과제 1: 탐색을 통한 문제해결 [5팀]

과목명: 인공지능(목 7-9)
교수명: 한경수 교수님
팀원: 20190274 정택원
20201092 정유진
20210833 남민지

목차

1. 문제 소개

→ 본 팀의 주제는 적대적 탐색 알고리즘을 이용하여 오셀로(Othello, 리버시)게임을 탐색한다.

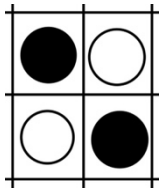
→ 오셀로 게임이란 두 사람이 하는 반상(盤上) 게임의 하나. 64구획의 반에 흑백 표리(表裏)로 된 동그란 말을 늘어 놓고 상대편의 말을 자기의 말 사이에 끼이게 하여 자기 말의 색깔로 바꾸어 가면서 승패를 결정한다.[1]

2. 문제 선택 배경

→ 적대적 탐색은 탐색을 하는 과정에 직접적으로 영향을 주는 입력을 사용자로부터 받아 직접 플레이할 수 있다는 것에 흥미를 느꼈다.

→ 작업 환경 특징이 완전 관측 가능, 결정적, 순차적, 정적, 이산적인 게임 중에서 보드게임을 선정했으며, 게임의 룰이 명확한(알려진 환경) 오셀로 게임을 선정하였다.

3. 문제 정의



상태: 보드의 가능한 상태 경우의 수 집합

초기 상태: 보드의 중앙에 흑 돌 2개 백 돌 2개가 위치한 상태

행동: 상대의 돌을 최소 1개 뒤집을 수 있는 좌표의 집합

이행 모델: 플레이어가 돌을 놓은 이후 보드 상태

종료 검사: 흑 돌과 백 돌 플레이어 모두 돌을 둘 수 없으면 종료

효용 함수: 종료 상태에서 상대와의 돌 개수의 차이(돌을 두는 비용은 모두 1로 같으며, 본 팀은 행동비용함수를 활용하지 않아 효용함수로 대체)

[1] https://ko.dict.naver.com/#/entry/koko/59e4e2207a5d47c1b19c9f3ae0bac6a2 (네이버 국어사전)

4. 목표

→ 알파-베타 가지치기 탐색, 휴리스틱 알파-베타 가지치기 탐색, 무작위 보행 탐색을 이용하여 게임의 결과를 사용자에게 보여줄 수 있다.

→ 더 나아가 하나의 에이전트는 사용자, 다른 하나는 원하는 탐색 방법을 지정하여 사용자가 직접 플레이할 수 있다.

→ 본래의 게임은 8x8크기의 보드판이지만, 사용자 입력으로 원하는 크기를 설정할 수 있다.

5. 코드 설명

```
def play_game(game, strategies: dict, verbose=False):
    """번갈아 가면서 두는 게임 진행 함수.
    strategies: {참가자 이름: 함수} 형태의 딕셔너리.
    여기서 함수(game)는 참가자의 수(move)를 찾는 함수"""

    while not game.is_terminal():
        player = game.to_move
        move = strategies[game.to_move](game)
        if move:
            game = game.result(move)
        else:
            game.turn_change()
        if verbose:
            print('Player', player, 'move:', move)
            game.display()
    return game
```

→ 게임을 시작할 수 있게 하는 함수이다.

→ 매개변수 game에는 사용자가 원하는 크기의 보드판으로 만들어진 오셀로 게임 객체가 들어간다.

→ 매개변수 strategies에는 플레이어의 돌 색깔(흑 돌or백 돌) : 플레이어의 탐색 방법이 딕셔너리 형태로 들어간다.

→ 매개변수 verbose를 True로 하면 한 번 수를 놓을 때마다 각 플레이어가 놓은 위치와 현재 보드판을 출력한다.

```
def alphabeta_search(game, null):
    """알파-베타 가지치기를 사용하여 최고의 수를 결정하기 위한 게임 트리 탐색."""
    def max_value(game, alpha, beta):
        if game.is_terminal():
            return game.utility(), None
        v, move = -infinity, None
        for a in game.actions():
            v2, _ = min_value(game.result(a), alpha, beta)
            if v2 > v:
                v, move = v2, a
                alpha = max(alpha, v)
            if v >= beta:
                return v, move
        if not move:
            game.to_move = '○'
            return min_value(game, alpha, beta)
        return v, move

    def min_value(game, alpha, beta):
        if game.is_terminal():
            return game.utility(), None
        v, move = +infinity, None
        for a in game.actions():
            v2, _ = max_value(game.result(a), alpha, beta)
            if v2 < v:
                v, move = v2, a
                beta = min(beta, v)
            if v <= alpha:
                return v, move
        if not move:
            game.to_move = '●'
            return max_value(game, alpha, beta)
        return v, move

    if(game.to_move=='○'):
        return min_value(game, -infinity, +infinity)
    else :
        return max_value(game, -infinity, +infinity)
```

