Sevens Card Game Competition Framework

Advanced C++ Project [MU4RBI02]

1 Project Overview

This document outlines the implementation of a competitive framework for the *Sevens* (7, *Nana Narabe*) card game. You will implement a strategy that competes against those of your classmates to see whose algorithm most effectively wins at the Sevens card game.

1.1 The Game of Sevens

Sevens (also known as Fan Tan or Parliament) is a traditional card game that has been played for generations across Japan and many other countries. It is a game of skill, timing, and strategy where players aim to empty their hands as quickly as possible.

The game unfolds on a table, where cards of the same suit are arranged in ascending or descending order, starting from the 7s. As the game progresses, the table layout evolves, creating opportunities for strategic play. A skilled player must decide not only which card to play at any given moment but also when to hold cards for later advantage.

The simplicity of the rules masks a rich strategic depth. Players must monitor what cards have been played, track the cards likely held by other players, and decide whether to play aggressively or conservatively based on the evolving game state.

In our competition framework, your challenge is to develop an algorithm that can navigate these strategic decisions and outperform other players. Will your strategy be aggressive, defensive, or adaptive? The choice is yours, but the goal remains the same: be the first to empty your hand.

1.2 Game Rules

Sevens is played with a standard 52-card deck. The objective is to get rid of all your cards first:

1. The game starts with the 7 of each suit already on the table.
2. Players can only play cards that are adjacent to cards already on the table (i.e., one rank higher or lower in the same suit).
3. If a player cannot play, they must pass.
4. The first player to empty their hand wins, followed by subsequent players, with the last player receiving the lowest rank.

1.3 Framework Features

The competition framework offers:

- **Internal mode**: Run games with a default random play strategy.
- **Demo mode**: Run games with built-in sample strategies.
- **Competition mode**: Dynamically load and pit different student-implemented strategies against each other.

2 Style Guide

In this project, you are constrained to use the following style guide to maintain your code efficiently:

- **Using compiler flags** `-Wall -Wextra -Werror -pedantic -pedantic-errors -O3` with `g++` to guarantee C++ norms in your code implementation, optimizing compilation.
- **Using C++ 17** standard for modern features. The compiler flag `-std=c++17` should be used.
- **Minimizing dynamic memory allocation**: Avoid raw `new`, `new[]`, `delete`, and `delete[]` to optimize security and readability.
- **Dynamic memory using STL containers** for safer memory management.
- **Variable naming according to their usage**, to minimize redundancy in comments.
- **Perfect indenting of your code** to guarantee readability and avoid implementation errors.
- **Minimizing duplication of your code** to avoid copy-pasting and maintenance issues.
- **Using `snake_case`**, with method naming in `object_verb`, variables in `object_noun`.
- **New types should start with a capital letter**.
- **Constants declared with all capitals** (SCREAMING_SNAKE_CASE).
- **Class members called using `this`** to find easily where class members are used and modified.
- **Using `const` where input should be constant**, to guarantee correct read/write permissions.
- **Using reference or pointer when passing arguments**.

3 Provided Files

You are provided with a collection of files that implement the game framework:

Generic_card_parser.hpp Base class for card handling, defines the Card structure.

Generic_game_parser.hpp Game state management, maintains the table layout.

Generic_game_mapper.hpp Game simulation interface, defines how games are run.

PlayerStrategy.hpp The interface you will implement for your strategy.

MyCardParser.hpp/cpp Implementation of card parsing.

MyGameParser.hpp/cpp Implementation of game state initialization.

MyGameMapper.hpp/cpp Main game engine implementation.

RandomStrategy.hpp/cpp A simple random strategy implementation as a reference.

GreedyStrategy.hpp/cpp A more advanced strategy implementation as a reference.

StrategyLoader.hpp Utility for dynamic loading of strategy implementations.

StudentTemplate.cpp Starting template for your own strategy.

