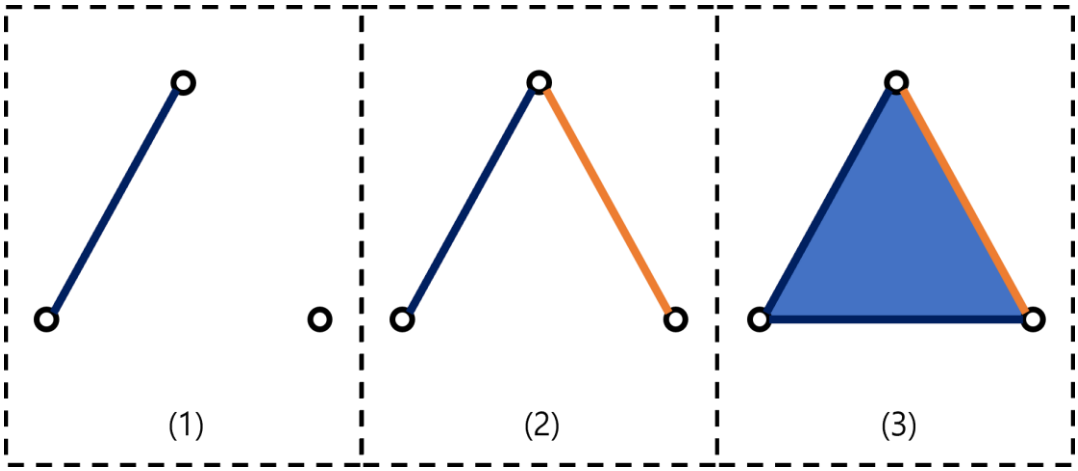
인공지능 팀프로젝트 보고서

9팀 | 권민혁, 김종권, 김현민, 이희철

1. 상황 분석

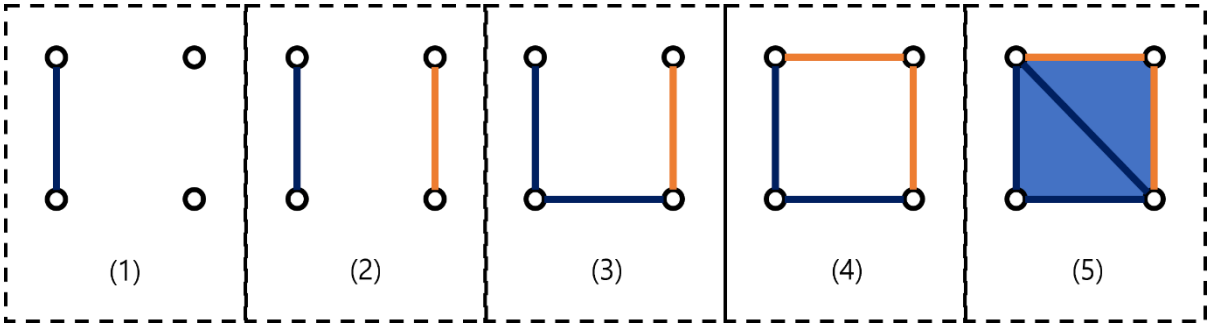
상황에 따라 어떤 점을 어느 순서로 연결하는 것이 유리한지 파악하기 위해 점의 개수가 적은 경우부터 점점 많아지는 경우로 진행했습니다.

