گزارش پروژه هوش مصنوعی

درین رستمی ۹۷۲۴۳۰۳۴

زهرا هاشمی ۹۷۲۴۳۰۷۲

متغیرهای اصلی بازی:

```
dim = 10 #max 100 if you want the visual representation of the hive to be clean
hive = []
red_ins = {"Queen": 1, "Beetle": 2, "Cicada": 3, "Spider": 2, "Ant": 3} #red's insects off board
green_ins = {"Queen": 1, "Beetle": 2, "Cicada": 3, "Spider": 2, "Ant": 3} #green's insects off board
red_q = [] #coordinates for the red queen
green_q = [] #coordinates for the green queen
greens_onboard = []
reds_onboard = []
co = [[1,1], [1,-1], [-1,1], [-1,-1], [2,0], [-2,0]] #coordinates for checking neighbors
```

hive یک لیست سه بعدی است که بعد اول، سطر، بعد دوم، ستون و بعد سوم که دو بعدی است، به صورت زیر، مهره‌ای از بازی که در آن خانه قرار دارد را نمایش می‌دهد:

hive[r][c][0] = insect, hive[r][c][1] = (r,c)

red_ins و green_ins تعداد هر مهره از دو بازیکن را نشان می‌دهد که وارد بازی نشده است.

لازم است که مختصات ملکه بازیکن سبز و قرمز در green_q و red_q نگه داشته شود.

greens_onboard و reds_onboard لیستی از مختصات مهره‌های سبز و قرمز هستند که بر روی برد بازی قرار دارند.

متغیر dim تعداد سطر و ستون‌ها را تعیین می‌کند که با تعیین مقدار ۱۰ برای این متغیر، برد بازی به شکل زیر خواهد بود:

```
0,0     0,2     0,4     0,6     0,8
    1,1     1,3     1,5     1,7     1,9
2,0     2,2     2,4     2,6     2,8
    3,1     3,3     3,5     3,7     3,9
4,0     4,2     4,4     4,6     4,8
    5,1     5,3     5,5     5,7     5,9
6,0     6,2     6,4     6,6     6,8
    7,1     7,3     7,5     7,7     7,9
8,0     8,2     8,4     8,6     8,8
    9,1     9,3     9,5     9,7     9,9
--------------------------------
```

برای اینکه برد را به حالت شش ضلعی و کندوی زنبور پرینت کنیم لازم است زمانی که سطر و ستون هم‌زمان زوج و یا فرد هستند، پرینت شود:

```
#func for printing hive
def print_board(hive):
    for i in range(dim):
        for j in range(dim):
            if (i%2==0 and j%2==0) or (i%2==1 and j%2==1):
                if hive[i][j][0][1] == 'r':
                    print(Fore.RED + hive[i][j][0], end=" ")
                elif hive[i][j][0][1] == 'g':
                    print(Fore.GREEN + hive[i][j][0], end=" ")
                else:
                    print(Fore.WHITE + hive[i][j][0], end=" ")
            else:
                print("     ", end=" ")
        print()
    print(Fore.WHITE + "--------------------------------")
```

متغیری به نام turn داریم که مشخص می‌کند نوبت بازی با سبز است یا قرمز. (در صورت بازی با هوش مصنوعی، بازیکن قرمز، AI است و شروع کننده بازی، قرمز می‌باشد.)

```
turn = 0 # even numbers red, odd numbers green
```

در ابتدای تابع main، برد بازی باید پرینت شود. اگر مهره‌ای بر روی برد قرار داده شده بود با فرمت زیر نمایش داده می‌شود.

