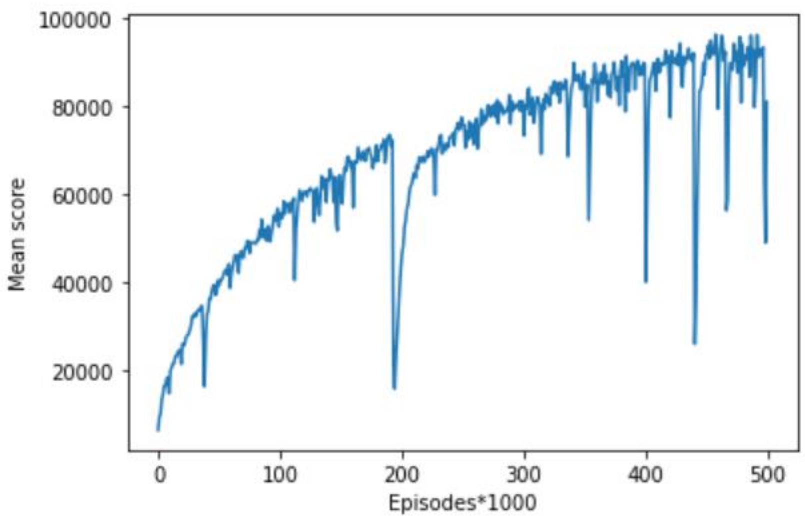
Lab7

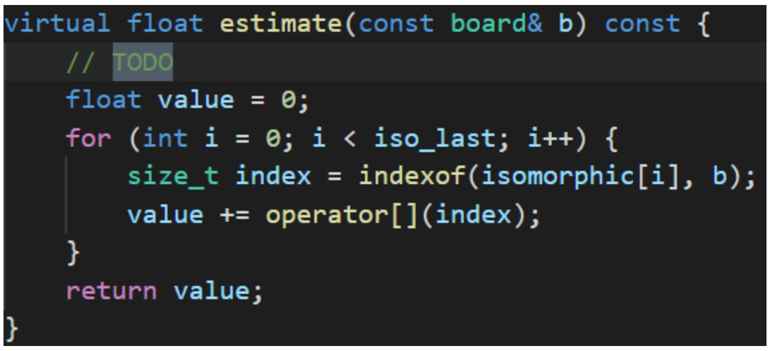
0856108 謝宗祐

- A plot shows episode scores of at least 100,000 training episodes



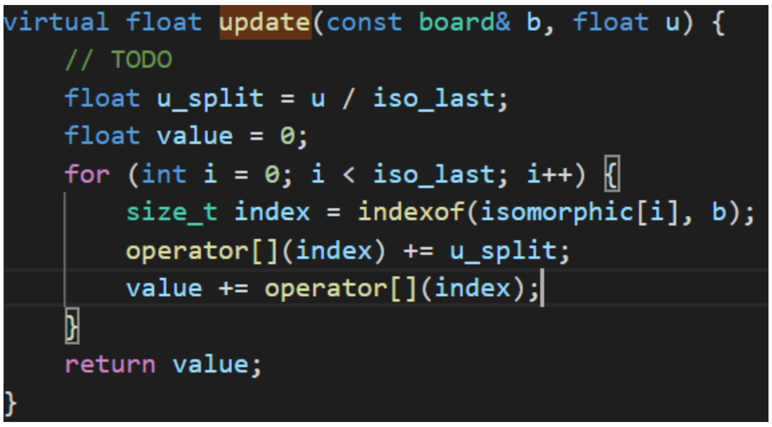
- Describe your implementation in detail

There are totally 5 TODOs.



