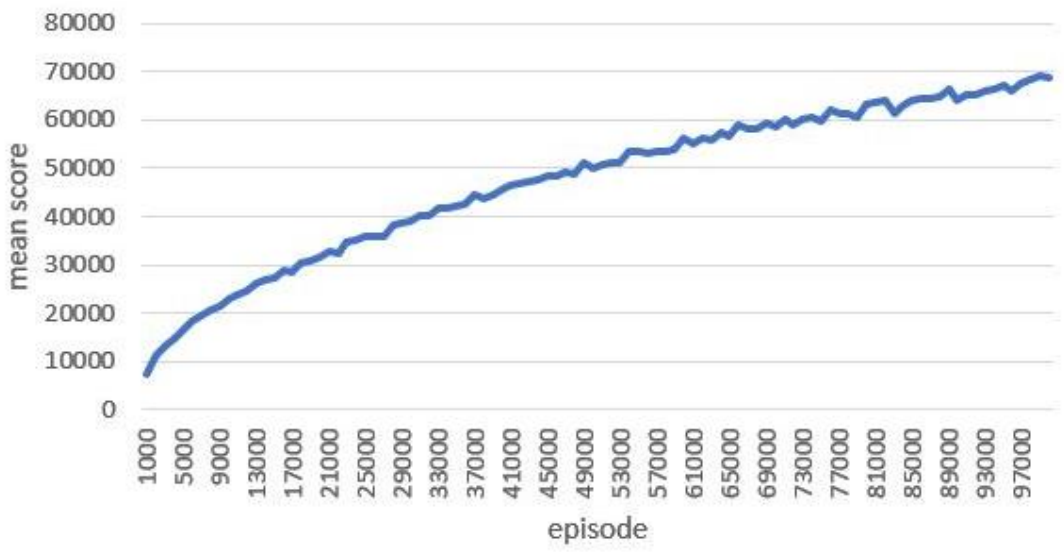
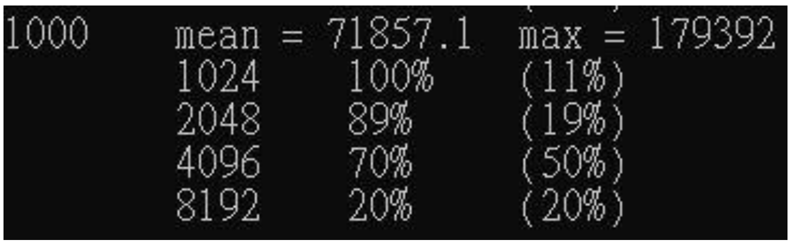
Lab 1 – 2048

312553024 江尚軒

Training:



Testing:



```
1000    mean = 71857.1   max = 179392
        1024     100%     (11%)
        2048     89%      (19%)
        4096     70%      (50%)
        8192     20%      (20%)
```

Describe the implementation and the usage of n-tuple network.

The n-tuple network is used for temporal difference learning in the game of 2048. Temporal Difference (TD) learning is a reinforcement learning method. The n-tuple network represents a set of features used to estimate the value of a state in the game.

The implementation involves several classes:

- Board Class: Represents the state of the 2048 game using a 64-bit bitboard. It provides methods for manipulating the game board, such as moving, rotating, and reflecting.
- Feature Class: Abstract base class for features used in TD learning. It has methods for estimating and updating the value of a given board.
- Pattern Class: A specific implementation of the Feature class representing a pattern in the game. It considers isomorphic patterns (rotations and reflections).
- State Class: Represents a state in the game, including before and after states, action taken, reward, and estimated value.
- Learning Class: Represents the learning agent. It maintains a set of features (n-tuple network) and provides methods for estimating, updating, and selecting the best move.

Explain the mechanism of TD(0).

Temporal Difference learning, specifically TD(0), is a model-free reinforcement learning algorithm. In the context of this implementation:

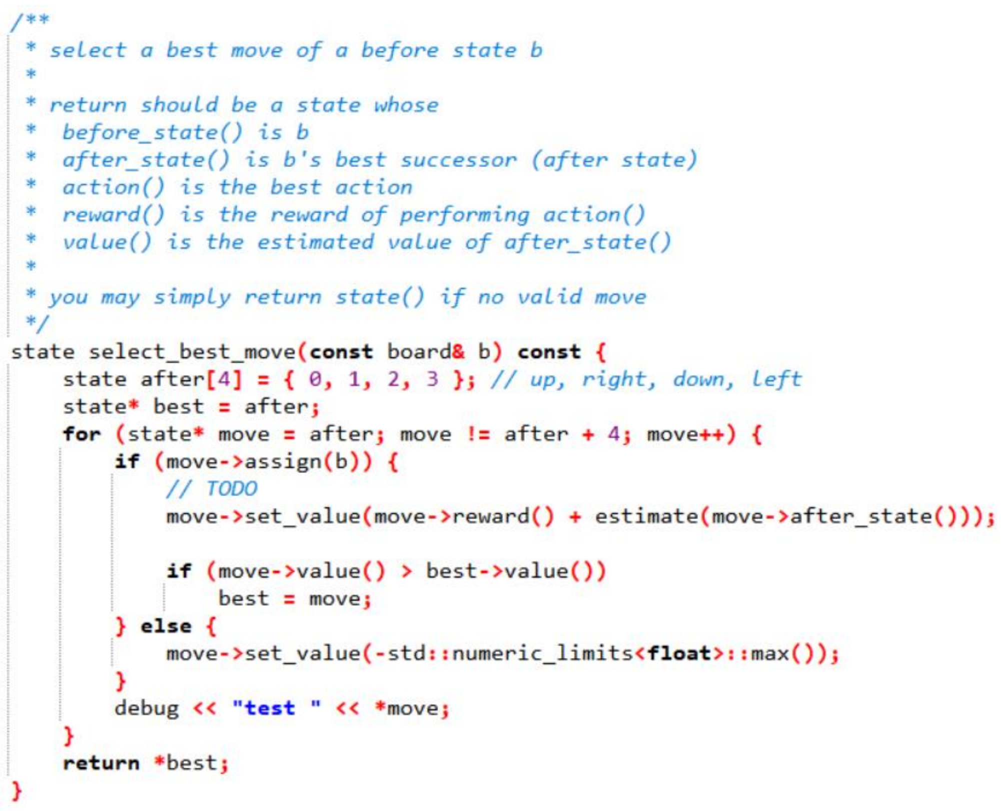
- Estimation (V(s)): The `estimate` method in the `Learning` class computes the value of a given board using the n-tuple network. It sums up the values from all features.

- Update (V(s) → V(s') + α(R + γV(s') - V(s))): The `update` method in the `Learning` class updates the value of a given board. It iterates through all features and updates their weights based on the TD(0) update rule. The target value (V(s') + α(R + γV(s') - V(s))) is used to adjust the weights.

Describe your implementation in detail including action selection and TD backup diagram.

- Action Selection:

  The `select_best_move` method in the `Learning` class is responsible for selecting the best move from a given state. It evaluates all possible moves (up, right, down, left) and selects the move with the highest estimated value.



```cpp
/**
 * select a best move of a before state b
 *
 * return should be a state whose
 *  before_state() is b
 *  after_state() is b's best successor (after state)
 *  action() is the best action
 *  reward() is the reward of performing action()
 *  value() is the estimated value of after_state()
 *
 * you may simply return state() if no valid move
 */
state select_best_move(const board& b) const {
    state after[4] = { 0, 1, 2, 3 }; // up, right, down, left
    state* best = after;
    for (state* move = after; move != after + 4; move++) {
        if (move->assign(b)) {
            // TODO
            move->set_value(move->reward() + estimate(move->after_state()));

            if (move->value() > best->value())
                best = move;
        } else {
            move->set_value(-std::numeric_limits<float>::max());
        }
        debug << "test " << *move;
    }
    return *best;
}
```

- TD Backup Diagram:

  The TD backup diagram involves the `update_episode` method in the `Learning` class. This method is called after an episode of the game has been played. It updates the weights of the features based on the TD(0) update rule.

  The update is performed backward through the episode path. For each state, the TD error is calculated, and the weights of the features are updated accordingly.

```
/**
 * update the tuple network by an episode
 *
 * path is the sequence of states in this episode,
 * the last entry in path (path.back()) is the final state
 *
 * for example, a 2048 games consists of
 *  (initial) s0 --(a0,r0)--> s0' --(popup)--> s1 --(a1,r1)--> s1' --(popup)--> s2 (terminal)
 *  where sx is before state, sx' is after state
 *
 * its path would be
 *  { (s0,s0',a0,r0), (s1,s1',a1,r1), (s2,s2,x,-1) }
 *  where (x,x,x,x) means (before state, after state, action, reward)
 */
void update_episode(std::vector<state>& path, float alpha = 0.1) const {
    // TODO
    float target = 0;
    for (path.pop_back() /* terminal state */; path.size(); path.pop_back()) {
        state& move = path.back();
        float error = target - estimate(move.before_state());
        target = move.reward() + update(move.before_state(), alpha * error);
        debug << "update error = " << error << " for" << std::endl << move.before_state();
    }

}
```