4 Implementation Roadmap

A roadmap is proposed to guide you through the various tasks in this project:

1. **Understand the strategy interface**: Study the `PlayerStrategy` interface to understand how strategies interact with the game.
2. **Study the provided strategies**: Examine the `RandomStrategy` and `GreedyStrategy` to understand different approaches.
3. **Create your strategy**: Copy `StudentTemplate.cpp` and implement your own approach.
4. **Test against internal mode**: Run your strategy against the basic random strategy.
5. **Test against sample strategies**: Compare your strategy against the provided implementations.
6. **Refine your approach**: Based on test outcomes, improve your strategy.
7. **Test in a tournament setting**: Run multiple games with multiple players to assess performance.

5 Your Task

Your task is to implement a strategy for playing the Sevens card game that outperforms the provided sample strategies and those of your classmates.

5.1 Implementation Steps

1. Make a copy of `StudentTemplate.cpp` and rename it to reflect your strategy (e.g., `AggressiveStrategy.cpp`).
2. Rename the class inside to match your strategy name.
3. Implement the required methods, focusing particularly on `selectCardToPlay`, which is where your strategy logic will reside.
4. Implement methods to track game progress in `observeMove` and `observePass` if your strategy needs this information.
5. Change the `getName()` method to return your unique strategy name.

5.2 The PlayerStrategy Interface

Your strategy must implement this interface:

```cpp
class PlayerStrategy {
public:
    virtual ~PlayerStrategy() = default;

    // Initialize the strategy with player ID and setup
    virtual void initialize(uint64_t playerID) = 0;

    // Select a card to play from the hand
    // Returns index in hand of card to play, or -1 if passing
    virtual int selectCardToPlay(
        const std::vector<Card>& hand,
        const std::unordered_map<uint64_t,
        std::unordered_map<uint64_t, bool>>& tableLayout) = 0;

    // Called to inform the strategy about other players' moves
    virtual void observeMove(uint64_t playerID, const Card& playedCard) =
   0;

    // Called when another player passes
    virtual void observePass(uint64_t playerID) = 0;

    // Get a name for this strategy (for display purposes)
    virtual std::string getName() const = 0;
};
```

6 Strategy Development Guidelines

When implementing your strategy, consider:

6.1 Selecting Cards to Play

The key method is `selectCardToPlay`, which receives two parameters:

- `hand`: A vector of `Card` objects representing your current cards.
- `tableLayout`: A nested map indicating which cards are already on the table.

The method must return:

- An index into the `hand` vector for the card you want to play, or
- -1 if you cannot play any card (which results in passing your turn).

6.2 Table Layout Understanding

The `tableLayout` is structured as:

```
tableLayout[suit][rank] = true/false
```

Where:

- `suit` is 0-3 (Clubs=0, Diamonds=1, Hearts=2, Spades=3).
- `rank` is 1-13 (Ace=1, 2=2, ..., King=13).
- The boolean value is `true` if the card is on the table, `false` otherwise.

6.3 Card Playability

A card is playable if it is adjacent to a card already on the table:

- For example, if the 7 of Hearts is on the table, you can play either the 6 or 8 of Hearts.
- If the 6, 7, and 8 of Hearts are on the table, you can play either the 5 or 9 of Hearts.

6.4 Tracking Other Players Using `observeMove` and `observePass`

Use these methods to track actions by other players:

- `observeMove(playerID, playedCard)` is called when another player successfully plays a card.
- `observePass(playerID)` is called when another player passes their turn.

For example, you might keep a structure mapping `playerID` to the cards you suspect they still hold. In `observeMove`, you could remove `playedCard` from your internal tracking of that player's possible cards. This can help you predict what they could play next and adjust your strategy accordingly.

7 Strategy Implementation Ideas

Consider implementing one of these strategies or create your own:

- **Random Strategy**: Play a random valid card (baseline strategy, already provided).
- **Greedy Strategy**: Prioritize cards based on certain criteria (also provided).
- **Aggressive Strategy**: Play the highest rank cards first to empty your hand quickly.
- **Conservative Strategy**: Keep pairs of playable cards together.
- **Blocking Strategy**: Play cards to prevent other players from playing.
- **Learning Strategy**: Adapt to other players' patterns.

