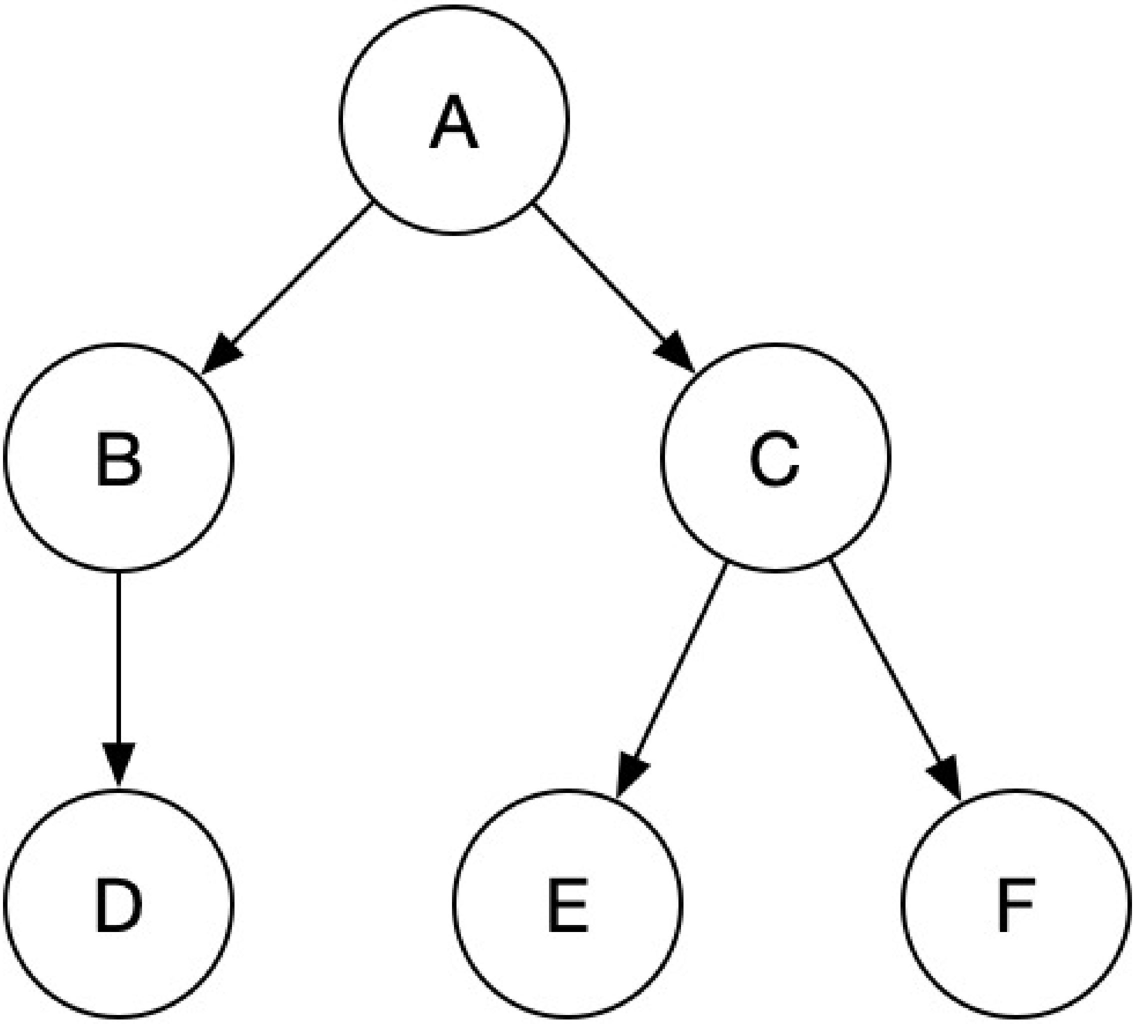
Homework 2

Part 1: Graph or Tree

1.