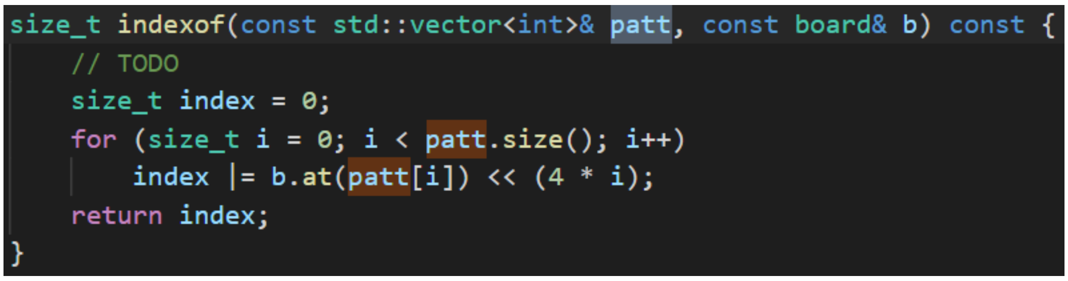
```
virtual float estimate(const board& b) const {
    // TODO
    float value = 0;
    for (int i = 0; i < iso_last; i++) {
        size_t index = indexof(isomorphic[i], b);
        value += operator[](index);
    }
    return value;
}
```

In order to estimate the value of a given board, for all isomorphisms of the patten, find its index and then look up the table to add the weights to the value.



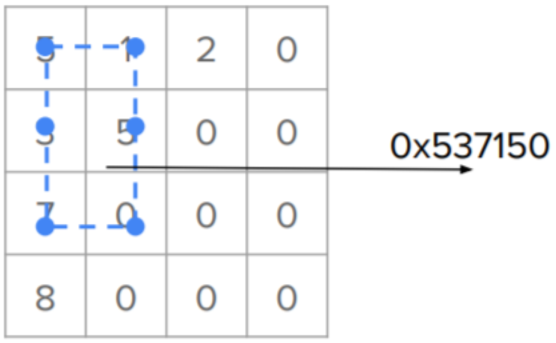
```
virtual float update(const board& b, float u) {
    // TODO
    float u_split = u / iso_last;
    float value = 0;
    for (int i = 0; i < iso_last; i++) {
        size_t index = indexof(isomorphic[i], b);
        operator[](index) += u_split;
        value += operator[](index);
    }
    return value;
}
```

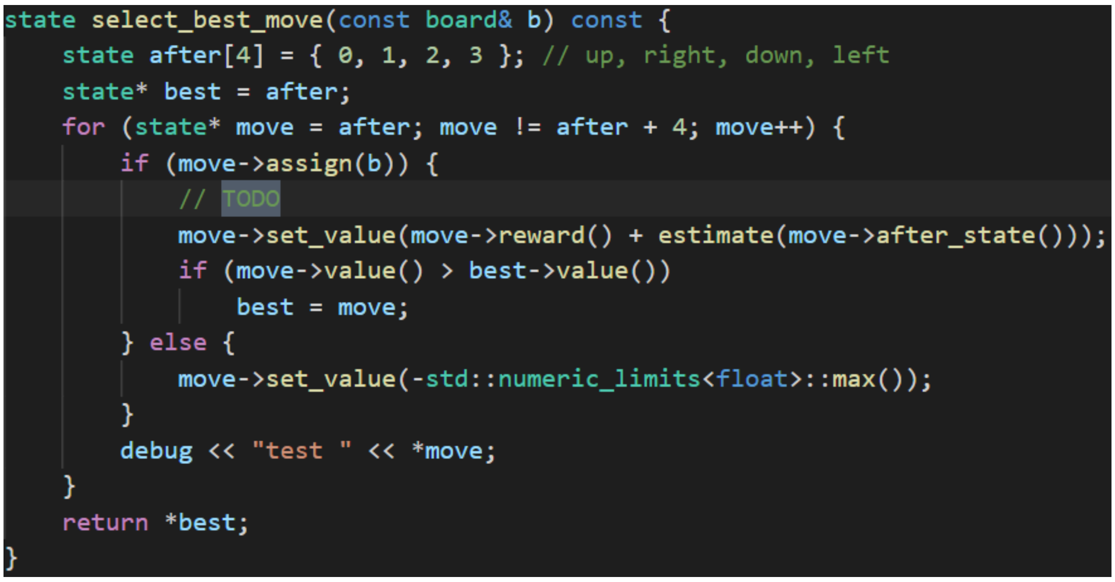
Similar to the estimate, update the given board and return its updated value.



```
size_t indexof(const std::vector<int>& patt, const board& b) const {
    // TODO
    size_t index = 0;
    for (size_t i = 0; i < patt.size(); i++)
        index |= b.at(patt[i]) << (4 * i);
    return index;
}
```