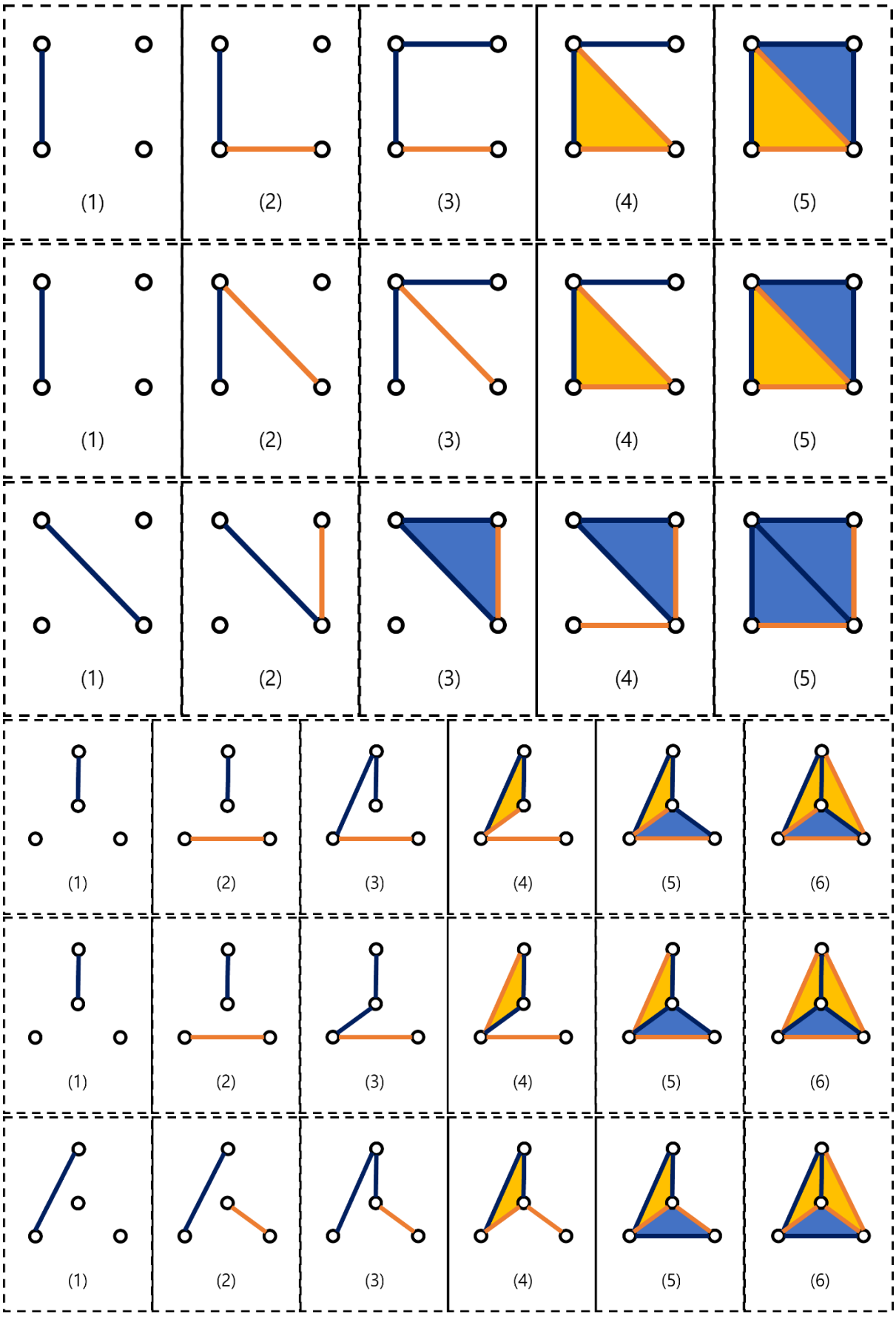
1) 점 3개

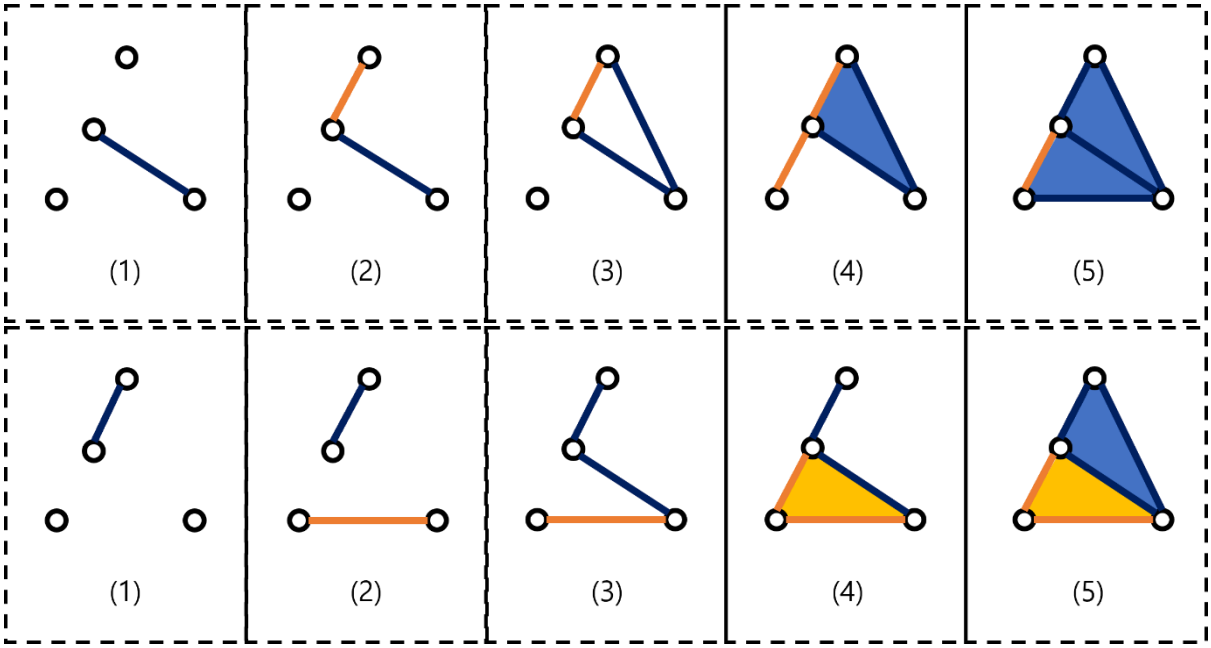


점이 3개이고 연결할 수 있는 모든 선분의 개수가 3개로 결정되는 경우에는 어떠한 2개의 점을 선택해서 연결할지에 상관없이 반드시 선공에게 유리했습니다.

2) 점 4개인 경우 중 일부

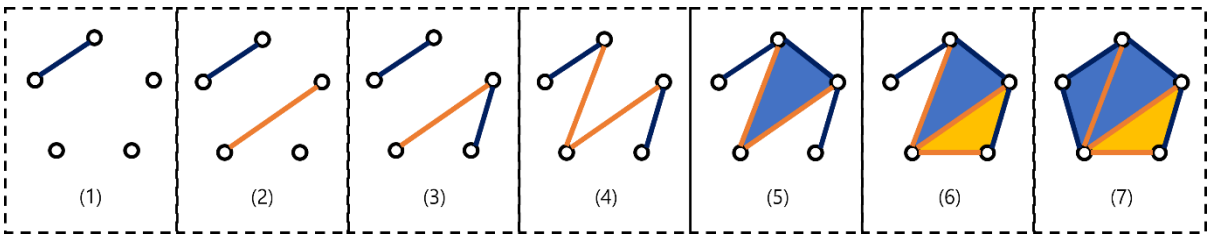


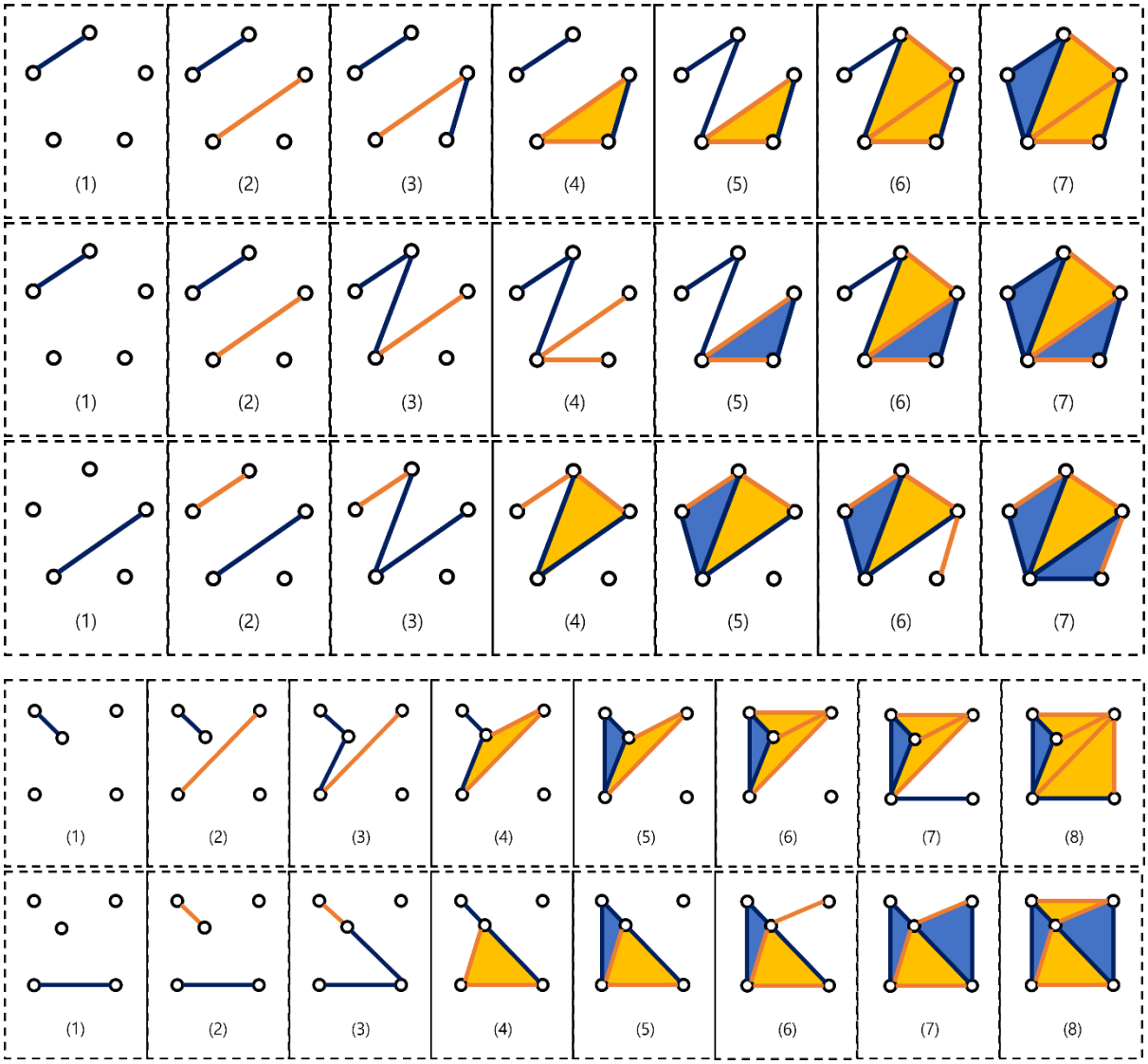

(1)
(2)
(3)
(4)
(5)
(1)
(2)
(3)
(4)
(5)
(1)
(2)
(3)
(4)
(5)
(1)
(2)
(3)
(4)
(5)
(6)
(1)
(2)
(3)
(4)
(5)
(6)
(1)
(2)
(3)
(4)
(5)
(6)



