Milestone 1

Wills Liou

February 20, 2020

1 Milestone 1

Goal: Understand an advanced data structure or algorithm to be used in Generals, including analysis and implementation details

Two Ideas: (MCTS implementation was inspired by the search algorithm used by Deep Mind's Alpha Zero AI)

1. Use Monte Carlo Tree Search with a neural network (reinforcement learning). Discover states on the go as opposed to DP.
   Search every possible move that exists after each turn and select the node with highest probability of winning. Our win probability is calculated by comparing how close the output of the child node's decision was to final output needed to win the game; we rewarding best child node.

   Use **Bellman equation** to estimate returns from each state for given policy (must use expected values as our information is imperfect)
   Bellman Equation
   $V(s) = max_a(R(s,a) + \gamma V(s'))$ where $Q(s,a) = R(s,a) + \gamma V(s'))$ for in state S we estimate Q for every possible action.
   Then we choose action with the highest Q value. **This function is looking for the return value we get from getting the best action.**

   Policy function gets the best action for every state
   $\pi(s) = max_a(Q(s,a))$ reads as *the policy for state s is to choose the action with the highest Q value.*
   This is policy function makes the same calculations as the Bellman equation, but instead is returning what the best action is.

   Use Q Learning
   Assume return is G. Our formula for returns we received at step t of the game. $G_t = r_{(t+1)} + \gamma G_{(t+1)}$
   This reads as G_t equals the immediate return received plus all discounted rewards thereafter received after following our current policy.

Algorithm to calculate returns after each episode

(a) Inititalize G = 0

(b) empty list statesReturns = []

(c) Loop backwards through list of statesReturns after playing game

(d) append (s,G) to statesReturns list

(e) $G = r + \gamma \times G$

(f) reverse back statesReturns to original order

Idea: Use Episilon greedy algorithm have balanced results Improve runtime

1. Don't visit same state more than once
2. take the first visit and discard all other visits

Steps for Reinforcement Learning

1. Selection
   (a) Start at root node
   (b) If left node (L) is more optimal or has a larger probability of winning, select left node; if this left node is a leaf, end the game.
2. Expand
   (a) Calculate, create child nodes for each action based on our selected nodes, select the first one of these nodes
   (b) Use confidence bound (upper) to select $v_i + C \cdot \sqrt{\frac{lnN}{n_i}}$ where C is some constant, v_i is value of state at i, n_i is number of visits to state i, N is number of visits on the same level
   (c) Markov Decision process - maximize the rewards
3. Simulation (S_i)
   (a) Take random actions, and retrieve state. (repeat until at terminal node and return that terminal node's value)
   (b) Use reinforcement learning to reward children nodes that
4. Backprogation (backwards propagation of errors) using gradient descent
   (a) Reduce error by analyzing and fine tuning our weights based on error rate in previous epoch or transition

Some downsides are that it might need millions of examples, solve by fetching public replays

Idea 2: Use dynamic programming - brute force and loop through all possible states and actions

1. Consider n-tiles around yourself.
2. Assume the worst possible scenarios for yourself (minimum gain by doing any move)
3. Store those in the dp table
4. Calculate back to your current state
5. You can write it like you are maximizing your minimum gain that chooses the strategy to have the highest minimum gain.

Some downsides are that it might be too slow, usually unpractical for large state spaces.

Other algorithms: A simulating annealing algorithm is probably not a good choice here. Too randomized compared to our other options.

Genetic algorithms - Snapshots or states of the game can be represented by $(s0, a0), s1, a1), ..., (sN, aN)$