players.get(player);
        // add points of missions so far
        tempUtility += p.points;

        for (CardInPlay mc: p.missions.pileCards){
                // for each mission
                MissionInPlay m = (MissionInPlay) mc;
                double missionUtility = 0.0;
                for (ShipInPlay sc: m.getShipsHere()){
                        double shipUtility = 0.0;
                        shipUtility +=sc.getPrintedCost();
                        shipUtility +=
(double)sc.getPersonnelHere().size()/(double)sc.getPrintedCost();
                        shipUtility += (double)sc.deckNum*0.00001; // attempt to
differentiate between copies of same ship by using a unique value, prevent
oscillation between same-scored states
                        for (PersonnelInPlay sp: sc.getPersonnelHere()){
                                shipUtility += sp.getPrintedCost();
                        }
                        // if mission solvable by ship, ship gets bonus
                        if
((!m.isHeadquarters())&&(!m.isSolved())&&(m.canSolve(sc))){
                                shipUtility *= 2.0;
                        }
                        // if mission already solved, gets less, not useful here
                        if
((!m.isHesadquarters())&&((m.isSolved())||(!m.canSolve(sc)))){
                                shipUtility *= 0.75;
                        }
                        missionUtility += shipUtility;
                }
                if (m.hasPlanet()){
                        double personnelUtility = 0.0;
                        for (PersonnelInPlay sp: m.getPersonnelHere()){
                                personnelUtility += sp.getPrintedCost();
                        }
                        // if mission solvable by personnel, gets bonus
                        if
((!m.isHeadquarters())&&(!m.isSolved())&&(m.canSolve(null))){
                                personnelUtility *= 3;
                        }
                        // if mission already solved, personnel get less
                        if
((!m.isHeadquarters())&&((m.isSolved())||(!m.canSolve(null)))){
                                personnelUtility *= 0.75;
                        }
                        missionUtility += personnelUtility;
                }
                tempUtility += (missionUtility*0.25);
        }
        return tempUtility;
}
```

7. RESULTS

During testing, run with both opponents using the same search program (Alpha beta search with an initial evaluation function), the players would often draw without a clear winner. After revising the evaluation function, games between players running the same search were always played to a winner. Limited testing was performed with the program against a player modeled to always perform random selection of actions (initially developed for use during correctness testing of the game abstraction); the search program always won.

From the web interface, the search program could be launched from the current state to perform the best next move, or moves until the end of the current player's turn. The program was also always invoked for the opposing player's turn. In practice, a competent human player that did not delay could handily beat the program. If the human player "skipped" a few turns by making neutral in value moves, the computer player could win. This was due to depth of the search tree, as well as the simple evaluation function; while the computer player would always make the best moves as determined by the search program, the evaluation function's estimate was not as focused on individual goal accomplishment as having a good position to rapidly accomplish goals in series, which, if the game had not been lost yet, it was able to do.

8. CONCLUSION

The game abstraction was able to accurately modify a simplified version of the game, and the interface developed to the search algorithm allowed for a reasonable computer opponent.

9. FUTURE WORK

I did not get as far as I had planned to, as the complexity of the rules of the game made implementation of the game itself more difficult than anticipated. There is still room for significant improvements to the game and search.

The game implementation itself would benefit from implementation of equipment, dilemmas, and special actions. Dilemmas and special actions are particularly challenging, as some sort of interpreted language for determining action applicability and effects operating against the game state would be preferable to hard-coding for each individual card.

The search implementation would benefit from the implementation of chance nodes, moving the game from a fully observable game to a partially observable game, as otherwise the agent is able to "cheat" as it is currently. Precomputing ideal dilemma combinations or the equivalent of "opening books" would also be a possible direction.

The evaluation function was more tuned to having a player be in a generally good position, which would be more beneficial were dilemmas (requiring personnel skills not as easily anticipated by a human as it could be by a computer that can plan for likely possibilities) implemented, than focused narrowly on reporting personnel specifically useful for missions as human players (correctly) would tend to.

REFERENCES

[1] Wikipedia, Star Trek Customizable Card Game, https://en.wikipedia.org/wiki/Star_Trek_Customizable_Card_Game
[2] C. D. Ward and P. I. Cowling, Monte Carlo Search Applied to Card Selection in Magic: The Gathering, IEEE Symposium on Computational Intelligence and Games, 2009. Proceedings, IEEE, 2009.
[3] http://www.trekcc.org
[4] http://aima.cs.berkeley.edu

Received May 2013;