8 Compiling and Running

8.1 Compiling the Main Game

To compile the complete game framework:

```
g++ -std=c++17 -Wall -Wextra -Werror -pedantic -pedantic-errors -O3 -ldl
    *.cpp -o sevens_game
```

8.2 Compiling Your Strategy as a Shared Library

Compile your strategy separately as a shared library:

```
# On Linux
g++ -std=c++17 -Wall -Wextra -fPIC -shared YourStrategy.cpp -o
    your_strategy.so

# On macOS
g++ -std=c++17 -Wall -Wextra -fPIC -dynamiclib YourStrategy.cpp -o
    your_strategy.dylib
```

The `-fPIC` flag creates position-independent code required for shared libraries.

9 Testing Your Strategy

To effectively test your strategy:

1. **Run it against the provided strategies** (`Random` and `Greedy`) to gauge performance.
2. **Try different numbers of players** (2, 3, 4, etc.) to see how your strategy scales.
3. **Run multiple games** to account for the randomness in card dealing.
4. **Analyze and refine** based on your average ranking across these tests.

10 The StudentTemplate.cpp File

The `StudentTemplate.cpp` file provides a starting point with a random strategy implementation. Key sections to modify include:

```cpp
// ...
class StudentStrategy : public PlayerStrategy {
    // Rename this class to your strategy name

    int selectCardToPlay(...) {
        // MODIFY THIS METHOD with your strategy logic
    }

    void observeMove(...) {
        // MODIFY to track other players' moves if needed
    }

    std::string getName() const override {
        // Change to your strategy name
        return "MyStrategy";
    }
};
```

11 Collaboration Policy

Students are permitted to work in pairs (exactly two members). Each pair must:

- Register both team members' names on Moodle before submitting the final work.
- Submit a single solution for the pair.
- Ensure that both students contribute substantially to the project.

No groups of more than two students will be permitted. If you choose to work alone, that is also acceptable.

12 Submission Requirements

Submit all code you create. We will test all provided code against each other and rank teams based on their performance. Ensure that:

- Your file compiles without warnings using the specified compilation flags.
- Your class is renamed from `StudentStrategy` to a unique name.
- The `getName()` method returns the same unique name.
- Your implementation includes clear comments explaining your strategy.
- It contains the required `extern "C"` factory function at the end.

When submitting as a team, clearly list both members at the top of your `main.cpp` file and in Moodle.

13 Course Policies

13.1 Academic Integrity & Plagiarism

All solutions will be routinely checked for academic integrity. In addition to running your submissions in a competitive setting, we will employ automated measures to compare the similarity of your code with other submissions (including past offerings). Please ensure your solution is genuinely your own work (or that of your pair). Minor code similarities can be coincidental, but extensive duplication or replication of other teams' solutions is a violation of academic policy.

13.2 Late Submissions

Assessment tasks submitted after the due date without an extension will be subject to a penalty. Points will be deducted according to the schedule below:

Up to 1 day late: 5% deduction

1 to 2 days late: 10% deduction

2 to 3 days late: 20% deduction

3 to 4 days late: 30% deduction

4 to 5 days late: 40% deduction

5 to 6 days late: 50% deduction

7th day and onwards: 100% deduction

Thus, if you do not have approval for an extension and the assessment is one week late, you will be given a score of zero for the project.

13.3 Extensions

If you cannot submit the assessment item before the due date due to special circumstances, you can request additional time in the form of an extension. Extension requests should be made as early as possible and before the assessment task is due. Special circumstances include personal health issues, unexpected commitments, bereavement, or other extenuating circumstances. For the extension to be considered, students must present documentation supporting their circumstances.

14 Evaluation Criteria & Rubric

Your strategy will be evaluated based on:

1. **Win rate** against other strategies in a tournament setting.