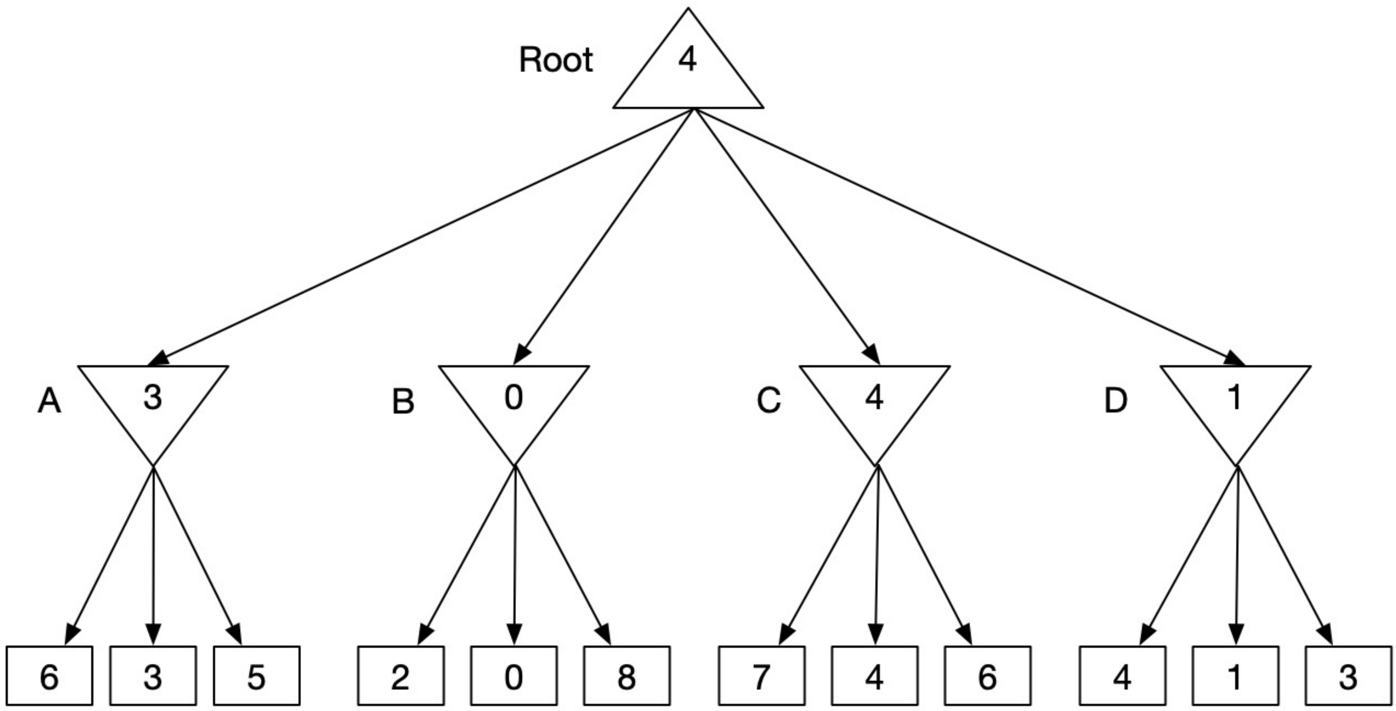


2. A: {2} B: {1, 2} C: {1} D: {1, 2} E: {0, 1, 2} F: {1, 2}
3. C=1
   If (C,E,F)=(2,2,2), there is a conflict that E should be greater than C, So we reassign C,　there are two options 0 and 1. If we choose C=0, then it conflicts that C>=F, but F!=G, so if G=0 we can't assign F right. But if we

assign C=1, F could be 0 or 1, no matter what G is there is at least one choice for F to be assigned. So based on the LCV heuristics, we should reassign C=1.

Part 2: Boring Matrix "Game"

1.



