Matthew Oey

Mike Wang

CS 3110 Problem Set 6: POKEJOUKI

Design Outline

Summary

We found implementing the bot to be one of the hardest parts of the assignment. We treated the game as a function that required optimization and the bot as an information search. This required a lot of linear algebra, probability theory, and exhaustive bookkeeping as we tried to formulate an algorithm that found the optimal attack for every possible state.

Another challenge was maintaining and distributing all of the state information necessary to implement the game. With so much communication required between the different modules, it was important to rigorously ensure that accurate information was being passed at the correct times to the correct places.

One of the major design decisions we decided to do was to implement the game rules in the state.ml file rather than in the game.ml file. We did this because many of the game rules involved handling commands and updating the state of the game. It also made dividing the assignment easier, especially given the long break in the middle which made it difficult to meet up, because the game.ml file handled the game implementation while state handled the game rules and they did not require that much code of each other's code.

Specification

In our module state.ml, we had to implement many of the game rules in functions that were called during the game's execution in game.ml. The most important functions in state.ml were add_steammon, select_inventory, switch_steammon, attack, and use_item. These five methods represented the core of the game.

Design and Implementation

Modules: Our modules are separated into three folders: game, shared, and team. The modules in the shared folder handles the shared files including utility functions, definitions, and constants. These were all left unchanged throughout the assignment. The modules in the game folder handle the implementation of the game. It contains our game, state, netgraphics, and server modules. The module netgraphics.ml handles the GUI and server.ml handles the concurrency between the game, the bots, and the GUI. The module state.ml handles many of the game rules and keeps track of the current game state, however since state.ml is merely a representation of data during each stage of the game, it does not call execute any code nor can it communicate with the server. We talked about if we should create mutable variables however we decided that we were better off creating a new game state every turn. The module game.ml handles implementation of the game as well as its various requests and commands, calling function in state.ml for information on what to do next. Game.ml also serves as a mediator between the server, client, and game state by receiving requests, issuing commands and updating the game state accordingly. In the teams folder we have the bots which handle all the AI operations. This bots wait for requests from the server and respond with calculated actions.

Architecture: The architecture of our program is fairly straightforward. Game.ml communicates between the client and server, and updates state.ml. State serves as a snapshot of the game at a certain point and is only accessed by game.ml. All three main classes we had to implement (game.ml, state.ml, the bot) are dependent on information from constants.ml, definitions.ml, and util.ml.

Code Design: The main data structure we used was game_status_data. Game state contained all the information about every steammon on each player's teams and kept track of both player's inventories. To organize this amount of data it used tuples to keep track of the various aspects and records to keep track of things that required a lot of data such as steammon and attack. Our implementations did not require any complex algorithms, and relied primarily on condition checks and state management. While we considered mutability of state data, we decided that although it would make for easier to write and less cluttered code, mutability could pose many problems that would make debugging significantly more difficult. So we opted instead to copy and paste those gigantic records.

Programming: Our implementation strategy consisted largely of creating many useful modulized helper functions that would be used in the larger parts of the code. One of the biggest challenged in coding a program of this magnitude was making sure the different components of the program communicated correctly. The distribution and manipulation of data was by far the most important aspect of this project.

The work distribution was relatively even throughout, and we worked together to discuss the design of each component before implemented them. Matt coded the attack function, part of state.ml, and the majority of the bot. Mike coded the majority of game.ml and a significant portion of state.ml, including GUI, and took the lead in debugging and testing.

Testing

Testing and debugging the code required an extremely thorough examination of all outlets of the code, to make sure that communication between each component was being processed correctly. One challenge initially was becoming familiar with the way the server and client interact with game.ml.

Our testing consists largely of trial and error, running the game simulation repeatedly and modifying the code accordingly to achieve the desired results. From these simulation, for the most part, we are able to confirm that game-server-client communication, initialization of the game, drafting steammon, basic attacking and status effects, item use, switching steammon, and game conclusion work correctly. We did not, however conduct any exhaustive tests to confirm that all possible battle scenarios function correctly, ie. if a steammon is confused and paralyzed. The GUI is confirmed to work, though the accuracy of what is being displayed is somewhat questionable at times. However, print statements in the console confirm that the game is processing data correctly despite the GUI being faulty.

Known Problems

The GUI does not always display correctly.

Comments

We thought the problem set was interesting although rather tedious and too exhaustive at times. It was not as fun as expected. At this moment the sun has just risen above the horizon; you may infer from that the amount of time we spent on this assignment. It didn't require the implementation of any challenging algorithms like we saw in earlier problem sets but it did provide good practice in software engineering. For future implementations of the pokemon game, I would suggest more difficult problems but less volume of coding. For example, maybe give some of the game rules such as calculating damage but require the student to code the server.