Sztuczna inteligencja i inżynieria wiedzy

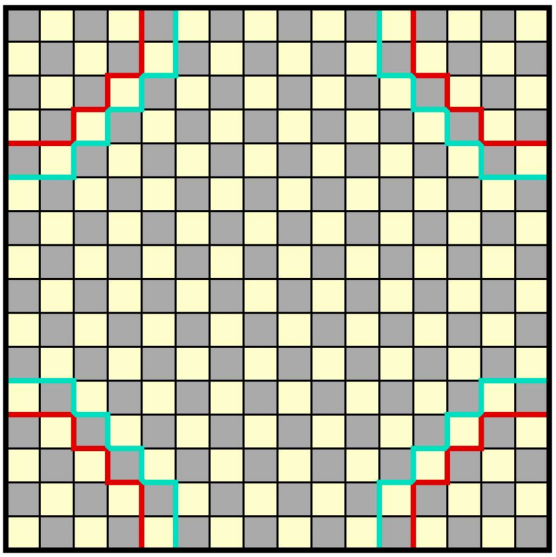
Lista 1

Dawid Walkiewicz

Halma

Halma to strategiczna gra planszowa, która została wynaleziona przez George'a Howarda Monksa w 1883 roku. Przeważnie grana przez 2 lub 4 osoby na kwadratowej planszy 16x16, lecz istnieją inne wersje. Celem gry jest przemieszczenie wszystkich swoich pionków ze strefy startowej do przeciwnego narożnika planszy (strefy startowej przeciwnika). Wygrywa gracz, który dokona tego jako pierwszy.

Gracze wykonują ruchy na przemian, przemieszczając jeden pionek. Pionek można o jedno pole dookoła niego lub wykonać jeden lub serię skoków nad innymi pionkami w zasięgu. Skok może być wykonany nad własnym pionkiem lub pionkiem przeciwnika. Po każdym skoku gracz może kontynuować skakanie, jeśli jest to możliwe.



Rysunek 1 - Plansza od 2 do 4 graczy, źródło: https://pl.wikipedia.org/wiki/Halma

Implementacja

Planszę traktuję jako tablicę tablic z biblioteki numpy. Taka tablica jest dodatkowo opakowana w klasę, zawierającą jeszcze informację, do którego gracza należy tura oraz funkcje wykonywania ruchów. Sam ruch również traktowany jest jako klasa. Poza tym jest również funkcja generowania możliwych dla ruchów.

```
class HalmaState:
    def __init__(self, board, current_player):
        self.board = board
        self.current_player = current_player
```

```
    def generate_moves(self):
        moves = []
        for x in range(16):
            for y in range(16):
                if self.board[x][y] == self.current_player:
                    moves.extend(generate_moves_with_jumps(self.board,
self.current_player, x, y))

        moves = set(moves)
        return list(moves)

    def make_move(self, move):
        self.board[move.end[0]][move.end[1]] = self.current_player
        self.board[move.start[0]][move.start[1]] = 0

    def unmove(self, move):
        self.board[move.start[0]][move.start[1]] = self.current_player
        self.board[move.end[0]][move.end[1]] = 0

    def __str__(self):
        return str(self.board)

    def __repr__(self):
        return str(self.board)

    def __hash__(self):
        return hash(self.board.tobytes() + bytes([self.current_player]))


class Move:
    def __init__(self, start, end):
        self.start = start
        self.end = end

    def check_if_legal(self, state):
        start_x, start_y = self.start
        end_x, end_y = self.end

        if state.board[start_x][start_y] != state.current_player:
            return False

        if state.board[end_x][end_y] != 0:
            return False

        if abs(end_x - start_x) <= 1 and abs(end_y - start_y) <= 1:
            return True

        return self._check_jumps(start_x, start_y, end_x, end_y, state,
set())

    def _check_jumps(self, current_x, current_y, end_x, end_y, state,
visited):
        if (current_x, current_y) == (end_x, end_y):
```

```
            return True
        visited.add((current_x, current_y))

        directions = [(-2, -2), (-2, 2), (2, -2), (2, 2), (2, 0), (-2, 0),
(0, 2), (0, -2)]
        for dx, dy in directions:
            nx, ny = current_x + dx, current_y + dy
            mx, my = current_x + dx // 2, current_y + dy // 2

            if 0 <= nx < 16 and 0 <= ny < 16 and (nx, ny) not in visited:
                if state.board[nx][ny] == 0 and state.board[mx][my] in [1,
2]:
                    if self._check_jumps(nx, ny, end_x, end_y, state,
visited):
                        return True
        return False

    def __str__(self):
        return f"{self.start} -> {self.end}"

    def __repr__(self):
        return f"{self.start} -> {self.end}"


def generate_moves_with_jumps(board, current_player, x, y, visited=None):
    if visited is None:
        visited = set()
    visited.add((x, y))

    if board[x][y] != current_player:
        return []

    directions = [(1, 0), (-1, 0), (0, 1), (0, -1), (1, 1), (1, -1), (-1,
1), (-1, -1)]
    moves = []
    for dx, dy in directions:
        new_x, new_y = x + dx, y + dy
        jump_x, jump_y = new_x + dx, new_y + dy
        if 0 <= new_x < 16 and 0 <= new_y < 16 and (new_x, new_y) not in
visited:
            if board[new_x][new_y] == 0:  # Simple move
                # move = Move((x, y), (new_x, new_y))
                # if check_if_legal_2(board, move, current_player):
                #     moves.append(move)
                moves.append(Move((x, y), (new_x, new_y)))
            elif board[new_x][new_y] in [1, 2] and 0 <= jump_x < 16 and 16
> jump_y >= 0 == board[jump_x][jump_y] and (
                    jump_x, jump_y) not in visited:
                # move = Move((x, y), (jump_x, jump_y))
                # if check_if_legal_2(board, move, current_player):
                #     moves.append(move)
                moves.append(Move((x, y), (jump_x, jump_y)))
```

```
                moves.extend(generate_moves_with_jumps(board,
current_player, jump_x, jump_y, visited.copy()))
    return moves
```

