I. INTRODUCTION

A. Purpose

This homework consists of designing and implementing a program that plays Chinese Checker. It will exemplify the minimax algorithm, and alpha-beta pruning, and the use of heuristic (evaluation/static) functions to prune the adversarial search.

Chinese checkers is a perfect information game for even-number players. A Chinese Checkers board is shown in Figure 1. The goal of the game is to get 10 pegs or marbles from one's starting position to one's ending position as quickly as possible. Starting and ending positions are always directly across from each other on the board, and players are placed as symmetrically as possible around the board. In a two-player game, the players would start at the top and bottom of the board. The goal of the game is moving all marbles of one color from starting point to the star point on the opposite side of the board.

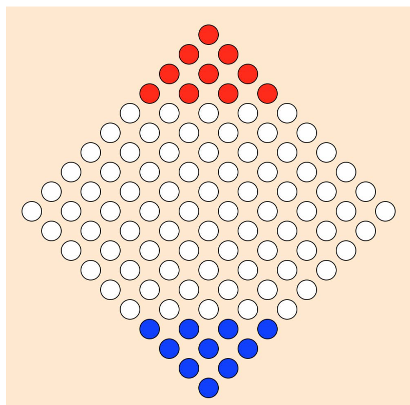


FIG. 1: A typical Two-player Chinese Checker board

Our **purpose** is to occupy the opposite side of the board as fast as possible while following two *special rules* listed as follows:

- One second is limited for each action
- In order to prevent the situation that one of the players may not leave its triangle and prevent the opponent to occupy its space, after 100 turns, if any of the marbles of one player are still in its own triangle, the player loses its game immediately.

Here, we use the minimax algorithm, and alpha-beta pruning, and the use of evaluation functions to prune the adversarial search. Our agent is called **iRoboticAgent**.

B. Environment

- Ubuntu 16.04
- Python 3.7.1
- Vscode 2019
- Pycharm 2018.2.4 x64

C. Procedure

1. *evaluation()* quantizes current situation for each board state, parameters are tuned along side
2. *minimaxValue()* implements alpha-beta pruning based on *evaluation()*
3. *defAction()* updates action based on *minimaxValue()*, and eliminates ”stuck” situations

II. IMPLEMENTATION

A. Evaluation as Utility Function

In Chinese Checker game, the agent should try his best to jump further and prevent the opposite agent's peg to do so. As a result, we mainly consider two aspects concerning the *evaluation()* function.

For vertical part, we consider that for each action(move or hop), the longer the action's vertical component is, the better it will be. This part resembles the *SimpleGreedyAgent* to a great extent. Emphasis on the vertical component makes sense, since it indicates how fast the peg is heading towards the destination.

For horizon part, we also consider in two aspects. First, it is best for our agent to stay in the center of the board, while the opponent stays in two sides. Second, however, it is not always the best to stay exactly in the center. This is because the board is not exactly vertical and horizon, and it is kind of leaned. Therefore, we assume in *evaluation()* that the middle is a little bit big, and the place adjacent to the center in each row is the smallest, and from that on, the farther from the middle of row, the larger the evaluation is. Figure 2 shows our horizon evaluation parameters for each position.

For vertical vs horizon, undoubtedly we consider the vertical part more important than the horizon part, since vertical is a matter of victory or failure, while horizon is only concerned with how fast and appropriate to reach the goal.

Based on these analysis and relevant tests using different parameters, we tuned relevant parameters and finally come up with our *evaluation()* function as our **utility function** concerning each action.