점이 4개이고 만들 수 있는 모든 선분의 개수가 5개로 결정되는 경우에는 사각형의 대각선을 처음부터 만드는 것이 유리합니다. Case 4의 (1)에서 대각선을 만드는 것과 같이 어디에도 연결되지 않은 점 2개를 연결하는 선분(**독립선분**이라고 명명했습니다.)이 있다면 그중에서 게임판에서 독립선분의 개수가 최소가 되도록 선분을 만드는 것이 유리하다고 판단했습니다. 예를 들어 Case 1, 2, 3과 같이 선공이 (1)에서 대각선이 아닌 세로로 선분을 만든다면 (2)에서 1개의 독립선분을 더 만들 수 있게 됩니다. 그러나 Case 4는 선공이 (1)에서 대각선을 만들었기 때문에 (2)에서 더 이상 만들 수 있는 독립선분은 없습니다.(더 이상 독립선분이 없는 상태를 **포화상태**라고 명명했습니다.) 만약 선분의 개수가 5개여서 선공이 유리한 게임판임에도 불구하고 선공이 (1)에서 대각선을 만들지 않고 후공이 그 다음 턴에서 대각선을 만든다면 비기는 상황이 됩니다. 그리고 삼각형을 만들 수 있는 상황이라면 다른 선분을 만드는 것보다 점수를 얻을 수 있는 선분 먼저 만드는 것이 유리했습니다. Case 5, 6, 7에서는 포화상태까지 반드시 독립선분이 2개가 생길 수밖에 없고 만들어지는 총 선분의 개수가 짝수여서 후공에게 유리합니다. Case 8, 9를 비교하면 선공 입장에서 처음에 독립선분의 개수가 적어지도록 선분을 만들었는지에 따라 승리와 무승부로 나뉘었습니다. Case 8은 가운데에 선분을 만들어서 포화상태까지 독립선분이 하나가 되도록 했습니다. 그리고 승리했습니다. 그러나 Case 9에서는 선공이 포화상태까지 독립선분을 2개가 되도록 허용해서 무승부가 되었습니다.

3) 점 5개인 경우 중 일부





점이 5개인 경우 중에 모든 점이 convex hull 위에 있다면 Case 2와 Case 4를 비교했을 때 선공은 Case 4처럼 convex hull 위가 아닌 선분을 그리는 것이 유리합니다.

포화상태 이후로 임의의 선분을 만들고자 할 때는 해당 턴에서 본인이 한 번에 몇 개의 삼각형을 만들 수 있고 다음 턴에서 상대방에게 주어지는 삼각형이 몇 개인지에 따라 유불리가 달라집니다. 그리고 앞서 점 4개인 경우에서 작성한 내용처럼 본인 턴에서 선분을 하나 만들었을 때 얻을 수 있는 점수가 있다면 해당 선분을 만드는 것이 좋습니다. 그리고 본인의 턴에서 하나의 선분을 만들어 점수를 얻을 수 있는 것과 별개로 해당 선분으로 인해 다음 상대 턴에 몇 개의 삼각형을 주는지를 고려해야 합니다.

- 위의 경우들을 종합적으로 보면 선공과 후공이 모두 매 턴마다 최선의 수를 두고 게임이 끝났을 때 생성된 총 선분의 개수가 홀수면 선공이 유리하고, 짝수면 후공이 유리하다는 것도 알 수 있습니다.

2. 게임 초반 규칙

상황 분석을 토대로 다음과 같은 규칙을 설정했습니다.

1) 삼각형이 만들어지는 경우가 있다면 해당 선분 선택합니다.

    (1) 이때 삼각형이 만들어지는 경우가 여러 가지라면 한 번에 가장 높은 점수를 받을 수 있는 경우를 선택합니다.

2) 만약 삼각형이 만들어지는 경우가 없다면 독립선분이 포화될 때까지 독립선분을 만듭니다.

    (1) 이때 포화상태까지 만들어지는 독립선분 case를 모두 가져옵니다.

    (2) 그리고 게임판에서 형성될 수 있는 convex hull의 점과 선분도 모두 가져옵니다.

    (3) 독립선분 중에서 아래 조건에 따라 best case가 되는 독립선분을 찾고 그것을 선택합니다.

        ① 이미 그려지지 않은 선분 중에서 선택해야 하고

        ② 만약 case의 모든 점이 convex hull 위의 점이면, convex hull 위의 선분이 아닌 선을 선택합니다.

3) 만약 독립선분이 포화상태면 mix-max tree search로 넘어갑니다.

3. Min-Max

독립선분이 포화상태가 되면 min-max tree search를 진행하고 가장 결과가 좋은 수를 리턴합니다. min-max tree search를 할 때 alpha-beta cut off를 사용하여 탐색 시간을 줄였습니다. 또한 한 턴당 1분 미만으로 주어지는 게임 시간을 고려하여 높이 제한을 두었고 휴리스틱을 활용하여 확장할 자식의 수를 조정했습니다.

min-max tree search가 계속해서 자식을 확장하다가 높이 제한에 걸리거나 게임이 종료될 경우 evaluation 함수를 호출하여 어느 팀에게 유리한 형세인지 판단했습니다. evaluation 함수는 해당 턴에서 우리 팀이 만들 수 있는 삼각형의 개수와 그 다음 턴에서 상대방이 만들게 될 수 있는 삼각형의 개수에 따라 서로 다른 값을 리턴합니다. 해당 턴에서 삼각형을 2개 만들 수 있으면 45점, 1개를 만들 수 있으면 15점, 다음 턴에서 삼각형을 2개 만들 수 있으면 -30점, 1개를 만들 수 있으면 -10점을 할당했습니다. 해당 턴에서 삼각형을 만들 수 있다면 바로 만드는 것이 우선이라고 판단하여 이번 턴에 만들 수 있는 삼각형에 개수에 따른 점수(45점, 15점)와 다음 턴에서 만들 수 있는 삼각형의 개수에 따른 점수(30점, 10점)의 절댓값이 서로 다릅니다. 그리고 min에서 끝났는지, 아니면 max에서 끝났는지에 따라 리턴값 반대가 되도록 서로 다른 evaluation 함수를 만들었습니다. min 노드에서 게임 상황을 evaluation하는 경우에는 상대방 팀의 입장에서 점수를 계산하는 것이고 max 노드에서 게임 상황을 evaluation하는 경우 우리 팀의 입장에서 점수를 계산하는 것이기 때문입니다.