Początkowo przy każdym wykonaniu ruchu tworzony był nowy stan gry, przy czym wykonywano głęboką kopię planszy gry. Taka operacja była czasochłonna i spowalniała grę. W obecnej wersji wszystkie ruchy wykonywane są na tym samym stanie, jednak po sprawdzeniu ruchy są wycofywane, przywracając stan gry. Ta zmiana przyspieszyła działanie gry, w zależności od algorytmu i heurystyk, od 2 do 5 razy.

Prócz tego stworzyłem również kilka dodatkowych funkcji pomagających w zarządzaniu planszą gry.

```
def initialize_game_board_13():
    board = np.zeros((16, 16), dtype=int)

    player_1 = get_target_positions(2, 13)
    player_2 = get_target_positions(1, 13)

    for x, y in player_1:
        board[x][y] = 1

    for x, y in player_2:
        board[x][y] = 2

    return board


def initialize_game_board_19():
    board = np.zeros((16, 16), dtype=int)

    player_1 = get_target_positions(2)
    player_2 = get_target_positions(1)

    for x, y in player_1:
        board[x][y] = 1

    for x, y in player_2:
        board[x][y] = 2

    return board


def get_target_positions(player, number_of_pieces=19):
    if number_of_pieces == 19:
        if player == 1:
            return [(15, 15), (15, 14), (15, 13), (15, 12), (15, 11), (14,
15), (14, 14), (14, 13), (14, 12),
                    (14, 11), (13, 15), (13, 14), (13, 13), (13, 12), (12,
15), (12, 14), (12, 13), (11, 15), (11, 14)]
        else:
            return [(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (1, 0), (1, 1),
```

```
(1, 2), (1, 3), (1, 4), (2, 0), (2, 1),
                    (2, 2), (2, 3), (3, 0), (3, 1), (3, 2), (4, 0), (4, 1)]
    else:
        if player == 1:
            return [(15, 15), (15, 14), (15, 13), (15, 12), (14, 15), (14,
14), (14, 13), (13, 15), (13, 14), (12, 15),
                    (11, 15), (11, 14), (10, 15), (9, 15), (8, 15), (7,
15), (6, 15), (5, 15), (4, 15)]
        else:
            return [(0, 0), (0, 1), (0, 2), (0, 3), (1, 0), (1, 1), (1, 2),
(2, 0), (2, 1), (3, 0), (4, 0), (4, 1),
                    (5, 0), (5, 1), (6, 0), (6, 1), (7, 0), (7, 1), (8, 0)]


def check_game_finished(board):
    player_2 = get_target_positions(2)
    player_1 = get_target_positions(1)

    player_1_count = 0
    player_2_count = 0

    for x, y in player_1:
        if board[x][y] == 1:
            player_1_count += 1

    for x, y in player_2:
        if board[x][y] == 2:
            player_2_count += 1

    if player_1_count == 19:
        return 1

    if player_2_count == 19:
        return 2

    return 0
```

Minimax

Algorytm Minimax – fundamentalny algorytm używany do znajdowania optymalnego ruchu dla gracza w grach o sumie zerowej z dwoma uczestnikami, np. szachy, warcaby, kółko i krzyżyk, go, halma i wiele innych. Algorytm przeszukuje drzewo gry do określonej głębokości i używa funkcji heurystycznej do oceny wartości pozycji na planszy, kiedy osiągnie limit głębokości.

Opis kroków działania:

1. Dla obecnego stanu planszy generowane są wszystkie możliwe ruchy na przemian dla obu graczy, tworząc drzewo możliwych gier. Węzły reprezentują pozycje na planszy, a krawędzie ruchy.
2. Minimax przeszukuje drzewo rekurencyjnie, oceniając stan gry na liściach przy użyciu funkcji heurystycznej.
3. Po ocenie liści, wartości są propagowane wstecz do korzenia: każdy ruch MAX wybiera maksymalną wartość swoich dzieci, a każdy ruch MIN wybiera minimalną wartość swoich dzieci.
4. Algorytm kończy się, gdy osiągnie korzeń drzewa, wybierając ruch, który prowadzi do najlepszego możliwego wyniku przy założeniu optymalnej gry przez przeciwnika.