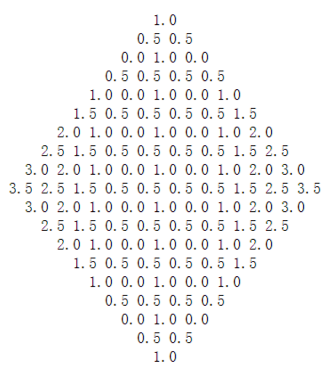


FIG. 2: Horizon evaluation parameters for each position

B. Minimax Using Alpha-Beta Pruning

Based on the *evaluation()* function, we now have a way to evaluate each action, thus implementing minimax algorithm to explore the potential action to maximize our agent's advantage while minimizing the opponent's. For the exploration **depth** which is greatly concerned with the outcome, we use **iterative depth minimax**

algorithm which aggregate the current depth by one automatically if the potential action for the current depth is acquired within one second. During our test, due to the one second limit, normally the depth is two. That is to say, we could actually set the depth to be exactly two. Nevertheless, for the robustness of the program, we recurse the depth.

Also, the idea of **Monte Carlo Tree Search**(MCTS)is utilized here for the **breadth** of each layer. We do not take all the nodes in current layer into the next layer. Instead, we choose some node based on our *evaluation()* function, and only explore the sons of these nodes. That is to say, we assume the leaves of top prioritized nodes to be the internode, which finally leads to victory. This resembles MCTS's idea, which select, expand and simulate the node which is rarely explored but has a promising outcome.

Here, there is a major **difference** from typical minimax algorithm to declare. We use two parallel grading systems. One is for our agent iRoboticAgent, the other is for the opponent agent. For our agent the larger the value is, the better the action's outcome will be. And we always assume that the opponent chooses his largest-value action to go, which is our smallest-value action.

C. getAction function

Based on the *minimaxValue()* function, we now have a powerful tool to determine the next action. However, there are two things to consider.

The first is the **one-second** limit, which means we have to update each action each time it pops out. This makes the structure of getAction function a little bit resemble the **Depth-first Search**(DFS) structure.

Besides, we have to deal with the **"stuck"** situation, which means that iRoboticAgent repeats his previous actions again and again without jumping out of the loop. This is doomed to fail. Therefore, we create a list of length three to store previous actions and compare the new action with previous ones to judge if there is anything wrong. If iRoboticAgent does repeat his action, we call the SimpleGreedyAgent's function to instruct iRoboticAgent on the next action. If, however, unfortunately again, the action called from SimpleGreedyAgent is just the same as the previous action, we have no choice but to call the RandomAgent's function to give iRoboticAgent guidance. As long as there's other possible action, there will be no loops. Thus, the problem is addressed.

III. CONCLUSION & DISCUSSION

Conclusion Our iRoboticAgent could beat SimpleGreedyAgent at 10:0(see Figure 3), let alone RandomAgent. We are pleased at this. However, much can be polished further.

Discussion The Chinese Checker task is difficult since feedback takes a much longer time. Whenever we are to put forward a new idea in our algorithm, it needs much time to see the results run by the program. This is time-consuming and a test of our patience. Feedback's importance is needless to say, but it consumes quite a lot of time. We will still go on searching methods which could significantly reduce the amount of time to get feedback.

Further Trial We also take **machine learning** method into account during this homework, you can see our implementation below, with relevant code in Appendix V B and neural network structure in Figure 4.

A. Machine Learning Trial

In this subsection we will also introduce how we try to use machine learning to solve the Chinese Checker problem.

