Michael Ostrow

12-12-2014

Final Project Report

In my final project, I constructed an AI agent which used minimax search with alpha beta pruning to conduct Pokemon battles. Pokemon is a classic video game in which two teams of six unique entities battle against each other (these entities being the Pokemon after which the game is named). The goal of these battles is for one player to reduce the health or HP of all of the opponent’s Pokemon to 0. A player would accomplish this by using various attacks each of their six Pokemon have, or by switching to a different pokemon in their team. Thus, legal actions in my gamestates can be either using an attack or switching to a different pokemon. Attacks vary in power, type, and whether they are physical attacks or special attacks. The damage done by these attacks is determined by the following formula:

Damage = ((((2 \* Level / 5 + 2) \* AttackStat \* AttackPower / DefenseStat) / 50) + 2) \* STAB \* Weakness/Resistance

In this formula, level refers to the level of the attacking pokemon (level ranges from 1 to 100). AttackStat refers to the number representing the appropriate attack value of the attacking pokemon (either special attack or just attack). AttackPower refers to the base power of the attack itself. DefenseStat refers to the appropriate defense value of the pokemon being attacked (again, either special defense or just defense). STAB is 1.5 if the type of the attack used is the same as the type of the pokemon using said attack, and 1 otherwise. Weakness/Resistance can range from 0 to 4, and denotes the type of the attack used being either more effective or less effective against the type of the defending pokemon. This formula was used in my project to determine state transitions, as the goal of a player in a given state using an attack is to reduce the HP of an opponent’s pokemon. This formula was necessary to determine what that HP reduction would be and to then determine the value of states in which such HP reduction was inflicted. A game state in my project is composed of two lists of pokemon. These pokemon are given a name which is a string, an integer representing their level, and floats representing current HP, max HP, attack value, defense value, special attack value, and special defense value. Pokemon are also given a field called ‘Type’ which represents the elemental type of the pokemon (i.e. fire, water, ice, ghost, etc..), which determines weaknesses or resistances to attacks used against that pokemon. Lastly, pokemon have a final ‘attacks’ field, which is a list containing all attacks usable by that pokemon.

Inputs to the program at large are text files containing all relevant information to construct two teams of six pokemon. Once the files are parsed, a gamestate is created containing the two teams obtained from the text file. This gamestate is then passed to the alpha-beta search agent, which takes the gamestate and performs a series of searches of fixed depth to determine the actions taken by each player. These actions are then applied to the gamestate to produce the next gamestate to be used in the search. These actions are also displayed on the command line as the program runs. When a terminal state is reached, the program will notify the user and terminate.

The main algorithm used in this search algorithm is minimax search with alpha-beta pruning. Minimax search is used to determine best courses of action in games where one agent wants to maximize the value they obtain in their actions within the gamestate and other agents hope to minimize the value obtained by the aforementioned maximizing agent. This is very appropriate in pokemon, as one player wants to maximize the value of their actions, and the other player wants to minimize that value (and in the case of my implementation, minimizing your opponents value of a gamestate means maximizing your own). In this implementation, maximizing value implies having a high percentage of HP remaining amongst all your pokemon, your opponent having a low percentage of HP amongst theirs, and also having a favorable matchup currently in play (by this I mean that the types of the pokemon you currently have in play are strong against the types of the pokemon your opponent has in play). The way I use minimax search in my project is identical to the way it is used classically. The maximizing agent (player 1 in the case of my project) will always take the action leading to the highest value for them assuming the minimizer (player 2) will always choose to minimize.

This algorithm worked very well given games where the correct moves are the obvious ones. For example, in the ‘basic’ run of my algorithm, it performs optimally even given a small maximum search depth (3 in this case). The agent is able to quickly recognize the proper move for player 1 to win, and player 2 quickly recognizes they have zero chance to win so they inflict as much damage as possible in the meantime. It was difficult to test how well the algorithm worked when given more challenging problems, however, simply because of how quickly the search trees would grow in size as minimax ran, and thus cause an extremely long runtime for the algorithm. For example, depth 1 produces 9 new nodes to search, which themselves produce 9 more nodes. This continues in this fashion, leading to 59000 nodes at only search depth 5. Alpha-beta pruning did help alleviate this to some degree however. In the ‘intermediate’ run of my project, used for testing purposes rather than to illustrate any points, 15580 nodes are expanded on the first turn in which player 1 chooses a move to reduce the HP of the opponents pokemon to 0. On the next play, however, only 1853 nodes were expanded for the second player to decide the pokemon they would switch in, which in this case granted a very obvious type advantage (this example was run at depth 5). However, even given the limited depths that were efficiently searchable, the algorithm would still produce expected results. For example, in my ‘advanced’ run of my algorithm, a search depth of 4 would still result in the team which I intended to have the advantage winning the game, showing that exceptionally deep searches were not required for the algorithm to still perform in an expected fashion.