Minimax jest skuteczny w grach o stosunkowo małej przestrzeni stanów lub tam, gdzie drzewo gry można przeszukać do znacznej głębokości bez dużego kosztu obliczeniowego. W przypadku gier o większej przestrzeni stanów, jak szachy czy go, Minimax może wymagać znacznych zasobów lub zaawansowanych technik redukcji przestrzeni przeszukiwań.

Implementacja

Funkcja implementująca algorytm zwraca 3 wartości: ewaluację ruchu, najlepszy ruch oraz liczbę odwiedzonych odnóg drzewa.

```
def minimax(state, depth, maximizing_now, heuristic):
    node_count = 1

    if depth == 0:
        return heuristic(state, state.current_player), None, node_count

    if maximizing_now:
        max_eval = float('-inf')
        best_move = None
        total_nodes = 0

        for move in state.generate_moves():
            state.make_move(move)
            state.current_player = 2 if state.current_player == 1 else 1

            eval,_, nodes_visited = minimax(state, depth - 1, False,
heuristic)
            total_nodes += nodes_visited
```

```
            state.current_player = 2 if state.current_player == 1 else 1
            state.unmove(move)

            if eval > max_eval:
                max_eval = eval
                best_move = move

        return max_eval, best_move, total_nodes + node_count
    else:
        min_eval = float('inf')
        total_nodes = 0

        for move in state.generate_moves():
            state.make_move(move)
            state.current_player = 2 if state.current_player == 1 else 1

            eval,_, nodes_visited = minimax(state, depth - 1, True,
heuristic)
            total_nodes += nodes_visited

            state.current_player = 2 if state.current_player == 1 else 1
            state.unmove(move)
            min_eval = min(min_eval, eval)
        return min_eval, None, total_nodes + node_count
```

Cięcie Alfa-beta

Algorytm alfa-beta – technika optymalizacji dla algorytmu minima. Znacznie zmniejsza liczbę ocenianych węzłów drzewa gry podczas szukania najlepszego ruchu. Przycinanie alfa-beta polega na pomijaniu tych gałęzi, które nie mogą wpłynąć na decyzję ostateczną, co skutkuje szybszym znalezieniem najlepszego ruchu bez przeglądania każdej możliwości.

Alfa – najlepsza zbadana do tej pory opcja na ścieżce do korzenia przy maksymalizowaniu. Inaczej dolna granica, którą maksymalizator wie, że może osiągnąć.

Beta – najlepsza zbadana do tej pory opcja na ścieżce do korzenia przy minimalizowaniu. Inaczej górna granica, którą minimalizator wie, że może osiągnąć.

Działanie:

1. Algorytm rozpoczyna z wartościami alfa = - nieskończoność, a beta = + nieskończoność.
2. Kiedy algorytm rozważa ruchy MAX, aktualizuje wartość alfa, gdy znajduje lepszy ruch (większa wartość). Jeśli wartość beta stanie się mniejsza lub równa alfa, dalsze przeszukiwanie tej gałęzi jest bezcelowe, ponieważ minimalizator nigdy nie pozwoli dojść do tego punktu.
3. Kiedy przeszukiwanie dotyczy MIN, aktualizuje wartość beta, gdy znajdzie ruch lepszy (mniejsza wartość). Jeśli alfa staje się większa lub równa beta, dalsze przeszukiwanie tej gałęzi jest przerywane, gdyż maksymalizator nigdy nie wybierze tej opcji.

Dzięki tym zmianom algorytm minimax z przycinaniem alfa-beta staje się znacznie szybszy niż czysty algorytm minimax.

Implementacja

```
def minimax_alpha_beta(state, depth, alpha, beta, maximizing_now,
heuristic):
    node_count = 1

    if depth == 0:
        return heuristic(state, state.current_player), None, node_count

    if maximizing_now:
        max_eval = float('-inf')
        best_move = None
        total_nodes = 0

        for move in state.generate_moves():
            state.make_move(move)
            state.current_player = 2 if state.current_player == 1 else 1

            eval, _, nodes_visited = minimax_alpha_beta(state, depth - 1,
alpha, beta, False, heuristic)
            total_nodes += nodes_visited
```

```
            state.current_player = 2 if state.current_player == 1 else 1
            state.unmove(move)
            if eval > max_eval:
                max_eval = eval
                best_move = move
            alpha = max(alpha, eval)
            if beta <= alpha:
                break
        return max_eval, best_move, total_nodes + node_count
    else:
        min_eval = float('inf')
        total_nodes = 0

        for move in state.generate_moves():
            state.make_move(move)
            state.current_player = 2 if state.current_player == 1 else 1

            eval, _, nodes_visited = minimax_alpha_beta(state, depth - 1,
alpha, beta, True, heuristic)
            total_nodes += nodes_visited

            state.current_player = 2 if state.current_player == 1 else 1
            state.unmove(move)

            min_eval = min(min_eval, eval)
            beta = min(beta, eval)
            if beta <= alpha:
                break
        return min_eval, None, total_nodes + node_count
```

