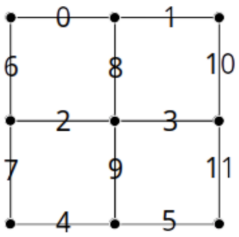
First, we need a way to store the values and policy in a tabular way. We note that each line must have one of two states: exists or doesnt exist. Hence, we use a binary representation to store all 4096 board states (12bits) To do so, we sum over i of state(i)<<i to get a single unique integer representation of the board state. Similarly, to check the current number of completed 1x1 squares, we note that each 1x1 square can be represented by a sequence of bits. That is, the first square can be represented by the binary equivalent of (1<<0)+(1<<6)+(1<<2)+(1<<8) = 325, etc.




```cpp
int tobinary(const Eigen::Vector<int, 12>& state){
  int binarystate=0;
  for(int i=0; i<12; i++){
    binarystate |= (state(i) << i);
  }
  return binarystate;
  //Access values using value[binarystate]
  //Access actions using policy[binarystate]
}
```

Then, to check if the first square is completed, we just have to check whether (state&323)==325 is true where “state” is the binary representation of the board state. This way, we can check the number of completed 1x1 squares in the current board state in a time efficient way.

```cpp
int countsquares(int state){
  int count = 0;
  if((state&325) == 325){
    count++; //upper left square
  }
  if((state&1290) == 1290){
    count++; //upper right square
  }
  if((state&660) == 660){
    count++; //bottom left square
  }
  if((state&2600) == 2600){
    count++; //bottom right square
  }
  return count;
}
```

We initialize two arrays “value” and “policy” arrays as global variables and three main functions that operate one these arrays: “policy_iteration”, “evaluate_policy”, and “improve_policy”. In “policy_iteration”, we begin by initializing the values and actions using for-loops. Then we do policy iteration while the policy is not stable. Inside this loop, we do policy evaluation while the on-policy value function has not converged then improve the policy by taking the argmax over all possible actions of the expected reward under the current value function. The code is as follows (“stable” and “converged” are initialized to 0):

```cpp
  while(!stable){
    stable=1;
    while(!converged){
      converged = 1;
      evaluate_policy(); //Evaluate on-policy value
    }
    converged=0;

    //policy improvement: set policy as argmax over current value function
    improve_policy();
  }
}
```

Inside “evaluate_policy”, we do the following: we loop over all states starting from 4094 (one empty space) and ending with 0(empty board). This will help increase the convergence speed as true values get propagated to previous states. We need to consider two case: 1.an action from the current state receives reward and 2.an action receives no reward. In case 1, the next state is simply the state after the action so we just add the reward and the value of this state to estimate the value of the current state. In case 2, the next state is determined by the opponent’s action. Since the opponent is acting randomly, there is a uniform probability of moving to any one of the possible next states so we just the the average of the values of the next state in using Bellman’s equation to estimate the value of the current state (since there is no reward). While doing this we check for converge by observing where there are any value updates that change the current estimate. If so we set “converged” to 0.

Inside “improve_policy”, we do the following: we loop over all possible states as we had in “evaluate_policy” but the order is not so important now so we just start from 0 and loop to 4094. This code is a little longer due to the additional step of taking argmax over all actions whereas for “evaluate_policy” the action was just the action of the policy. In general, we do: for all states s, for all possible actions from s, check for reward and find the value of the next state to bootstrap to. For each action, we check if this action yields the greater reward and if it does than we save that reward and action for future comparison. After finding argmax over a, we update policy[state] to that action. If the policy has changed, we set “stable” to 0.

```
numstates =0;
valuesum=0;
for(int i=0; i<12; i++){ //loop to sum over V(s')
  if(!(stateaction&(1<<i))){
    nextstate = stateaction | (1<<i);
    valuesum += value[nextstate];
    numstates++;
  }
}

if(value[state] != (reward + valuesum/numstates)){
  value[state] = reward + valuesum/numstates; //update value
  converged = 0;
```

```
    if(actionvalue>maxactionvalue){
      maxactionvalue = actionvalue;
      argmax = j;
    }
  }
}
if(policy[state] != argmax){
  policy[state] = argmax;
  stable = 0;
```

*Left: “evaluate_policy”* *Right: “policy_improvement”*

After the policy has stabilized, the program exits out of policy_iteration and returns the optimal value/optimal action corresponding to the given state:

```
/// DO NOT CHANGE THE NAME AND FORMAT OF THIS FUNCTION
double getOptimalValue(const Eigen::Vector<int, 12>& state){
  policy_iteration();
  return value[tobinary(state)];
}

/// DO NOT CHANGE THE NAME AND FORMAT OF THIS FUNCTION
int getOptimalAction(const Eigen::Vector<int, 12>& state){
  policy_iteration();
  return policy[tobinary(state)];
}
```