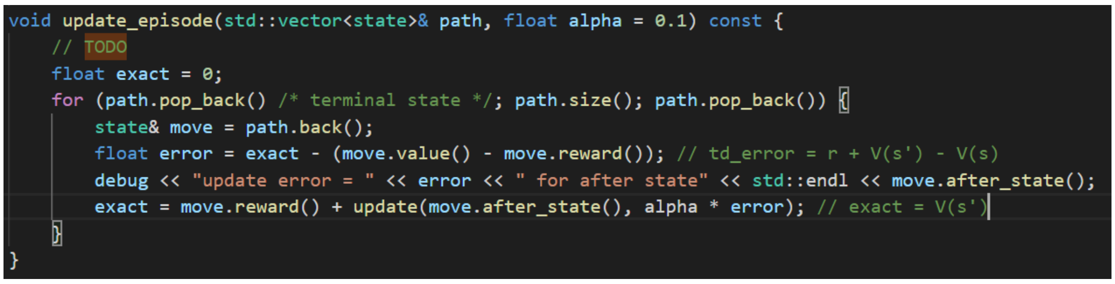
Find the index of the given pattern. For example like the below figure, the index value in the for loop is updated from 0x000000 -> 0x000050-> 0x000150 -> 0x007150 -> 0x037150 -> 0x537150.





```
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;
}
```

Find out which actions(up, right, down, left) can lead to the largest value, which is the summation of the reward and the value of the after state.

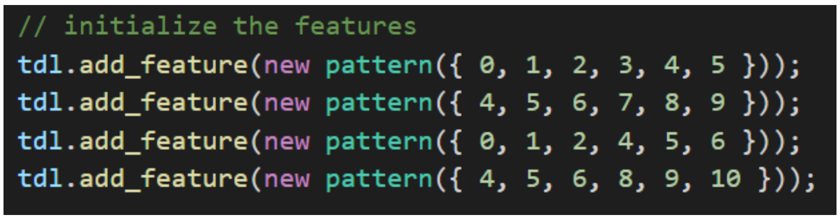


```
void update_episode(std::vector<state>& path, float alpha = 0.1) const {
    // TODO
    float exact = 0;
    for (path.pop_back() /* terminal state */; path.size(); path.pop_back()) {
        state& move = path.back();
        float error = exact - (move.value() - move.reward()); // td_error = r + V(s') - V(s)
        debug << "update error = " << error << " for after state" << std::endl << move.after_state();
        exact = move.reward() + update(move.after_state(), alpha * error); // exact = V(s')
    }
}
```

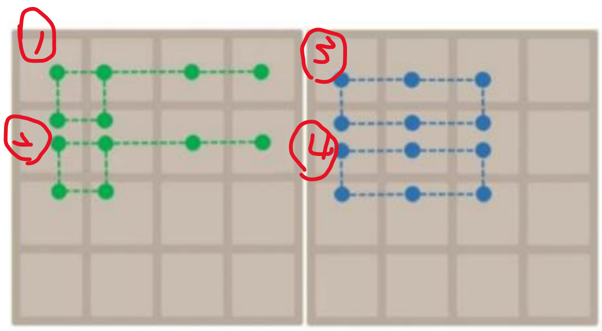
Since the TD learning will use the value of the next state, so we start from the terminal state. In the for loop is the TD(0) leraning and the comments describe the mapping between the variables and the algorithm notations.