4. 휴리스틱

Min-Max 상황에서 확장할 자식 노드의 수를 줄이기 위해 휴리스틱을 사용했습니다. 확장된 자식 노드 중 휴리스틱 평가 값이 좋은 몇몇 자식 노드만 선택하여 확장했고 나머지는 확장하지 않음으로써 탐색 시간을 줄이고 메모리 공간의 낭비를 방지하고자 했습니다.

휴리스틱 평가함수는 **[선분을 만들었을 때 얻은 점수 – 선분을 만들었을 때 상대방 턴에서 만들어지는 삼각형의 개수]**로 구성했습니다. 수식으로 나타내면 다음과 같습니다.

$\hat{h}(x_i) = getScore(x_i) - nextTriangle(x_i), x_i$는 만들 수 있는 선분 x_n 중 하나

getScore(x)는 만든 선분 x로 인해 우리 팀이 얻을 수 있는 점수에 대해 특정 값을 리턴하는 함수입니다. nextTriangle(x)은 만든 선분 x로 인해 상대 턴에서 만들어지는 삼각형의 개수에 따라 특정 값을 리턴하는 함수로 일종의 패널티로 볼 수 있습니다.

getScore(x)의 리턴값은 해당 선분으로 얻은 점수 * 3입니다. 얻은 점수에 3을 곱하는 이유는 nextTriangle(x)로 발생하는 값보다 getScore(x)로 발생하는 값에 가중치를 주기 위함입니다. 즉, 선분을 만들었을 때 상대 턴에 만들어지는 삼각형의 개수보다 이번 턴에 우리 팀이 얻을 수 있는 점수에 더 높은 가치를 주었습니다.

nextTriangle(x)의 리턴값은 상대방 턴에서 만들 수 있는 삼각형의 개수에 따라 3가지 Case로 나뉩니다. Case 1은 우리 팀에서 선분을 만들었을 때 다음 상대방 턴에서 만들 수 있는 삼각형이 0개라면 nextTriangle(x)의 리턴값은 0입니다. 상대방 턴에서 점수를 주지 않는다는 점에서 0점을 부여했습니다. Case 2는 상대방 턴에서 만들 수 있는 삼각형의 개수가 1개인 경우입니다. 이때 nextTriangle(x)의 리턴값은 2입니다. 경우에 따라서 상대방 턴에서 만들 수 있는 삼각형이 하나일 때 상대방이 그것을 만들면 다시 우리 턴에서 삼각형을 만들 수 있는 가능성이 새롭게 생길 수 있지만 반대로 상대방 턴에서 삼각형이 하나 만들어지고 다시 우리 팀 턴이 됐을 때 만들 수 있는 삼각형이 없는 경우도 있습니다. 그렇게 되면 우리 팀은 연속으로 점수를 얻지 못할 수 있습니다. 따라서 상대방 턴에서 만들 수 있는 삼각형이 하나라면 다시 우리 팀 턴이 됐을 때 점수를 획득을 보장할 수 없다는 측면에서 2를 부과했습니다. Case 3은 상대방 턴에서 만들 수 있는 삼각형의 개수가 2개 이상인 경우입니다. 이때 nextTriangle(x)의 리턴값은 1입니다. 상대방 턴에서 만들 수 있는 삼각형이 2개 이상이라면 상대방이 그 중에서 삼각형을 만들어서 점수를 얻을 수 있도록 되기는 하지만 다시 우리 팀 턴이 됐을 때 우리도 점수를 얻을 수 있는 것이 보장됩니다. 따라서 이때는 패널티로 1점만 부과했습니다.

5. 구현

위에서 제시한 게임 초반 규칙, min-max, 휴리스틱은 아래와 같은 코드로 구현했습니다.

find_best_selection 함수 – 가능한 선분 개수 찾기

```
    def find_best_selection(self):

        self.posible_lines = []
        self.independent_lines_case = []

        #가능한 line 가져오기
        for (point1, point2) in list(combinations(self.whole_points, 2)):
            temp_line = [point1, point2]
            if self.check_availability(temp_line):
                self.posible_lines.append(temp_line)
```

find_best_selection 함수를 호출하면 현재 게임 상황에서 만들 수 있는 모든 선분을 가져옵니다.

find_best_selection 함수 – 삼각형이 생기는 경우

```
        #삼각형을 만드는 경우가 생기면 해당 선분을 우선적으로 생성
        max_value = 0
        best_line = self.posible_lines[0]
        for line in self.posible_lines:
            temp = self.check_temp_triangle_return_num(self.drawn_lines,line)
            if temp > max_value:
                max_value = temp
                best_line = line

        if max_value != 0:
            print("triangle case")
            return best_line
```

find_best_selection 함수에서 다른 조건보다 삼각형을 만들 수 있는 경우가 있다면 해당 선분을 먼저 생성하도록 합니다. 삼각형을 만들 수 있는 선분이 여러 개라면 그중에서 가장 높은 점수를 얻을 수 있는 선분을 리턴합니다.

find_best_selection 함수 – 독립선분을 찾는 과정

```
        #독립선분 긋기 --> DFS 완전탐색
        simul_line_set = self.drawn_lines
        self.minimum_independent_line_num = 1000000000
        self.simulation_independent_line(0,simul_line_set)
```

당장에 만들 수 있는 삼각형이 없다면 DFS 탐색을 하면서 독립선분을 찾습니다. 생성할 수 있는 독립선분을 list에 담아둡니다.

find_best_selection 함수 – convex hull 위의 점과 선분 찾기

```
        #convex hull points와 convex hull lines 생성
```

```
        convex_points = self.ConvexHull(self.whole_points)
        convex_lines = []
        convex_lines.append(sorted([convex_points[0], convex_points[-1]]))
        for i in range(0,len(convex_points)-1):
            convex_lines.append(sorted([convex_points[i],convex_points[i+1]]))
```

convex hull이 있는 경우에 독립선분을 최소로 하는 선분을 만들기 위한 준비 단계로 convex hull 위의 점과 선분을 가져옵니다.

find_best_selection 함수 – 독립선분 중 best case인 독립선분 찾기

```
        # 독립선분 최소 case 에서 convex hull 위의 선분이 아닌 것을 선택
        min_length, shortest_element = min([(len(x),x) for x in
self.independent_lines_case])
        for line_set in self.independent_lines_case:
            if len(line_set) == min_length:
                for line in line_set:
                    if line in self.drawn_lines:
                        continue
                    else:
                        if len(self.whole_points) == len(convex_points):
                            #독립 선분 중에 convex hull 위의 선이 아닌것을 출력
                            if sorted(line) in convex_lines:
                                continue
                            else:
                                print("return = ", line)
                                print("independent line")
                                return [line[0], line[1]]
                        else:
                            print("return = ", line)
                            print("independent line")
                            return [line[0], line[1]]
```

독립선분 중에서 포화상태까지 독립선분의 개수가 최소가 되도록하는 case를 찾습니다. 우선 연결할 수 있는 선분 중에서 이미 연결되어 있는 선분은 제외합니다. 그리고 모든 점이 convex hull 위의 점이면 convex hull 위의 선분이 아닌 선분을 선택합니다.