1. Environment

- Ubuntu 18.04

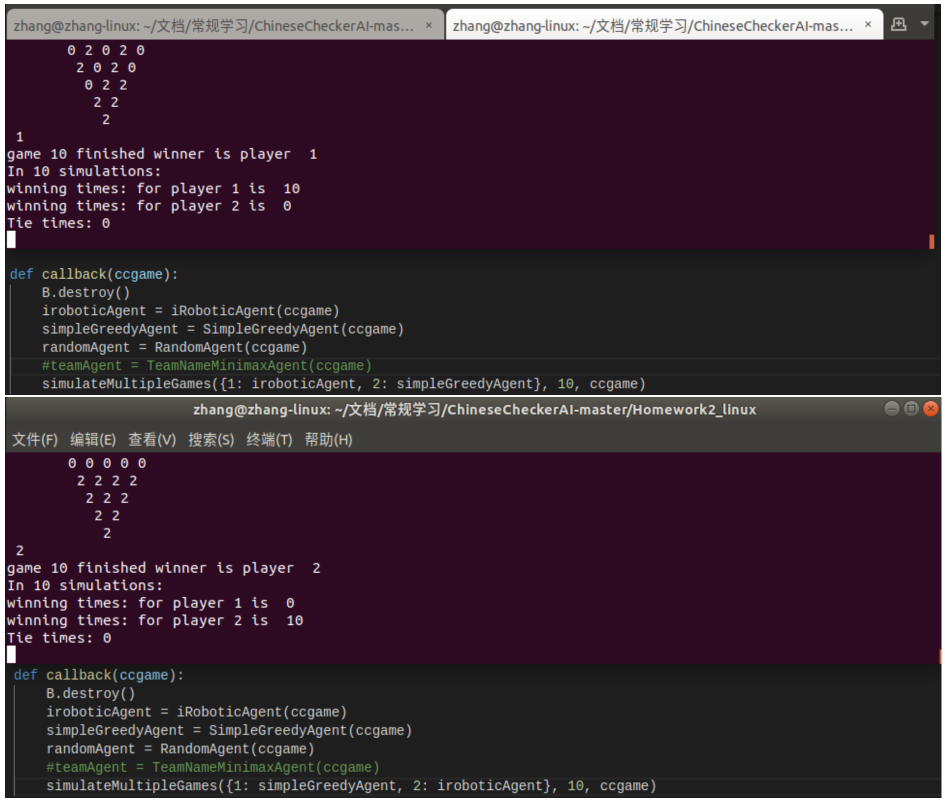


FIG. 3: iRoboticAgent VS SimpleGreedyAgent

- Python 3.7.0
- CUDA 10.1
- pytorch 1.2.0

2. Ideas

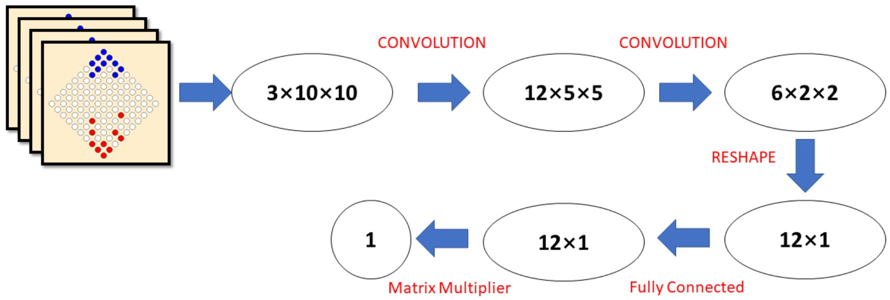
It's too difficult to design a model to play the game like AlphaGo which can totally make the decision by itself. So we still use Minimax with Alpha-Beta Pruning to find the best solution, with a brand-new evaluation function given by network.

However, we don't have any data for ChineseCheckerwhich means we can only use unsupervised method to training our model.So the essential issue is that how to design our loss functions and how to train the model.

We randomly generated many chess boards for the network to evaluate. The network will output a number for each board as its evaluation for the state. Then we use traditional method(using raw and col to compute) as ground truth to evaluate the board and use it to supervise the training of model.

3. Reasons for the failure

1. The structure of network is terrible, which means we can't extract the features of the board.



structure.png

FIG. 4: Neural Network Structure

2. We don't have enough time to design some better loss functions to guide the training of our model.
3. We don't have powerful GPU for our training, which reduced our efficiency.

IV. ACKNOWLEDGEMENT

We appreciate the teaching assistants and our professor Yue Gao, who have participated in constructing a good platform for us to concentrate on the design of our our AI agent.