Heurystyki

Jako główną metodę przyznawania wag różnym polom planszy przyjąłem odległość euklidesową, choć z dodatkowymi wagami dla specyficznych pól. Jednym z powodów było „utykanie” pionków blisko celu. Problem ten rozwiązałem poprzez zdefiniowanie dwóch „stef”, jednej z ujemną wagą, drugiej z dodatnimi wagami.

```
def distance_heuristic(state, current_player=1):
    player_target = (15, 15) if current_player == 1 else (0, 0)
    dead_zone_player = (12, 12) if current_player == 1 else (3, 3)
    golden_zone_player = {(11, 15), (11, 14), (13, 13), (15, 11), (14, 11)}
if current_player == 1 else {(0, 4), (1, 4),

(2, 2), (4, 0),

(4, 1)}

    player_score = 0

    for y in range(16):
        for x in range(16):
            current_piece = state.board[x][y]
            if current_piece == current_player:
                dx, dy = x - player_target[0], y - player_target[1]
                distance = math.sqrt(dx * dx + dy * dy)
                player_score += 22 - distance
                if distance <= 4:
                    player_score += 5

                if (x, y) == dead_zone_player:
                    player_score -= 20

                if (x, y) in golden_zone_player:
                    player_score += 15

    return player_score
```

Na podstawie tej heurystyki zdefiniowałem kolejne dwie. Pierwsza z nich odmiennie przypisuje punkty przy blisko celu. W tym celu ma słownik pól wraz z ich wagami.

```
def wall_corner_heuristic(state, current_player):
    target_x, target_y = (15, 15) if current_player == 1 else (0, 0)

    edges_player_1 = {(11, 14): 9, (11, 15): 4, (12, 15): 4, (13, 15): 5,
(14, 15): 6, (15, 15): 10,
                      (15, 14): 6, (15, 13): 5, (15, 12): 4, (15, 11): 4,
(14, 11): 9}
    edges_player_2 = {(1, 0): 6, (2, 0): 5, (3, 0): 4, (4, 0): 4, (4, 1):
9, (0, 0): 10, (0, 1): 6,
                      (0, 2): 5, (0, 3): 4, (0, 4): 4, (1, 4): 9}

    edge_bonuses = edges_player_1 if current_player == 1 else
```

```
edges_player_2

    score = 0

    for x in range(16):
        for y in range(16):
            if state.board[x][y] == current_player:
                distance = math.sqrt((x - target_x) ** 2 + (y - target_y)
** 2)
                score += 22 - distance

                if (x, y) in edge_bonuses:
                    score += edge_bonuses[(x, y)]

    return score
```

Druga z pokrewnych heurystyk jest bardziej skomplikowana. Opiera się na założeniu, że pionki gracza nie powinny być zbyt skupione i nie wywoływać „korków”. W tym celu wylicza środek masy pionków gracza. Niestety ze względu na większą złożoność obliczeń jest ona najwolniej działającą z moich heurystyk.

```
def goal_dispersion_heuristic(state, current_player):
    player_target = (15, 15) if current_player == 1 else (0, 0)
    dead_zone = (12, 12) if current_player == 1 else (3, 3)
    golden_zone_player = {(11, 15), (11, 14), (13, 13), (15, 11), (14, 11)}
if current_player == 1 else {(0, 4), (1, 4),

(2, 2), (4, 0),

(4, 1)}
    center_mass_x = 0
    center_mass_y = 0
    count = 0
    progression_score = 0

    penalty_zone_player_1 = {(0, 0): 10, (0, 1): 8, (0, 2): 6, (0, 3): 4,
(0, 4): 2, (1, 0): 8, (1, 1): 6, (1, 2): 4,
                             (1, 3): 2, (1, 4): 1, (2, 0): 6, (2, 1): 4,
(2, 2): 2,
                             (2, 3): 1, (3, 0): 4, (3, 1): 2, (3, 2): 1,
(4, 0): 2, (4, 1): 1}
    penalty_zone_player_2 = {(11, 14): 1, (11, 15): 2, (12, 13): 1, (12,
14): 2, (12, 15): 4, (13, 12): 1, (13, 13): 2,
                             (13, 14): 4, (13, 15): 6, (14, 11): 1, (14,
12): 2, (14, 13): 4, (14, 14): 6, (14, 15): 8,
                             (15, 11): 2, (15, 12): 4, (15, 13): 6, (15,
14): 8, (15, 15): 10}

    penalty_zone = penalty_zone_player_1 if current_player == 1 else
penalty_zone_player_2

    for y in range(16):
```

```
        for x in range(16):
            if state.board[x][y] == current_player:
                distance = math.sqrt((x - player_target[0]) ** 2 + (y -
player_target[1]) ** 2)
                progression_score += 22 - distance
                if distance <= 4:
                    progression_score += 5

                if (x, y) == dead_zone:
                    progression_score -= 20

                if (x, y) in golden_zone_player:
                    progression_score += 15

                if (x, y) in penalty_zone:
                    progression_score -= penalty_zone[(x, y)]

                center_mass_x += x
                center_mass_y += y
                count += 1

    if count > 0:
        center_mass_x /= count
        center_mass_y /= count

    dispersion_score = 0
    for y in range(16):
        for x in range(16):
            if state.board[x][y] == current_player:
                distance_to_center = math.sqrt((x - center_mass_x) ** 2 +
(y - center_mass_y) ** 2)
                dispersion_score += distance_to_center

    return progression_score * 0.6 + dispersion_score * 0.4
```