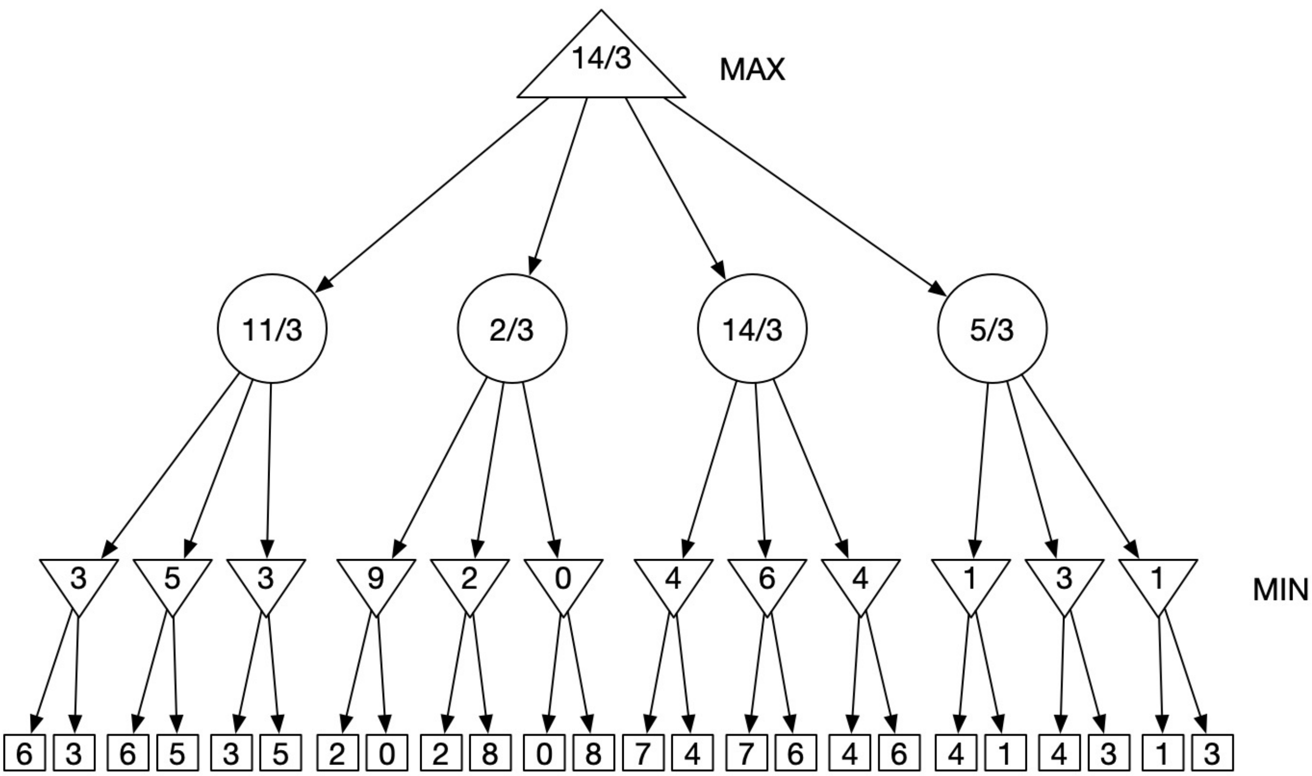
2.

step	node	α	β	value	children skipped
1	Root	$-\infty$	∞	4	-
2	A	$-\infty$	∞	3	-
3	6	$-\infty$	∞	6	-
4	3	$-\infty$	6	3	-
5	5	$-\infty$	3	5	-
6	B	3	∞	0	-

step	node	α	β	value	children skipped
7	2	3	∞	2	0,8
8	C	3	∞	4	-
9	7	3	∞	7	-
10	4	3	7	4	-
11	6	3	4	4	-
12	D	3	∞	1	-
13	D	4	∞	4	1,3

So the player A should choose the third row.

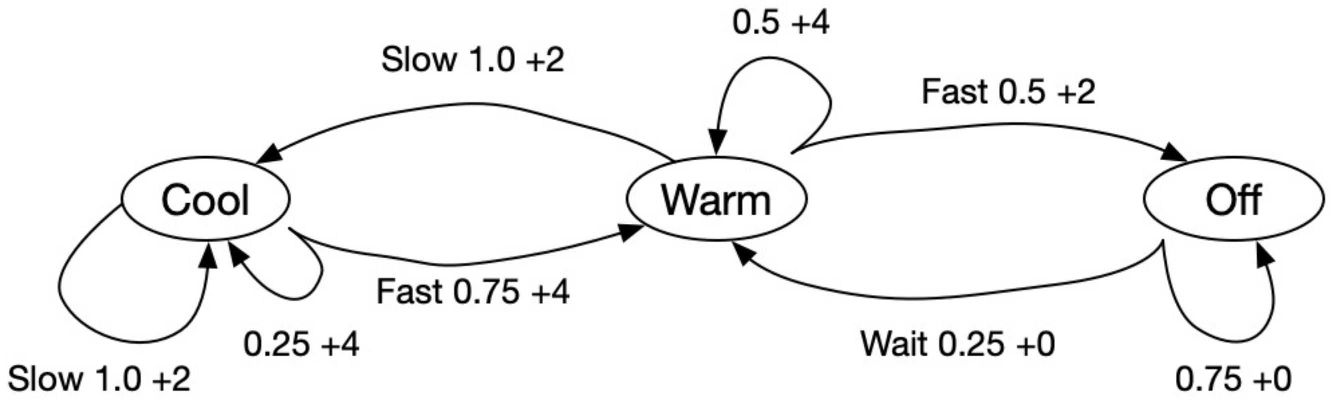
3.



A should choose the third row, and the expected game value is $14/3$.