[ ][r or g (representing red or green)],[first letter of game bead] r,S

```
0,0       0,2       0,4       0,6       0,8
     1,1       1,3       1,5       1,7       1,9
2,0       2,2       2,4       2,6       2,8
     3,1       3,3       3,5       3,7       3,9
4,0       4,2       r,S       4,6       4,8
     5,1       r,Q       g,Q       5,7       5,9
6,0       6,2       6,4       6,6       6,8
     7,1       7,3       7,5       7,7       7,9
8,0       8,2       8,4       8,6       8,8
     9,1       9,3       9,5       9,7       9,9
--------------------------------
```

```
if check_can_put(turn) == 0 and check_can_move(turn) == 0:
      turn += 1
      continue
```

قبل از اینکه کسی که نوبتش است بخواهد حرکتی کند یا مهره ای را قرار دهد، باید بررسی شود که اصلا می‌تواند این کار را بکند یا نه. اگر نمیتوانست، باید نوبت بازی به بازیکن حریف داده شود.

برای این کار از دو تابع check_can_put و check_can_move استفاده می‌کنیم:

در تابع check_can_put ابتدا بررسی می‌کنیم که مهره‌ای برای گذاشتن مانده است یا نه. حال اگر اطراف مهره‌های بازیکن، محلی خالی باشد، باید اطراف محل خالی چک شود. اگر مهره حریف قرار داشته باشد، نمی‌تواند به حریف بچسبد.

```
def check_can_put(turn):
    global reds_onboard
    global greens_onboard
    global hive
    global co


    main_onboard = reds_onboard if turn % 2 == 0 else greens_onboard
    op_color = 'g' if turn % 2 == 0 else 'r'

    if len(main_onboard) == 11:
        return 0
    elif len(main_onboard) == 0:
        return 1

    for i in range(len(main_onboard)):
        for j in range(len(co)):
            row = co[j][0] + main_onboard[i][0]
            col = co[j][1] + main_onboard[i][1]
            tmp = is_insect(row, col)
            if tmp == 'n':
                for k in range(len(co)):
                    tmp2 = is_insect(row + co[k][0], col + co[k][1])
                    if tmp2 == op_color:
                        break
                    elif k == len(co) -1:
                        return 1

    return 0
```

در تابع check_can_move ، برای مهره‌های بازیکن بررسی می‌کنیم که اگر حرکت کند، پیوند کندو را از بین می‌برد یا نه (is_bridge) اگر از بین نبرد، مهره‌ای که در آن محل قرار دارد را برمیداریم. اگر سوسک یا ملخ باشد، چون میتواند بپرد به هرجا که بخواهد، پس حرکت امکان پذیر است. اگر مهره ای دیگر باشد، باید اطرافش بررسی شود، در آن حوالی باید بررسی شود که اگر مهره‌ای وجود دارد، قانون لغزیدن را رعایت می‌کند یا نه (can_enter) اگر با این شرایط بتواند حرکت کند، تابع مقدار یک را برمی‌گرداند که یعنی حرکتی می‌تواند صورت بگیرد.

```
def check_can_move(turn):
    global reds_onboard
    global greens_onboard
    global hive
    global co

    main_onboard = reds_onboard if turn % 2 == 0 else greens_onboard

    for i in range(len(main_onboard)):
        row = main_onboard[i][0]
        col = main_onboard[i][1]
        if not is_bridge(row, col):
            ins = hive[row][col].pop(0)
            if ins[3] == 'C' or ins[3] == 'B':
                hive[row][col].insert(0, ins)
                return 1
            for j in range(len(co)):
                row2 = co[j][0] + row
                col2 = co[j][1] + col
                if has_insect_around(row2, col2):
                    if can_enter(row, col, row2, col2):
                        hive[row][col].insert(0, ins)
                        return 1
            hive[row][col].insert(0, ins)
    return 0
```