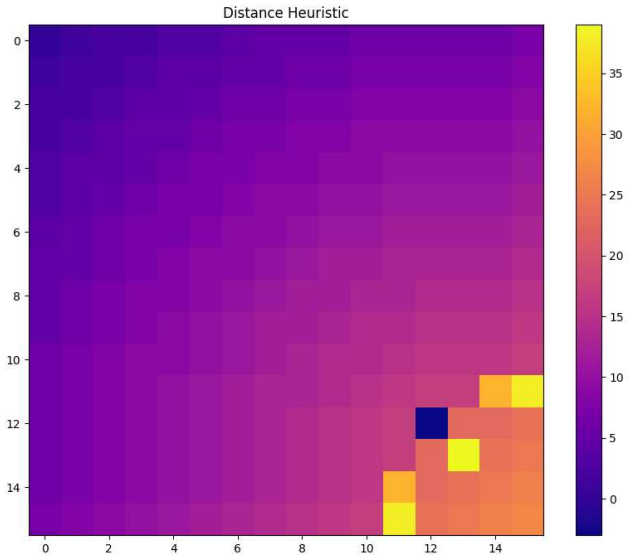
Ostania z moich heurystyk nie opiera się na odległości euklidesowej, zamiast tego wykorzystuje odległość Manhattan. Tutaj również potrzeba było wprowadzić kilka poprawek, gdyż funkcja „utykała” przy samym celu. Dodatkowo istniały przypadki, gdy algorytm nie wyprowadzał w porę wszystkich pionków z „bazy” i tym sposobem blokowana była cała gra. Z tego powodu dodałem ujemne wagi na obszarach startowych obu graczy.

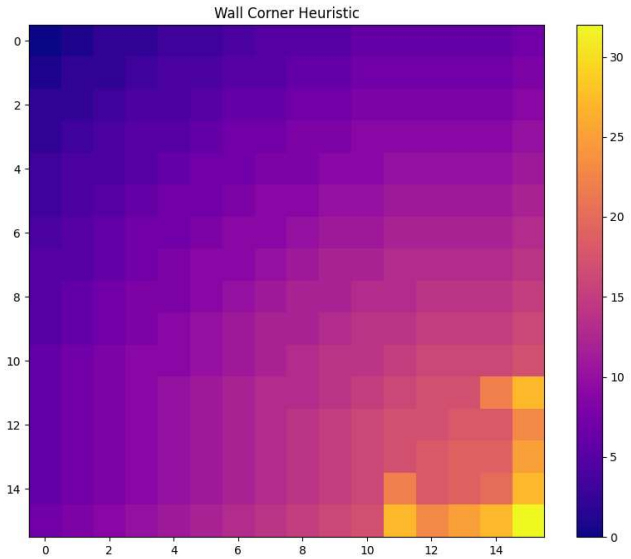
```
def manhattan_heuristic(state, current_player):
    player_target = (15, 15) if current_player == 1 else (0, 0)
    dead_zone = [(10, 15), (15, 10)] if current_player == 1 else [(0, 5),
(5, 0)]

    penalty_zone_player = {(0, 0): 10, (0, 1): 8, (0, 2): 6, (0, 3): 4, (0,
4): 2, (1, 0): 8, (1, 1): 6, (1, 2): 4,
                           (1, 3): 2, (1, 4): 1, (2, 0): 6, (2, 1): 4, (2,
2): 2,
                           (2, 3): 1, (3, 0): 4, (3, 1): 2, (3, 2): 1, (4,
0): 2,
```