Part 3: Immortal Race Car

1.



2. $V^\pi(Cool) = 1 \times (2 + 0.8(V^\pi(Cool))$
   $V^\pi(Warm) = 1 \times (2 + 0.8(V^\pi(Cool))$
   $V^\pi(Off) = 0.25 \times (0 + 0.8(V^\pi(Warm)) + 0.75 \times (0 + 0.8(V^\pi(Warm))$
   Solve the equations above, we can get:
   $V^\pi(Cool) = 10$
   $V^\pi(Warm) = 10$
   $V^\pi(Off) = 5$
   As for V(Cool) and V(Warm), because no matter the car is cool or warm, the policy just tells it to goes slow, so there is no chance for the car to change from warm to off, so they produce the same result for the car.
   As for V(wait), The car just waits to get the chance to be warm again and during the waiting there is no reward for the car, so the value of it is smaller than the other two values.
3. $V_1^\pi(Cool) = max(1 \times 2 + 0.8 \times 10, 0.75 \times (4 + 0.8 \times 10) + 0.25 \times (4 + 0.8 \times 10)) = max(10, 12) =$**12**
   $V_1^\pi(Warm) = max(1 \times 2 + 0.8 \times 10, 0.5 \times (4 + 0.8 \times 10) + 0.5 \times (2 + 0.8 \times 5) = max(10, 9) =$**10**
   $V_1^\pi(Off) = max(0.25 \times 0.8 \times 10 + 0.75 \times 0.8 \times 5) =$**5**
   The policy changes to direct the car to go fast when it's cool, other actions

remain the same.
For the state of Cool, the car can simply go fast than slow to gain more reward, and no matter it becomes cool or fast, the values of these two states are the same in the previous steps, so there is no risk of get into a low-value states when going fast(no risk to be overheated).

4.

State	$V_0(s)$	$V_1(s)$	$V_2(s)$
Cool	0	4	6.6
Warm	0	3	5.2
Off	0	0	0.6

$V_1(Cool) = max(1 \times (2), 0.75 \times 4 + 0.25 \times 4) = max(2, 4) = 4$
$V_1(Warm) = max(1 \times 2, 0.5 \times 4 + 0.5 \times 2) = max(2, 3) = 3$
$V_1(Off) = max(0) = 0$
$V_2(Cool) = max(1 \times (2 + 0.8 \times 4), 0.75 \times (4 + 0.8 \times 3) + 0.25 \times (4 + 0.8 \times 4)) = max(5.2, 6.6) = 6.6$
$V_2(Warm) = max(0.5 \times 2 + 0.5 \times (4 + 0.8 \times 3), 1 \times (2 + 0.8 \times 4)) = max(4.2, 5.2) = 5.2$
$V_2(Off) = max(0.25 \times (0 + 0.8 \times 3)) = 0.6$

5. Based on the answer above, the optimal policy is that when car is Cool, go fast; when car is Warm, go slow; When car is Off, just wait.
For the policy in step 2, when the car is Cool it lets it go slow, which is suboptimal because the car will gain more if it goes fast.
For the policy in step 3, the two policies are the same because in the step 2 through one step policy iteration it has learned that when it's Cool it's better to go fast and when it's warm, there is a relatively great chance for it to get overheated and the penalty of being overheated is huge cause the value of state Off is really low. In all, using just one step policy iteration the policy has converged to a optimal policy.

Programming Portion

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*

"""
COMS W4701 Artificial Intelligence - Programming Homework 2

An AI player for Othello. This is the template file that you need to
complete and submit.

@author:  Haoyu Yan hy2574
"""

import random
import sys
import time

# You can use the functions in othello_shared to write your AI
from othello_shared import find_lines, get_possible_moves, get_score

def compute_utility(board, color):
    anti_color = -color + 3
    count = 0
    anti_count = 0
    for i in range(len(board)):
        for j in range(len(board)):
            if board[i][j] == color:
                count += 1
            elif board[i][j] == anti_color:
                anti_count += 1

    return count - anti_count


############ MINIMAX ###############################

def minimax_min_node(board, color):
    anti_color = 3 - color
    moves = get_possible_moves(board, anti_color)
    if not moves:
        return compute_utility(board, color)
    mini_score = float("inf")

    for move in moves:
```

```python
        new_board = play_move(board, anti_color, move[0], move[1])

        score = minimax_max_node(new_board, color)

        if score<mini_score:
            mini_score = score

    return mini_score


def minimax_max_node(board, color):