Also, we pays tribute to relevant papers and theses online to guide us jumping out of the box to finalize our design. These links are appended as follows:

1. http://bnaic2010.uni.lu/Papers/Category%20B/Nijssen.pdf
2. https://www.cs.du.edu/ sturtevant/papers/UCT-endgame.pdf

V. APPENDIX

A. Source Code

Source code in agent.py is appended as follows for reference. This is also our **final** code for the tournament.

```

import random, re, datetime


class Agent(object):
    def __init__(self, game):
        self.game = game

    def getAction(self, state):
        raise Exception("Not implemented yet")


class RandomAgent(Agent):
    def getAction(self, state):
        legal_actions = self.game.actions(state)
        self.action = random.choice(legal_actions)
```

```

class SimpleGreedyAgent(Agent):
    # a one-step-lookahead greedy agent that returns action with max vertical advance
    def getAction(self, state):
        legal_actions = self.game.actions(state)

        self.action = random.choice(legal_actions)

        player = self.game.player(state)
        if player == 1:  # playing from bottom to top, hence later vertical value is smaller
            max_vertical_advance_one_step = max([action[0][0] - action[1][0] for action in legal_actions])
            max_actions = [action for action in legal_actions if action[0][0] - action[1][0] == max_vertical_advance_one_step]
        else:  # playing from top to bottom, hence later vertical value is larger
            max_vertical_advance_one_step = max([action[1][0] - action[0][0] for action in legal_actions])
            max_actions = [action for action in legal_actions if action[1][0] - action[0][0] == max_vertical_advance_one_step]
        self.action = random.choice(max_actions)


class iRoboticAgent(Agent):
    def getAction(self, state):
        legal_actions = self.game.actions(state)
        self.action = random.choice(legal_actions)

        player = self.game.player(state)
        ### START CODE HERE ###

        global staticDepth, cnt, repeat  # store the initial value for depth for minimax
        depth = 2                        # starting from 2
        staticDepth = depth              # store original depth
        memory = 20                      # breadth for each layer, max is 42
        board = state[1]
        player_status = board.getPlayerPiecePositions(player)

        temp = (player_status in q)
        q.append(player_status)

        cnt += 1

        if cnt >= 3 and temp:            # in case of "stuck" situation, use "greedy"
            q.remove(q[0])
            repeat += 1
            if repeat == 2:              # in case of repeated "stuck" situation
                self.action = random.choice(legal_actions)
            else:
                if player == 1:          # playing from bottom to top, hence later vertical value is smaller
                    max_vertical_advance_one_step = max([action[0][0] - action[1][0] for action in legal_actions])
                    max_actions = [action for action in legal_actions if action[0][0] - action[1][0] == max_vertical_advance_one_step]
                else:                    # playing from top to bottom, hence later vertical value is larger
                    max_vertical_advance_one_step = max([action[1][0] - action[0][0] for action in legal_actions])
                    max_actions = [action for action in legal_actions if action[1][0] - action[0][0] == max_vertical_advance_one_step]
```

```
                self.action = random.choice(max_actions)

        else:
            if cnt >= 3:
                q.remove(q[0])
            repeat = 0
            v = -float('inf')

            order = PriorityQueue()      # search with preference
            for action in legal_actions:
                order.put((-(3 - 2 * player) * (action[0][0] - action[1][0]), action))
            order_next = PriorityQueue()

            while True:
                count = 0
                while (not order.empty()) and (memory > count):
                    action = order.get()[1]
                    if self.evaluation(self.game.succ(state, action), player) == 1000:
                        self.action = action
                        break
                    count += 1
                    v_next = self.minimaxValue(player, self.game, self.game.succ(state,
                                                                  action), depth, v, float('
                                                                  inf'),
                                                memory)
                    order_next.put((-v_next, action))
                    if v_next > v:
                        v = v_next
                        self.action = action

                depth += 1
                staticDepth = depth
                del order
                order = PriorityQueue()
                while not order_next.empty():
                    order.put(order_next.get())

    # minimax algorithm
    def minimaxValue(self, player, game, state, depth, alpha, beta, memory):

        if depth == 0:
            return self.evaluation(state, player)
        else:
            depth -= 1

        if (staticDepth - depth) % 2 == 1:
            if self.evaluation(state, player) == 1000:
                return 1000

            v = float('inf')
            order = PriorityQueue()
            for action in game.actions(state):
                order.put(((3 - 2 * player) * (action[0][0] - action[1][0]), action))

            count = 0
            while (not order.empty()) and memory > count:
                action = order.get()[1]
                count += 1
                v = min(v, self.minimaxValue(player, game, game.succ(state, action), depth,
                                                              alpha, beta, memory))
                if v <= alpha: return v
                beta = min(beta, v)
            # print("depth:", depth)
            return v