find_best_selection 함수 – 포화상태면 min-max tree search 시작

```
        # 독립선분이 포화 상태일 때 min-max 적용
        simul_line_set = self.drawn_lines
        print("minmax")
        self.min_max_start(simul_line_set)
        return self.best_line
```

더 이상 만들 수 있는 독립선분이 없다면 min-max tree search를 적용합니다.

find_best_selection 함수 – simulation_independent_line 함수

```
    # DFS 완전탐색 for 독립선분
    def simulation_independent_line(self, i, simul_line_set):
```

```
        # 독립선분 case 조사하는 개수 제한
        if len(self.independent_lines_case) == 12:
            return

        # 독립선분 case 조사할 때, 선분 수가 가장 적은 case 보다 simulation set 이
많아지면 즉시 종료
        if(len(simul_line_set) > self.minimum_independent_line_num):
            return

        check = True
        for j in range(i, len(self.posible_lines)):
            if(self.check_temp_availability(simul_line_set,
self.posible_lines[j])):
                if(self.check_next_triangle(simul_line_set,
self.posible_lines[j])):
                    check = False
                    simul_line_set.append(self.posible_lines[j])
                    self.simulation_independent_line(i,simul_line_set)
                    simul_line_set.remove(self.posible_lines[j])

        if(check):
            self.organize_points(simul_line_set)
            if(not simul_line_set in self.independent_lines_case):
                self.independent_lines_case.append(simul_line_set.copy())
                if(self.minimum_independent_line_num > len(simul_line_set)):
                    self.minimum_independent_line_num = len(simul_line_set)
```

독립선분을 찾을 때 사용하는 함수입니다. 게임 시간을 고려하여 찾는 독립선분의 개수는 임의의 개수(예: 12개)로 제한합니다. 또한 탐색 시간을 줄이기 위해 독립선분이 만들어지는 개수를 조사할 때 이미 가장 적은 독립선분이 만들어지는 case보다 예상 독립선분 개수가 많아지는 경우 즉시 종료합니다. 독립선분은 DFS로 탐색하며 독립선분을 찾으면 list에 추가합니다.

find_best_selection 함수 – check_triangle 함수

```
    # 삼각형인지 판별 from system.py
    # True/False 만 반환하게 custom
    def check_triangle(self, line):

        point1 = line[0]
        point2 = line[1]

        point1_connected = []
        point2_connected = []
```

```
        for l in self.drawn_lines:
            if l==line: # 자기 자신 제외
                continue
            if point1 in l:
                point1_connected.append(l)
            if point2 in l:
                point2_connected.append(l)

        if point1_connected and point2_connected: # 최소한 2 점 모두 다른 선분과
연결되어 있어야 함
            for line1, line2 in product(point1_connected, point2_connected):

                # Check if it is a triangle & Skip the triangle has occupied
                triangle = self.organize_points(list(set(chain(*[line, line1,
line2]))))
                if len(triangle) != 3 or triangle in self.triangles:
                    continue

                empty = True
                for point in self.whole_points:
                    if point in triangle:
                        continue
                    if bool(Polygon(triangle).intersection(Point(point))):
                        empty = False

                if empty:
                    return True

        return False
```

삼각형이 만들어지는지 확인하는 함수입니다. system.py에 있는 기능 중 리턴값이 True/False가 되도록 수정했습니다. 인자로 받은 line으로 삼각형이 만들어지는지 판별합니다.

find_best_selection 함수 – check_temp_availability 함수

```
    # simulation_line_set 에 포함된 선분을 대상으로 그릴 수 있는 선분인지를 판단
    def check_temp_availability(self, simul_line_set ,line):
        line_string = LineString(line)

        # Must be one of the whole points
        condition1 = (line[0] in self.whole_points) and (line[1] in
self.whole_points)

        # Must not skip a dot
        condition2 = True
```

```
        for point in self.whole_points:
            if point==line[0] or point==line[1]:
                continue
            else:
                if bool(line_string.intersection(Point(point))):
                    condition2 = False

        # Must not cross another line
        condition3 = True
        for l in simul_line_set:
            if len(list(set([line[0], line[1], l[0], l[1]]))) == 3:
                continue
            elif bool(line_string.intersection(LineString(l))):
                condition3 = False

        # Must be a new line
        condition4 = (line not in simul_line_set)

        if condition1 and condition2 and condition3 and condition4:
            return True
        else:
            return False
```

simuation_line_set에 포함된 선분을 대상으로 규정에 부합한 선분인지 확인하는 함수입니다. 인자로 simul_line_set과 line을 받고 line이 simul_line_set 안에서 만들어질 수 있는 선분인지 판별합니다.

find_best_selection 함수 – check_next_triangle 함수

```
    # simulation_line_set에 포함된 선분을 대상으로 상대방에게 삼각형을 만드는
경우를 제공하는 선분인지를 판단
    def check_next_triangle(self, simul_line_set, line):
        simul_line_set.append(line)

        point1 = line[0]
        point2 = line[1]

        connected_with_line = []

        for next_line in self.posible_lines:
            if next_line in simul_line_set:
                continue
            elif point1 in next_line or point2 in next_line:
                connected_with_line.append(next_line)
```

```
        for next_line in connected_with_line:
            if(self.check_temp_triangle(simul_line_set,next_line)):
                simul_line_set.remove(line)
                return False

        simul_line_set.remove(line)
        return True
```

다음 턴에 상대방이 삼각형을 만들 수 있게 되는지 판별하는 함수입니다. 인자로 받은 line이 다음 턴에 상대방에게 삼각형을 내주는지 검사합니다.

find_best_selection 함수 – check_temp_triangle 함수

```
    # simulation_line_set 에 포함된 선분을 대상으로 삼각형을 만드는 선분인지를 판단
    def check_temp_triangle(self, simul_line_set,line):

        point1 = line[0]
        point2 = line[1]

        point1_connected = []
        point2_connected = []

        for l in simul_line_set:
            if l==line: # 자기 자신 제외
                continue
            if point1 in l:
                point1_connected.append(l)
            if point2 in l:
                point2_connected.append(l)

        if point1_connected and point2_connected: # 최소한 2 점 모두 다른 선분과
연결되어 있어야 함
            for line1, line2 in product(point1_connected, point2_connected):

                # Check if it is a triangle & Skip the triangle has occupied
                triangle = self.organize_points(list(set(chain(*[line, line1,
line2]))))
                if len(triangle) != 3 or triangle in self.triangles:
                    continue

                empty = True
                for point in self.whole_points:
                    if point in triangle:
                        continue
                    if bool(Polygon(triangle).intersection(Point(point))):
```

```
                        empty = False

                if empty:
                    return True

        return False
```

인자로 simul_line_set와 line을 받습니다. line이 simul_line_set에서 삼각형을 만들 수 있는지 판단합니다.