در تابع is_bridge، اگر چند سوسک یا ملخ در آن محل باشند، با حرکت آنها همچنان پیوند برقرار می‌ماند. این تابع حالتی خاص از dfs است که از تابع hive_dfs استفاده می‌کند.

```
#funcs for checking if removing insect disconnects hive
def hive_dfs(row, col, visited, cnt_visited = 1):
    global hive
    global co

    visited[row][col] = 1
    for i in range(len(co)):
        r = co[i][0] + row
        c = co[i][1] + col
        is_ins = is_insect(r, c)
        if (is_ins == 'r' or is_ins == 'g') and visited[r][c] == 0: #has unvisited
insect
            cnt_visited += len(hive[r][c]) - 1
            cnt_visited = hive_dfs(r, c, visited, cnt_visited)
    return cnt_visited

def is_bridge(row, col):
    global hive
    global dim
    global co
    global red_ins
    global green_ins

    if(len(hive[row][col]) > 2): #insects on top of eachother
        return 0

    moving_ins = hive[row][col].pop(0) #remove ins temporarily

    visited = []
    for _ in range(dim):
        visited. append([0 for _ in range(dim)])

    for i in range(len(co)):
        tmp = is_insect(co[i][0] + row, co[i][1] + col)
        if  tmp == 'r' or tmp == 'g': #has insect
            cnt_visited = hive_dfs(co[i][0] + row, co[i][1] + col, visited) + 1 #1 =
the insect we're moving
            cnt_onboard = 22 - sum(red_ins.values()) - sum(green_ins.values()) #all -
off the board
            hive[row][col].insert(0, moving_ins)
            return 0 if cnt_visited == cnt_onboard else 1
```

اولین حرکت با AI است. نوبت‌های فرد با انسان است که باید ورودی را وارد کند. از فرد شماره سطر، ستون، نوع حرکت (move-put) و نوع مهره بازی خواسته می‌شود.

```
turn  1
choose where to put what insect
row: 5
col: 5
insect: Ant
 0,0        0,2        0,4        0,6        0,8
      1,1        1,3        1,5        1,7        1,9
 2,0        2,2        2,4        2,6        2,8
      3,1        3,3        3,5        3,7        3,9
 4,0        4,2        r,S        4,6        4,8
      5,1        5,3        g,A        5,7        5,9
 6,0        6,2        6,4        6,6        6,8
      7,1        7,3        7,5        7,7        7,9
 8,0        8,2        8,4        8,6        8,8
      9,1        9,3        9,5        9,7        9,9
 -------------------------------
```

تابع check_action، بررسی می‌کند که آیا این درخواست بازیکن قابل انجام هست یا نه.

```
#check if this action can be done (based on queen existing)
def check_action(turn):
    global red_ins
    global green_ins

    action = input("move or put: ")
    if action != "move" and action != "put":
        print("enter a valid action")
        return 0
    if action == "move":
        if (turn%2 == 0 and red_ins["Queen"] == 1) or (turn%2 == 1 and
green_ins["Queen"] == 1):
            print("You don't have a queen")
            print("enter a valid action")
            return 0
        elif (turn == 6 and red_ins["Queen"] == 1) or (turn == 7 and
green_ins["Queen"] == 1):
            print("You have to put the queen on the board now")
            return 0
    return action
```

تابع check_coordinate بررسی می‌کند آیا سطر و ستون به درستی وارد شده است یا نه.

```
def check_coordinate(row, col):

    if (row%2 == 0 and col%2 == 1) or (row%2 == 1 and col%2 == 0):
        return 0
    return 1
```

تابع check_insect بررسی می‌کند که نام مهره به درستی وارد شده است یا نه.

```
def check_ins(ins):
    if ins != 'Queen' and ins != 'Cicada' and ins != 'Beetle' and ins != 'Ant'
and ins != 'Spider':
        return 0
    return 1
```

تابع is_insect یکی از چهار حالت را برای یک خانه برمی‌گرداند.

```
#check if cell has an insect inside or not (or is out of bounds)
def is_insect(row, col):
    global dim
    global hive

    if row >= dim or col >= dim or row < 0 or col < 0:
        return 'o' #out of bounds
    elif hive[row][col][0][1] == 'g':
        return 'g'
    elif hive[row][col][0][1] == 'r':
        return 'r'
    else:
        return 'n' #no insect
```