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

n-tuple network can be seen as a large lookup table. In this lab, n =6, there are 4 patterns. In addition, a patten has 8 isomorphisms by rotating and mirroring



```
  // initialize the features
1 tdl.add_feature(new pattern({ 0, 1, 2, 3, 4, 5 }));
2 tdl.add_feature(new pattern({ 4, 5, 6, 7, 8, 9 }));
3 tdl.add_feature(new pattern({ 0, 1, 2, 4, 5, 6 }));
4 tdl.add_feature(new pattern({ 4, 5, 6, 8, 9, 10 }));
```



■ Explain the TD-backup diagram of V(state)

$$V(s) \leftarrow V(s) + \alpha(r + V(s'') - V(s))$$

S is the current state.
Alpha is the learning rate.
R is the reward.
S’’ is the next state. After doing an action and the environment producing a new tile.
The above formula is a method for updating our value function.

■ Explain the action selection of V(state) in a diagram

$$a \leftarrow \underset{a' \in A(s)}{\operatorname{argmax}} \ \text{EVALUATE}(s, a')$$

```
function EVALUATE$(s, a)$
    $s', r \leftarrow$ COMPUTE AFTERSTATE$(s, a)$
    $S'' \leftarrow$ ALL POSSIBLE NEXT STATES$(s')$
    return $r + \Sigma_{s'' \in S''} P(s, a, s'') V(s'')$
```

According to the state s and action a, we can first get the reward and the after state, and then get all possible next states by the after state. Finally, calculate the expected return by the return and the transition probability and the value function. Since there are 4 actions, we choose the action which leads to the largest expected return.

- Explain the TD-backup diagram of V(after-state)

$$V(s') \leftarrow V(s') + \alpha(r_{\text{next}} + V(s'_{\text{next}}) - V(s'))$$

Similar to the V(state), the only difference is that s becomes s' now. And s' is the after state of s, which is the state after moving the tile but before producing a new tile. s' next is the after state of s''.

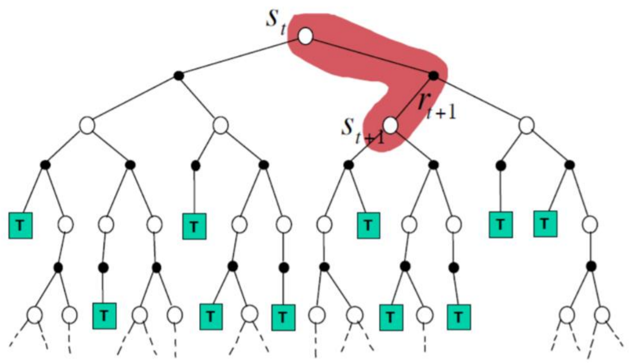
- Explain the action selection of V(after-state) in a diagram

```
function EVALUATE($s, a$)
    $s', r \leftarrow$ COMPUTE AFTERSTATE($s, a$)
    return $r + V(s')$
```

Differs from the action selection of V(state), the expected return is the summation of reward and the value of the after state. So we don't need to consider so many possible states like above.

- Explain the mechanism of temporal difference learning

$$V(S_t) \leftarrow V(S_t) + \alpha\big(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\big)$$



TD learning exploits Markov property, it is efficient because it can learn from the incomplete episodes and don't need to wait until the end of episodes. It only uses the value of next states to learn. And the formula is interpreted in the above question.