find_best_selection 함수 – check_endgame 함수

```
    # simulation_line_set 에 포함된 선분을 대상으로 게임이 끝났는지를 판단
    def check_endgame(self, simul_line_set):
        remain_to_draw = [[point1, point2] for (point1, point2) in
list(combinations(self.whole_points, 2)) if
self.check_temp_availability(simul_line_set, [point1, point2])]
        return False if remain_to_draw else True
```

게임이 끝났는지 판단하는 함수입니다.

find_best_selection 함수 – min-max 함수

```
    # min_max function
    def min_max_start(self, simul_line_set):
        print("min_max running...")
        alpha = -1000000
        beta = 100000
        evaluation_value = 0
        self.best_line = [(0, 0), (1 , 0)]
        depth = 0
        # 무조건 machine 의 turn 이니까 max_move 먼저 호출
        self.max_move(depth, simul_line_set, alpha, beta, evaluation_value)
```

min-max tree search을 시작하는 함수입니다. alpha-beta cutoff를 위한 값, evaluation 값, best case, depth를 초기화하고 max_move를 먼저 호출합니다.

find_best_selection 함수 – max_move 함수

```
    def max_move(self, depth, simul_line_set, alpha, beta, evaluation_value):
        if self.check_endgame(simul_line_set):
            return evaluation_value
        elif depth > 5:
            return evaluation_value
        else:
            next_lines = self.GenerateMove(simul_line_set) #line list 를 반환
            best_value = -1000000000
            temp_score = self.score
            temp_evaluation_value = evaluation_value
            for next_line in next_lines:
                # machine 이 이기면 점수 +1
```

```
                if(self.check_temp_triangle(simul_line_set, next_line)):
                    self.score[1] += 1

                #simul_line_set에 next line 넣고, evaluation_value 계산해서, 다음
min_move 보기
                simul_line_set.append(next_line)
                evaluation_value += self.EvalGameStateMax(simul_line_set)
                next_node_value = self.min_move(depth+1, simul_line_set, alpha,
beta, evaluation_value)
                # 나온 결과 값의 최댓값만 저장
                if next_node_value > best_value:
                    best_value = next_node_value
                    self.best_line = next_line
                    alpha = max(alpha, best_value)
                # 점수랑 simul_line_set랑 evaluation_value 다시 원래대로 되돌리기
                simul_line_set.remove(next_line)
                self.score = temp_score
                evaluation_value = temp_evaluation_value

                #alpha cuttoff
                if beta <= alpha:
                    break

            return best_value
```

5의 깊이 제한이 걸리거나 게임이 종료되면 evaluation을 계산합니다. 만약 끝나지 않았으면 자식 노드를 확장하면서 계속 탐색합니다. 자식 노드를 확장할 때 휴리스틱을 사용하여 자식 노드 중 평가함수 값이 가장 높은 3개의 노드만 확장합니다. 자식 노드의 값을 부모 노드로 올릴 때는 alpha-beta cutoff를 사용합니다.

find_best_selection 함수 – min_move 함수

```
    def min_move(self, depth, simul_line_set, alpha, beta, evaluation_value):
        if self.check_endgame(simul_line_set):
            return evaluation_value
        elif depth > 5:
            return evaluation_value
        else:
            next_lines = self.GenerateMove(simul_line_set) #line list를 반환
            best_value = 1000000000
            temp_score = self.score
            temp_evaluation_value = evaluation_value
            for next_line in next_lines:
                # user가 이기면 점수 +1
                if(self.check_temp_triangle(simul_line_set, next_line)):
```

```
                    self.score[0] += 1

                #simul_line_set에 next line 넣고, evaluation_value 계산해서, 다음
max_move 보기
                simul_line_set.append(next_line)
                evaluation_value -= self.EvalGameStateMin(simul_line_set)
                next_node_value = self.max_move(depth+1, simul_line_set,alpha,beta,
evaluation_value)
                # 나온 결과 값의 최솟값만 저장
                if next_node_value < best_value:
                    best_value = next_node_value
                    beta = min(beta, best_value)
                # 점수랑 simul_line_set랑 evaluation_value 다시 원래대로 되돌리기
                simul_line_set.remove(next_line)
                self.score = temp_score
                evaluation_value = temp_evaluation_value

                #beta cutoff
                if beta <= alpha:
                    break

            return best_value
```

min_move 함수는 max_move와 같은 방식으로 작동합니다.

find_best_selection 함수 – EvalGameStateMax 함수

```
    # max 노드에서 evaluation
    def EvalGameStateMax(self, simul_line_set):
        node_value=0

        if(self.check_next_triangle_return_num(simul_line_set[0:len(simul_line_set)
-1:],simul_line_set[-1])==2):
            node_value-=30
        elif(self.check_next_triangle_return_num(simul_line_set[0:len(simul_line_se
t)-1:],simul_line_set[-1])==1):
            node_value-=10

        if(self.check_temp_triangle_return_num(simul_line_set[0:len(simul_line_set)
-1:],simul_line_set[-1])==2):
            node_value+=45
        elif(self.check_temp_triangle_return_num(simul_line_set[0:len(simul_line_se
t)-1:],simul_line_set[-1])==1):
            node_value+=15
```

```
        return node_value
```

Max 노드에서 게임 상황에 대한 evaluation을 계산합니다. 선분을 만들었을 때 삼각형을 2개 얻을 수 있으면 45점, 1개 얻을 수 있으면 15점을 추가합니다. 반대로 상대방 턴에서 삼각형을 2개 주면 -30점, 1개를 주면 -10을 합니다.

find_best_selection 함수 – EvalGameStateMin 함수

```
    # min 노드에서 evaluation
    def EvalGameStateMin(self, simul_line_set):
        node_value=0

        if(self.check_next_triangle_return_num(simul_line_set[0:len(simul_line_set)
-1:],simul_line_set[-1])==2):
            node_value-=45
        elif(self.check_next_triangle_return_num(simul_line_set[0:len(simul_line_se
t)-1:],simul_line_set[-1])==1):
            node_value-=15

        if(self.check_temp_triangle_return_num(simul_line_set[0:len(simul_line_set)
-1:],simul_line_set[-1])==2):
            node_value+=30
        elif(self.check_temp_triangle_return_num(simul_line_set[0:len(simul_line_se
t)-1:],simul_line_set[-1])==1):
            node_value+=10

        return node_value
```

Min 노드에서 게임 상황에 대한 evaluation을 계산합니다. 상대방 입장이기 때문에 선분을 만들었을 때 삼각형을 2개 만들 수 있으면 -45점, 1개를 만들 수 있으면 -15점 합니다. 반대로 우리 턴에서 삼각형을 2개 만들 수 있도록 하면 30점, 1개를 만들 수 있도록 하면 10점을 더합니다.