→ 알파-베타 가지치기 탐색 함수이다.

→ 해당 플레이어의 가능한 행동이 비어 있으면 턴을 상대에게 넘기는 기능이 추가되었으며, 플레이어의 돌에 따라 먼저 실행하는 함수가 달라진다. (흑 돌이면 max_value실행, 백 돌이면 min_value실행)

```
def h_alphabeta_search(game, c_depth):
    """휴리스틱 알파-베타 탐색"""
    cutoff=cutoff_depth(c_depth)
    player = game.to_move

    def max_value(game, alpha, beta, depth):
        if game.is_terminal():
            return game.utility(), None
        if cutoff(game, depth):
            return h(game), None
        v, move = -infinity, None
        for a in game.actions():
            v2, _ = min_value(game.result(a), alpha, beta, depth+1)
            if v2 > v:
                v, move = v2, a
                alpha = max(alpha, v)
            if v >= beta:
                return v, move
        if not move:
            game.to_move = '○'
            return min_value(game, alpha, beta, depth)
        return v, move

    def min_value(game, alpha, beta, depth):
        if game.is_terminal():
            return game.utility(), None
        if cutoff(game, depth):
            return h(game), None
        v, move = +infinity, None
        for a in game.actions():
            v2, _ = max_value(game.result(a), alpha, beta, depth+1)
            if v2 < v:
                v, move = v2, a
                beta = min(beta, v)
            if v <= alpha:
                return v, move
        if not move:
            game.to_move = '●'
            return max_value(game, alpha, beta, depth)
        return v, move

    if(game.to_move=='○'):
        return min_value(game, -infinity, +infinity, 0)
    else :
        return max_value(game, -infinity, +infinity, 0)
```

→ 휴리스틱 알파-베타 탐색 함수이다.

→ 알파-베타 탐색 함수에 추가로 cutoff를 할 수 있는 매개변수 c_depth가 추가되었으며, cutoff가 되면 해당 상태에서의 평가함수를 반환한다.

```
def cutoff_depth(d):
    """깊이 d까지만 탐색하도록 하는 중단 함수: depth > d이면 True 리턴."""

    return lambda game, depth: depth > d
```

→ 휴리스틱 알파-베타 탐색에서 사용할 함수이다.

→ 현재 깊이를 받아 cutoff인지 확인하는 함수이다. (cutoff라면 True를 반환한다.)

```
def h(game):
    """터미널 노드까지 도달하지 않고 평가함수를 반환하기 위해, 오셀로 판의 각 모서리에 가중치를 부여
    판의 각 꼭짓점 => +
    판의 모든 모서리 => +
    판의 각 꼭짓점을 둘러싼 세 점 => +
    판의 각 꼭짓점이 채워지지 않은 상태에서 각 꼭짓점을 둘러싼 세 점 => - """

    cnt = 0
    for i in range(game.height):
        for j in range(game.width):
            if game.board[i][j] == '●':
                cnt += 1
            elif game.board[i][j]=='○':
                cnt -= 1

            if i==game.height-1 or i==0 or j==game.width-1 or j==0 :
                if game.board[i][j]=='●':
                    cnt += 1
                elif game.board[i][j]=='○':
                    cnt -= 1

    corners = [[[0,0],[[1,0],[1,1],[0,1]]],
               [[0,game.width-1],[[1,game.width-1],[1,game.width-2],[0,game.width-2]]],
               [[game.height-1,0],[[game.height-2,0],[game.height-2,1],[game.height-1,0]]],
               [[game.height-1,game.width-1],[[game.height-2,game.width-1],[game.height-2,game.width-2],[game.
    for corner in corners:
        if(game.board[corner[0][0]][corner[0][1]]=='●'): cnt+=3
        elif(game.board[corner[0][0]][corner[0][1]]=='○'):cnt-=3
        else:
            for xy in corner[1]:
                if(game.board[xy[0]][xy[1]]=='●'): cnt-=2
                elif(game.board[xy[0]][xy[1]]=='○'):cnt+=2
    return cnt
```

→ cutoff가 되었을 때의 평가함수를 반환하는 함수이다.