تابع put_insect ، اگر شرایط قرار دادن مهره در آن خانه وجود داشته باشد، (به مهره حریف نچسبد) مهره را قرار می‌دهد.

```
def put_insect(row, col, ins, turn):
    global hive
    global red_ins
    global green_ins
    global red_q
    global green_q
    global greens_onboard
    global reds_onboard

    for i in range(len(co)):
        if turn%2 == 0 and is_insect(co[i][0] + row, co[i][1] + col) == 'g':
            print("there is a green around here")
```

```
            return 0
        elif turn%2 == 1 and is_insect(co[i][0] + row, co[i][1] + col) == 'r':
            print("there is a red around here")
            return 0

    if is_insect(row, col) == 'n': #no insect is here
        if turn%2 == 0:
            if red_ins[ins] == 0:
                print("You don't have any of this insect left")
                return 0
            if turn == 6 and ins != "Queen" and red_ins["Queen"] == 1: #queen
needs to be inside when r_turn <= 4
                print("put the queen on the board")
                return 0

            hive[row][col].insert(0, " r," + ins[0] + " ")
            red_ins[ins] -= 1
            reds_onboard.append([row, col])
            if ins == "Queen":
                red_q = [row, col]
        else:
            if green_ins[ins] == 0:
                print("You don't have any of this insect left")
                return 0
            if turn == 7 and ins != "Queen" and green_ins["Queen"] == 1: #queen
needs to be inside when g_turn <= 4
                print("put the queen on the board")
                return 0

            hive[row][col].insert(0, " g," + ins[0] + " ")
            green_ins[ins] -= 1
            greens_onboard.append([row, col])
            if ins == "Queen":
                green_q = [row, col]

        return 1
    else:
        print("can't put insect here")
        return 0
```

تابع `move_insect` ، مهره را به خانه مورد نظر جابه‌جا می‌کند. (چون تابع برای هر مهره یک مدل خاص نوشته شده و طولانی است، لذا کد ملکه فقط در گزارش آورده شده است.)

```
#func for moving insects from
def move_ins(src_row, src_col, dst_row, dst_col, ins_co = []):
    global hive
    global co
    global reds_onboard
    global greens_onboard

    if is_insect(dst_row, dst_col) == 'o':
        print("destination out of bounds")
        return 0

    if is_bridge(src_row, src_col) == 1:
        print("moving this insect disconnects the hive")
        return 0

    ins = hive[src_row][src_col][0]

    if ins[3] == 'Q': #Queen
        if is_insect(dst_row, dst_col) != 'n':
            print("invalid destination")
            return 0
        if has_insect_around(dst_row, dst_col) == 0:
            print("destination is not connected to hive")
            return 0
        if not can_enter(src_row, src_col, dst_row, dst_col):
            print("unable to move to destination")
            return 0
        for i in range(len(co)):
            if dst_row == co[i][0] + src_row and dst_col == co[i][1] + src_col:
                hive[dst_row][dst_col].insert(0, ins)
                hive[src_row][src_col].pop(0)
                if ins[1] == 'r':
                    red_q.remove([src_row, src_col])
                    red_q.append([dst_row, dst_col])
                else:
                    green_q.remove([src_row, src_col])
                    green_q.append([dst_row, dst_col])
        print("Queens can only move 1 cell around")
        return 0
```

تابع `who_wins()` اگر کسی بازی را ببرد مقدار یک را برمی‌گرداند و برنده را تعیین کرده و پرینت می‌کند.

```
def who_wins():
    global hive
    global green_q
    global red_q
    global co

    r_win = 2
    g_win = 2

    if red_q:
        for i in range(len(co)):
            counter = 0
            tmp = is_insect(co[i][0] + red_q[0], co[i][1] + red_q[1])
            if not(tmp == 'r' or tmp == 'g'): #a cell with no insect is around
the red queen
                counter += 1
        if counter == 6:
            g_win = 1

    if green_q:
        for i in range(len(co)):
            counter = 0
            tmp = is_insect(co[i][0] + green_q[0], co[i][1] + green_q[1])
            if not(tmp == 'r' or tmp == 'g'): #a cell with no insect is around
the green queen
                counter += 1
        if counter == 6:
            r_win = 1

    if g_win == 1 or r_win == 1:
        if g_win == 1 and r_win == 1:
            print("The score is equal")
        elif g_win == 1:
            print("Player green wins")
        else:
            print("Player red wins")
        return 1
    return 0
```

