The use of Artificial Intelligence in teaching: Bridge and other trick-taking card games

CPSC 490 Final Report
Hari Ganesan
Dana Angluin, Advisor

Abstract

This project is an application that serves as both a tutorial for someone learning how to play Bridge as well as an interface for someone to play Bridge with three AI players. The application is created in the style of a video game and its structure is highly reflective of this style. The game is created with SDL and OpenGL, and uses JSON to store lesson data. The main feature of the game is the AI that calculates decisions within the play of the hand in order to try and make a specified contract or to try and defend against a contract. This contract can be selected beforehand, and all bids are stored in a bid history. The application also introduces lessons and scenarios. A lesson includes several slides to represent various sections of the tutorial. A scenario is a representation of a point in time in the play of a Bridge hand which holds the information of the bid as well as all of the cards left in players’ hands. Also included in the game is functionality to include new lessons and create new scenarios. The executable has two flags as command-line arguments that deal with scenarios: one is to test a specified scenario, and the other is to output a random scenario. The game also serves as a boilerplate for other trick-taking card games, which could reuse many of the same structural and graphical aspects. Future considerations include bidding systems, scenario analysis, advanced graphics, a user interface for navigating through multiple lessons, and other trick-taking card games.

Background

Card games have been around for centuries, and possibly the most popular game of the last century is Bridge. Bridge is a trick-taking card game that requires four people in teams of two and a standard 52-card deck. Bridge is often played with a standard set of rules and small variations that encompass bidding systems and style of play. Unlike many other card games, Bridge requires a great amount of memory and logic.

Software for card games – and therefore, AI for card games – has been around for the last couple decades. The World Computer-Bridge Championship started in 1997, and there have been a couple AIs that have accounted for 11 of the 16 titles – Jack and WBridge5. Since Bridge is a game of incomplete information, these programs generate many samples of possible hands that other players might have, and then make decisions by weighing the various options differently.
Recently, there has been a transition to web and mobile applications to support these games. However, my main interest remains standard game development, and along with it, engines able to support high-end graphics and sound. Therefore, I have built this project using SDL, a popular game programming interface, and OpenGL, a standard graphics library. All code is written in C++, and JSON is used to store data from lessons.

Main Structure

I have split up the objects in my application into the following main structs and classes: Cards, Players, CardGame, and BridgeGame.

The CardGame class contains properties and functions that serve to make the gameplay for any trick-taking game possible. I have made functions for dealing, playing, and discarding cards. The methods I used for dealing and shuffling were not intuitive; in order to display and retrieve information from players’ cards in order, I kept the deck in order and randomly dealt cards to hands until each was full.

The BridgeGame adds additional features that are specific to Bridge. Among them is an array of the bidding history, which holds all of the information of the bids for a specific hand. This information is crucial to the play of the game, and often is retrieved when making early decisions.

I have also created a new file, “Decisions”, which handles all of the logic within the BridgeGame. I have made several auxiliary functions to aid my decision making process. Simple functions include picking the lowest or highest cards in a hand given a suit, and more complex functions include picking a card that is least valuable to a person (a.k.a. junk). My main decisions function that picks a card will first find out who is playing (contract vs. defending) and then what turn they are in the round (e.g., lead), and then make a decision based on several factors: the cards already played on the felt, the cards already played in the hand, and the bid history and the final contract. It should also be noted that during many Bridge games, understanding that another team is playing a certain convention could alter decisions made during the course of the game. In order to focus on the mechanical aspects of the game (i.e., the play), I have assumed a particular bidding convention has been used for all players and that no miscommunication could have been developed as a result.

Lessons and Scenarios

The teaching component of the application has a rigid structure that allows for easy inclusion of new lessons and components. The application is displayed as a series of slides that function as a tutorial. These slides are created as JSON objects, and a lesson is an array of these objects in the order of their intended appearance. I have
used a third-party library – JSON Spirit\(^1\) – in order to parse the JSON into my CardGame class and render the values of the objects. The navigation through each lesson’s slides is very simple – forward and back buttons, and a “run” button that will run a loaded scenario of a bridge hand. Encoded in the JSON is an integer that represents the number of cards played for each simulation of a scenario (in most circumstances, this would be some multiple of four). If the number is zero, no scenario will be loaded, and the players’ hands will not appear.