2. **Final ranking** across multiple games.
3. **Code quality** and originality of your strategy.
4. **Efficiency** of your implementation (runtime and memory usage).

Scores (Out of 100%)	**Description of Metric**	**Advanced**	**Proficient**	**Approaching Proficiency**	**Beginner**
Syntax (15%)	Ability to understand and follow the rules of the programming language.	Program compiles and contains no evidence of misunderstanding or misinterpreting the syntax of the language.	Program compiles and is free from major syntactic misunderstandings, but may contain non-standard usage or superfluous elements.	Program compiles, but contains errors that signal misunderstanding of syntax.	Program does not compile or (in a dynamic language) contains typographical errors leading to undefined names.
Logic (25%)	Ability to specify conditions, control flow, and data structures that are appropriate for the problem domain.	Program logic is correct, with no known boundary errors, and no redundant or contradictory conditions.	Program logic is mostly correct, but may contain an occasional boundary error or redundant or contradictory condition.	Program logic is on the right track with no infinite loops, but shows no recognition of boundary conditions.	Program contains some conditions that specify the opposite of what is required (e.g., $<$ instead of $>$), confuse Boolean AND/OR, or lead to infinite loops.
Correctness (25%)	Ability to code formulae and algorithms that reliably produce correct answers or appropriate results.	Program produces correct answers or appropriate results for all inputs tested.	Program produces correct answers or appropriate results for most inputs.	Program approaches correct answers or appropriate results for most inputs, but can contain miscalculations in some cases.	Program does not produce correct answers or appropriate results for most inputs.
Clarity (10%)	Ability to format and document code for human consumption.	Program contains appropriate documentation for all major functions, variables, or non-trivial algorithms. Formatting, indentation, and other white space aids readability.	Program contains some documentation on major functions, variables, or non-trivial algorithms. Indentation and other formatting is appropriate.	Program contains some documentation (at least the student's name and program's purpose), but has occasionally misleading indentation.	Program contains no documentation, or grossly misleading indentation.
Modularity (10%)	Ability to decompose a problem into coherent and reusable functions, files, classes, or objects (as appropriate for the language and platform).	Program is decomposed into coherent and reusable units, and unnecessary repetition has been eliminated.	Program is decomposed into coherent units, but may still contain some unnecessary repetition.	Program is decomposed into units of appropriate size, but they lack coherence or reusability. Program contains unnecessary repetition.	Program is one big function or is decomposed in ways that make little sense.
Performance & Ranking (15%)	The final ranking of the code in the final competition.	Students will receive their final score based on the performance of their code in both internal and competitive mode. This includes how the code ranks in the competition, with students ranking higher receiving greater marks.			

Table 1: Project Grading Rubric

15 Advanced Strategy Tips

For more sophisticated strategies, consider:

- Maintaining a count of cards played/remaining in each suit.
- Tracking which cards each player seems to hold (via `observeMove` and `observePass`).
- Calculating the probability of other players having certain cards.
- Implementing a multi-turn lookahead to anticipate future states.
- Adjusting your strategy based on the current game context (early game vs. late game).

16 Game Framework Architecture

To help you understand the underlying system, here's a brief overview of the framework architecture:

- **Card Representation**: Simple struct with suit (0-3) and rank (1-13).
- **Table Layout**: Nested map tracking which cards are on the table.
- **Player Hands**: Each player has a vector of `Card` objects.
- **Game Flow**: Players take turns in sequence until all but one have emptied their hands.
- **Strategy Interface**: Provides methods for each player to decide their moves.
- **Dynamic Loading**: Shared libraries loaded at runtime for competition mode.

17 Conclusion

This project provides an opportunity to demonstrate your algorithm design and C++ skills in a competitive, game-playing context. Through careful strategy development and implementation, you'll create an agent that can outperform others in the Sevens card game.

Your success will depend on understanding the game mechanics, implementing an effective decision-making algorithm, and efficiently tracking the game state to make informed choices.

Good luck!