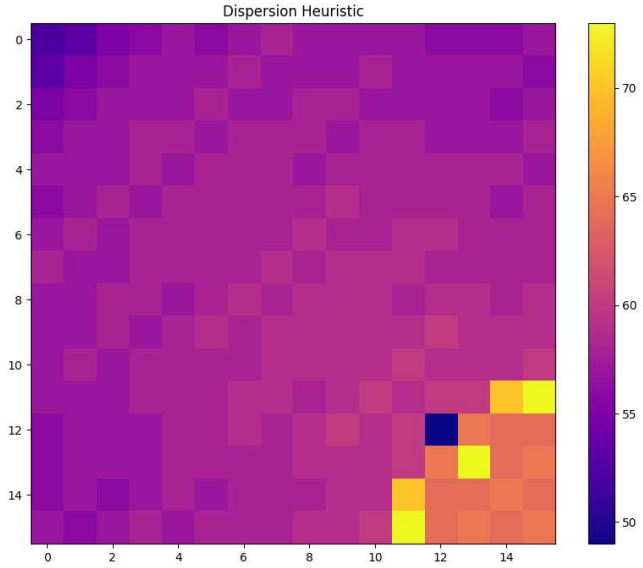
```
                            (4, 1): 1} \
        if current_player == 1 else {(11, 14): 1, (11, 15): 2, (12, 13): 1,
(12, 14): 2,
                                     (12, 15): 4, (13, 12): 1, (13, 13): 2,
(13, 14): 4,
                                     (13, 15): 6, (14, 11): 1, (14, 12): 2,
(14, 13): 4,
                                     (14, 14): 6, (14, 15): 8, (15, 11): 2,
(15, 12): 4,
                                     (15, 13): 6, (15, 14): 8, (15, 15):
10}

    player_score = 0

    for y in range(16):
        for x in range(16):
            current_piece = state.board[x][y]
            distance = abs(x - player_target[0]) + abs(y -
player_target[1])
            if current_piece == current_player:
                player_score -= distance

                if distance <= 5:
                    player_score += 10

                if (x, y) in dead_zone:
                    player_score -= 10

                if (x, y) in penalty_zone_player:
                    player_score -= penalty_zone_player[(x, y)]

    return player_score
```

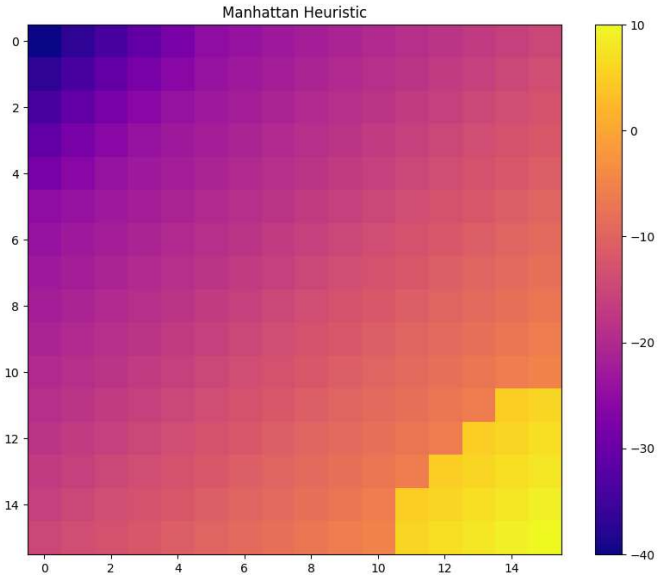
Wizualizacja wag poszczególnych pól

W celach efektywniejszego wykrywania błędów skorzystałem z biblioteki matplotlib aby wygenerować mapy termiczne (heat map) planszy z wagami pól. Poniżej widać te wykresy dla wcześniej zdefiniowanych heurystyk.






Dispersion Heuristic
0
2
4
6
8
10
12
14
0
2
4
6
8
10
12
14
70
65
60
55
50


Manhattan Heuristic
0
2
4
6
8
10
12
14
0
2
4
6
8
10
12
14
10
0
−10
−20
−30
−40

Eksperymenty

Przeprowadziłem dwa rodzaje eksperymentów: gracz stosujący minimax bez cięcia alfa-beta przeciwko graczowi stosującemu minimax z cięciem alfa-beta oraz starcie graczy stosujących minimax z cięciem alfa-beta z użyciem różnych heurystyk.

Z przyczyn czasowych testy heurystyk wykonałem jedynie z użyciem cięcia alfa-beta oraz ograniczyłem liczbę eksperymentów algorytmu minimax bez cięcia alfa-beta.

Minimax z i bez cięcia alfa-beta

Przy wykonywaniu testów użyłem heurystyki dystansowej (distance_heuristic).

Minimax bez cięcia alfa-beta

Czas gry: około 287 sekund (4 min i 47 sekund)

Liczba odwiedzonych węzłów drzewa: około 3 000 000

Z uwagi na czas działania algorytmu został on włączony tylko 3 razy (w porównaniu do wersji z cięciem alfa-beta, która była włączona 10 razy)

Minimax z cięciem alfa-beta

Czas gry: około 26 sekund

Liczba odwiedzonych węzłów drzewa: około 220 000

Porównianie wyników gry pomiędzy minimax z i bez alfa-beta

Z wykonanych 10 gier wynika, że

Wnioski

Jak widać dzięki zastosowaniu cięcia alfa-beta algorytm minimax stał się zauważalnie szybszy (około 10 razy). Wynika to z faktu, że musiał odwiedzić znacznie mniej (ponad 10 razy mniej) węzłów drzewa.

