Parallel Genetic Hyperparameter Tuning for Playing 2048 with Adversarial Search

COMS4995_003 Final Project Proposal

Colin Brown cdb2167 - Jonathan Rich jrr2193

2048 is a single person game where you try to maximize your score by combining numbered tiles of the same value on a 4x4 square grid without letting it completely fill up. Your score in the game is roughly the sum of all the tile values on the grid, so it’s roughly proportional to how long you survive in the game. As the tiles spawn randomly, it is possible to build an agent that plays this game using adversarial search, modelling the random tile appearances instead as the deliberate choices of an opponent trying to make you lose as quickly as possible. To play against this, the agent can choose their moves with an alpha-beta pruning minimax algorithm to try to survive as long as possible by choosing the best-worst case scenario possible over agent moves and tile placements. As the alpha-beta minimax algorithm is a recursive tree search, it is well suited to be implemented in Haskell (while the overall game could be represented by a State monad where both the agent’s move choice and the random tile generation are the alternating transitions from state to state). However, a full tree search for each move is impossible as the game is not finite (or at least of a depth where a full tree would be impossible to generate), so a heuristic function is needed to estimate how good a given board state is at a particular depth down the tree. This heuristic is hard to directly code, as there are many possible measures of how good a state could be (such as tile value only increasing in a given direction, having closer neighboring tile values, having full rows and corners, etc), so tuning the weights or exponents given to these different measures is needed to result in an agent that can play well. As there is a random component to the actual tile placements in the game, averaging the scores of a few games played with a given set of heuristic weights is needed to get an accurate measure of the quality of that hyperparameter set.

Rather than doing this manually or through grid search, as the search space of weights and exponents is continuous rather than discrete, finite, and/or convex, a genetic algorithm can be used to evolve continuously improving sets of parameters. This could work by starting with a fixed number of random or chosen parameter sets, from which more could be chosen by combining two previous parameter sets (swapping a random subset of parameters) and then giving each individual parameter a chance of mutating a bit from its current value. The fitness of each generated parameter set could be tested by running a few games as above with that set of weights, then averaging the scores. Parameter sets that don't perform well enough would be eliminated, while well-performing sets would be used to generate successors in proportion to their relative performance. Rather than using fixed generations, the genetic algorithm could be run by maintaining a set of a fixed number of the best performing parameter sets encountered so far. A new parameter set could be generated by choosing two random parents (with probability proportional to performance), combining them, then mutating slightly. This new parameter set would be testing by averaging the score over multiple games, then added into the list of the best encountered parameter sets. The worst performing set would be dropped, enforcing selection of the better performing heuristics. There will be a small chance for an underperforming candidate to be kept in favor of one of the top performing candidates, in order to incentivize exploration of different candidates and avoid the algorithm getting "stuck" in a local maxima. Furthermore, as testing a given candidate parameter set is independent of any other parameter set, multiple candidates could be tested in parallel at a given time. As each test finishes, it's set would be merged into the best performing as described above and a newly generated candidate could take its place, so the fixed generations of traditional genetic search would be replaced with continuous parallel testing and selection. A log of the best performing parameters could be maintained as the algorithm runs, so it could be stopped at any point and return the best possible parameters so far (or it could run until convergence, although given that genetic search can be slow and is not guaranteed to be optimal, no convergence is guaranteed in a given time frame).