AI Part

الگوریتم minimax را همراه با هرس آلفا بتا پیاده سازی کردیم.

```
def minimax(turn, depth, maximizing, board, red_ins, green_ins, red_q, green_q,
        reds_onboard, greens_onboard, alpha, beta):
```

این تابع برد را می‌گیرد و با استفاده از تابع available_moves حرکات قابل انجام را در قالب درخت minimax پیدا کرده و وقتی به برگ رسید، برگ ها را evaluate می‌کند و سپس بهترین حرکت را در متغیر best_move می‌ریزد.

```
    color = ' r,' if turn % 2 == 0 else ' g,'

    if depth < 0 :
        return 0, None
    elif depth == 0:
        return evaluate (turn, board, red_q, green_q), None

    new_red_ins = deepcopy(red_ins)
    new_green_ins = deepcopy(green_ins)
    new_red_q = deepcopy(red_q)
    new_green_q = deepcopy(green_q)
    new_reds_onboard = deepcopy(reds_onboard)
    new_greens_onboard = deepcopy(greens_onboard)
    moves = available_moves(board, turn, new_red_ins, new_green_ins, new_red_q,
new_green_q,
        new_reds_onboard, new_greens_onboard)

    if maximizing == 1: #always for red during AI vs human
        best_sib = -inf
        best_move = None
        for move in moves:
            new_board = deepcopy(board)
            if move[1] !=
None:                                                    # action = move
                new_board[move[1][0]][move[1][1]].pop(0)
                new_board[move[2][0]][move[2][1]].insert(0, color + move[0] + "
")
                #we know it's always red
                new_reds_onboard.append([move[2][0], move[2][1]])
                new_reds_onboard.remove([move[1][0], move[1][1]])
                if move[0] == 'Q':
                    new_red_q = [move[2][0], move[2][1]]
            else:
      # action = put
```

```
                new_board[move[2][0]][move[2][1]].insert(0, color + move[0] + "
")
                #we know it's always red
                new_reds_onboard.append([move[2][0], move[2][1]])
                if move[0] == 'Q':
                    new_red_q = [move[2][0], move[2][1]]
                    new_red_ins["Queen"] -= 1
                elif move[0] == 'B':
                    new_red_ins["Beetle"] -= 1
                elif move[0] == 'S':
                    new_red_ins["Spider"] -= 1
                elif move[0] == 'C':
                    new_red_ins["Cicada"] -= 1
                elif move[0] == 'A':
                    new_red_ins["Ant"] -= 1


            value, _ = minimax(turn + 1, depth - 1, not maximizing, new_board,
new_red_ins, new_green_ins,
                               new_red_q, new_green_q, new_reds_onboard,
new_greens_onboard, alpha, beta)
            if value > best_sib:
                best_sib = value
                best_move = move
            alpha = max(alpha, best_sib)

            if beta <= alpha:
                break

        return best_sib, best_move

    else: #always for green during AI vs human
        best_sib = inf
        best_move = None
        for move in moves:
            new_board = deepcopy(board)
            if move[1] != None:                             # action = move
                new_board[move[1][0]][move[1][1]].pop(0)
                new_board[move[2][0]][move[2][1]].insert(0, color + move[0] + "
")
                #we know it's always green
                new_greens_onboard.append([move[2][0], move[2][1]])
                new_greens_onboard.remove([move[1][0], move[1][1]])
                if move[0] == 'Q':
                    new_green_q = [move[2][0], move[2][1]]
```

```
            else:                                           # action = put
                new_board[move[2][0]][move[2][1]].insert(0, color + move[0] + "
")
                #we know it's always green
                new_greens_onboard.append([move[2][0], move[2][1]])
                if move[0] == 'Q':
                    new_green_q = [move[2][0], move[2][1]]
                    new_green_ins["Queen"] -= 1
                elif move[0] == 'B':
                    new_green_ins["Beetle"] -= 1
                elif move[0] == 'S':
                    new_green_ins["Spider"] -= 1
                elif move[0] == 'C':
                    new_green_ins["Cicada"] -= 1
                elif move[0] == 'A':
                    new_green_ins["Ant"] -= 1

            value, _ = minimax(turn + 1, depth - 1, not maximizing, new_board,
new_red_ins, new_green_ins,
                               new_red_q, new_green_q, new_reds_onboard,
new_greens_onboard, alpha, beta)
            if value < best_sib:
                best_sib = value
                best_move = move
            beta = min(beta, best_sib)

            if beta <= alpha:
                break

        return best_sib, best_move
```

