Report of Theory of Computer Game, Project 2

109550027 紀竺均

Part a. Explanation of the Code and Methods I used.

In this assignment, I mainly modified the file agent.h. I have implement the simple 8*4-tuple network and 4*6-tuple network.

First, I created a class weight_slider which inherit from the class weight_agent. Then, I started implement five functions that will list below.

1. open_episode()

```
virtual void open_episode(const std::string& flag = "") {
    trained = 0;
}
```

➔ Set “trained” to zero while a new game begins so the first step of the game won’t pass thru function TDlearn().

2. get_value()

```
double get_value(board& b){
    double value=0;
    //for 8*4 tuple
    /*
    for(int i=0;i<n;i++){
        value += net[i][b2feature(b,i];
    }*/

    //for 4*6 tuple
    for(int i=0;i<2;i++){
        for(int j=0;j<4;j++){
            for(int f=0;f<n;f++){
                value += net[f][b2feature(b,f)];
            }
            b.rotate_clockwise();
        }
        b.reflect_vertical();
    }
    return value;
}
```

➔ Get the value of a “board” by add up all the features’ value. Here, we add up 4 (features) * 8 (directions) = 32 (number of single value) for a board.

3. TDlearn()

```
void TDlearn(double reward){
    double TDerr;
    if(reward==-1) TDerr = alpha*(-get_value(prev));
    else TDerr = alpha*(reward+get_value(next)-get_value(prev));

    /* for 8*4-tuple
    for(int i=0;i<n;i++){
        net[i][b2feature(prev,i)] += TDerr;
    }*/
    // for 4*6-tuple
    for(int i=0;i<2;i++){
        for(int j=0;j<4;j++){
            for(int f=0;f<n;f++){
                net[f][b2feature(prev,f)] += TDerr;
            }
            prev.rotate_clockwise();
        }
        prev.reflect_vertical();
    }
}
```

➔ Calculate the TD error, TDerr = $\alpha(r_{t+1} + V(s'_{t+1}) - V(s'_t))$
reward = -1 means it is the last afterstate, so TD error should be 0.
Then, adjust all the features with the TD error.

4. b2feature() //board to feature

```
long long int  b2feature(board& b,int f){//board to feature
    long long int ret=0;
    int weight[6] = {1,15,225,3375,50625,759375};
    for(int c=0;c<6;c++){
        int cell_index = tup[f][c];
        board::cell tmpcell=b(cell_index);
        ret+=weight[c]*tmpcell;
    }
    return ret;
}
```

➔ This function aims to transfer the current board to a return number that represent the feature. I accomplish this by adding up the number of cell times the power of 15. I'll discuss more in part c about why I did it this way.

5. take_action()

```
virtual action take_action(const board& before) {
    double bestval=-10000000;
    int bestop=-1;
    if(!trained) prev=before; // if the first step
    for(int op=0;op<4;op++){
        board board1 = before;
        board::reward reward1 =board1.slide(op);
        // choose the best operation base on current weight table.
        if(reward1 != -1){
            double value = get_value(board1);
            if(reward1+value>bestval){
                bestval = reward1+value;
                bestop=op;
            }
        }
    }
    next = before;
    board::reward nextreward = next.slide(bestop);
    if(trained) TDlearn(nextreward); // if not the first step
    trained = 1;
    if(bestop!=-1){
        prev = next;
        return action::slide(bestop);
    }
    else{ // bestop==-1 -> no available move
        return action();
    }

}
```

➔ This is the most important function of weight agent. First, tries to choose the best reward and best operation. I get the current board's value by calling get_value(). The agent takes its best operation based on reward + value.
Then, I set the "next" and "prev" board and call function TDlearn() to update my weight table. It's also worth mentioning that I use forward method to updates the table. Finally, I return the action if it's not the last step.

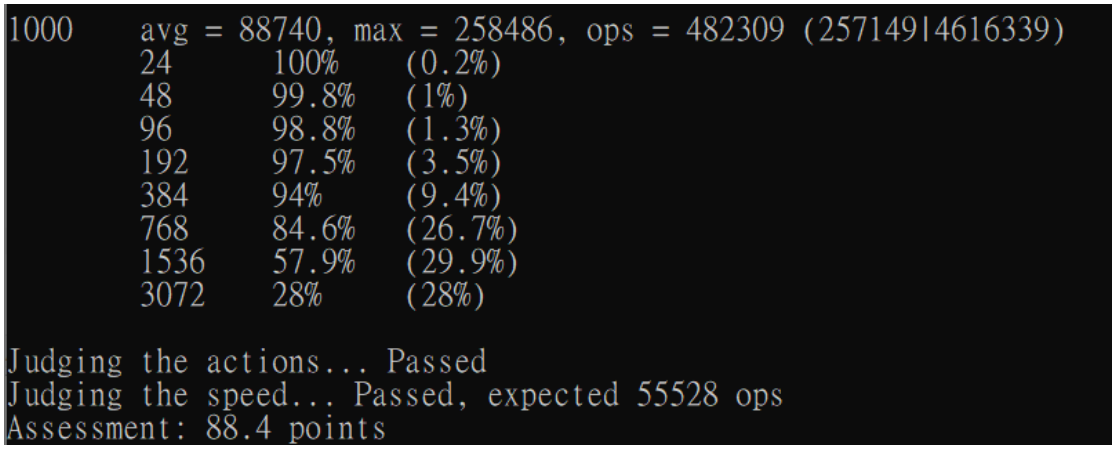
Part b. The Training Process and Result.

For the training of 4*6-tuple:

At first, I set learning rate alpha = 0.1 / 32,

after 100,000+ runs of training, the number of 384 get stuck at 88%, so I set a lower alpha from 0.001 to 0.0001.

Finally, I think the network is converged. I have trained it more than 500,000 runs.



Part c. Problems Encountered and Solution.

The biggest problem I encountered is the speed limit. My program only did about 49,000 operations while the limit is 54,000+ operations (per second). I check everywhere and finally found out it was the function pow(15,n) that occupied a lot of time because I calculate it every time I call the b2feature() function. Thus, I store the power of 15 into an array and call the array instead of calculate it. As a result, my program passed the speed limit and did about 560,000 operations per second.