→ 각 돌의 가중치는 모두 1로 간주하고, 각 돌이 놓여진 위치를 가중치 삼아 평가함수를 계산한다. (ex: 보드의 꼭짓점은 상대에게 빼앗길 위험이 없으므로 가중치를 더한다.)

→ 가중치를 부여 할 판의 위치들의 좌표를 corners변수에 넣어, 현재 보드판을 탐색한 결과가 평가함수로 반환된다.

```
class Othello(Game):
    """Othello 게임. 보드 크기: height * width.
    게임 종료시 더 많은 색을 가진 사람이 승리
    '●'와 '○'가 게임 플레이. '●'가 먼저 플레이.
    (0, 0) 위치는 보드의 좌상단 끝 위치."""
```

→ 기존의 코드는 Board와 TicTacToe(Game)를 따로 두고, 게임에서 필요한 기능과 게임판의 기능을 분리하여 매개변수로 전달했다.

→ 하지만 본 팀은 보드와 게임이 매개변수의 형태로 각각 전달되는 해당 구조가 전체 로직을 이해하기 어렵다고 생각하였고, Board의 모든 기능을 Game에 이전하여 각각의 Game객체가 현재의 보드판을 가지고 생성되도록 하였다.

```
def __init__(self, height, width, to_move):
    self.height=height
    self.width=width
    self.board = [[ '.' for x in range(width)] for y in range(height)]
    self.board[width//2][height//2]='●'
    self.board[width//2-1][height//2-1]='●'
    self.board[width//2-1][height//2]='○'
    self.board[width//2][height//2-1]='○'
    self.to_move = to_move
```

→ 행동 시 단순히 돌을 놓는 것으로 끝나는 것이 아니라 해당 돌을 놓았을 때 8방향에 위치한 모든 돌에 영향을 줄 가능성이 있으므로, 딕셔너리 형태보다는 리스트 형태가 적합할 것이라고 생각하여 board를 height*width 크기의 리스트로 만들었다.

→ Othello게임의 초기상태는 양측의 돌이 일부 놓여있는 형태이므로, 리스트의 가운데 위치에 초기 돌을 세팅했다.

```
def actions(self):
    move = ((0,1),(1,0),(0,-1),(-1,0),(1,1),(-1,1),(1,-1),(-1,-1))
    """내 돌을 놓을 수 있는 좌표를 담은 배열 반환"""
    action = []
    for i in range(self.height):
        for j in range(self.width):
            for d in move:
                if(self.board[i][j]=='.' and self.feasible(i+d[0], j+d[1], d) > 0):
                    action.append([i,j])
                    break
    return action
```

→ Othello게임은 상대의 돌을 하나 이상 뒤집을 수 있는 위치에만 수를 놓는 것이 가능하다. 따라서 모든 빈 칸을 검사하여 돌을 하나 이상 뒤집을 수 있는 위치가 현재 플레이어가 가능한 행동이다.

→ 반복문을 통해 8방향을 수월하게 검사하기 위해 방향을 담고 있는 move 튜플을 선언한다.

→ feasible함수를 통해 뒤집을 수 있는 돌이 해당 방향에 있는 것을 확인한다면 action에 추가하고 다음 위치를 확인한다.

→ 모든 방향에서 뒤집을 수 없다면 해당 위치는 action에 추가되지 않는다.

```
def feasible(self, x, y, dxy):
    """해당 위치에 돌을 놓을 수 있는지 확인하는 함수"""

    if(x < 0 or y < 0 or x >= self.height or y >= self.width or self.board[x][y] == '.'):
        return -1
    elif(self.board[x][y] == self.to_move):
        return 0
    else:
        tmp = self.feasible(x + dxy[0], y + dxy[1], dxy)
        if(tmp != -1):
            return tmp + 1
        return -1
```

→ action에서 특정 방향과 이동좌표를 매개변수로 feasible을 호출한다.