find_best_selection 함수 – GenerateMove 함수

```
def GenerateMove(self, simul_line_set):
        return_next_lines = []
        posible_lines = []
        evalueted_lines_value = []

        #posible_lines 고르기
        for (point1, point2) in list(combinations(self.whole_points, 2)):
            temp_line = [point1, point2]
            if self.check_temp_availability(simul_line_set, temp_line):
                posible_lines.append(temp_line)

        num_of_child_node = 3
```

```
        #각각 가능한 선분에 대해 heuristic 값 구하기
        # h(x) = getScore(x) - nextTriangle(x)
        for next_line in posible_lines:
            h_x =0   # heuristic value

            # getScore()
            getScore = self.check_temp_triangle_return_num(simul_line_set,
next_line)
            if(getScore != 0):
                h_x += 3*getScore

            # nextTriangle()
            nextTriangle = self.check_next_triangle_return_num(simul_line_set,
next_line)
            if nextTriangle == 0:
                h_x -= 0
            elif nextTriangle == 1:
                h_x -= 2
            else:
                h_x -= 1

            evalueted_lines_value.append([h_x, next_line])

        evalueted_lines_value.sort()
        if(len(evalueted_lines_value) < num_of_child_node):
            for element in evalueted_lines_value:
                return_next_lines.append(element[1])
        else:
            for i in range(num_of_child_node):
                return_next_lines.append(evalueted_lines_value[i][1])

        return return_next_lines
```

자식 노드를 확장하는 함수입니다. 게임 시간을 고려하여 확장하는 자식 노드의 수는 3개로 제한했습니다. 확장할 수 있는 모든 자식 중에서 휴리스틱 평가함수 값이 가장 높은 3개의 노드만 확장합니다. 휴리스틱의 평가함수 h(x) = getScore(x) – nextTriangle(x)로 [선분을 만들었을 때 (이번 턴에 얻을 수 있는 점수 * 3) – (상대방 턴에서 만들 수 있는 삼각형의 개수에 따른 리턴값)]입니다. nextTriangle(x)의 리턴값은 임의의 선분을 만들었을 때 만약 상대방 턴에서 만들 수 있는 삼각형이 없으면 0, 삼각형이 1개라면 2, 삼각형이 2개 이상이면 1입니다.

find_best_selection 함수 – check_temp_triangle_return_num 함수

```
# 현재 내가 긋는 선분이 몇개의 삼각형을 만드는지를 return 하는 function
    def check_temp_triangle_return_num(self, simul_line_set,line):
```

```
        point1 = line[0]
        point2 = line[1]

        point1_connected = []
        point2_connected = []

        for l in simul_line_set:
            if l==line: # 자기 자신 제외
                continue
            if point1 in l:
                point1_connected.append(l)
            if point2 in l:
                point2_connected.append(l)

        return_value = 0

        if point1_connected and point2_connected: # 최소한 2 점 모두 다른 선분과
연결되어 있어야 함
            for line1, line2 in product(point1_connected, point2_connected):

                # Check if it is a triangle & Skip the triangle has occupied
                triangle = self.organize_points(list(set(chain(*[line, line1,
line2]))))
                if len(triangle) != 3 or triangle in self.triangles:
                    continue

                empty = True
                for point in self.whole_points:
                    if point in triangle:
                        continue
                    if bool(Polygon(triangle).intersection(Point(point))):
                        empty = False

                if empty:
                    return_value += 1

        return return_value
```

휴리스틱에서 getScore(x)에 적용하기 위한 함수입니다. 이번 턴에서 연결하는 선분이 몇 개의 삼각형을 만들 수 있는지 그 값을 리턴합니다.

find_best_selection 함수 – check_next_triangle_return_num 함수

```
# simulation_line_set 에 포함된 선분을 대상으로 상대방에게 삼각형을 몇개 제공하게
되는지를 return 하는 함수
```

```
    def check_next_triangle_return_num(self, simul_line_set, line):
        simul_line_set.append(line)

        point1 = line[0]
        point2 = line[1]

        connected_with_line = []
        return_value =0

        posible_lines = []
        for (point1, point2) in list(combinations(self.whole_points, 2)):
            temp_line = [point1, point2]
            if self.check_temp_availability(simul_line_set,temp_line):
                posible_lines.append(temp_line)

        for next_line in posible_lines:
            if next_line in simul_line_set:
                continue
            elif point1 in next_line or point2 in next_line:
                connected_with_line.append(next_line)

        for next_line in connected_with_line:
            if(self.check_temp_triangle(simul_line_set,next_line)):
                return_value += 1

        simul_line_set.remove(line)
        return return_value
```

휴리스틱에서 nextTriangle(x)에 적용하기 위한 함수입니다. 이번 턴에서 연결한 선분으로 인해 다음 턴에 상대방이 새로운 선분을 만들어 몇 개의 삼각형을 얻을 수 있는지 계산합니다.

find_best_selection 함수 – ConvexHull 함수와 ccw 함수

```
def ConvexHull(self, points):
        upper = []
        lower = []
        for p in sorted(points):
            while len(upper) > 1 and self.ccw(upper[-2], upper[-1], p) > 0:
                upper.pop()
            while len(lower) > 1 and self.ccw(lower[-2], lower[-1], p) < 0:
                lower.pop()
            upper.append(p)
            lower.append(p)

        convex_points = upper + lower
```

```
        result = []
        [result.append(x) for x in convex_points if x not in result]
        return result

    def ccw(self, a,b,c):
        return a[0]*b[1] + b[0]*c[1] + c[0]*a[1] - (b[0]*a[1] + c[0]*b[1] +
a[0]*c[1])
```

convex hull을 계산하고 그때 쓰이는 ccw 함수입니다.

- 코드가 모두 작성되어 있는 machine.py 파일은 별도로 첨부했습니다.

6. 장단점

1) 장점

제한된 시간 안에 최적의 수를 찾기 위해 최대한 효율적인 방법을 다수 활용하였습니다.

첫 번째로는 게임 초반 규칙 기반을 적용할 때 가능한 독립선분 후보군에서 best case를 찾을 때입니다. 후보군에서 하나씩 골라 해당 선분이 만들어졌을 때 게임 상황에서 앞으로 몇 개의 독립선분이 더 연결될 수 있는지 검사하는 과정은 포화상태까지 시뮬레이션을 해야 한다는 점에서 많은 시간이 소요되었습니다. 이에 대해 각 독립선분 후보가 게임 상황에서 만들어낼 수 있는 독립선분의 개수를 계산하고 best case를 저장하도록 했습니다. 만약 현재의 best case보다 독립선분의 개수를 더 줄일 수 있는 case가 있다면 best case를 해당 case로 교체합니다. 그리고 독립선분 후보군에서 하나씩 골라 계산하는 도중에 best case보다 독립선분의 개수가 더 많아지게 된다면 해당하는 후보의 계산을 종료하고 후보군에서 제외하게 됩니다.

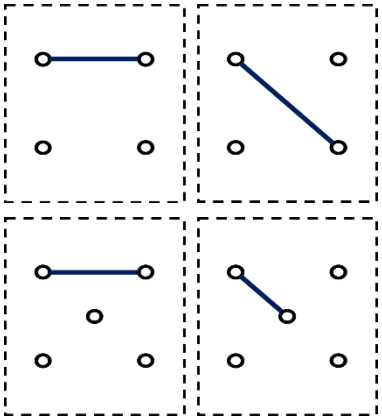
두 번째는 min-max tree search에서 탐색 시간을 줄이기 위해 Top-Down과 Bottom-Up에서 쓸 수 있는 방법을 모두 사용했고 그 중 하나가 alpha-beta cutoff 입니다. alpha-beta cutoff는 Bottom-Up에서 탐색 시간을 줄이는 방법으로 alpha와 beta 값을 통해 계산할 필요가 없는 가지를 자르는 것입니다. 저희 머신에서도 max_move와 min_move 모두에 해당 방법을 적용했습니다.