```

```python
        else:
            if self.evaluation(state, player) == -1000:
                return -1000

            v = -float('inf')
            order = PriorityQueue()
            for action in game.actions(state):
                order.put((-(3 - 2 * player) * (action[0][0] - action[1][0]), action))

            count = 0
            while (not order.empty()) and memory > count:
                action = order.get()[1]
                count += 1
                v = max(v, self.minimaxValue(player, game, game.succ(state, action), depth, alpha, beta, memory))
                if v >= beta: return v
                alpha = max(alpha, v)
            return v

    # heuristic evaluation
    def evaluation(self, state, player):  # this function returns a number as the evaluation value of a given state

        board = state[1]
        player_status = board.getPlayerPiecePositions(player)
        opponent_status = board.getPlayerPiecePositions(3 - player)

        # vertical dimension
        player_vertical_count = 0
        for position in player_status:
            player_vertical_count += position[0]

        opponent_vertical_count = 0
        for position in opponent_status:
            opponent_vertical_count += position[0]

        # horizon dimension
        player_horizontal_count = 0
        for position in player_status:
            if position[0] % 2 == 1:
                if position[1] == (position[0] + 1) / 2:
                    player_horizontal_count += 1
                else:
                    player_horizontal_count += abs(position[1] - (position[0] + 1) / 2) - 1
            else:
                if position[1] == (position[0] / 2) or (position[0] / 2 + 1):
                    player_horizontal_count += 0.5
                else:
                    player_horizontal_count += abs(position[1] - (position[0] + 1) / 2) - 1

        opponent_horizontal_count = 0
        for position in opponent_status:
            if position[0] % 2 == 1:
                if position[1] == (position[0] + 1) / 2:
                    opponent_horizontal_count += 1
                else:
                    opponent_horizontal_count += abs(position[1] - (position[0] + 1) / 2) - 1
            else:
                if position[1] == (position[0] / 2) or (position[0] / 2 + 1):
                    opponent_horizontal_count += 0.5
                else:
                    opponent_horizontal_count += abs(position[1] - (position[0] + 1) / 2) - 1
```

```
        # final calculation
        if player == 1:
            if player_vertical_count == 30:                                          # you
                                                             win!
                return 1000
            if opponent_vertical_count == 170:                                       # you
                                                             lose!
                return -1000
            else:
                return 400 - (player_vertical_count + opponent_vertical_count) + (  # the
                                                                 more the better
                        opponent_horizontal_count - player_horizontal_count) / 2
        else:
            if player_vertical_count == 170:
                return 1000
            if opponent_vertical_count == 30:
                return -1000
            else:
                return (player_vertical_count + opponent_vertical_count) + (        # the
                                                                 more the better
                        opponent_horizontal_count - player_horizontal_count) / 2

# three global variables( out of the class)
cnt = 0
repeat = 0
q = []

from queue import PriorityQueue

        ### END CODE HERE ###
```

B. Trial Using Machine Learning Method

We have tried machine learning method, but the outcome is not as good as the original one without utilizing machine learning. We append our machine learning code as follows:

1. NNmodel.py

```
import pdb

import sys
import os

import random
import numpy as np
from collections import deque
import torch
from torch.autograd import Variable
import torch.nn as nn


class StateNet(nn.Module):
    def __init__(self,):
        super(StateNet,self).__init__()
        #input [4, 3, 10, 10] [B, S, H, W]
        self.conv1 = nn.Sequential(
            nn.Conv2d(in_channels=3, out_channels=12, kernel_size=4, stride=2, padding=1),
            nn.ReLU(inplace=True)
        )  # torch.Size([4, 12, 5, 5])
```

```
        self.conv2 = nn.Sequential(
            nn.Conv2d(in_channels=12, out_channels=6, kernel_size=4, stride=1, padding=0),
            nn.ReLU(inplace=True)
        )  # torch.Size([4, 6, 2, 2])

        self.fc1 = nn.Sequential(
            nn.Linear(24,12),
            nn.ReLU()
        )
        self.out = nn.Linear(12,1)

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = x.view(x.size(0),-1)
        x = self.fc1(x)
        return self.out(x)
```

2. *loss$_f$unction.py*

```
import torch
from torch.autograd import Variable

import torch.nn as nn
import numpy as np

# heuristic evaluation
def Vertical_loss(board):  # this function returns a number as the evaluation value of a given state
    b,_,_,_ = board.size()
    board_judgement = []
    for i in range(0,b):# very stupid method but i dont have time to find a better way
        # print(board.size())
        state = converse_board_to_state(board[i,:,:,:])
        player_vertical_count = 0
        opponent_vertical_count = 0

        # print(state)

        for location, chess in state.items():
            # print(location)
            if chess is 2:
                player_vertical_count += location[0]
            if chess is 1:
                opponent_vertical_count += location[0]


        # final calculation

        if player_vertical_count == 170:
            player_vertical_count =  1000
        if opponent_vertical_count == 30:
            opponent_vertical_count =  -1000

        board_judgement.append(torch.tensor(player_vertical_count+opponent_vertical_count))


    return torch.cat((board_judgement[0].unsqueeze(0),
                      board_judgement[1].unsqueeze(0),
                      board_judgement[2].unsqueeze(0),
                      board_judgement[3].unsqueeze(0))
```

```
                     ,0)



def converse_state_to_board(state,OurPlayer = 2):

    def converse_one_by_one(location,player):
        if player is 1:
            if location[0] >10:
                x,y = -location[1]+10, location[0]+location[1]-11
            else :
                x,y = location[0]-location[1], location[1]-1
            opponent_board[x, y] = 1
            blank_board[x, y] = 0

        if player is 2 :
            if location[0] >10:
                x,y = -location[1]+10, location[0]+location[1]-11
            else :
                x,y = location[0]-location[1], location[1]-1
            our_board[x, y] = 1
            blank_board[x, y] = 0

    board = state
    our_board = np.zeros((10,10))
    opponent_board = np.zeros((10,10))
    blank_board = np.ones((10,10))
    for location, player in board.board_status.items() :
        converse_one_by_one(location, player)

    if OurPlayer is 1:
        return np.concatenate((opponent_board[np.newaxis,:,:],our_board[np.newaxis,:,:],
                                                    blank_board[np.newaxis,:,:]),axis=0)
    else :
        return np.concatenate((our_board[np.newaxis,:,:],opponent_board[np.newaxis,:,:],
                                                    blank_board[np.newaxis,:,:]),axis=0)


def converse_board_to_state(board):
    """
    return a dic as (x,y):1/2/0
    """
    state = {}
    our_board = board[0,:,:]
    opponent_board = board[1,:,:]

    for a in range(0,10):
        for b in range(0,10):
            if a+b<10:
                x = a+b+1
                y = b+1
            else :
                x = a+b+1
                y = 10-a
            if abs(our_board[a,b].item() - 1)<0.1:
                state[(x,y)] = 2

            if abs(opponent_board[a,b].item()-1)<0.1:
                state[(x,y)] = 1

    # print(len(state)) 20

    return state
```