→ feasible은 상대의 돌이 나오면 해당 방향으로 재호출된다. 반환 값이 -1이 아니라면 뒤집는 것이 가능하므로 return값에 1을 가산하여 반환한다.

→ 만약, 호출된 좌표가 판의 범위를 넘었거나, 빈칸이라면 뒤집을 수 없는 방향이므로 -1을 반환한다.

→ 해당 플레이어의 돌이 나오면 사이에 있는 돌은 뒤집을 수 있으므로 0을 반환한다.

→ 최종적으로 feasible의 return값은 뒤집을 수 없다면 0 이하, 뒤집은 돌이 있다면 1 이상이다.

```
def turn(self, x, y, dxy):
    """상대 돌을 뒤집는 함수"""

    if(x < 0 or y < 0 or x>=self.height or y>=self.width or self.board[x][y]=='.'):
        return -1
    elif(self.board[x][y]==self.to_move):
        return 0
    else:
        tmp = self.turn(x+dxy[0], y+dxy[1], dxy)
        if(tmp != -1):
            self.board[x][y] = self.to_move
            return 0
        return -1
```

→ turn함수는 feasible과 유사한 방식으로 작동한다.

→ 다만 상대의 돌을 뒤집을 수 있음이 확인되었을 때, 가산을 하여 반환하는 것이 아니라 실질적으로 그 점을 플레이어의 돌로 변경한다.

```
def check_all_dirt(self,action):
    """8방향에 대해 돌을 놓을 수 있는지 확인하는 함수"""

    move = ((0,1),(1,0),(0,-1),(-1,0),(1,1),(-1,1),(1,-1),(-1,-1))

    for d in move:
        self.turn(action[0]+d[0], action[1]+d[1], d)
```

→ check_all_dirt 함수는 8방향으로 turn함수를 호출하는 역할을 한다.

```
def result(self, action):
    """반환할 새로운 게임(보드)를 위해 새로운 객체 board2생성
    깊은복사를 위해 새로운 보드에 현재 보드(2차원 배열) 복사
    돌을 놓고 바뀐 결과를 담은 객체 board2반환"""

    board2 = Othello(self.height, self.width, self.to_move)
    board2.board = [self.board[x][:] for x in range(self.height)]
    board2.check_all_dirt(action)
    board2.board[action[0]][action[1]] = self.to_move
    board2.turn_change()

    return board2
```

→ 본 팀은 주어진 Board 클래스를 Game 클래스와 합쳐서 코드를 작성하였으므로 result의 ruturn값도 Game객체여야 한다.

→ 새 Othello 객체를 생성하고 기존 보드판과 똑같이 복사한다. (깊은 복사)

→ 선택한 좌표(action)를 매개변수로 check_all_dirt를 호출하여 가능한 돌들을 모두 뒤집고 action의 위치에 현재 플레이어의 돌을 둔다.

→ turn_change로 다음 순서를 상대방으로 변경한 후 game자체를 반환한다.

```
def turn_change(self):
    """수를 놓는 차례 변경"""

    if self.to_move=='○':
        self.to_move = '●'
    else:
        self.to_move = '○'
```

→ turn_change는 게임의 차례를 바꾸는 함수이다.

```
def is_terminal(self):
    """두 플레이어 모두 수를 둘 수 없을 때, 터미널 상태로 인지하고 중단"""

    a = self.actions()
    self.turn_change()
    b = self.actions()
    self.turn_change()
    if(not a and not b):
        return 1
    return 0
```

→ 종료조건은 양 플레이어 모두 수를 둘 수 있는 위치가 없을 때로 정의하였다.

→ 단순히 보드의 빈칸을 세어 보기도 하였는데, Othello는 게임이 끝나지 않더라도 상대의 패를 더 이상 뒤집을 수 없다면 무름 수가 발생한다.

→ 이때, 빈칸이 있음에도 양 측 모두 둘 수 있는 위치가 없으면 is_terminal 함수에 의해 종료되지 않으면서 서로 턴을 무한으로 미뤄 게임이 끝나지 않는 문제점이 발견되었다.