Scenarios are loaded from files using the \(\text{.bridge} \) extension\(^3\). The required file format is simple and easy to use. Each file lists all of the data of the cards in the players’ hands. A card of a player’s hand is represented by two integers\(^2\) separated by whitespace that represent a suit and a number, and each card is separated by a newline. Each hand is then separated by another newline. An example file might start as follows:

```
0 4
2 10
3 0

1 0
1 2
...
```

It is important to note that neither full hands nor hands that have the same number of cards are required for these scenarios.

One can also use the command-line flag “\(-g\)” in order to output a random scenario to a file, such as:

```
./main \(-g\) out.bridge
```

and one can also test out a scenario, using the “\(-t\)” flag:

```
./main \(-t\) in.bridge
```

Graphical Elements and UI

There is a lot of boilerplate code that serves as functionality for a variety of games that use both SDL and OpenGL. I have organized my project such that the main function that runs the application and controls the main loop is separate from the rendering process. There are several parameters and functions that are required in order to initialize SDL, fonts, music, and images.

I have also made an elements file to display graphical elements within the application. These could include buttons, navigation bars, etc. These buttons were
first created as an object that had certain properties such that some event be fired with those properties once a mouse is clicked in a certain area. These buttons were extended to cards for the user when playing against the AI. Play is then advanced either by hitting the space bar (required for AI and optional for user) or clicking on a card in the user’s hand.

There are several elements that are rendered within the main loop, depending on the display invoked. In order to display these elements in an efficient manner, I must structure my objects in a clear manner. For example, when deciding what order in which to place cards on the felt, I decided to put them in order of position relative to the players on the board, rather than the order in which they were played, which allowed me to render them easily.

**Future considerations**

**Unique generation of scenarios.** Using the [-g] flag allows the user to generate a random scenario using the program. This flag could easily be used to generate specific scenarios – for example, if one wanted to ensure that the contract side had a specified number of trumps, or that each player had a certain number of cards left in their hand.

**Auxiliary functions that analyze scenarios.** Likewise, if one wanted to mine information from particular scenarios, whether they be user-created or randomly created, it would be possible to include a new set of tools that extracts relevant data from the scenario.

**Inclusion of multiple lessons.** I have only included a sample lesson about the opening lead. Other lessons could be constructed and serve as a continuation of the tutorial. I should note that this might require a new user interface in order to select available lessons.

**Inclusion of native bidding systems.** There are dozens of recognized bidding systems. It would be possible to extract the main features of popular ones and include them either as lessons in the tutorial or as part of the AI in the bidding display of the game. This could also be extended to including the bidding history within a scenario instead of only including the final bid, with a particular bidding system embedded into the code.

**Rendering PNGs and other images.** Currently, all information is either displayed using GL_QUADS, GL_LINE_LOOP, or rendering of text on a black background. A more aesthetically pleasing simulation could pre-load and render assets, such as card pictures and other card game related images. These images could also include different states of elements in the program, such as navigation buttons with hover and active states.
General overload of graphics. The use of graphics is quite minimal given the extraordinarily large amount of code that is required to render all of the elements in the game. For instance, apart from the text, there has been no use of textures or 3D graphics, including lighting, shadows, etc.

Other trick-taking card games. The CardGame class was created for two purposes: to be able to further organize and separate functions in order to allow for ease-of-access and make programming easier, and to allow for other trick-taking card games to extend the class in addition to BridgeGame. In conjunction with the basic structure, any of the graphical elements of the program could also be reused to help create these similar games.

References


SDL - http://www.libsdl.org/

OpenGL - http://www.opengl.org/

1JSON Spirit is a parser for C++ that uses the boost library. The documentation is available online: http://www.codeproject.com/Articles/20027/JSON-Spirit-A-C-JSON-Parser-Generator-Implemented

2These integers are based on enumerations made in the program. For example, in the suits, 0 == SPADES, and in the numbers 0 == TWO, 1 == THREE, and so on (in order to keep a fixed valuation of numbers).

3This extension is not required – I originally used .txt files for scenarios.