    moves = get_possible_moves(board, color)
    if not moves:
        return compute_utility(board, color)
    max_score = float("-inf")

    for move in moves:
        new_board = play_move(board, color, move[0], move[1])
        score = minimax_min_node(new_board, color)
        if score>max_score:
            max_score = score

    return max_score


def select_move_minimax(board, color):
    moves = get_possible_moves(board, color)
    max_score = float("-inf")
    best_move = [moves[0][0],moves[0][1]]
    for move in moves:
        new_board = play_move(board, color, move[0], move[1])
        score = minimax_min_node(new_board, color)
        if score>max_score:
            max_score = score
            best_move[0] = move[0]
            best_move[1] = move[1]

    return best_move[0], best_move[1]


############ ALPHA-BETA PRUNING #####################
```

```
states_cache = {}
limit = 7
#alphabeta_min_node(board, color, alpha, beta, level, limit)
def alphabeta_min_node(board, color, alpha, beta, level, limit):
    anti_color = 3 - color
    moves = get_possible_moves(board, anti_color)
    if not moves:
        states_cache[board] = compute_utility(board, color) #update
        return compute_utility(board, color)
    mini_score = float("inf")
    boards = []

    for move in moves:
        boards.append(play_move(board, anti_color, move[0], move[1])
    boards.sort(key=lambda x:compute_utility(x,color))

    for new_board in boards:
        # new_board = play_move(board, anti_color, move[0], move[1])
        if new_board in states_cache:
            score = states_cache[new_board]
        else:
            if level>=limit:
                score = compute_utility(new_board, color)
            else:
                score = alphabeta_max_node(new_board, color, alpha,
                states_cache[new_board] = score

        if score<mini_score:
            mini_score = score
        if mini_score<= alpha:
            return score
        beta = min(beta, mini_score)
    return mini_score


#alphabeta_max_node(board, color, alpha, beta, level, limit)
def alphabeta_max_node(board, color, alpha, beta, level, limit):

    moves = get_possible_moves(board, color)
    if not moves:
        states_cache[board] = compute_utility(board, color) #update
        return compute_utility(board, color)
    max_score = float("-inf")
```

```
    boards = []
    for move in moves:
        boards.append(play_move(board, color, move[0], move[1]))
    boards.sort(key=lambda x:compute_utility(x,color), reverse=True)

    for new_board in boards:
        # new_board = play_move(board, color, move[0], move[1])
        if new_board in states_cache:
            score = states_cache[new_board]
        else:
            if  level>=limit:
                score = compute_utility(new_board, color)
            else:
                score = alphabeta_min_node(new_board, color, alpha, 
                states_cache[new_board] = score

        if score>max_score:
            max_score = score
        if max_score>=beta:

            return max_score
        alpha = max(alpha, max_score)

    return max_score



def select_move_alphabeta(board, color):
    moves = get_possible_moves(board, color)
    max_score = float("-inf")
    best_move = [0, 0]
    alpha = float("-inf")
    beta = float("inf")
    for move in moves:
        new_board = play_move(board, color, move[0], move[1])
        score = alphabeta_min_node(new_board, color, alpha, beta,1, 
        if score>max_score:
            max_score = score
            best_move[0] = move[0]
            best_move[1] = move[1]
        alpha = max(alpha, max_score)

    return best_move[0], best_move[1]
```

```
####################################################
def run_ai():
    """
    This function establishes communication with the game manager.
    It first introduces itself and receives its color.
    Then it repeatedly receives the current score and current board s
    until the game is over.
    """
    print("Minimax AI") # First line is the name of this AI
    color = int(input()) # Then we read the color: 1 for dark (goes 
                         # 2 for light.

    while True: # This is the main loop
        # Read in the current game status, for example:
        # "SCORE 2 2" or "FINAL 33 31" if the game is over.
        # The first number is the score for player 1 (dark), the sec
        next_input = input()
        status, dark_score_s, light_score_s = next_input.strip().spl
        dark_score = int(dark_score_s)
        light_score = int(light_score_s)

        if status == "FINAL": # Game is over.
            print
        else:
            board = eval(input()) # Read in the input and turn it in
                                  # object. The format is a list of
                                  # squares in each row are represen
                                  # 0 : empty square
                                  # 1 : dark disk (player 1)
                                  # 2 : light disk (player 2)

            # Select the move and send it to the manager
            # movei, movej = select_move_minimax(board, color)
            movei, movej = select_move_alphabeta(board, color)
            print("{} {}".format(movei, movej))


if __name__ == "__main__":
    run_ai()
```