→ 해당 문제를 해결하기 위해 다소 연산의 횟수가 많아지더라도 action을 직접 확인하는 것이 좋다고 판단했다.

```
def utility(self):
    """●가 먼저 시작
    반환되는 값이 양수면 ●승리
    반환되는 값이 음수면 ○승리"""

    cnt = 0
    for i in range(self.height):
        for j in range(self.width):
            if self.board[i][j] == '●':
                cnt += 1
            elif self.board[i][j] =='○':
                cnt -= 1
    return cnt
```

→ 게임의 결과점수는 양 플레이어 간의 돌 개수 차이로 두었다.

→ 검은색이 흰색보다 돌이 몇 개나 더 많은지를 반환한다. 따라서 양수 값이 클수록 검은색이 압도적으로 승리한 것이며, 0이라면 비긴 게임, 음수 값이라면 흰색이 승리한 것이다.

```
def display(self, util = False):
    """현재 배열 출력 함수"""

    for i in range(self.height):
        for j in range(self.width):
            if(self.board[i][j]=='.'): print('',end=" ")
            print(self.board[i][j], end=" ")
        print()
    if util:
        return self.utility()
```

→ 빈 칸을 의미하는 ' . ' 과 양측의 돌을 의미하는 '●'와 '○'의 크기가 달라 단순 출력 시 보드가 밀려 가독성이 떨어지는 현상을 발견했다. 따라서 빈 칸의 경우 한 칸을 추가로 출력하여 바둑판 모양 그대로 출력될 수 있도록 display 구현하였다.

```python
def query_player(game):
    """다음 수(move)를 직접 입력하는 형태의 플레이어"""

    cheerup = ["좋습니다.", "허를 찌르는 수입니다.", "훌륭하군요!", "한 수 배워갑니다.", "예상치 못한 수...(사실 그런거 없음)", "대단합니
    print("현재 상태:")
    game.display()
    print("")
    move = None
    actions = game.actions()
    if actions:
        while True:
            print(f"가능한 수: {actions}")
            move_string = input('당신의 수는? (돌을 둘 위치 입력; 예: (1,1)(세로, 가로)): ')
            col, row = int(move_string.split(',')[0]), int(move_string.split(',')[1])
            if [col, row] in actions:
                print()
                print(random.choice(cheerup))
                print()
                break

            print()
            print("==============================")
            print("가능한 수만 둘 수 있습니다! 다시 두세요")
            print("==============================")
            print()
            print("현재 상태:")
            game.display()
            continue
        try:
            move = eval(move_string)

        except NameError:
            move = move_string
    else:
        print("================================")
        print('가능한 수가 없음: 상대방에게 순서가 넘어감.')
        print("================================")
    return move
```

→ 사용자 플레이를 위한 함수이다.

→ 사용자가 수를 놓을 수 있는 좌표를 출력하며, 현재 보드의 상태도 출력한다.

→ 사용자가 가능한 수 외의 좌표 값을 입력하면 오류 메시지를 띄우고, 다시 입력하도록 유도한다.

→ 만약 사용자가 둘 수 있는 좌표가 없다면, 가능한 수가 없다는 메시지를 띄우고 다른 상대에게 턴이 넘어간다.

```
def random_player(game):
    """허용되는 수(move) 중에서 무작위로 하나를 선택하는 플레이어"""

    act = game.actions()
    if not act:
        return None
    return random.choice(game.actions())
```

→ 돌을 놓을 수 있는 좌표를 받아 랜덤으로 선택하여 수를 두는 함수이다.

→ 수를 두는 것이 현재 상태를 좋게 만드는 것 이므로 언덕등반으로 보일 수 있지만, 판의 꼭짓점에 돌이 놓여져 있을 때 그 꼭짓점을 둘러싸는 위치에 돌을 놓는 것은 좋지 않은 결과값을 초래한다.

→ random_player는 그러한 상황을 인지하지 못하고 무작위로 수를 두기 때문에, 무작위 보행 탐색이라고 할 수 있다.

```
def player(search_algorithm, c_depth=0):
    """지정된 탐색 알고리즘을 사용하는 플레이어: (game)을 입력 받아 move를 리턴하는 함수."""

    return lambda game : search_algorithm(game, c_depth)[1]
```

→ alphabeta_search 혹은 h_alphabeta_search를 입력받아, 해당 탐색으로 게임을 수행하는 함수이다.

→ 매개변수 c_depth는 h_alphabeta_search를 사용할 때 입력한다.

→ 사용자가 지정해주지 않으면 기본값으로 0을 가지며, alphabeta_search에는 해당 매개변수가 입력 되어도 사용되지 않는다.