با استفاده از مقاله ای که در کنار پروژه قرار داده شده است، حرکاتی که بهینه و بهترین هستند و در چهار مرحله اول بازی برای هوش مصنوعی و برای انسان پیش بینی میشود را در تابع `available_moves` پیاده سازی کردیم. برای مراحل بعد نیز تمام حرکاتی که در آن میتوان مهره‌ای را روی برد قرار داد را در نظر میگیریم.

```
def available_moves(board, turn, red_ins, green_ins, red_q, green_q,
        reds_onboard, greens_onboard):

    moves = []
    if turn == 0:
        for i in ['S', 'G', 'B']:
            moves.append([i, None, (mid_row,mid_col)])
    elif turn == 1:
```

```
        for i in ['S', 'G', 'B']:
            for j in range(len(co)):
                moves.append([i, None, (mid_row+co[j][0],mid_col+co[j][1])])

    elif turn == 2: #red's 2nd time
        for i in range(len(co)):
            row = mid_row+co[i][0]
            col = mid_col+co[i][1]
            if is_insect(row, col, board) != 'n':
                continue
            if has_enemy_around(row, col, board, 'g') == 0:
                moves.append(['Q', None, (row,col)])
                moves.append(['S', None, (row,col)])

    elif turn == 3: #green's 2nd time
        for cc in range(len(co)):
            roww = mid_row+co[cc][0]
            coll = mid_col+co[cc][1]
            tmp = is_insect(co[cc][0] + roww, co[cc][1] + coll, board)
            if tmp == 'g': #found where the first green was put
                for i in range(len(co)):
                    row = roww+co[i][0]
                    col = coll+co[i][1]
                    if is_insect(row, col, board) != 'n':
                        continue
                    if has_enemy_around(row, col, board, 'r') == 0:
                        if green_ins["Queen"] == 1: #if human player didn't put
Queen on their 1st turn
                            moves.append(['Q', None, (row,col)])
                        moves.append(['S', None, (row,col)])
                break

    elif turn == 4: #red's 3rd time
        for r in range(len(reds_onboard)):
            roww = reds_onboard[r][0]
            coll = reds_onboard[r][1]
            for i in range(len(co)):
                row = roww+co[i][0]
                col = coll+co[i][1]
                if is_insect(row, col, board) != 'n':
                    continue
                if has_enemy_around(row, col, board, 'g') == 0:
                    if red_ins["Queen"] == 1:
                        moves.append(['Q', None, (row,col)])
                    if red_ins["Beetle"] > 0:
```

```
                    moves.append(['B', None, (row,col)])
                if red_ins["Spider"] > 0:
                    moves.append(['S', None, (row,col)])

    elif turn == 5: #green's 3rd time
        for g in range(len(greens_onboard)):
            roww = greens_onboard[g][0]
            coll = greens_onboard[g][1]
            for i in range(len(co)):
                row = roww+co[i][0]
                col = coll+co[i][1]
                if is_insect(row, col, board) != 'n':
                    continue
                if has_enemy_around(row, col, board, 'r') == 0:
                    if green_ins["Queen"] == 1:
                        moves.append(['Q', None, (row,col)])
                    if green_ins["Beetle"] > 0:
                        moves.append(['B', None, (row,col)])
                    if green_ins["Spider"] > 0:
                        moves.append(['S', None, (row,col)])

    elif turn == 6: #red's 4th time
        for r in range(len(reds_onboard)):
            roww = reds_onboard[r][0]
            coll = reds_onboard[r][1]
            for i in range(len(co)):
                row = roww+co[i][0]
