CS4731: Capstone Write-Up

Harrison Katz John Rafferty Stephen Whipple

July 31, 2013

1 Instructions

You need NodeJS if you plan on running on a local simulator (useful if the main server is down or breaking changes have been applied upstream). We tested using NodeJS v0.10.12 and simulator version 0.9.1. To run the test server, do the following:

1. Navigate to <extracted directory>/simulator/ in a console window.
2. Run `npm install` to install dependencies
3. Run `node app.js` to launch the server (by default it will run on port 8000)
4. Go to `http://localhost.psim.us` in Chromium to ensure that you can connect to the local server.

You need a recent version of Chrome or Chromium; it has been tested on Chromium version 28.0.1500.71 Debian 7.1 (209842). To add the extension to Chromium, do the following:

1. Extract the zip archive to a known directory on the system
2. Chromium open the extensions tab by using Menu > Tools > Extensions
3. Check the "Developer mode" checkbox
4. Click the "Load unpacked extension..." button

5. Locate <extracted directory>/extension/ and open it using the dialog

6. Ignore any warnings about key files if they are present; they were generated locally from the `npm install` command if you installed a local server and only matter if you plan on redistributing the packed extension.

Once the extension has been added to Chromium, do the following:

1. If you want to play on the local server and you are running one with `node app.js`, open a window and go to `http://localhost.psim.us/`.

2. Otherwise, you should be able to play on the public servers as long as they are running and no breaking changes have been applied at `http://play.pokemonshowdown.com/`. Note that since this tracks upstream and there is a lot of development, breakage over time is likely – the local server is more reliable.

3. Click the extension button in the far right corner of the URL bar. The AI should begin logging in and searching for games to play within a few seconds.

2 Approach

Our approach was originally to use Expectimax with alpha-beta pruning to narrow down states, however we soon realized that this could not be done since pokemon moves were chosen at the same time (and each side does not know which move the other chose). This made any kind of alpha-beta pruning impossible since we'd want to optimize based on maximums and minimums that are on different levels, instead of each level representing either a maximization or minimization as in standard Expectimax. We then tried to use 2-tuples with each move choice (including swaps) as the edges of the decision tree, with the states being represented by a score of the entire game state (always trying to maximize). No kind of pruning can be applied in this scenario either, and we soon found out that simulating these states per level results in millions of states just looking a few moves ahead. We instead decided to apply a kind of learning algorithm and use Case Based Reasoning to solve our problem. We have two AIs – one that learns to play

the game by itself and bases its preferences on a "game state score" which it calculates based on the difference between this state and the last one, and one that learns based on watching other humans play (the idea here is that, internally, the human scores each move according to how they rank it, and always chooses what they consider to be the best, so if we are in a similar situation as them, we should choose a similar move even if we don't have a specific number score for that move). The second AI falls back to learning by itself if it hasn't seen a human play in a similar enough situation before (since the other alternative would just be randomly flailing around, which is always a worse choice).

3 Challenges

Scoring (Preference of states)
It was difficult to figure out the best way to score a state. We went with assigning a value to different parts of the state on set scales for each part. For example: our health could get between 0 and 30 points, 0 for losing more than 50% of our health, 30 points for gaining more than 50% of our health, and 25 points for staying at the same health. We had values for our health, our status, our hazards, enemy health, enemy status, and enemy hazards. After we calculated the scores for each part we added them up and the total was the score. The score was on a scale of 0-110. It was hard to come up with numbers for each thing that rated states in a way that we thought made sense.

Comparing (Similarity function)
Deciding the best way to narrow down states that were the similar to ours was a challenge. This posed a problem because even if we had a state where we had used the same pokemon before we might have not faced the enemy pokemon, or enemy pokemon type, with it before. If we had faced the enemy pokemon, or enemy pokemon type, before with a pokemon that was of the same type as our pokemon then we would want to choose that state instead. Figuring out the best way to handle cases like this was the biggest problem. Once that problem had been handled we had to decide how to represent swapping pokemon. We decided to represent swaps as moves. This worked out very well and made choosing a move easier.

Other challenges (Non-AI related)
We had many other challenges when working on this project, which proved

to be much more difficult to debug than the challenges we faced with the AI. In particular, parsing the game to extract information and synchronization problems between our polling method of determining game state and the actual game state were the cause of many, many bugs that were difficult to trace. There was one instance of needing to drop a feature because it couldn't be done reliably with the current platform (due to Chromium's restrictive security model – it wouldn't allow foreign JavaScript to be executed with the local extension JavaScript), which was actually sending text to the other player (though the audio on the host computer's end works fine for demonstration purposes, or if the AI was fighting a local opponent). It was incredibly hard to debug and test certain functionality with JavaScript and the Chromium web inspector, and documentation on certain features was scarce.

4 Goals

In this project our goal was to create two AIs; one of which learns from other human decisions that it observed and the other learns from playing the game and evaluating each game state using a scoring function. Both AIs attempt to categorize game states and choose moves that either score highly or were chosen as moves by real players in similar situations.

We do get a potentially decent AI in the end, depending on specific moves and scores it collects as it runs. This AI could probably beat new players to the game, but would lose to veteran players. The reason it does pretty poorly against experienced players is because it lacks a sense of continuous strategy and meta-game, which turns out to be much more important than we anticipated. Whereas our bot and newer players tend to choose good moves that benefit them in the short run, veterans will choose moves in certain sequences to reach a specified goal. In addition, context is important, which our AI doesn't possess. For example, the moves are scored based on current game state, but don't take into account the entire sequence of actions (this would be much too large to store in a database for retrieval), so moves that work twice in a row but have 100% chance of failure in the third turn would get high rankings the first two times, so the AI might try it again a third time. To combat this, we could either use some kind of planning or sequence matching to do well in the many edge cases and special cases.

5 Design

FSM
The automation uses an FSM to run. It has 6 states:

1. start : click the home button -> logged in or not logged in
2. not logged in : log in -> logged in
3. logged in : search -> searching
4. searching : wait -> in game
5. in game : perform action -> change name
6. change name : logged out -> logged out

Database
We used Web SQL (a dying standard) to store our recorded moves and states for the AI to request and compare. This database has 1 table (tb_state) which has many rows:

1. my_pokemon
2. en_pokemon
3. my_status
4. en_status
5. my_type
6. en_type
7. my_hazards
8. en_hazards
9. weather
10. move
11. rating

Audio

We used a JSON structure to hold possible audio, containing a trigger function that would determine if the audio had been triggered:

1. name - Name of audio
2. chance - Chance to activate given a trigger
3. max - maximum times this audio can be used
4. audio - object containing the WAV file
5. used - number of times this audio has been used (always initialized to 0)
6. trigger - Boolean function for determining if this sound should be used