6. 향후 발전 가능성

게임 탐색을 이용하여 만든 다른 에이전트와 대결을 함으로써 어떠한 조건에서 승리하고 패배하는지 파악하여, 본 팀의 탐색에 승률을 높일 수 있다.

본래의 게임과 같이 보드의 판이 8x8 크기일 때로 고정한다면, 주어진 정보[2]를 통해 평가함수를 위한 가중치를 정확하게 제공하여 cutoff를 했을 때 더 정확한 평가함수 값을 얻을 수 있다.

유지보수를 위해 코드 리팩토링을 하였지만, 이를 하지 않고 오직 알고리즘의 효율을 높여 코드를 작성한다면 현재보다 빠른 시간 내에 결과를 도출할 수 있다.

특정 상태에서 필승 전략이 존재한다면, 머신 러닝을 통해 탐색을 하는데 필요한 시간을 줄이고 즉시 다음 상태를 도출할 수 있다.

[2] https://greenothello.com/library/ffo_guide0 (오셀로 보드판 가중치 참고자료)

7. 참고자료

https://ko.dict.naver.com/#/entry/koko/59e4e2207a5d47c1b19c9f3ae0bac6a2
(네이버 국어사전: 오셀로 게임 정의 참고자료)

https://greenothello.com/library/ffo_guide0
(오셀로 보드판 가중치 참고자료)

컴퓨터공학과 20190274 정택원

이번 과제를 통해 탐색을 통한 문제해결을 하였습니다. 팀원들과 주제에 대해 회의를 하여 적대적 탐색에서 사용자 입력을 통한 플레이를 구현하려는 계획에 흥미로웠습니다.

먼저 코드를 어떻게 수정할 것인가에 대해서 설계를 진행하였습니다. 본 팀의 게임에 맞추어 각 단계별 어떤 함수가 호출되는 등의 절차를 추적했습니다. 코드 동작은 재귀적으로 수행되어 2-3 단계 까지만 분석하니 반복적으로 실행되는 사이클을 발견하여 팀원 모두가 해당 로직을 이해할 수 있었습니다.

팀원간 시간과 장소의 제약이 존재하여, 코드 수정과 테스트는 남민지 학우와 짝 프로그래밍을 통해 완성했습니다. 오프라인으로는 수업이 끝난 후, 온라인으로는 각자의 집에서 화면 공유를 통해 한명이 코드를 수정하면 다른 한명이 잘못된 부분을 바로잡아 주며 진행하였습니다. 주어진 코드에서 상위 클래스인 Game 클래스를 제외한 모든 코드를 남민지 학우와 함께 작성하였습니다.

수정된 코드 전달에 불편함이 존재하여, 깃 허브 레파지토리[3]를 만들었습니다. 형상 관리를 통한 작업으로, 언제든 되돌릴 수 있다는 마음에 과감한 코드 수정을 할 수 있었습니다.

코드완성을 하고 리팩토링을 통해 보는 사람으로 하여금 시각적으로 불편함이 없도록 친절한 주석과 플레이 방법을 명시하였습니다.

수업시간에 배웠던 적대적 탐색을 그대로 활용할 수 있어서 로직의 이해는 어렵지 않았습니다. 하지만 주어진 코드에서 수정하려고 하니, 어려움이 존재하여 과감하게 본 팀의 주제에 맞추어 코드를 변경해 나갔습니다. 수 많은 계산을 해야 하는 탐색이기 때문에, 효율적으로 코드를 작성하려고 노력했습니다.

이번 과제를 통해 적대적 탐색을 더욱 잘 이해할 수 있었으며, 머신 러닝을 통한 딥 러닝을 배워보고자 하는 마음이 생겼습니다. 감사합니다.

---

[3] https://github.com/chtw2001/aiclass

산업경영공학과 20201092 정유진

과제를 진행하며 휴리스틱 탐색과 국지적 탐색, 적대적 탐색이 다양한 선택적 기로에 있거나 여러 게임에서 탐색 알고리즘을 통해 해결 할 수 있다는 것을 알게 되었다. 과제를 진행하기 전에는 수업 시간에서 예시로 나온 문제들과 유사한 문제들만 생각 했지만 팀과제를 진행하면 내가 생각에 문제 뿐만이 아니라 다양한 게임도 해당 된다는 것을 깨달았다.