마지막으로 min-max tree search의 Top-Down에서 탐색 시간을 줄이는 방법인 휴리스틱을 활용했습니다. 부모 노드에서 확장할 수 있는 자식 노드를 확인하고 각 노드에 휴리스틱 평가함수의 값을 매겨 자식 노드를 내림차순으로 정렬합니다. 그중 평가함수의 값이 높은 특정 개수의 자식만 확장하여 계속 탐색하도록 했습니다. 덕분에 짧은 시간 안에 유리할 것으로 기대되는 노드를 집중적으로 확장할 수 있게 되었습니다.

2) 단점

탐색 시간을 줄이기 위한 노력에도 불구하고 턴당 1분이라는 시간을 맞추기 위해서는 게임 초반 규칙 가반에서 조사할 독립선분 후보의 수와 min-max tree search에서 확장할 자식 노드의 개수와 깊이를 제한하게 되었습니다. 이로 인해 탐색하지 못한 노드에 대해서 더 적절한 해가 있을 수 있다는 한계가 존재합니다.

또다른 문제로, 특수한 경우에 대한 처리가 부족합니다. 예를 들어 다음과 같은 경우를 고려할 우 있습니다.



위와 같은 예시에서, 그려질 수 있는 삼각형의 개수는 짝수(2, 4)입니다. 이 때 선공이 오른쪽과 같이 대각선으로 그은 경우 무승부를 유도할 수 있습니다(그렇지 않은 경우 패배할 가능성이 높음). 그러나 왼쪽과 같이 convex hull 위의 선분을 선택한 경우, 후공의 최선에 따라 반드시 패배하게 됩니다. 이들은 비교적 간단한 예시이나, **포화상태의 형태에 따라 게임의 양상이 크게 달라짐**을 확실히 알 수 있습니다. 두 번째 규칙(convex hull 위의 선분을 피함)은 이러한 예시로부터 기인하였으나, 초기 주어질 수 있는 점의 개수와 위치에 따른 경우의 수가 무한하므로 항상 좋은 전략임을 장담할 수 없습니다.

7. 역할 분담

이름	역할
권민혁	팀장, 게임 초반 규칙 설계, 규칙 기반/휴리스틱/min-max 동작 코드 구현
김종권	게임 상황 분석, 잘못 둔 수 모니터링, 보고서 작성
김현민	min-max evaluation 함수 설계 및 코드 구현
이희철	게임 상황 분석, 휴리스틱 함수 설계, 보고서 작성

8. 추가적인 고찰

게임을 진행하면서 몇 가지 문제점을 발견하였습니다.

우선 첫 번째 규칙을 적용하는 것이 항상 이득이 아니라는 점을 고려하지 않았습니다. 가장 간단하고 직관적인 예시로, 삼각형 ABC 내부의 한 점 P가 존재하고, 선분 PA가 이어진 상태를 가정할 수 있습니다. 이 경우, 선분 PB를 선택하여 긋는다면 1점을 얻을 수 있으므로, 구현한 알고리즘에 따라 해당 선분을 긋습니다. 그러나 상대에게 나머지 선분 PC를 그어 2점을 얻을 수 있는 상황을 제공합니다. **n이 4 이상인 닫힌 n각형이 그려진 경우, 마지막에 2점을 얻는 상황이 항상 발생함**을 알 수 있었습니다. 포화상태에서 이러한 조건을 만족하는 n각형이 만들어지지 않는 경우는 발생할 수 있으나, 독립선분을 선택하는 과정에서 이러한 상황의 출현 빈도가 꽤 높음을 확인하였습니다. 따라서 점수를 얻을 수 있는 상황에서 해당 선분을 그었을 때 상대방이 얻는 점수(0, 1, 2점)를 추가적으로 고려하여 다음과 같이 알고리즘을 개선할 수 있습니다.

0점 : 상대가 어떠한 이득도 취하지 못하므로, 반드시 긋습니다.

1점 : 서로 번갈아가며 삼각형을 그릴 수 있는 상황이 발생할 수 있습니다. (휴리스틱으로 해결)

2점 : 닫힌 다각형에서만 발생하며 이후 자신이 삼각형을 그릴 수 없으므로 긋지 않습니다.

이러한 방식을 통해 상대가 얻을 수 있는 점수가 1점인 경우에 대하여 min-max tree를 적용하지 않고, 더 빠른 방식을 채택할 수 있습니다. 포화상태에서 현재 정보를 복사하여 저장한 후, 서로 번갈아가며 점수를 1점씩 얻을 수 있을 때, 해당 선분들을 list에 기록합니다(실제로 게임을 진행하는 것이 아니라, 자체적으로 메모리 공간을 할당하여 진행). 이 때 list의 크기가 홀수인지 짝수인지에 따라 해당 선분의 가치를 판단할 수 있습니다.

홀수인 경우, 자신이 마지막으로 삼각형을 긋기 때문에 상대가 x점을 얻은 경우, 자신은 x+1점을 얻어 유리합니다. 이는 점수에서 이득일 뿐만 아니라, 포화상태에서 상대방에게 차례를 넘기게 되므로 매우 좋은 선택이 됩니다.

짝수인 경우, 상대가 마지막으로 삼각형을 긋기 때문에 상대가 x점을 얻은 경우, 자신도 x점을 얻게 됩니다. 이는 언뜻 나쁘지 않은 선택으로 보이나, 포화상태에서 자신의 차례가 되므로 큰 의미가 없습니다. 오히려 앞으로 선택할 선분의 개수가 줄어듦으로써 상대가 잘못된 선택을 할 가능성을 크게 줄이게 됩니다.

따라서 연속적으로 그려지는 삼각형 개수의 홀짝에 따라 선분을 선택하는 전략이 유효합니다.

9. 느낀점

컴퓨터에게 게임 상황을 판단하고 좋은 수를 두는 방법에 대해 규칙 기반으로 알려주는 것이 매우 어렵다는 것을 알게 되었습니다. 그리고 예외 상황까지 고려한다면 개발자가 일일이 다 조작을 하는 것은 거의 불가능에 가깝다는 생각이 들었습니다. 이 때문에 사람이 직접 일일이 규칙을 설계하는 것보다 다양한 탐색 방법을 고려하거나 기계학습 등 새로운 방식의 필요성을 느끼게 되었습니다.

과제 주제에 대한 내용 이외에도 전체적으로 좋은 팀원 분들과 함께 팀프로젝트를 진행하면서 많은 점들을 경험할 수 있었습니다. 혼자서는 하지 못할 다양하고 심화된 사고를 할 수 있었고 새로운 알고리즘에 대해 습득할 수 있었습니다. 그리고 이번 프로젝트를 통해 컴퓨터공학을 전공하는 학생으로서 항상 새로운 지식과 기술에 도전하고 배우려는 태도를 갖는 것이 필요하다고 느꼈습니다. 프로젝트를 마무리 짓기까지 쉽지 않은 과정이었지만 학구적인 고민과 협력을 할 수 있는 시간이었습니다.