Porównanie heurystyk

Sprawdziłem wszystkie 16 kombinacji heurystyk, dla każdej wykonując po 10 testów. Podsumowane wyniki znajdują się poniżej.

Średni czas gry

	distance	wall_corner	dispersion	manhattan
distance	26,72s	25,29s	36,08s	26,04s
wall_corner	24,23s	24,34s	36,02s	26,02s
dispersion	35,53s	36,60s	53,01s	39,35s
manhattan	26,14s	26,14s	36,16s	26,95s

Średnia liczba ruchów

	distance	wall_corner	dispersion	manhattan
distance	339	339	315	316
wall_corner	335	332	313	311
dispersion	313	312	360	320
manhattan	315	311	304	318

Kto najczęściej wygrał

	distance	wall_corner	dispersion	manhattan
distance	-	wall_corner(8)	distance(10)	remis(4:6)
wall_corner	wall_corner(9)	-	wall_corner(10)	wall_corner(9)
dispersion	distance(10)	wall_corner(10)	-	manhattan(10)
manhattan	remis(5:5)	wall_corner(7)	manhattan(10)	-

Wygrane pomiędzy graczem 1 i 2:

Wygrane bez liczenia gier przeciwko tej samej heurystyce

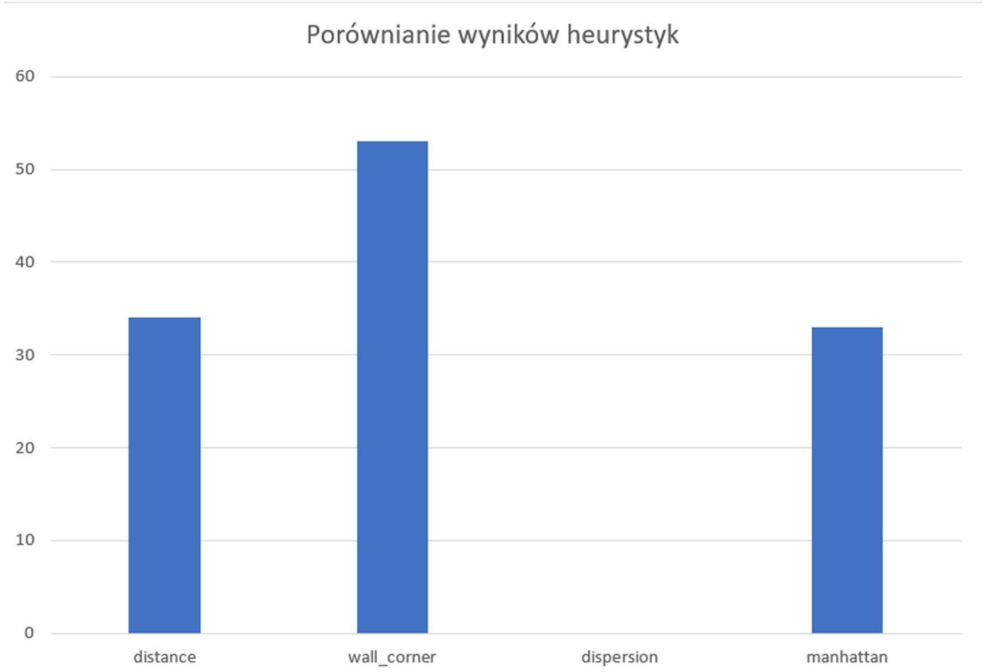
	Liczba wygranych	Procent wygranych
Gracz 1	16 + 25 + 0 + 15 = 56	47%
Gracz 2	14 + 5 + 30 + 15 = 64	53%