회의를 진행했을 때 생각했던 문제는 서울 ~ 부산 최적 경로(이동 거리, 수단, 비용 등)에 대한 것과 한라산 등산 등 거리에 대한 문제들에 대한 의견을 제시했다. 다른 게임들에 대한 의견을 제시해 주셔서 그런 방향도 있다는 것을 알게 되었다.

휴리스틱 탐색과 국지적 탐색, 적대적 탐색에 대해 강의에서 나온 예시와 체스나 빙고게임, 오목 등 외에는 생각 하는게 없었으며 팀원분들이 '오셀로 게임'을 제시해 주셔서 주제로 진행하기 정해졌다. 처음 알게 된 게임이라 게임에 대해 규칙과 게임의 승리조건, 진행 방향에 대해 찾아 보았다.

힘든 점.
탐색에 맞는 주제 선정이 어려웠다. 강의에 나온 예시 외의 문제를 생각한다는 것이 어려웠으며 각 알고리즘 탐색에 맞는 게임을 찾는게 어려웠다.
파이썬 프로그램에 대해 잘 모르는 상태에서 과제를 진행한다는 것이 조금 버거웠다.

컴퓨터공학과 20210833 남민지

주제를 정하는 것이 가장 큰 문제라고 생각하여 오히려 실제 구현 시간보다 더 긴 시간을 주제 선정에 사용했다. 팀원들과 대화하며 주제를 정의하고자 했지만, 탐색 알고리즘을 적절하게 사용할 수 있는 문제를 찾기는 쉽지 않았다. 문제를 새로 정의하는 것보다는 기존에 있는 문제를 가져다 쓰는 것이 좋겠다는 생각에 1인 혹은 2인 플레이인 고전 퍼즐게임을 많이 찾아보았다. 그중 오셀로는 초심자도 쉽게 이해할 수 있을 만큼 규칙이 간단하면서도, 현재 판에서 승률을 계산하기 위해서는 제법 고민이 필요할 것 같아 좋은 주제가 될 것이라 생각했다. 사람이 직접 입력 값을 넣어 직접 작성한 AI와 대결하며 성능을 확인할 수 있는 적대적 탐색에 매력을 느낀 것도 주제 선정 이유 중 하나이다.

TicTacToe 게임은 동일한 보드 상태에 양측이 취할 수 있는 행동도 동일했다. 따라서 actions 함수는 (초기 가능 행동 - 이미 취한 행동)의 형태였다. 하지만 오셀로는 플레이어마다 취할 수 있는 행동이 달랐고, 이러한 형태의 집합 연산은 불가능했다. 구조를 바꿔야 한다고 생각하여 보드의 형태를 익숙한 방식인 리스트로 변경했다. 리스트에서 8방향 위치를 확인할 수 있도록 새로운 함수를 정의하여 actions 함수를 완성하였다. 하지만 이때까지는 기존 코드를 많이 바꾸기에는 두려워 기존의 형식을 유지하고자 노력하였고, 그로 인해 진행이 더뎠다.

이때 정택원 학우가 깃 레파지토리를 만들고 기존 코드를 백업한 뒤 Board의 모든 기능을 게임 클래스로 옮겨주었다. 이미 큰 틀이 변경된 상황과 기존의 자료가 남아있을 것이라는 확신 덕에 이후 코드 수정은 좀 더 과감 해졌다. 온/오프라인으로 팀원과 실시간으로 소통하며 코드 수정을 이어 나갔고, 번갈아 가며 코드를 수정하는 과정에서 코드에 대한 이해도가 깊어 졌다.

구조를 파악하자 휴리스틱 함수의 정의는 어렵지 않았다. 자료 조사를 통해 어떤 위치에 가중치를 부여해야 할지 정하는 것이 핵심이었다. 연속적인 if-else 문을 줄이기 위해 검사해야 할 좌표 값을 리스트로 저장하여 반복시켰다.

오셀로를 잘 알고 있는 것은 아니라 휴리스틱 값이 아주 적절한 것은 아닐 것이고, 파이썬에 미숙하여 효율적인 코드 역시 아니었을 것이다. 그럼에도 팀원과 함께 직접 코드를 짜고 그 코드와 대결해 보는 것은 재미있는 경험이었다.