                col = coll+co[i][1]
                if is_insect(row, col, board) != 'n':
                    continue
                if has_enemy_around(row, col, board, 'g') == 0:
                    if green_ins["Queen"] == 1:
                        moves.append(['Q', None, (row,col)])
                    else:
                        if green_ins["Beetle"] > 0:
                            moves.append(['B', None, (row,col)])
                        if green_ins["Spider"] > 0:
                            moves.append(['S', None, (row,col)])
                        if green_ins["Ant"] > 0:
                            moves.append(['A', None, (row,col)])

    elif turn == 7: #green's 4th time
        for g in range(len(greens_onboard)):
            roww = greens_onboard[g][0]
            coll = greens_onboard[g][1]
```

```
            for i in range(len(co)):
                row = roww+co[i][0]
                col = coll+co[i][1]
                if is_insect(row, col, board) != 'n':
                    continue
                if has_enemy_around(row, col, board, 'r') == 0:
                    if green_ins["Queen"] == 1:
                        moves.append(['Q', None, (row,col)])
                    else:
                        if green_ins["Beetle"] > 0:
                            moves.append(['B', None, (row,col)])
                        if green_ins["Spider"] > 0:
                            moves.append(['S', None, (row,col)])
                        if green_ins["Ant"] > 0:
                            moves.append(['A', None, (row,col)])
    else: #mid game
        available_cells = []
        friends_onboard = reds_onboard
        enemy = 'g'
        friends_ins = red_ins
        if turn%2==1: #red
            friends_onboard = greens_onboard
            enemy = 'r'
            friends_ins = green_ins
        for ins in friends_onboard:
            ins_row = ins[0]
            ins_col = ins[1]
            for i in range(len(co)):
                row = ins_row + co[i][0]
                col = ins_col + co[i][1]
                if is_insect(row, col, board) != 'n':
                    continue
                if has_enemy_around(row, col, board, enemy) == 0:
                    available_cells.append([row,col])
        for cell in available_cells:
            if friends_ins["Beetle"] > 0:
                moves.append(['B', None, (cell[0],cell[1])])
            if friends_ins["Cicada"] > 0:
                moves.append(['C', None, (cell[0],cell[1])])
            if friends_ins["Spider"] > 0:
                moves.append(['S', None, (cell[0],cell[1])])
            if friends_ins["Ant"] > 0:
                moves.append(['A', None, (cell[0],cell[1])])
    random.shuffle(moves)
    return moves
```

تابع `evaluate` با توجه به تعداد مهره‌های اطراف ملکه حریف و ملکه ما، مقداری را به حرکت مورد نظر می‌دهد و آن `score` را برمی‌گرداند.

```
def evaluate (turn, board, red_q, green_q):
    around_red = 0
    around_green = 0
    score = 0
    for i in range(len(co)):
        if len(red_q) > 0:
            r = red_q[0] + co[i][0]
            c = red_q[1] + co[i][1]
            ins = is_insect(r, c, board)
            if ins != 'n' and ins != 'o':
                around_red += 1
        if len(green_q) > 0:
            r = green_q[0] + co[i][0]
            c = green_q[1] + co[i][1]
            ins = is_insect(r, c, board)
            if ins != 'n' and ins != 'o':
                around_green += 1

    if turn%2 == 0: #red
        score = around_green - around_red
    else:
        score = around_red - around_green
    return score
```