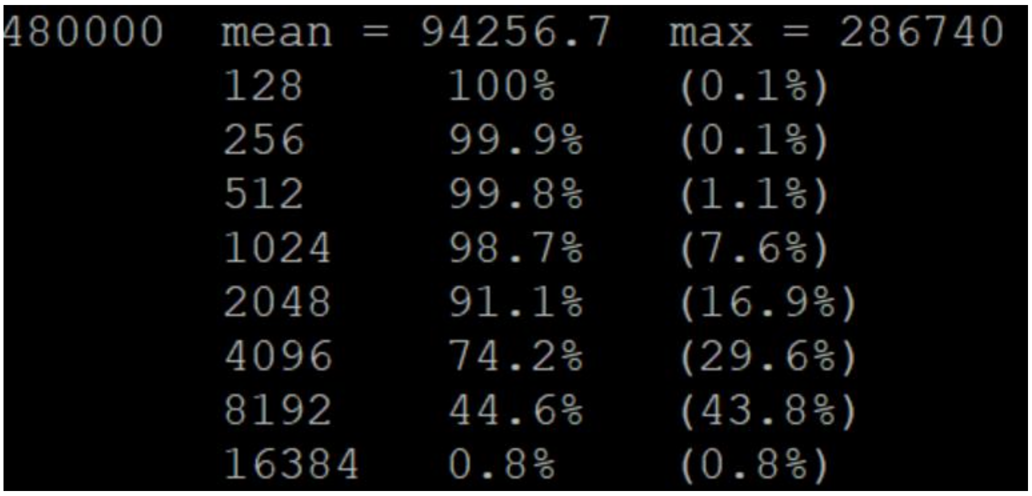
- Explain whether the TD-update perform bootstrapping

Yes, since it uses the value of next states to estimate the sample return (TD target).

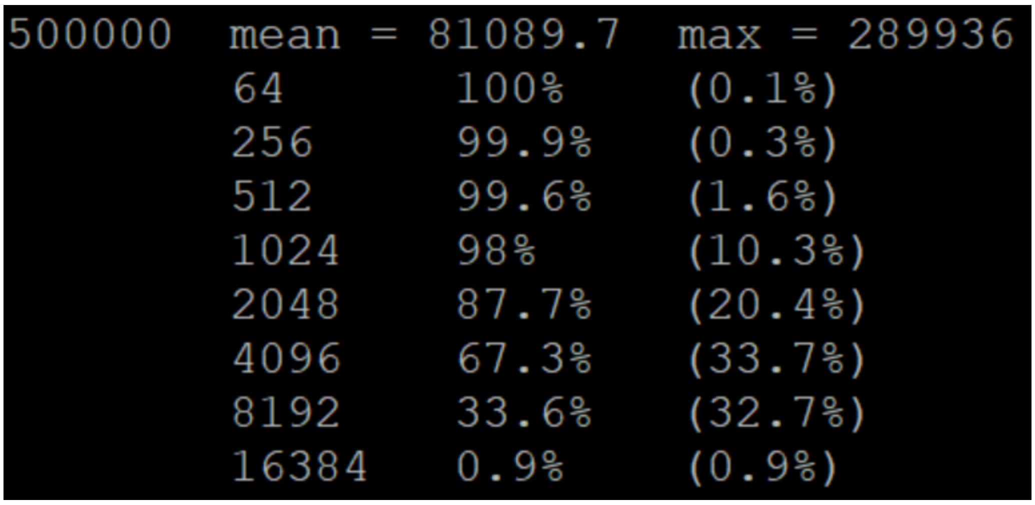
- Explain whether your training is on-policy or off-policy
  On-policy. Because the policies we learn and behave are the same.

- Other discussions or improvements
  Initially, I set the episodes to 100000, and get the win rate about 84%, finally, I tune the episodes to 500000, and the win rate improves from 84% to 90%.

- The 2048-tile win rate in 1000 games
  I totally train the network for 500000 episodes, and the highest win rate in 1000 games occurs on the 480000 episodes. It has 91.1% to win the 2048.



```
480000  mean = 94256.7  max = 286740
        128     100%    (0.1%)
        256     99.9%   (0.1%)
        512     99.8%   (1.1%)
        1024    98.7%   (7.6%)
        2048    91.1%   (16.9%)
        4096    74.2%   (29.6%)
        8192    44.6%   (43.8%)
        16384   0.8%    (0.8%)
```



```
500000  mean = 81089.7  max = 289936
        64      100%    (0.1%)
        256     99.9%   (0.3%)
        512     99.6%   (1.6%)
        1024    98%     (10.3%)
        2048    87.7%   (20.4%)
        4096    67.3%   (33.7%)
        8192    33.6%   (32.7%)
        16384   0.9%    (0.9%)
```