Wygrane z liczeniem gier przeciwko tej samej heurystyce

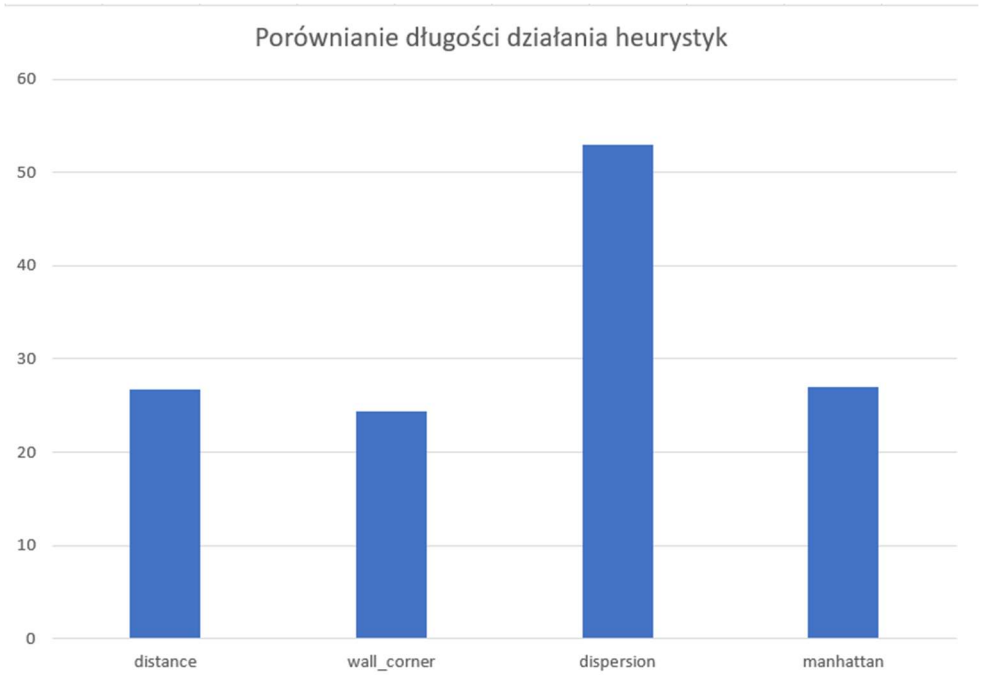
	Liczba wygranych	Procent wygranych
Gracz 1	56 + 7 + 4 + 3 + 3 = 73	46%
Gracz 2	64 +3 + 6 + 7 + 7 = 87	54%

Według tych danych gracz 2 ma lekko większą szansę na wygraną, lecz może to być jedynie błąd statystyczny wynikający z małej ilości danych.

Graficzne przedstawienie wybranych statystyk



Jak widać na powyższym wykresie heurystyką z najlepszymi wynikami okazała się heurystyka wall_corner przypisująca wagi dla krawędzi pola docelowego. Zdecydowanie najgorszą heurystyką okazała się natomiast heurystyka dipersion, która przeciwdziała tworzeniu się „korków". Nie wygrała ona ani razu.



Z kolejnego wykresu wynika natomiast, że czas działania poszczególnych heurystyk jest podobny, z wyjątkiem heurystyki dispersion, która jest zauważalnie wolniejsza.

Algorytm minimax dynamicznie wybierający heurystykę

Zmodyfikowana wersja algorytmu minimax wybierającą największą wartość spośród heurystyk. Wykorzystuje cięcie alfa-beta.

Implementacja

Implementacja różni się od zwykłego minimax jedynie tym, że w liściu wywoływane są wszystkie heurystyki i zwracana jest spośród nich maksymalna wartość.

```
def minimax_alpha_beta_2(state, depth, alpha, beta, maximizing_now):
    node_count = 1

    if depth == 0:
        evals = [distance_heuristic(state, state.current_player),
                 goal_dispersion_heuristic(state, state.current_player),
                 wall_corner_heuristic(state, state.current_player),
manhattan_heuristic(state, state.current_player)]
        return max(evals), None, node_count

    if maximizing_now:
        max_eval = float('-inf')
        best_move = None
        total_nodes = 0

        for move in state.generate_moves():
            state.make_move(move)
            state.current_player = 2 if state.current_player == 1 else 1

            eval, _, nodes_visited = minimax_alpha_beta_2(state, depth - 1,
alpha, beta, False)
            total_nodes += nodes_visited

            state.current_player = 2 if state.current_player == 1 else 1
            state.unmove(move)
            if eval > max_eval:
                max_eval = eval
                best_move = move
            alpha = max(alpha, eval)
            if beta <= alpha:
                break
        return max_eval, best_move, total_nodes + node_count
    else:
        min_eval = float('inf')
        total_nodes = 0

        for move in state.generate_moves():
            state.make_move(move)
            state.current_player = 2 if state.current_player == 1 else 1

            eval, _, nodes_visited = minimax_alpha_beta_2(state, depth - 1,
alpha, beta, True)
            total_nodes += nodes_visited
```

```
            state.current_player = 2 if state.current_player == 1 else 1
            state.unmove(move)

            min_eval = min(min_eval, eval)
            beta = min(beta, eval)
            if beta <= alpha:
                break
        return min_eval, None, total_nodes + node_count
```

Eksperyment 1

Aby porównać skuteczność tego algorytmu wykonałem 10 gier przeciwko algorytmowi minimax alfa-beta z heurystyką wall_corner (według wcześniejszych eksperymentów jest najlepsza)

Oto wyniki:

Średni czas trwania gry: 72 sekundy

Średnia liczba ruchów: 338

Wygrane: 1

Wyniki okazały się rozczarowujące. Powodem może być użycie heurystyki dispersion, dlatego przeprowadziłem jeszcze jeden eksperyment bez jej użycia.

Eksperyment 2

Średni czas trwania gry:

Średnia liczba ruchów: 334

Wygrane: 2

Ponownie algorytm w większości przypadków przegrał. Powodem mogą być wartości wag heurystyk. Różne heurystyki mogą zwracać różne wartości na postawie różnych wag, co przełoży się na wybieranie tej z największą wartością, mimo niekoniecznie najlepszego ułożenia pionków.